

Formale Grundlagen der Informatik 1 (Sommer 2012)

Vorlesung: *Automaten, Grammatiken, Formale Sprachen und Berechenbarkeit*

Michael Köhler-Bußmeier

(basierend auf einer Vorlage von Matthias Jantzen)

Arbeitsbereich *Theoretische Grundlagen der Informatik*

Fachbereich Informatik

Universität Hamburg



Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Anmerkungen zur vorliegenden Ausgabe:

Das Skript basiert auf Skripten der alten Diplomstudienordnung. Im Übergang zu Bachelorordnung wurden einige Inhalte anders dargestellt, ungeordnet und einige Teile gekürzt oder sogar weggelassen.

Mit Unterstützung von Übungsgruppenleiter(inne)n und Besuchern früherer Veranstaltungen konnten (einige) Fehler in dieser Ausgabe vermieden werden. Sie halten eine korrigierte und ergänzte Version in den Händen. Dennoch sind Fehler nie ganz auszuschließen. Wenn Sie meinen, einen Fehler gefunden zu haben, dann wenden Sie sich bitte an Michael Köhler-Bußmeier.

Das vorliegende Skript wurde mit L^AT_EX unter Verwendung diverser Zusatzpakete erstellt.

Copyright:

Der vorliegende Text ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen sowie der Speicherung in Datenverarbeitungsanlagen oder auf elektronischen Medien, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Einführendes Beispiel	3
2. Notationen und Grundbegriffe	9
2.1. Notation von Mengen	10
2.2. Relationen	14
2.3. Funktionen	24
2.4. Beweistechniken	27
2.5. Strukturen	31
2.6. Wortmengen	34
2.7. Paar- und Tupelfunktionen	40
2.8. Nummerierungen von Wortmengen	44
3. Endliche Automaten und reguläre Mengen	51
3.1. Deterministische endliche Automaten	51
3.2. Nichtdeterministische endliche Automaten	55
3.3. Endliche Automaten mit Ausgabe	60
3.4. Rationale Ausdrücke	68
3.5. Einfache Abschlusseigenschaften	70
3.6. Ein Anwendungsbeispiel	73
3.7. Äquivalenzbegriffe und minimale Automaten	75
3.8. Das Pumping-Lemma	86
3.9. Operatoren auf Sprachfamilien	87
3.10. Entscheidungsprobleme	92
4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken	95
4.1. Grammatiken	96
4.2. Chomsky-Normalform kontextfreier Grammatiken	100
4.3. Lineare Grammatiken und reguläre Mengen	103
4.4. Ableitungen und Ableitungsbäume	104
4.5. Das Pumping-Lemma der kontextfreien Sprachen	110
4.6. Ein Entscheidbarkeitsresultat	112
4.7. Wichtige Abschlusseigenschaften	114
5. Kellerautomaten	119
5.1. Nichtdeterministische Kellerautomaten	119
5.2. Deterministische Kellerautomaten und weitere Eigenschaften von PDAs	123

Inhaltsverzeichnis

5.3. Kellerautomaten und kontextfreie Sprachen	124
5.4. Deterministisch kontextfreie Sprachen	128
5.5. Abschlusseigenschaften der deterministisch kontextfreien Sprachen	130
6. Syntaxanalyse kontextfreier Sprachen	135
6.1. Aufbau eines Compilers	135
6.2. Deterministische Syntaxanalyse	139
6.3. LR(k)-Grammatiken	141
6.4. LL(k)-Grammatiken	147
7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit	151
7.1. Turing-Maschinen	152
7.2. Berechenbarkeit	156
7.3. Varianten der Turing-Maschine	157
7.4. Nichtdeterministische Turing-Maschinen	159
7.5. Entscheidbarkeit und Aufzählbarkeit	161
7.6. Die universelle Turing-Maschine (UTM)	165
7.7. Unentscheidbarkeit des Halteproblems	166
7.8. k -Keller-Automaten	173
8. Chomsky Typ-0 und Chomsky Typ-1 Grammatiken	177
8.1. Typ-0 Grammatiken, Semi-Thue Systeme	177
8.2. Typ-1 Grammatiken, kontextsensitive Grammatiken	180
8.3. Linear beschränkte Automaten	183
8.4. Die Chomsky-Hierarchie	186
9. Strukturelle Komplexitätstheorie	189
9.1. Zeit- und Platzkomplexität	190
9.2. Komplexitätsklassen	193
9.3. Problemreduktionen und vollständige Probleme	197
A. Historischer Ursprung	207

1. Einleitung

„Die Grenzen meiner Welt sind die Grenzen meiner Sprache.“

„Ich kann in einer Disziplin nur soviel an Wissenschaft entdecken, wie Mathematik in ihr enthalten ist.“

(Ludwig Wittgenstein, 1889 – 1951, Philosoph und Mathematiker)

Informatik ist eine Wissenschaft, die sich mit der Beschreibung, Analyse und Gestaltung informationsverarbeitender Prozesse beschäftigt. Zur Analyse und Aufbereitung von Problemen aus dem täglichen Leben wie auch der Wissenschaft, ist es nötig, ihre strukturellen und mathematischen Aspekte zu erkennen und zu formulieren. Oft erkennt man diese erst, wenn auch die Möglichkeiten zu deren Formulierung zur Verfügung stehen. Da eine Hauptaufgabe der Informatik die Entwicklung von maschinell durchführbaren Verfahren zur Lösung von Problemen der Informationsverarbeitung ist, kommt der formalen Beschreibung der Strukturen, Modelle und Algorithmen eine besondere Bedeutung zu. Während jedoch die Mathematik wie selbstverständlich mit unendlichen Objekten, wie zum Beispiel den reellen Zahlen, oder Zahlen, die nicht rational sind wie π oder e , aber auch mit unendlichen Mengen umgeht, diese definiert, umrechnet und darüber Beweise führt, ist das in der Informatik manchmal nur sehr eingeschränkt möglich. Hier benötigen wir reale Rechner, die keine wirklich unendlichen Objekte bearbeiten können. An deren Stelle werden immer endliche Repräsentationen treten müssen, mit denen dann weiter gearbeitet werden kann. In der (nicht nur theoretischen) Informatik, haben wir es zum größten Teil mit Fragen zu endlichen Repräsentationen beliebiger Objekte und deren adäquaten algorithmischen Behandlung zu tun.

Programme zur Ausführung von Algorithmen auf Rechenmaschinen sind nun kaum brauchbar, wenn sie das gestellte Problem nicht korrekt lösen; oder dies zwar richtig bearbeiten, jedoch nicht mit der nötigen Effizienz. Aussagen über die Korrektheit und Effizienz von Programmen sind aber nur möglich, wenn Syntax und Semantik der verwendeten Konstruktionen und Programmiersprachen exakt festgelegt sind. Die Methoden, die solches erst möglich machen, sind Gegenstand der Theoretischen Informatik. Gleichzeitig mit der formal einwandfreien, mathematischen Formulierung von Modellen, wird der Rahmen geliefert, in dem versprochene Eigenschaften bewiesen werden können.

Die **Theoretische Informatik** befaßt sich mit formalen Grundlagen. In dem Teilgebiet **Automatentheorie** wird zum Beispiel danach gefragt, welche einfachen mathematischen Modelle dem Computer zu Grunde liegen. Die **Theorie der Berechenbarkeit** untersucht, wie man das Berechenbare von dem Nichtberechenbaren abgrenzen kann, indem man Probleme benennt, die ein Computer unter keinen Umständen lösen kann. Die **Komplexitätstheorie** fragt danach, welchen rechnerischen Aufwand die Lösung gewisser Problem erfordert. In dem Teilgebiet **Formale Sprachen** wird der prinzipielle, strukturelle Aufbau von Programmier- und Spezifikationssprachen (deren **Syntax**) untersucht. Die **Logik** bildet die Grundlage für eine formale Semantik von Konstruktionen z.B. mit Programmier- und Spezifikationssprachen.

1. Einleitung

Die Mathematik spielt in der Informatik die Rolle eines „Denkzeugs“, mit dem wir knapp und präzise, ohne Vagheiten und Mehrdeutigkeiten den (abstrakten) Kern einer Sache beschreiben können. Erst auf der Basis eines sauberen theoretischen Fundaments wird es möglich, solche Beschreibungen zu formulieren und deren Analysen vorzunehmen.

Zum Verständnis der (nicht nur theoretischen) Grundlagen der Informatik wird im Anhang, quasi als Zusammenschau und Überblick, ein kurzer Einblick in die historischen Ursprünge dargestellt.

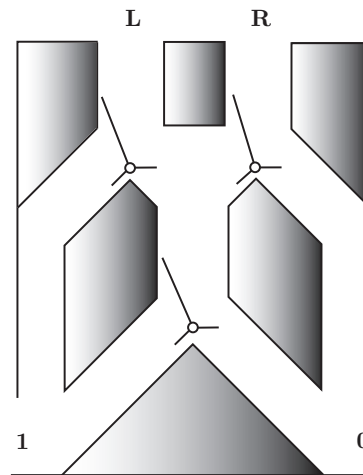
Das vorliegende Skript soll die wesentlichen Notationen und Ergebnisse dieses grundlegenden Teils der Theoretischen Informatik wiedergeben, wobei zur Motivation Anwendungen angesprochen werden. Die zum Verständnis notwendigen mathematischen Notationen und Konzepte werden angesprochen, dürften aber zum großen Teil auch in den vorangegangenen Mathematik-Veranstaltungen bekannt gemacht worden sein und können entsprechenden Lehrbüchern entnommen werden. Trotzdem werden einzelne Beweise relativ ausführlich aufgenommen, weil dadurch die Leserinnen und Leser weitere Anwendungen der in der Mathematik vorgestellten Beweistechniken vorfinden und mit diesen vertraut gemacht werden sollen. Die dabei auftauchende formale Strenge ist jedoch keine Feindin der Einfachheit. Schon Hilbert wies darauf hin, dass die Benutzung streng formaler Methoden dazu anhält, nach klareren und einfacheren Beweisen zu suchen!

Am Ende dieser Unterlagen werden weitere Quellen angegeben, die zum Nachschlagen wie zur Vertiefung des Verständnisses herangezogen werden können.





Zur Einübung sind Übungsaufgaben in den angebotenen Übungsgruppen zu lösen. Diese werden entsprechend ihres unterschiedlichen Schwierigkeitsgrades mit Punkten bewertet. Die aktive Teilnahme an den angebotenen Übungsgruppen ist erforderlich. Es wird nachdrücklich empfohlen, den Stoff in kleinen Arbeitsgruppen von zwei bis höchstens vier Studierenden zu bearbeiten. Besser als durch eigenes Auseinandersetzen mit dem Stoff und zusätzliche Diskussion mit Kommilitonen kann sich ein(e) Studierende[r] kaum für eine Prüfung vorbereiten!


1.1. Einführendes Beispiel

Das folgende Bild zeigt das Innere eines Gerätes, in das eine Kugel oben bei **L** oder **R** eingeworfen werden kann. Die Hebel an den Verzweigungsstellen werden jeweils durch die Kugel beim Passieren umgestellt. Je nach dem wo die Kugel eingeworfen wird und wie dann gerade die Stellung der Hebel ist, wird die Kugel das Gerät bei dem mit **1** oder mit **0** bezeichneten Ausgang verlassen.



Um das Verhalten des Gerätes vollständig zu beschreiben, könnten wir zum Beispiel die nachstehende Liste vervollständigen:

1. Wenn in der Situation  die Kugel bei **L** eingeworfen wird, dann kommt sie beim Ausgang **0** heraus und die Hebelstellung danach ist: .
2. Wenn in der Situation  die Kugel bei **R** eingeworfen wird, dann kommt sie beim Ausgang **0** heraus und die Hebelstellung danach ist: .
3. bis 16. (analog)

Wir können uns die kleinen Bildchen  wie durch kleine Gummistempel erzeugte „Symbole“ vorstellen, und werden in diesem Sinne eine kurze Zeit damit weiterarbeiten.

Die fertiggestellte Liste wäre am Ende sicher nicht sehr übersichtlich, und wenn wir das Verhalten des Gerätes nach mehrfach wiederholtem Einwerfen der Kugel notieren sollten, wären wir wohl etwas länger beschäftigt.

Offensichtlich hängt das Verhalten des Gerätes nur davon ab, welche Schalterstellung gerade vorliegt und wo die Kugel eingeworfen wird. Zudem wird das Ergebnis dadurch offensichtlich eindeutig festgelegt.

Eine angemessenere Form der Beschreibung ist dann die einer mathematischen Funktion δ , mit der Urbildmenge:

$$\{(x_1, x_2) \mid x_1 \in \left\{ \begin{array}{c} \diagdown \diagdown \\ \diagup \diagup \end{array}, \begin{array}{c} \diagdown \diagup \\ \diagup \diagdown \end{array}, \begin{array}{c} \diagdown \diagdown \\ \diagup \diagdown \end{array}, \begin{array}{c} \diagdown \diagup \\ \diagup \diagup \end{array}, \begin{array}{c} \diagup \diagdown \\ \diagdown \diagdown \end{array}, \begin{array}{c} \diagup \diagup \\ \diagdown \diagup \end{array}, \begin{array}{c} \diagup \diagdown \\ \diagup \diagdown \end{array}, \begin{array}{c} \diagup \diagup \\ \diagup \diagup \end{array} \right\} \text{ und } x_2 \in \{\mathbf{L}, \mathbf{R}\}\}$$

1. Einleitung

und der Bildmenge:

$$\{(y_1, y_2) \mid y_1 \in \left\{ \begin{array}{|c|} \hline \diagdown \diagup \\ \hline \end{array}, \begin{array}{|c|} \hline \diagup \diagdown \\ \hline \end{array}, \begin{array}{|c|} \hline \diagdown \diagdown \\ \hline \end{array}, \begin{array}{|c|} \hline \diagup \diagup \\ \hline \end{array}, \begin{array}{|c|} \hline \diagdown \diagup \\ \hline \end{array}, \begin{array}{|c|} \hline \diagup \diagdown \\ \hline \end{array}, \begin{array}{|c|} \hline \diagdown \diagdown \\ \hline \end{array}, \begin{array}{|c|} \hline \diagup \diagup \\ \hline \end{array} \right\} \text{ und } y_2 \in \{\mathbf{1}, \mathbf{0}\}\}.$$

Dies notiert sich natürlich viel leichter, wenn die Menge der Hebelstellungen besser – im Sinne einer einfacheren Notation – dargestellt wird!

Bezeichnen wir die Hebelstellung **von links oben nach rechts unten** mit der Ziffer „0“ und diejenige **von rechts oben nach links unten** mit der Ziffer „1“, so kann die Stellung der drei Hebel als Binärzahl dargestellt werden. Wir wählen die Binärzahl „001“ für die Situation $\begin{array}{|c|} \hline \diagdown \diagup \\ \hline \end{array}$ und „101“ für die Situation $\begin{array}{|c|} \hline \diagup \diagdown \\ \hline \end{array}$! In Dezimaldarstellung (z.B.: $3 = [011]_2$) sind dies die Ziffern 0, 1, 2, 3, 4, 5, 6, 7. Die acht Hebelstellungen bedeuten letztlich die möglichen internen Zustände des Gerätes, weswegen wir diese als Elemente z_i der Menge $Z := \{z_0, z_1, z_2, z_3, z_4, z_5, z_6, z_7\}$ notieren wollen. Aus dem Index i des Zustands z_i läßt sich ebenso eindeutig die Stellung der Hebel im Inneren des Gerätes in diesem Zustand ablesen. Die soeben natürlichsprachlich beschriebene (Übergangs-)Funktion lässt sich nun, wie in Abbildung 1.1 dargestellt, mathematisch formal notieren.

$$\begin{array}{lll} \delta : Z \times \{L, R\} & \longrightarrow & Z \times \{1, 0\} \\ (z_0, L) & \mapsto & (z_6, 0) \\ (z_0, R) & \mapsto & (z_1, 0) \\ (z_1, L) & \mapsto & (z_7, 0) \\ (z_1, R) & \mapsto & (z_2, 0) \\ (z_2, L) & \mapsto & (z_4, 1) \\ (z_2, R) & \mapsto & (z_3, 0) \\ (z_3, L) & \mapsto & (z_5, 1) \\ (z_3, R) & \mapsto & (z_0, 1) \\ (z_4, L) & \mapsto & (z_0, 1) \\ (z_4, R) & \mapsto & (z_5, 0) \\ (z_5, L) & \mapsto & (z_1, 1) \\ (z_5, R) & \mapsto & (z_6, 0) \\ (z_6, L) & \mapsto & (z_2, 1) \\ (z_6, R) & \mapsto & (z_7, 0) \\ (z_7, L) & \mapsto & (z_3, 1) \\ (z_7, R) & \mapsto & (z_4, 1) \end{array}$$

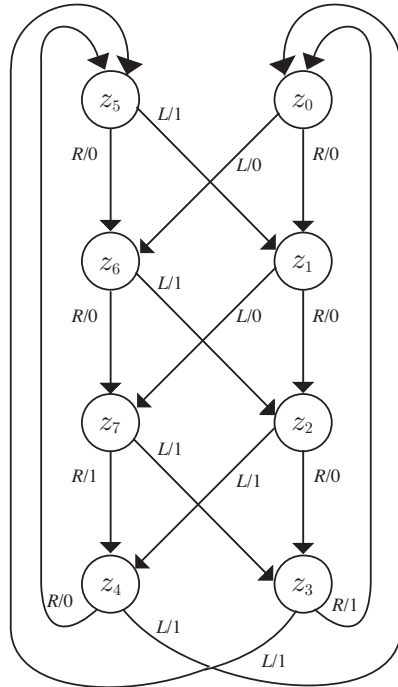


Abbildung 1.1.: Übergangsfunktion (links) und Zustandsgraph (rechts)

Die Übergangsfunktion haben wir hier mathematisch beschrieben durch die Zeile

$$\delta : Z \times \{L, R\} \longrightarrow Z \times \{1, 0\},$$

die ausdrückt, dass jeweils ein Zustand aus der Menge Z zusammen mit einem der Symbole L oder R überführt werden in einen neuen Zustand – wiederum aus Z – und einen der Ausgänge 1 oder 0. Hier werden also Vor- und Nachbereich der Funktion festgelegt.

$Z \times \{L, R\}$ ist (mathematisch gesehen) die Menge aller Paare aus Zustand und Eingang, während $Z \times \{1, 0\}$ die Menge aller Paare aus Zustand und Ausgang darstellt. (\times bildet das **cartesische Produkt** zweier Mengen.)

In den folgenden Zeilen wird die Funktion durch explizite Angabe der Zuweisungsvorschrift spezifiziert. Die entspricht der konkreten Angabe der Menge aller Paare aus Urbild und Bild der Funktion. Dabei ist zu beachten, dass Funktionen immer rechtseindeutig sind, d.h. für jedes Element aus dem Vorbereich das Bild eindeutig bestimmt ist. Eine Funktion weist Elementen aus dem Vorbereich **deterministisch** einen Funktionswert zu.

Natürlich ist solch eine Liste von Funktionswerten nicht sehr hübsch. Etwas übersichtlicher wird es, wenn wir stattdessen einen Graph zeichnen (rechtes Bild), dessen Knoten den Zuständen zugeordnet sind und der 16 gerichtete und beschriftete Kanten für die Abbildungen besitzt. Diesen Graphen bezeichnet man gewöhnlich als **Zustandsgraph** (genauer: Zustandsübergangsdiagramm). An einer Kante steht hier die Beschriftung **L/0** (bzw. **L/1**, **R/0**, **R/1**), wenn die Kugel bei **L** (bzw. **R**) eingeworfen wird und dann bei **0** (bzw. **1**) wieder herauskommt.

Falls wir nun wissen möchten, in welchen Zustand das Gerät gelangt, wenn wir beginnend im Zustand z_0 die Kugel z.B. in der Folge

$$\mathbf{F} := \mathbf{L L R L R R L R L R L L R R L R L L R L R R L R R R L R R L R R R L R R}$$

einwerfen, so können wir dies an dem Zustandsdiagramm ablesen. Wir erhalten dabei sogar gleichzeitig die Informationen, wo die Kugel jeweils herauskommen wird. Wenn die Eingabesequenz allerdings recht lang ist, so ist dies Vorgehen sogar recht mühselig und fehleranfällig.

Bei sorgfältigem Betrachten des Zustandsdiagramms fällt auf, dass man nach viermaligem Einwerfen der Kugel bei **L** (bzw. nach viermaligem Einwerfen der Kugel bei **R**) das Gerät wieder in den gleiche Zustand versetzt bei dem man gestartet hatte. Wir notieren dies für jeden Zustand $z_i, i \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ durch $z_i \xrightarrow{LLLL} z_i$ und $z_i \xrightarrow{RRRR} z_i$. Ähnlich verifizieren wir, dass für alle $i, j \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ die gleichartig notierte Beziehung $z_i \xrightarrow{RL} z_j$ genau dann gilt, wenn $z_i \xrightarrow{LR} z_j$ zutrifft. Dies bedeutet, dass wir eine lange Eingabefolge F von Eingabesymbolen **L** und **R** mit dem im folgenden beschriebenen Verfahren umformen können, ohne den erreichten Zustand dabei zu verändern. Wir formulieren das Verfahren in Form von **Regeln**, die eine Folge von Aktionen (hier: Einwürfen der Kugel in die linke bzw. rechte Öffnung) in eine andere, nicht längere Folge umwandeln.

Transformationsverfahren:

1. Einleitung

Wende die folgenden Regeln solange an, wie dies möglich ist:

- Wenn in der Folge eine der Sequenzen **L L L L** oder **R R R R** vorkommt, lösche diese.
- Wenn in der Folge die Sequenz **R L** auftritt, ersetze diese durch die Sequenz **L R**.

Wenden wir diese Regeln auf die Sequenz **F** an, so ist zum Beispiel die in Abbildung 1.2 gezeigte Abfolge von Ersetzungen möglich, wobei die späteren Vertauschungsschritte bei maximaler Parallelität in einer Zeile notiert wurden. Die mit Klammern markierten Stellen deuten die Vertauschungen von Symbolen an, während unterstrichene Teilwörter entfernt werden.

```

L L R L R R L R L R L L R R L R L L R L R R R L R R L R R R L R R
L L R L R R L R L R L L R R L R L L R L R R R L R L R R R L R R
L L R L R R L R L R L L R R L R L L R L R R L R R R L R L L R R
L L R L R R L R L R L L R R L R L L R L R R R L R L L R R
L L R L R R L R L R L L R R L R L L R L R L L R L L R R
L L L R R L R L R L R L R L R L R L L R L R L L R L R R
L L L R L R L R L R L R L R L R L R L L R L R L L R R R
L L L L R L R L R L R L R L R L R L R L L R L R L R R R
L R L R L R L R L R L R L R L L R L R R R R
L L R L R L R L R L R L R L R L L R
L L L R L R L R L R L R L R L R L R L R
L L L L R L R L R L R L R L R L R L R R
L R L R L R L R L R L R L R R R
L L R L R L R L R L R L R R R R
L L L R L R L R L R L R
L L L L R L R L R L R R
L R L R L R R R
L L R L R R R R
L L L R

```

Abbildung 1.2.: Anwendung des Transformationsverfahrens auf die Folge **F**

Am Ende sind die erhaltenen Sequenzen stets so kurz, dass der mit einer beliebig langen Folge **F** erreichbare Zustand nun sehr leicht herauszufinden ist! Tatsächlich bestehen die so erhaltenen Folgen aus höchstens sechs Symbolen. Diese kurzen Sequenzen können mit obigen Regeln **alle** auf eine der folgenden Formen gebracht werden: λ , **L**, **R**, **LL**, **LR**, **RR**, **LLL**, **LLR**, **LRR**, **RRR**, **LLLR**, **LLRR**, **LRRR**, **LLLRR**, **LLRRR**, **LLLRRR**. Hierbei bezeichnet das griechische kleine Lambda „ λ “ die Tatsache, dass nach der Umformungsprozedur keines der Symbole **L** oder **R** mehr vorhanden ist, man also in den Ausgangszustand zurückkehrt.

Offensichtlich braucht man nur die Anzahlen der vorkommenden **L**'s und **R**'s modulo 4 zu zählen, um die entsprechende kurze Folge als eine **Normalform** zu erhalten! Die vorige Folge **F** enthält 15-mal das

Auftreten eines \mathbf{L} 's und 21 Vorkommnisse des Symbols \mathbf{R} . Modulo 4 sind das $3 \cdot \mathbf{L}$ und $1 \cdot \mathbf{R}$, was vom Effekt her der Folge $\mathbf{L L L R}$ entspricht, die wir durch die Ersetzungsregeln auch erhalten hatten. Wie man nun leicht nachprüft, wird im Zustandsdiagramm mit dieser Folge \mathbf{F} von jedem Zustand jeweils der benachbarte Zustand in der gleichen Höhe gegenüber eingenommen.

An diesem Beispiel zeigen sich bereits wichtige inhaltliche Bestandteile und Erkenntnisse (nicht nur) für diese Vorlesung:

- Die Notwendigkeit von **Abstraktionen** und die Beschränkung auf das Wesentliche. Es ist jedoch Vorsicht geboten, damit nicht wichtige Informationen verlorengehen!
- Die Nützlichkeit **mathematischer Notationen**, wie z.B. Funktionen und Relationen! Diese werden durchgängig in der Informatik verwendet.
- Die deutlichere Übersichtlichkeit bei Verwendung von **gerichteten und beschrifteten Graphen**. Diese werden häufig bei der Beschreibung von Automaten und anderen mathematischen Objekten verwendet. Sie dienen vorrangig der Visualisierung von (häufig) komplexen Zusammenhängen, haben jedoch immer auch eine mathematisch formale Darstellung!
- Die Möglichkeit **algorithmischer Verfahren**, zum Beispiel solche, die Transformationen von Zeichenketten benutzen. Solche Transformationen werden später im Zusammenhang mit Algorithmen und formalen Grammatiken bzw. Reduktionssystemen wie auch anderen Kalkülen untersucht.

2. Notationen und Grundbegriffe

Betrachten Sie folgende Aussage:

„Wenn $G = \sqrt[2]{ab}$ das geometrische Mittel (*geometric mean*) und $A = \frac{a+b}{2}$ das arithmetische Mittel (*arithmetic mean*) der positiven Zahlen a und b bezeichnen, so gilt stets $G \leq A$. Die Gleichheit $G = A$ gilt hier nur, wenn $a = b$ ist.“

Die hier verwendeten Gleichheitszeichen haben unterschiedliche Bedeutung: Die des ersten Satzes bedeuten die Definition der neuen Objekte G und A auf der Basis der schon bekannten Größen a und b , während die im zweiten Satz Beziehungen (Relationen) bezeichnen. Wir werden diese verschiedenen Gleichheitsbegriffe auch in der Notation von einander unterscheiden. Das relationale Gleichheitszeichen „ $=$ “ wird wie üblich verwendet (Auf dem europäischen Kontinent ist es erst seit dem achtzehnten Jahrhundert in Gebrauch, obwohl das Symbol „ $=$ “ von Robert Recorde schon 1577 eingeführt und empfohlen wurde! Vergl. [Gries&Schneider], Seite 16.) und unterschieden von demjenigen, das wir benutzen um Neues aus schon Bekanntem zu definieren. Wir werden in diesem Skript für: „sei definitionsgemäß gleich“ das mit einem Doppelpunkt beginnende Gleichheitszeichen „ $:=$ “ benutzen, welches in PASCAL- und ALGOL-ähnlichen Programmiersprachen, sowie häufig bei Beschreibungen von algorithmischen Verfahren als Zuweisungssymbol verwendet wird.

Weiterhin werden neue Begriffe bei deren Definition **fett** gedruckt und es werden die englischen Bezeichnungen in *kursiv* (wie oben geschehen) hinter neuen Begriffen notiert. Beweise werden mit dem kleinen Quadrat „ \square “ abgeschlossen, Definitionen, Theoreme (Sätze), Lemmata (Hilfssätze) sowie Korollare (Folgerungen aus Theoremen oder Lemmata) werden dagegen ohne eigenes „Endezeichen“ geschrieben, jedoch typographisch vom restlichen Text abgesetzt.

Wichtige, dem Verständnis dienende Anmerkungen werden mit einem Rahmen versehen, der jedoch nicht die Wichtigkeit von Definitionen und insbesondere Theoremen schmälern soll!

Wir beginnen das Skript mit einer Reihe von Definitionen und Notationen, werden aber – wo immer es geht – weitere mathematische Begriffe erst dann definieren, wenn diese aus Sicht der Anwendung und Benutzung in diesem Skript nötig werden. Dieses Kapitel ist vorwiegend als Referenz zu verstehen, in dem nachgeschlagen werden kann, welche Schreibweisen und Definitionen für dieses Skript verwendet werden, da in der Fülle der weiterführenden Literatur häufig unterschiedliche Notationen verwendet werden.

Als Begleitlektüre seien insbesondere [Hopcroft,Motwani&Ullman], [Hopcroft&Ullman] und [Kinber&Smith] empfohlen.

2.1. Notation von Mengen

Mengen Wir werden oft Mengen und darauf erklärte Relationen verwenden, und wollen hier die von uns verwendete Notationen für diese intuitiv bekannten Begriffe erklären. Cantor (Georg Cantor, 1845-1918) folgend, ist eine **Menge** (*set*) M eine Zusammenfassung bestimmter, wohlunterschiedener Objekte unserer Anschauung oder unseres Denkens (welche die Elemente von M genannt werden) zu einem Ganzen.

Mengen können wir auf unterschiedlichste Weise bilden: Wir können sie **extensional** (dem Umfang, der Ausdehnung nach) durch direkte Aufzählung aller Elemente zwischen den Mengenklammern „{“ und „}“ notieren, z.B. als $A := \{1, 3, 5, 7\}$ für die ersten vier ungeraden natürlichen Zahlen. Bei nicht zu umfangreichen, endlichen Mengen (*finite* [sprich: feineit] *sets*) ist das immer möglich. Unendliche Mengen (*infinite* [sprich: infinit] *sets*) könnte man etwas informal, wie im folgenden Beispiel der geraden, nicht negativen ganzen Zahlen, **aufzählend** in der Form $B := \{0, 2, 4, \dots\}$ notieren. Diese laxe Schreibweise hat natürlich ihre Nachteile, denn es ist durchaus nicht immer klar, wie eine begonnene Aufzählung (hier z.B. 0, 2, 4, ...) fortgesetzt werden soll! Sie alle kennen diese sogenannten Intelligenztests, wie sie auch bei manchen Einstellungsprüfungen vorgenommen werden.

Wie wird zum Beispiel die Folge $\langle 8, 3, 1, 5, 9, 0, 6, \dots \rangle$ fortgesetzt?

Sie wußten, dass es sich hierbei nur um die *alphabetische Reihenfolge* der Ziffernnamen handeln konnte und nun also noch 7, 4 und 2 folgen werden? Herzlichen Glückwunsch!

Folgen von Zahlen oder anderen Objekten wollen wir hier im folgenden immer in Spitze Klammern gesetzt notieren. So bezeichnet $\langle 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots \rangle$ die Folge der bekannten Fibonacci-Zahlen $f_n, n \geq 1$ mit der rekursiven Definition durch die Rekurrenzgleichung:

$$f_n := \begin{cases} 0 & \text{falls } n \leq 0 \\ 1 & \text{falls } n = 1 \\ f_{n-1} + f_{n-2} & \text{falls } n > 1. \end{cases}$$

Diese Rekurrenzgleichung induziert natürlich sogleich ein Verfahren zum Erzeugen der einzelnen Glieder dieser unendlichen Zahlenfolge und stellt somit eine der gewünschten endlichen Repräsentationen dieser unendlichen Menge (Folge) von Objekten (der Fibonacci-Zahlen) dar. Rekursiven und induktiven Definitionen dieser und anderer Art werden Sie in der Informatik noch häufiger begegnen. Es gibt jedoch oft andere, in mancher Hinsicht günstigere Darstellungen in geschlossener Form. Für die Fibonacci-Zahlen gilt z.B. für jedes $n \in \mathbb{N}$:

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

Beweise für solche Charakterisierungen werden in der Diskreten Mathematik geführt. Die Bücher von Martin Aigner, ([Aigner]), und Norman L. Biggs, [Biggs], sind gute und nicht zu einfache Quellen, letzteres in englischer Sprache. In einigen, meist einfachen, Fällen werden wir solche und ähnliche Beweise in diesem Skript jedoch auch direkt aufnehmen.

Logische Notationen Auch wenn die Zeichen der Logik schon in der Aussagen- und Prädikatenlogik (Vorlesung F1 – Logik) detailliert behandelt wurden, wollen wir hier die sogenannten logischen Junktoren wiederholen. Wir benutzen diese jedoch meist nur bei den Definitionen von Mengen:

- \neg für die Negation („nicht“, (*not*))
- \wedge für die Konjunktion („und“ (*and*))
- \vee für die Disjunktion („oder“ (*or*))
- \rightarrow für die Implikation („daraus folgt“, „impliziert“ (*implies*))
- \leftrightarrow für die Biimplikation „genau dann, wenn“ (*if and only if*, Abk.: *iff*))
- \forall für den Allquantor („für alle“ (*for all*))
- \exists für den Existenzquantor („es gibt“ (*exists*))

Die Bedeutung und die hier verwendete Notation der Quantoren sollen der Vollständigkeit halber an dieser Stelle noch einmal präzisiert werden:

- Die Formel $\forall x \in M : p(x)$ ist genau dann falsch, wenn es ein Element x in M gibt, für welches das Prädikat $p(x)$ nicht erfüllt ist.
- Die Formel $\exists x \in M : p(x)$ ist genau dann wahr, wenn es ein Element x in M gibt, für welches das Prädikat $p(x)$ erfüllt ist.

Dadurch ist der Ausdruck $\forall x \in \emptyset : p(x)$ ausnahmslos für jedes Prädikat p erfüllt!

Oft werden wir Mengen aber **intensional** (der Absicht, der Zielrichtung nach) durch Angabe einer charakteristischen Eigenschaft unter Bezugnahme auf schon bekannte Mengen definieren:

Wenn p eine Eigenschaft, d.h. ein Prädikat, ist, welches auf die Elemente der zu definierenden Menge M zutreffen soll, schreiben wir $M := \{x \mid p(x)\}$. Um auszudrücken, dass x ein Element der Menge M ist, schreiben wir $x \in M$, andernfalls $x \notin M$. Unter Rückgriff auf die üblichen Notationen (siehe weiter unten) können wir nun die Mengen A und B von eben auch auf folgende Weise definieren:

$A := \{n \in \mathbb{N} \mid \exists m \in \mathbb{N} : 1 \leq m \leq 4 \wedge n = 2m - 1\}$ und $B := \{m \mid \exists n \in \mathbb{N} : m = 2n\}$.

Die **leere Menge**, d.h., die eindeutig bestimmte Menge die kein Element enthält, könnte so definiert werden: $\emptyset := \{m \mid m \neq m\}$ oder auch als die Menge aller Drei-Pfennig Münzen nach 1948.

Eigenschaften und Prädikate Von Kenneth E. Iverson, (1962), dem Erfinder der Programmiersprache **APL**¹ stammt die folgende sehr nützliche Notation, mit der Wahrheitswerte von Prädikaten auf die Zahlenwerte 0 und 1 abgebildet werden:

2.1 Definition

Ist p ein Prädikat, so ist $[p]$ (also das p in eckigen Klammern), folgendermaßen definiert:

$$[p] := \begin{cases} 1 & \text{falls } p \text{ wahr ist} \\ 0 & \text{falls } p \text{ nicht wahr ist.} \end{cases}$$

2.2 Beispiel

Es sind „ $3 \cdot 2 = 6$ “ und „ $\pi = 3,14$ “ zwei Prädikate für die gemäß Definition 2.1 gilt:

$$1 = [3 \cdot 2 = 6] \quad \text{und} \quad 0 = [\pi = 3,14]$$

¹APL steht für **A** **P**rogramming **L**anguage

2. Notationen und Grundbegriffe

Wenn eine Eigenschaft durch mehrere gleichzeitig geltende Prädikate p_1, p_2, p_3, \dots definiert werden soll, so vereinfachen wir bisweilen die exaktere Schreibweise und schreiben z.B. statt $\{x \mid x \in \mathcal{N} \wedge p_1(x) \wedge p_2(x) \wedge p_3(x)\}$ oft nur $\{x \in \mathcal{N} \mid p_1(x), p_2(x), p_3(x)\}$. In der Regel werden die Leser(innen) in der Literatur allerdings aussagen- oder prädikatenlogische Formeln vorfinden.

Falls für jedes $x \in M_1$ stets auch $x \in M_2$ gilt, so ist die Menge M_1 in der Menge M_2 enthalten, und wir notieren das als $M_1 \subseteq M_2$ (oder $M_2 \supseteq M_1$) wobei eben auch $M_1 = M_2$ gelten darf. In manchen Werken wird dies auch nur als \subset geschrieben, aber diese Notation benutzen wir hier nicht, da diese zu leicht mit dem echten Enthaltensein verwechselt werden kann. Um auszudrücken, dass zwar $M_1 \subseteq M_2$ aber $M_1 \neq M_2$ gilt, schreiben wir $M_1 \subsetneq M_2$. Die Mengen M_1 und M_2 sind gleich, notiert als $M_1 = M_2$, wenn sowohl $M_1 \subseteq M_2$ als auch $M_2 \subseteq M_1$ gilt. Offensichtlich kommt es bei diesen Definitionen genausowenig auf die Reihenfolge, wie auf die Anzahl der Elemente der Mengen bei deren Aufschreibung an. Es gilt $\{1, 3, 5, 7\} = \{3, 1, 5, 5, 3, 7, 7, 1, 7\}$. Die Anzahl der Elemente in einer Menge M wird als **Kardinalität** (auch: Mächtigkeit) dieser Menge bezeichnet und mit $|M|$ notiert. Eine Menge ist **unendlich**, wenn sie die gleiche Kardinalität wie eine ihrer echten Teilmengen besitzt. Ist M keine endliche Menge, so schreiben wir $|M| = \infty$.

2.3 Definition

Sei $M \subseteq C$, dann ist die **charakteristische Funktion** von M die auf ganz C definierte Funktion $\chi_M : C \longrightarrow \{0, 1\}$ mit $\chi_M(x) := [x \in M]$.

Explizit notiert ergibt sich damit die übliche Schreibweise der charakteristischen Funktion:

$$\chi_M(x) := \begin{cases} 1 & \text{falls } x \in M \\ 0 & \text{sonst} \end{cases}$$

Zu jeder Menge M wird durch die *charakteristische Funktion* die Zugehörigkeit von Elementen zu dieser Menge durch die Zahlen 1 und 0 ausgedrückt.

2.4 Definition

Die Menge aller Teilmengen einer Menge M nennt man **Potenzmenge** und wir notieren sie als 2^M .

2.5 Beispiel

Die Potenzmenge von $M := \{1, 3, 5, 7\}$ ist:

$$2^M := \left\{ \begin{array}{l} \emptyset, \{1\}, \{3\}, \{5\}, \{7\}, \\ \{1, 3\}, \{1, 5\}, \{1, 7\}, \{3, 5\}, \{3, 7\}, \{5, 7\}, \\ \{1, 3, 5\}, \{1, 3, 7\}, \{1, 5, 7\}, \{3, 5, 7\}, \{1, 3, 5, 7\} \end{array} \right\}$$

Jede Menge wird eindeutig durch ihre charakteristische Funktion bestimmt. Für jede Teilmenge A einer Menge M gilt somit $A = \{x \mid \chi_A(x) = 1\}$ und es gibt genau so viele Teilmengen von M wie es Abbildungen von M auf die Menge $\{0, 1\}$ gibt, da ja jede solche Abbildung eine charakteristische Funktion darstellt. Das sind aber genau $2^{|M|}$ viele. Wenn wir später Mengen von Mengen betrachten, bezeichnen wir diese meistens als **Klassen** von Mengen oder auch als **Familien** von Mengen, wenn die Klassen besondere Eigenschaften erfüllen.

Das **Komplement** einer Menge A muss stets zu einer, diese enthaltenden, Obermenge C gebildet werden und wir schreiben dann, $C \setminus A := \{c \mid c \in C \wedge c \notin A\}$, womit die allgemeine **Mengendifferenz** bezeichnet wird. Wenn C als bekannt vorausgesetzt werden kann, schreiben wir auch \bar{A} anstelle von $C \setminus A$.

$A \cup B := \{c \mid c \in A \vee c \in B\}$ bezeichnet die **Vereinigung** der Mengen A und B , während $A \uplus B$ deren **disjunkte Vereinigung** bezeichnet, d.h. A und B waren schon disjunkt oder werden, durch Umbenennung der Elemente einer Menge, dazu gemacht. Der **Durchschnitt** der Mengen A und B wird mit $A \cap B := \{c \mid c \in A \wedge c \in B\}$ bezeichnet. Ist $M = \biguplus_{i \in I} A_i$ die disjunkte Vereinigung von Mengen A_i mit i aus einer nicht notwendig endlichen Indexmenge I , so nennt man $(A_i)_{i \in I}$ eine **Partition** von M .

Weitere Mengen, die uns häufig begegnen, sind die folgenden:

2.6 Definition

Entsprechend DIN 5473 ist $\mathbb{N} := \{0, 1, 2, 3, \dots\}$ die Menge der **natürlichen Zahlen** (*non-negative integers*), also inklusive der 0, anderswo bisweilen als \mathbb{N}_0 notiert. Die Menge der **ganzen Zahlen** (*integers*) wird mit \mathbb{Z} notiert. Die Menge $\mathbb{Z}^+ = \{x \in \mathbb{Z} \mid x > 0\}$ enthält genau die positiven ganzen Zahlen (ohne die 0).

Mit $\mathbb{Q} := \left\{ \frac{p}{q} \mid p \in \mathbb{Z} \wedge q \in \mathbb{Z}^+ \right\}$ werden die **rationalen**, mit \mathbb{R} die **reellen Zahlen** und mit \mathbb{C} die **komplexen Zahlen**. In den Fällen der Zahlenbereiche $M \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$ werden in Analogie zu \mathbb{Z}^+ durch M^+ und M^- folgende Einschränkungen bezeichnen: $M^+ := \{x \in M \mid x > 0\}$ sowie $M^- := \{x \in M \mid x < 0\}$.

Funktionen zur Manipulation von Zahlen Ebenfalls von Kenneth E. Iverson stammt die Schreibweise mit den „abgefeilten“ eckigen Klammern für „die größte (bzw. kleinste) ganze Zahl kleiner (bzw. größer) als r “, die z.B. in den Programmiersprachen JAVA und C++, wie auch in Definition 2.7 mit den üblichen englischen Ausdrücken *floor* und *ceiling* bezeichnet werden:

2.7 Definition

Für $r \in \mathbb{R}$ sei $\lfloor r \rfloor$ die größte ganze Zahl $z \in \mathbb{Z}$ mit $z \leq r$ (*floor*) und $\lceil r \rceil$ die kleinste ganze Zahl $z \in \mathbb{Z}$ mit $z \geq r$ (*ceiling*).

Wir stellen im Folgenden einige nützliche und einfache Anwendungshilfen zusammen.

2.8 Beispiel

Folgende Beziehungen gelten gemäß Definition 2.7:

$$\begin{aligned} \lfloor 3.14 \rfloor &= 3 = \lfloor 3.75 \rfloor \\ \lfloor -3.14 \rfloor &= -4 = \lfloor -3.75 \rfloor \\ \lceil 3.14 \rceil &= 4 = \lceil 3.75 \rceil \\ \lceil -3.14 \rceil &= -3 = \lceil -3.75 \rceil \end{aligned}$$

Wir betrachten einige **grundlegende Eigenschaften** der *floor*- und *ceiling*-Funktionen:

$$\begin{aligned} \lfloor x \rfloor &= x \quad \text{genau dann, wenn} \quad x \in \mathbb{Z} \quad \text{genau dann, wenn} \quad \lceil x \rceil = x. \\ x - 1 &< \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 && \text{für jedes } x \\ \lfloor -x \rfloor &= -\lceil x \rceil, \lceil -x \rceil = -\lfloor x \rfloor && \text{für jedes } x \end{aligned}$$

2. Notationen und Grundbegriffe

$$\lceil x \rceil - \lfloor x \rfloor = [x \notin \mathbb{Z}] = \begin{cases} 1 & \text{wenn } x \notin \mathbb{Z}. \\ 0 & \text{wenn } x \in \mathbb{Z}. \end{cases}$$

Außerdem können folgende **nützliche Äquivalenzen** gezeigt werden: $\forall n \in \mathbb{N} \forall x \in \mathbb{R}$:

$$\lfloor x \rfloor = n \iff n \leq x < n + 1$$

$$\lfloor x \rfloor = n \iff x - 1 < n \leq x$$

$$\lceil x \rceil = n \iff x \leq n < x + 1$$

$$\lceil x \rceil = n \iff n - 1 < x \leq n$$

$$x < n \iff \lfloor x \rfloor < n$$

$$n < x \iff n < \lceil x \rceil$$

$$x \leq n \iff \lceil x \rceil \leq n$$

$$n \leq x \iff n \leq \lfloor x \rfloor$$

Weiterhin gilt:

$$\forall x \in \mathbb{R}, n \in \mathbb{Z} : \lfloor x + n \rfloor = \lfloor x \rfloor + n$$

und

$$\forall x \in \mathbb{R}, n \in \mathbb{Z} : \lceil x + n \rceil = \lceil x \rceil + n$$

2.9 Beispiel

Zur Übung wollen wir die folgenden Ausdrücke vereinfachen:

1. $\lceil \lfloor x \rfloor \rceil$

2. $\lceil \frac{n}{2} \rceil - \lfloor \frac{n}{2} \rfloor$

zu 1.: Da $\lfloor x \rfloor \in \mathbb{Z}$ ist, folgt natürlich sofort $\lceil \lfloor x \rfloor \rceil = \lfloor x \rfloor$.

zu 2.: Nach der letzten Gleichung der grundlegenden Eigenschaften ist für alle Zahlen $n \in \mathbb{R}$

$$\left\lceil \frac{n}{2} \right\rceil - \left\lfloor \frac{n}{2} \right\rfloor = [n \text{ ist keine gerade Zahl aus } \mathbb{Z}] = \begin{cases} 0 & \text{falls } n \text{ eine gerade Zahl aus } \mathbb{Z} \text{ ist,} \\ 1 & \text{sonst.} \end{cases}$$

Mehr zu diesen nützlichen Notationen und vielen weiteren Ergebnissen aus der diskreten Mathematik finden die interessierten Leser(innen) im hervorragenden Buch von R.L. Graham, D.E. Knuth und O. Patashnik: „Concrete Mathematics“, [Graham et al.].

2.2. Relationen

Die folgenden Begriffe zu Relationen und deren Klassifizierung nach Eigenschaften sind ebenfalls Stoff der Mathematikveranstaltung. Damit wir bei Bedarf diese Notationen benutzen können, finden Sie diese jedoch auch an dieser Stelle noch einmal wieder. Im allgemeinen wollen wir die Notationen möglichst spät und erst dann einführen, wenn die mit ihnen beschriebenen Begriffe auch benötigt werden! Das gelingt verständlicher Weise nicht durchgehend, denn es ist oft besser, ähnliche und abgeleitete Begriffe zusammenhängend zu erklären. Relationen werden wir benötigen, um Graphen und Tabellen zu notieren, einige der hier bereits eingeführten Eigenschaften werden wir aber erst später verwenden.

2.10 Definition

Seien A_1, A_2, \dots, A_n Mengen und $x_1 \in A_1, \dots, x_n \in A_n$, dann heißt (x_1, \dots, x_n) ein **(geordnetes) n -Tupel** von Elementen über A_1, \dots, A_n . Die Menge aller geordneten n -Tupel (x_1, \dots, x_n) mit $x_1 \in A_1, \dots, x_n \in A_n$ heißt **cartesisches Produkt** der Mengen A_1 bis A_n und wird geschrieben als $A_1 \times A_2 \times \dots \times A_n$.

Es ist also $A_1 \times A_2 \times \dots \times A_n := \{(x_1, \dots, x_n) \mid x_1 \in A_1, \dots, x_n \in A_n\}$.

Spricht man von Tupeln, so sind in der Regel immer geordnete Tupel gemeint, wenn dies nicht explizit anders angegeben wird.

Sind die Mengen A_i selbst schon cartesische Produkte der Form $A_i = B_{i,1} \times \dots \times B_{i,k_i}$, so werden in den Elementen von $A_1 \times A_2 \times \dots \times A_n$ die Klammern der Tupel aus A_i alle weggelassen sofern dies nicht zu Mehrdeutigkeiten führen kann. Anstelle von $((a, b), (3, 2, 4), (\nabla, \Delta))$ schreiben wir also häufig einfach $(a, b, 3, 2, 4, \nabla, \Delta)$!

Eine Teilmenge $R \subseteq A_1 \times A_2 \times \dots \times A_n$ heißt **n -stellige Relation** über A_1 bis A_n .

Ist $n = 2$, so spricht man häufig von einer **binären** statt von einer 2-stelligen **Relation** (genauer: heterogene dyadische Relation oder auch Korrespondenz von A_1 nach A_2).

Falls $A_i = A$ für jedes i mit $1 \leq i \leq n$ ist, so sprechen wir von einer **n -stelligen Relation auf A** . Das n -fache cartesische Produkt $\underbrace{A \times A \times \dots \times A}_n$ wird mit A^n abgekürzt. Wenn Elemente aus A^n nicht als

Tupel (x_1, x_2, \dots, x_n) geschrieben und benutzt, sondern im Zusammenhang mit Matrizenrechnung in der Linearen Algebra als Vektoren gebraucht werden, so notiert man diese in der Regel als Spaltenvektoren $\vec{x} \in A^n$. Um Vektoren in Zeilendarstellung aufschreiben zu können, müssen diese transponiert werden: Es ist

$$\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}^\top := (x_1 \ x_2 \ \dots \ x_n)$$

und $(\vec{x}^\top)^\top = \vec{x}$. Genauso, wie die Elemente in Matrizen in der rechteckigen Anordnung ebenfalls nur durch Zwischenräume von einander getrennt werden, fehlen in dieser Darstellung die Kommata als Trennzeichen der Tupelschreibweise! Vergleichen Sie zum Beispiel auch [Möller].

Bei einer zweistelligen Relation $R \subseteq A^2$ benutzen wir anstelle von $(x, y) \in R$ oft die Infixschreibweise xRy (z.B. $3 \leq 5$).

2.11 Definition

Sei $R \subseteq A_1 \times A_2$ eine Relation, dann ist A_1 der **Vorbereich** von R und A_2 ist der **Nachbereich**² von R . Die Menge aller Elemente $x \in A_1$, die mit einem Element $y \in A_2$ in der Relation R stehen, wird als **Definitionsbereich von y** (auch: Argumentbereich, oder engl. *domain*) bezüglich R bezeichnet und

²Die Begriffe Vor- und Nachbereich werden in [Bronstein et al.] synonym mit Definitions- und Wertebereich verwendet. Wir unterscheiden die Begriffe hier, da wir z.B. auch partielle Funktionen betrachten wollen, bei denen nicht für jedes Element aus dem Vorbereich auch ein Funktionswert existieren muss.

2. Notationen und Grundbegriffe

als $\text{dom}(y) := \{x \in A_1 \mid xRy\}$ notiert. Mit $\text{dom}(R)$ wird der Definitionsbereich der gesamten Relation R bezeichnet und folgendermaßen erklärt:

$$\text{dom}(R) := \bigcup_{y \in A_2} \text{dom}(y) = \{x \in A_1 \mid \exists y \in A_2 : xRy\}$$

Entsprechend ist $\text{range}(x) := \{y \in A_2 \mid xRy\}$ der **Wertebereich** (engl. *codomain* oder *range*) **von** x bezüglich R . Der **Wertebereich** der gesamten Relation R wird mit $\text{range}(R) := \{y \in A_2 \mid \exists x \in A_1 : xRy\}$ bezeichnet.

Häufig wird auch $\text{Def}(R)$ und $\text{Bild}(R)$ anstelle von $\text{dom}(R)$ und $\text{range}(R)$ verwendet.

Mit $R^{-1} := \{(y, x) \mid (x, y) \in R\}$ wird die **inverse** Relation zu R bezeichnet.

Die **(Rechts-)Komposition** $R_1 \cdot R_2$ zweier Relationen R_1 und R_2 ist:

$$R_1 \cdot R_2 := \{(x, z) \mid \exists y \in \text{range}(R_1) \cap \text{dom}(R_2) : ((x, y) \in R_1) \wedge ((y, z) \in R_2)\}.$$

Sind R_1, R_2 Relationen mit $R_1 \subseteq R_2$, dann heißt R_1 **feiner** als R_2 und R_2 **größer** als R_1 .

Das cartesische Produkt begegnet uns in der Informatik in vielen Programmiersprachen (wie z.B. PASCAL oder MODULA) als Datenstruktur **RECORD**.

Eigenschaften von Relationen Die folgende Definition stellt einige wichtige Eigenschaften von Relationen zusammen. Diese Begriffe werden in vielen Bereichen der (theoretischen) Informatik verwendet und stellen somit ein Grundvokabular dar, das jedem Informatiker geläufig sein sollte.

2.12 Definition

Sei $R \subseteq A \times A$ eine Relation auf A . R heißt

1. **reflexiv** genau dann, wenn $(a, a) \in R$ für alle $a \in A$ gilt.
2. **irreflexiv** genau dann, wenn für kein $a \in A$ $(a, a) \in R$ gilt.
3. **symmetrisch** genau dann, wenn für $(a, b) \in R$ stets auch $(b, a) \in R$ gilt.
4. **asymmetrisch** genau dann, wenn aus $(a, b) \in R$ immer $(b, a) \notin R$ folgt. Eine asymmetrische Relation ist also immer irreflexiv, aber eine irreflexive Relation braucht nicht asymmetrisch zu sein!
5. **antisymmetrisch** (oder *identitiv*), falls für alle $a, b \in A$ gilt:
 $(a, b) \in R \wedge (b, a) \in R$ impliziert stets $a = b$.
6. **transitiv** genau dann, wenn aus $(a, b) \in R$ und $(b, c) \in R$ stets $(a, c) \in R$ folgt. R ist also transitiv, wenn $R \cdot R \subseteq R$ gilt.
7. **linear** genau dann, wenn für beliebige $a, b \in A$ genau eine der drei Bedingungen: $a = b$ oder $(a, b) \in R$ oder $(b, a) \in R$ erfüllt ist.

Aus diesen elementaren Eigenschaften von Relationen kann man für spezielle Typen von Relationen Zusammenstellungen solcher Eigenschaften betrachten. Ein spezieller Relationentyp, dem besondere Bedeutung zukommt, sind die Äquivalenzrelationen. Sie werden in Kapitel 3 benötigt und werden durch Reflexivität, Symmetrie und Transitivität charakterisiert. Weitere wichtige Relationentypen werden eingesetzt, um die Elemente von Mengen anzuordnen oder wenigstens miteinander vergleichen zu können. Hierzu gehören u.a. Größenvergleiche, wie \leq und \geq auf Zahlenmengen.

Eine „nicht symmetrische“ Relation muss weder „asymmetrisch“ noch „antisymmetrisch“ sein!

Ordnungsrelationen So wie die Relation „kleiner“ auf den natürlichen Zahlen \mathbb{N} , werden Relationen im allgemeinen als Ordnungsrelationen bezeichnet, wenn sie gewisse „ordnende“ Eigenschaften erfüllen. Die meisten dieser Notationen werden auch in den entsprechenden Mathematik-Veranstaltungen erklärt, oder sind schon aus der Schule bekannt. Leider gibt es bei den Definitionen auch in der Literatur einige Unstimmigkeiten über die Bezeichnungen, die sich zum einen mit der Zeit geändert haben, bzw. in anderen Ländern mit leichten Unterschieden verwendet werden. Wir wollen hier die Bezeichnungen so verwenden, wie sie sich auch in den meisten Lehrbüchern wiederfinden.

2.13 Definition

Eine Relation $R \subseteq A \times A$ heißt

1. **Quasiordnung** (*quasi order*) oder auch Vorordnung bzw. Präordnung (*preorder*) genau dann, wenn sie reflexiv und transitiv ist.³

Sei K eine Teilmenge von A , d.h. $K \subseteq A$. Ein Element $u \in A$ heißt **untere Schranke** von K genau dann, wenn $\forall x \in K : (u, x) \in R$ gilt. Die Menge aller unteren Schranken von K bezeichnen wir als $MI(K)$.

Ein Element $o \in A$ wird **obere Schranke** genannt, wenn $\forall x \in K : (x, o) \in R$ gilt. Die Menge aller oberen Schranken von K bezeichnen wir als $MA(K)$.

2. **partielle Ordnung** (*partial order*) genau dann, wenn sie antisymmetrisch, reflexiv und transitiv ist, d.h. wenn sie eine antisymmetrische Quasiordnung ist.⁴ Die Menge A heißt dann partiell geordnet bzgl. R und wird kurz **poset** (*partially ordered set*) genannt.

$a \in A$ heißt **minimal** (*least*) (bzw. **maximal** (*maximal*)) bzgl. R genau dann, wenn es **kein** $b \in A$ gibt mit $a \neq b$ und $(b, a) \in R$ (bzw. $(a, b) \in R$).

Ein Element $a \in A$ heißt **Minimum** (bzw. **Maximum**) wenn $\forall b \in A \setminus \{a\} : (a, b) \in R$ (bzw. $\forall b \in A \setminus \{a\} : (b, a) \in R$) gilt.

³In der Literatur findet man manchmal auch den Begriff der irreflexiven Quasiordnung, für die neben der Transitivität nur Irreflexivität gelten muss. In diesem Fall würde die hier definierte Quasiordnung zur Unterscheidung als reflexive Quasiordnung bezeichnet. Wir bezeichnen irreflexive Quasiordnungen als *Striktordnung*. Zum Beispiel in [Gries&Schneider] wird jedoch der Begriff *quasi order* gerade für solch transitive und irreflexive Relationen verwendet!

⁴Analog zur vorigen Fußnote zu Quasiordnungen, wird auch manchmal zwischen reflexiven und irreflexiven **Halbordnungen** unterschieden. Eine partielle Ordnung in unseren Sinne ist dann eine reflexive Halbordnung. Zum Beispiel in [Floyd&Beigel]) wird für eine partielle Ordnung jedoch die Reflexivität *nicht* gefordert und nur Transitivität und Antisymmetrie vorausgesetzt.

2. Notationen und Grundbegriffe

3. **Halbordnung** genau dann, wenn sie asymmetrisch und transitiv ist.⁵

Sei $K \subseteq A$. Als **Infimum**, auch größte untere Schranke (*greatest lower bound*) oder untere Grenze, bezeichnet man eine untere Schranke $u \in \text{MI}(K)$, wenn $\forall x \in \text{MI}(K) : (x \neq u \rightarrow (x, u) \in R)$ gilt. Ein Infimum ist also das Maximum der Menge der unteren Schranken von K bzgl. R . Falls ein Infimum existiert, so ist es eindeutig und wird mit $\inf(K)$ (oft auch $\bigwedge K$) bezeichnet.

Als **Supremum**, auch kleinste obere Schranke (*least upper bound*) oder obere Grenze, bezeichnet man eine obere Schranke $o \in \text{MA}(K)$, wenn $\forall x \in \text{MA}(K) : (x \neq o \rightarrow (o, x) \in R)$ gilt. Ein Supremum ist also das Minimum der Menge der oberen Schranken von K bzgl. R . Falls ein Supremum existiert, so ist es eindeutig und wird mit $\sup(K)$ (oft auch mit $\bigvee K$) bezeichnet.

4. **Striktordnung** (Abk. **spo** für *strict partial order*)⁶, falls sie asymmetrisch und transitiv ist. Jede irreflexive und transitive Relation ist asymmetrisch, so dass eine Striktordnung auch als transitive, irreflexive Relation definiert werden könnte. Der **strikte** Teil einer partiellen Ordnung $R \subseteq A \times A$ ist $R \setminus \text{Id}_A$, wobei $\text{Id}_A := \{(a, a) \mid a \in A\}$ die **Identitätsrelation** auf A bezeichnet.

5. **lineare** (oder **totale**) **Ordnung** genau dann, wenn sie partielle Ordnung ist und zusätzlich für alle $a, b \in A$ mit $a \neq b$ entweder $(a, b) \in R$ oder $(b, a) \in R$ gilt, d.h. R ist konnex. Man sagt, dass die Menge A total geordnet ist.

6. **wohl-fundiert** (*well founded*), wenn jede nicht leere Teilmenge von A ein minimales Element besitzt.

7. **Wohlordnung** (*well ordering*), wenn sie eine wohl-fundierte lineare Ordnung ist.

2.14 Beispiel

Die Relation \leq auf \mathbb{Z} ist eine lineare Ordnung ohne minimale oder maximale Elemente, während auf \mathbb{N} mit \leq total geordnet ist, als Minimum 0 hat, aber kein maximales Element besitzt. \leq ist eine Wohlordnung auf \mathbb{N} , nicht jedoch auf \mathbb{Z} .

Die Menge $K := \{x \in \mathbb{Q} \mid x^2 < 2\}$ besitzt keine obere Schranke, d.h., $\sup(K) = \emptyset$.

$(\mathbb{N}, <)$ ist kein poset, denn $<$ ist nicht reflexiv. Gleiches gilt für die Relation **verheiratet** $\subseteq (M \cup F) \times (M \cup F)$, wobei M die Menge der Männer und F die der Frauen abkürzt.

Die Relation $\text{div} \subseteq \mathbb{Z}^+ \times \mathbb{Z}^+$, definiert durch $(m \text{ div } n) \leftrightarrow (\frac{n}{m} \in \mathbb{Z}^+ \setminus \{1\})$, d.h., m ist echter Teiler von n , ist eine Striktordnung (aber keine totale Ordnung) und besitzt 1 als Minimum, sofern man auch die Zahl 1 als echten Teiler bezeichnet (siehe Abbildungen auf Seite 19).

Für jede Menge M ist $(2^M, \subseteq)$ eine partiell geordnete Menge.

Für partielle Ordnungen werden oft Zeichen wie \leq , \subseteq , \sqsubseteq oder \preceq verwendet, während $<$, \subset , \sqsubset oder \prec für Striktordnungen benutzt werden.

Um **endliche** Halbordnungen (partielle Ordnungen oder Striktordnungen) darzustellen werden häufig „Hasse-Diagramme“ als graphische Darstellungen verwendet.

⁵Eine Halbordnung ist also gerade eine asymmetrische, irreflexive Quasiordnung.

⁶auch *asymmetrische (irreflexive) Halbordnung* genannt

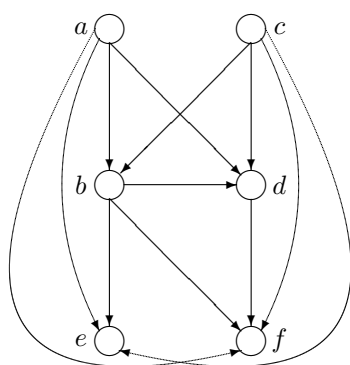
2.15 Definition

Das **Hasse-Diagramm** zu einer endlichen, asymmetrischen oder antisymmetrischen, transitiven Relation R , ist der gerichtete Graph $G(R_0)$ einer Teilmenge $R_0 \subseteq R$ mit folgender Eigenschaft: $(a, b) \in R_0$ gilt genau dann, wenn $(a, b) \in R$ und für kein Element $c \in \text{dom}(R) \cup \text{range}(R)$ mit $b \neq c \neq a$ sowohl $(a, c) \in R$ als auch $(c, b) \in R$ gilt. Gerichtete Kanten in $G(R_0)$ der Art $a \rightarrow b$ werden also genau dann gezeichnet, wenn $(a, b) \in R$ ist und „kein weiterer Knoten c dazwischen liegt“.

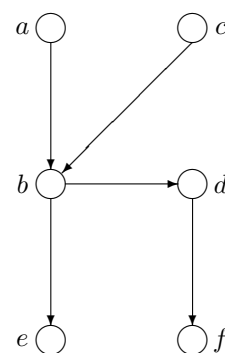
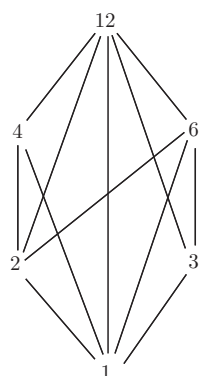
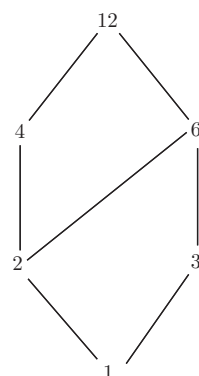
Hasse-Diagramme werden häufig nur zu Striktordnungen definiert, jedoch ist eine Erweiterung auf alle transitiven Relationen möglich, nur nicht ganz so einfach zu beschreiben. Für Striktordnungen $R \subseteq A \times A$ ist das Hasse-Diagramm $G(R_0)$ gegeben durch $R_0 := (R \setminus \text{Id}_A) \setminus (R \setminus \text{Id}_A)^2$. In der Regel zeichnen Mathematiker in Hasse-Diagrammen keine Pfeile, sondern ungerichtete Kanten. Die Richtung wird durch die Platzierung der Knoten gegeben: Geht nach unserer Definition eine Kante vom Knoten a zum Knoten b , so wird a oberhalb von b gezeichnet.

Hasse-Diagramm für die folgende Striktordnung:

$$R := \{ (a, b), (a, d), (a, e), (a, f), (b, d), (b, e), (b, f), (c, b), (c, d), (c, e), (c, f), (d, f) \}.$$

Graph der Striktordnung R

Wollten wir den vollständigen Graph einer Quasiordnung zeichnen, so müssten an alle Knoten noch Schleifen für die Reflexivität angefügt werden.

Hasse-Diagramm zu R Graph der Striktordnung div Hassediagramm zur Striktordnung div

Die Striktordnung div wurde als „echter-Teiler-von“ Relation auf Seite 18 definiert

Abschlussoperatoren Das Hasse-Diagramm ist der Graph einer in der Regel kleineren (Paar-)Menge, welche die gesamte Striktordnung durch transitiven Abschluss, bzw. eine partielle Ordnung durch refle-

2. Notationen und Grundbegriffe

xiven und transitiven Abschluss erzeugt.

2.16 Definition

Sei $R \subseteq A \times A$ eine Relation auf der Menge A . Dann seien R^+ , der **transitive Abschluss** (auch **transitive Hülle**), und R^* , der **reflexive, transitive Abschluss**, von R wie folgt erklärt:

$$R^+ := \bigcup_{i \geq 1} R_i \text{ mit } R_1 := R \text{ und } R_{i+1} := R_i \cdot R. \quad R^* := R^+ \cup Id_A.$$

Hier ist entsprechend Definition 2.13 $Id_A := \{(a, a) \mid a \in A\}$ die *Identitätsrelation* auf A und offensichtlich R_i die i -fache Komposition von R mit sich selbst, was nicht identisch mit $R^i = \underbrace{R \times R \times \cdots \times R}_{i\text{-mal}}$ ist!

Betrachten wir das Geschehen bei den Relationen und deren Graphen, so bilden wir von der Menge $R_0 := \{(a, b), (c, b), (b, d), (b, e), (d, f)\} \subseteq R$, deren Graph $G(R_0)$ gerade ein Hasse-Diagramm der halbgeordneten Menge R ist, den **transitiven Abschluss**, um R zu erhalten.

2.17 Beispiel

Die Quasiordnung $R_c := \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, c), (c, b), (c, a)\}$ besitzt verschiedene erzeugende Relationen R_e mit $R_e^+ = R_e^* = R_c$. Zwei davon sind z.B.: $R_e := \{(a, b), (b, c), (c, a)\}$ und $R_e := \{(a, b), (b, a), (b, c), (c, b)\}$.

Nun ist jedoch nach obiger Definition der transitive Abschluss einer Relation R als unendliche Vereinigung definiert worden, und das scheint für algorithmische Verfahren kein gangbarer Weg zu sein. Muss hier jedoch wirklich eine unendliche Vereinigung gebildet werden?

Wenn die Grundmenge A und auch $\text{range}(R)$ unendliche Mengen sind, so kann es passieren, dass tatsächlich eine unendliche Vereinigung gebildet werden muss. Wir werden sehen, dass dies nicht erforderlich ist, wenn R oder sogar nur $\text{range}(R)$ endlich ist!

2.18 Beispiel

Wir betrachten $G \subseteq \mathbb{Z} \times \mathbb{Z}$ mit $G := \{(x, 2x) \mid x \in \mathbb{Z}\}$. Offensichtlich ist $(x, 2^i x) \in G_i \setminus G_{i-1}$ für jedes $i \in \mathbb{N}$. Wenn jedoch die Menge A endlich ist, so kann es in R^+ höchstens $r := |A|^2$ viele verschiedene Elemente geben. Für jedes $i \in \mathbb{N}$ enthält die Menge R_i davon eine Auswahl, und auch von diesen möglichen Auswahlen gibt es nur begrenzt viele, nämlich maximal so viele wie die Anzahl aller verschiedenen Teilmengen von $A \times A$, die genau $\sum_{j=0}^r \binom{r}{j} = 2^r$ ist. (Einen kurzer Beweis für diese Summe folgt aus der binomischen Formel: $\forall a, b \in \mathbb{R} : (a + b)^n = \sum_{i=0}^n \binom{n}{i} a^{n-i} b^i$. Setzt man in dieser Formel $a = b = 1$, so folgt die zu beweisende Formel sofort.)

Also kann die unendliche Vereinigung $\bigcup_{i \geq 1} R_i$ aus nur endlich vielen verschiedenen Mengen gebildet werden!

Schubfachprinzip Ohne eine formale Analyse der Eigenschaften der Mengen R_i aus Beispiel 2.18 zu machen beweisen wir diese Endlichkeitseigenschaft exakter. Wir stellen dazu zunächst fest, dass die Menge R_i das Element (a, b) genau dann enthält, wenn es im Graphen $G(R)$ einen Pfad der Länge i

von a nach b gibt! Wir werden als Beweisprinzip das wichtige Schubfachprinzip (*box principle* oder auch *pigeonhole principle*) von Lejeune Dirichlet (1805-1859) in seiner einfachsten Variante verwenden.

Das **Schubfachprinzip** in seiner einfachsten Formulierung sagt, dass jedes Verteilen von mehr als k Objekten in k Kästen dazu führt, dass in einem Kasten mindestens zwei Objekte liegen werden. Etwas mathematischer formuliert lautet es so:

2.19 Theorem (Schubfachprinzip, einfache Variante)

Sind A und B Mengen mit $|A| > |B|$, so gibt es keine injektive Abbildung $f : A \rightarrow B$.

Betrachten wir einen Vektor $\vec{x} := (x_1, x_2, \dots, x_n)^\top \in \mathbb{R}^n$, so kann dieser auch als eine Abbildung k der n Komponenten von \vec{x} auf die Werte $x_i \in \mathbb{R}$ angesehen werden:

$$k : \{1, \dots, n\} \rightarrow \mathbb{R} \quad \text{mit} \quad k(i) := \begin{cases} x_1 & \text{falls } i = 1 \\ \vdots & \\ x_n & \text{falls } i = n \end{cases}$$

Die Komponenten können wir als n Schubfächer ansehen, in welche die reelle Zahl $\sum_{i=1}^n x_i$ auf irgendeine Weise aufgeteilt wird. Wenn wir die Größen $\text{mean}(\vec{x})$ für das arithmetische Mittel und $\max(\vec{x})$ für das Maximum der Komponenten von \vec{x} benutzen, so läßt sich eine leichte Verallgemeinerung des Schubfachprinzips für reelle Zahlen formulieren.

2.20 Definition

Sei $n \in \mathbb{N}$ und $\vec{x} := (x_1, x_2, \dots, x_n)^\top \in \mathbb{R}^n$ dann bezeichne

$$\text{mean}(\vec{x}) := \frac{1}{n} \cdot \sum_{i=1}^n x_i$$

das **arithmetische Mittel** (*mean*) und

$$\max(\vec{x}) := \max\{x_i \mid 1 \leq i \leq n\}$$

das **Maximum** der Komponenten von \vec{x} .

Ist $M \subseteq \mathbb{R}$ eine endliche Menge, so ist

$$\text{mean}(M) := \frac{1}{|M|} \cdot \sum_{x \in M} x$$

und

$$\max(M) := \max\{x \mid x \in M\}.$$

Mit diesen Abkürzungen gilt nun folgendes Theorem:

2.21 Theorem (Schubfachprinzip, leicht verallgemeinert)

Für jedes $n \in \mathbb{N}$ und jeden beliebigen Vektor $\vec{x} \in \mathbb{R}^n$ gilt:

$$\text{mean}(\vec{x}) > 1 \quad \text{impliziert} \quad \max(\vec{x}) > 1.$$

2. Notationen und Grundbegriffe

Wenn in Theorem 2.21 $\vec{x} \in \mathbb{N}^n$ gewählt ist, so ist das (arithmetische) Mittel (*mean*) bzw. der durchschnittliche Wert (*average*) $\frac{1}{n} \cdot \sum_{i=1}^n x_i$ nur dann größer 1, falls mehr als n Objekte mit dem Gesamtwert $\sum_{i=1}^n x_i$ auf die n Fächer verteilt werden. Also folgt die einfache Variante des Schubfachprinzips aus dieser leichten Verallgemeinerung.

In vielen Fällen ist eine weitere Verallgemeinerung des einfachen Schubfachprinzips nützlich, die wir hier nicht nur formulieren, sondern auch beweisen wollen, selbst wenn dies alles offensichtlich erscheint.

2.22 Theorem (verallgemeinertes Schubfachprinzip)

1. Für jedes $n \in \mathbb{N}$ und jeden beliebigen Vektor $\vec{x} \in \mathbb{R}^n$ gilt: $\text{mean}(\vec{x}) \leq \max(\vec{x})$.

2. Für jedes $n \in \mathbb{N}$ und jeden beliebigen Vektor $\vec{x} \in \mathbb{N}^n$ gilt: $\lceil \text{mean}(\vec{x}) \rceil \leq \max(\vec{x})$.

Beweis: Aus der Definition des arithmetischen Mittels und des Maximums folgt sofort $\text{mean}(\vec{x}) = \frac{1}{n} \cdot \sum_{i=1}^n x_i \leq \frac{1}{n} \cdot \sum_{i=1}^n \max(\vec{x}) = \frac{1}{n} \cdot n \cdot \max(\vec{x}) = \max(\vec{x})$. Somit gilt 1. Falls nur natürliche Zahlen verwendet werden, so ist auch $\max(\vec{x}) \in \mathbb{N}$ und 2 folgt direkt aus 1. \square

Es folgt aus dem verallgemeinerten Schubfachprinzip auch stets das einfache Schubfachprinzip, nicht jedoch umgekehrt! Nach dem verallgemeinerten Schubfachprinzip können wir $\text{mean}(\vec{x}) \leq \max(\vec{x})$ voraussetzen. Wenn nun $\text{mean}(\vec{x}) > 1$ gilt, ergibt sich $1 < \text{mean}(\vec{x}) \leq \max(\vec{x})$ und also $\max(\vec{x}) > 1$ wie in Theorem 2.21 behauptet. Aus Theorem 2.21 folgt aber, wie gezeigt, auch das einfache Schubfachprinzip, Theorem 2.19.

In manchen Fällen ist die Anwendung des Schubfachprinzips nicht ganz einfach, weil nicht sofort klar ist, was den „Objekten“ (Menge A aus Theore 2.19) und was den „Schubladen“ (Menge B aus Theore 2.19) entsprechen soll. Daher geben wir hier zwei Beispiele an.

2.23 Beispiel

Sei $A \subseteq \mathbb{Z}^+$ eine Menge von zehn beliebigen natürlichen Zahlen nicht größer als 106, d.h. $A \subseteq D := \{1, \dots, 106\}$ mit $|A| = 10$. Es gibt dann immer zwei disjunkte Teilmengen $B, C \subseteq A$ mit $\sum_{x \in B} x = \sum_{x \in C} x$.

Beweis (zu Beispiel 2.23): Es gibt $2^{10} = 1024$ viele unterschiedliche Teilmengen, die wir mit $A_i \in 2^A, 0 \leq i \leq 1023$ bezeichnen. Die leere Menge $A_0 := \emptyset$ wird als eine der möglichen Lösungsmengen B oder C natürlich niemals vorkommen können, da $\sum_{x \in \emptyset} x = 0 \neq \sum_{x \in C} x$ für jede nichtleere Menge C ist.

Wir definieren $f : 2^A \setminus \emptyset \rightarrow \mathbb{N}$ durch $f(A_i) := \sum_{x \in A_i} x$, und ordnen damit jeder nichtleeren Teilmenge von A ihre Elementsumme zu. Keine Summe von höchstens zehn verschiedenen Zahlen $n_j \in \{1, \dots, 106\}$ überschreitet jemals den maximalen Wert 1015. Folglich ist $f(A_i) \leq 1015$ für jedes $i \in \{1, \dots, 1023\}$ und somit gilt $|\text{range}(f)| \leq 1015$.

Die Abbildung $f : 2^A \setminus \emptyset \rightarrow \mathbb{N}$ kann nun wegen $|\text{dom}(f)| = 1023$ nach dem einfachen Schubfachprinzip (Theorem 2.19) nicht injektiv sein und es muss daher zwei verschiedene Indizes $p, q \in \{1, \dots, 1023\}$ mit $f(A_p) = f(A_q)$ geben. Da diese Mengen jedoch nicht disjunkt sein müssen, bilden wir $B := A_p \setminus (A_p \cap A_q) = A_p \setminus A_q$ und $C := A_q \setminus (A_p \cap A_q) = A_q \setminus A_p$. Die Mengen B und C sind nun disjunkt und aus $f(A_p) = f(A_q)$ folgt sogleich $f(B) = f(A_p) - f(A_p \cap A_q) = f(A_q) - f(A_p \cap A_q) = f(C)$. \square

Bei dem Versuch, die Menge D der zugelassenen Zahlen um die Zahl 107 zu erweitern, also $D := \{1, \dots, 107\}$ zu erlauben, wird das Argument fehlschlagen, da dann $\text{dom}(f) = 1023 < \text{range}(f) = 1025$ wird. Solch eine Menge A zu finden, für die es keine Partition der beschriebenen Art gibt, ist jedoch nicht ganz einfach, auch wenn wir wissen, dass es diese geben *könnte*! Die Tatsache, dass wir das Schubfachprinzip nicht für einen Beweis verwenden können, besagt aber nicht, dass es nicht möglicherweise eine andere Beweistechnik geben könnte, mit der ein Beweis gelingt

Allgemein: Wenn ein Beweisverfahren scheitert, so kann es immer noch in anderes geben, das erfolgreich angewandt werden kann – es sei denn man kann **beweisen, dass jedes Verfahren zum Scheitern verurteilt ist!** Ein solcher Beweis ist meist sehr schwierig.

Selbst wenn wir wissen, dass als Summe 1 und 2 bei disjunkten Mengen B und C nicht auftreten können, wären die Mengen $\text{dom}(f)$ und $\text{range}(f)$ noch gleich mächtig und das Schubfachprinzip würde immer noch nicht greifen!

2.24 Beispiel

Sei $\langle n_1, n_2, \dots, n_k \rangle$ eine Folge von k nicht notwendigerweise verschiedenen ganzen Zahlen. Dann gibt es immer eine Teilfolge, deren Summe ohne Rest durch k teilbar ist.

Beweis (zu Beispiel 2.24): Wir betrachten für jedes $i \in \{1, \dots, k\}$ die Teilsummen $s_i := \sum_{j=1}^i n_j$. Wenn schon eine dieser Teilsummen ohne Rest durch k teilbar ist, sind wir fertig. Ist das nicht der Fall, so müssen die Reste **mod** k alle ungleich 0 sein. Seien nun $r_i \equiv s_i \pmod{k}$ mit $1 \leq r_i \leq k-1$ alle echten Reste bei Division durch k .

Da es nur $k-1$ solche Reste gibt, aber k unterschiedliche Teilsummen s_i gebildet wurden, muss es zwei Teilsummen s_p und s_q geben, die den gleichen Rest **mod** k haben.

O.B.d.A. (ohne Beschränkung der Allgemeinheit) sei $p < q$. Mit $s_p \equiv s_q \pmod{k}$, ist $n_{p+1} + \dots + n_q$ eine durch k teilbare Zahl! Hier wurde wieder das einfache Schubfachprinzip angewendet. \square

2.25 Theorem

Sei A eine endliche Menge und $R \subseteq A \times A$. Dann gibt es $t \in \mathbb{N}$, so dass $R^+ = \bigcup_{i=1}^t R_i$ ist.

Beweis: Aus Definition 2.18 wissen wir, dass $R^+ := \bigcup_{i \geq 1} R_i$ ist.

Betrachten wir die Mengen R_i für jedes $i \geq 1$. Da es höchstens $2^{|A|^2} - 1$ viele verschiedene nichtleere Mengen R_i geben kann, muss in der Folge $R_1, R_2, R_3, \dots, R_{2^{|A|^2}}$ eine Menge R_j , $j \geq 1$, zweimal vorkommen. (Anwendung des einfachen Schubfachprinzips).

Trete z.B. die Menge $R_j = R_l$ in dieser Aufzählung nach k Schritten das erste Mal wieder als Menge $R_l := R_{j+k}$ auf. Mit $R_j = R_{j+k}$ gilt also auch $\forall m \in \mathbb{N} : R_{j+m} = R_{j+k+m}$. Also folgt

$$R^+ = \bigcup_{i \geq 1} R_i = \bigcup_{i=1}^t R_i \text{ für } t := j+k < 2^{|A|^2}.$$

\square

Die unendliche Vereinigung $R^+ = \bigcup_{i \geq 1} R_i$ kann also als endliche Vereinigung von effektiv konstruierbaren Mengen R_i ebenfalls in endlich vielen Schritten konstruiert werden!

2. Notationen und Grundbegriffe

Für Beweise sind bisweilen statische Beschreibungen, wie jene, die sich aus dem folgenden Theorem ergibt, leichter zu handhaben. Für algorithmische Verfahren liefern diese Beschreibungen und Beweise jedoch leider nur sehr selten Hinweise zu deren Umsetzung.

2.26 Theorem

Sei R eine Relation auf A , R^* wie in Definition 2.16 erklärt, und außerdem sei $Q_A := \{Q \mid Q \text{ ist Quasiordnung auf } A\}$ die Klasse aller reflexiven und transitiven Relationen auf A . Dann gilt:

$$R^* = \hat{R} := \bigcap_{\substack{R \subseteq Q \\ Q \in Q_A}} Q.$$

Das Theorem besagt in anderen Worten: Der reflexive, transitive Abschluss R^* einer Relation R ist gerade der Durchschnitt aller Quasiordnungen, die R enthalten. R^* ist also die kleinste Quasiordnung, die R enthält.

Beweis (zu Theorem 2.26): Zunächst ist \hat{R} als Durchschnitt von reflexiven transitiven Relationen wieder reflexiv und transitiv. Weiter ist $R \subseteq \hat{R} \subseteq R^*$ denn R^* ist reflexiv und transitiv nach Definition 2.16 und daher auch Element der Klasse Q_A . Da \hat{R} Durchschnitt von anderen Mengen mit R^* ist, kann \hat{R} nur kleiner sein. Zu zeigen ist nur noch $R^* \subseteq \hat{R}$, was sofort aus den Inklusionen $R^* \subseteq Q$ für **jedes** einzelne Q des Durchschnitts folgt. Dies zeigen wir durch Induktion über den Index i der Mengen R_i in der Definition von $R^* = R^+ \cup Id_A$. Die Identität $Id_A = R_0$ ist selbstverständlich in jeder reflexiven Relation $Q \in Q_A$ enthalten. Als Induktionsannahme sei $R_j \subseteq Q$ für jedes $Q \in Q_A$. Sei $(a, c) \in R_{j+1} = R_j \cdot R$, dann $\exists b \in A : (a, b) \in R_j \wedge (b, c) \in R$. Nach Definition des Durchschnittes zur Bestimmung von \hat{R} ist $R \subseteq Q$ und nach Induktionsannahme gilt auch $R_j \subseteq Q$. Da Q transitiv ist folgt sofort $R_{j+1} \subseteq Q$. Mithin gilt $R_i \subseteq Q$ für jedes i und folglich ist ihre unendliche Vereinigung R^* ebenfalls Teilmenge von jedem $Q \in Q_A$ des Durchschnitts. Daher ergibt sich wie behauptet $R^* \subseteq \hat{R}$ und insgesamt die Gleichheit. \square

R^* ist zwar stets eine Quasiordnung, aber nicht immer eine partielle Ordnung. Als ein einfaches Gegenbeispiel wählen wir $R := R_c$ des Beispiels auf Seite 20.

2.3. Funktionen

Relationen sind als beliebige Verknüpfungen zweier Mengen definiert worden, wobei nur eine Richtung der Kanten vorgegeben ist. Spezielle Relationen sind uns als **Abbildungen** oder **Funktionen** bekannt. In Abbildung 2.1 sei zunächst eine grafische Übersicht über die wichtigsten Begriffe im Zusammenhang mit Funktionen und Relationen gegeben. Im Folgenden werden diese Eigenschaften formal definiert.

2.27 Definition

Eine Relation $R \subseteq A \times B$ heißt

1. **rechtseindeutig** oder (**partielle**) **Funktion** genau dann, wenn aus $(a, b) \in R \wedge (a, c) \in R$ stets $b = c$ folgt. Das heißt, dass zu jedem $a \in A$ höchstens ein $b \in B$ existiert mit $(a, b) \in R$. Dieses Element werden wir dann mit $R(a)$ bezeichnen. Wir notieren Funktionen meist mit kleinen Buchstaben (z.B. g, f) und wie üblich durch Angabe von Argument- und Wertebereich in der Form $f : A \longrightarrow B$, wenn f total ist und durch $f : A \xrightarrow{p} B$, wenn $f(a)$ möglicherweise nicht für jedes $a \in A$, also nur partiell, definiert ist. Wenn A ein cartesisches Produkt von n Mengen ist, so heißt n die **Stelligkeit** der Funktion.

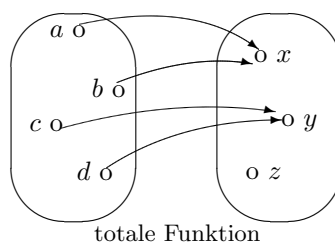
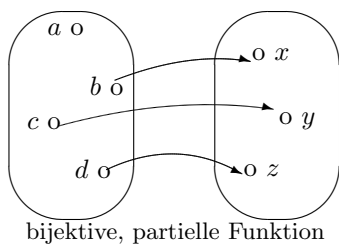
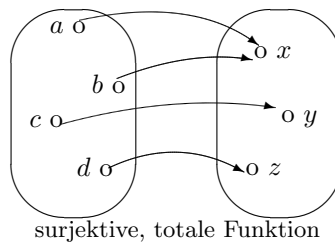
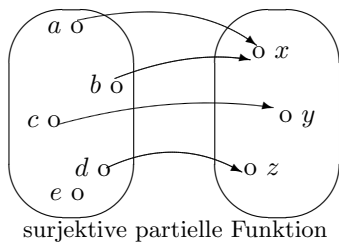
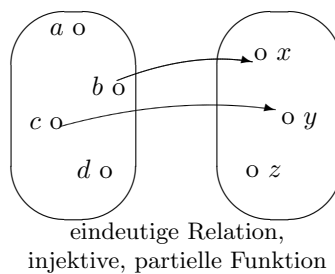
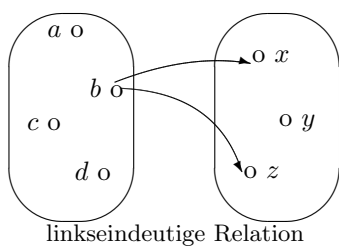
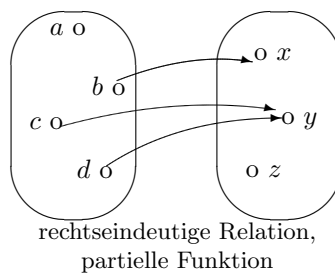
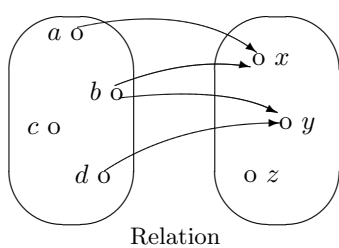


Abbildung 2.1.: Eigenschaften von Funktionen und Relationen

2. Notationen und Grundbegriffe

2. **linkseindeutig** genau dann, wenn aus $(a, b) \in R \wedge (c, b) \in R$ stets $a = c$ folgt.
3. **linkstotal** oder **total** genau dann, wenn $\text{dom}(R) = A$ ist.
4. **Abbildung** genau dann, wenn R eine totale Funktion ist, d.h. $|\text{range}(a)| = 1$ für jedes $a \in A$ ist.

Hier sollen also Abbildungen, im Gegensatz zu Funktionen, **nicht** partiell sein!

Wenn $f : A \xrightarrow{p} B$ eine Funktion mit $A = A_1 \times A_2 \times \cdots \times A_n$ ist, so schreiben wir anstelle von $f((x_1, x_2, \dots, x_n))$ kürzer $f(x_1, x_2, \dots, x_n)$.

2.28 Definition

Eine Funktion $f : A \xrightarrow{p} B$ heißt

1. **surjektiv** oder **Surjektion** genau dann, wenn $f(A) := \{f(a) \mid a \in A\} = B$ ist. Ist f total, dann sprechen wir von einer **Abbildung** von A auf B . Als Relation gesehen nennt man f **rechtstotal**.
2. **injektiv**, **eindeutig** oder **Injektion** genau dann, wenn f^{-1} eine (partielle) Funktion ist, d.h. aus $f(x) = f(y)$ stets $x = y$ folgt.
3. **bijektiv** oder **Bijektion** genau dann, wenn f surjektiv und injektiv ist.

2.29 Definition

Seien A und B Mengen. Wir notieren die Menge aller Abbildungen von A nach B , also der totalen Funktionen von A nach B , durch B^A .

Ist A zudem endlich und linear durch die Relation $<$ geordnet, so kann $f : A \rightarrow B$ als Vektor mit $|A|$ Komponente, die Einträgen aus B enthalten, betrachtet werden und wir notieren $f \in B^A$ (statt $f \in B^{|A|}$, wie häufig in der Linearen Algebra, oder statt umständlich $f \in \underbrace{B \times B \times \cdots \times B}_{|A|-\text{mal}}$). Die

Komponenten dieses $|A|$ -Tupels werden nun, statt mit $1, 2, \dots, |A|$, mit den Elementen aus der Menge A in aufsteigender Reihenfolge bezüglich der Ordnung $<$ benannt. Ist $A = A_1 \times A_2 \times \cdots \times A_n$ für endliche und linear geordnete Mengen A_i , so erhalten wir mit $f \in B^A$ eine $|A_1| \times \cdots \times |A_n|$ -Matrix.

Wir hatten bereits A^n als das n -fache cartesische Produkt der Menge A definiert. Alternativ könnten wir dies nun auch als die Menge aller Abbildungen $A^{\{1, 2, \dots, n\}}$ von $\{1, 2, \dots, n\}$ auf A verstehen. Mit $x \in A^n$ bezeichnet dann also der Funktionswert $x(i)$ das Element der i -ten Komponente des n -Tupels x . Diesen Wechsel in der Darstellungs- und Sichtweise ändern wir gerade so, wie dieser am besten passt, ähnlich wie ein Physiker Materie einmal als Teilchen und ein andermal als Welle beschreibt.

Die Notation für die Potenzmenge einer Menge A mit 2^A (siehe Definition 2.4) wird nun noch klarer, wenn wir dies als Abkürzung für $\{0, 1\}^A$ verstehen: Die charakteristische Funktion jeder Teilmenge von A ist mit dieser Schreibweise ein Element von $\{0, 1\}^A$!

Wir werden es in der Informatik, wie auch in der Mathematik, häufig mit unendlichen Mengen zu tun haben, die wir definieren und untersuchen müssen. Neben Verfahren zu ihrer Spezifikation werden wir Methoden benötigen, diese untereinander zu vergleichen und zu Klassen zusammenzustellen. Der in Definition 2.30 eingeführte Begriff der **Abzählbarkeit** beschreibt eine wichtige (notwendige) Eigenschaft unendlicher Mengen, ohne deren Erfüllung wir kein Verfahren finden könnten, mit dem wir die einzelnen Elemente der unendlichen Menge Schritt für Schritt erzeugen.

2.30 Definition

Eine Menge M heißt **abzählbar** genau dann, wenn sie entweder endlich ist oder gleich mächtig zur Menge \mathbb{N} der natürlichen Zahlen.

Für jede abzählbare Menge M gilt nach dieser Definition entweder $M = \emptyset$ oder es existiert eine surjektive Abbildung $g : \mathbb{N} \rightarrow M$ auf die Menge M , kurz $g(\mathbb{N}) = M$. Wir nennen g dann eine **Abzählung** von M . Bei abzählbaren unendlichen Mengen M kann immer auch eine Bijektion zwischen \mathbb{N} und M angegeben werden.

Ist M eine abzählbare Menge mit der Abzählung $g : \mathbb{N} \rightarrow M$ und g ist eine *berechenbare* Funktion, so heißt M **aufzählbar**. Es sei bereits an dieser Stelle darauf hingewiesen, dass nicht jede mathematisch wohldefinierte Funktion auch berechenbar ist! Der Grund dafür kann in der Nicht-Existenz einer endlichen Beschreibung eines endlichen Berechnungsverfahrens, also eines Algorithmus' begründet sein.

Zum Abschluss dieses Abschnittes wollen wir noch ein wichtiges Lemma beweisen.

2.31 Lemma

Jede Teilmenge einer abzählbaren Menge ist auch wieder abzählbar.

Beweis: Sei $f : \mathbb{N} \rightarrow M$ eine Abzählung der Menge M und $L \subseteq M$ eine beliebige Teilmenge. Gilt $L = M$ oder $L = \emptyset$, so ist nichts zu beweisen. Ist $L \neq M$, so gibt es wegen $L \neq \emptyset$ ein Element $m \in L$. Eine Abzählfunktion $g : \mathbb{N} \rightarrow L$ wird nun definiert durch

$$g(i) := \begin{cases} f(i) & \text{falls } f(i) \in L \\ m & \text{sonst} \end{cases}.$$

□

2.4. Beweistechniken

In der theoretischen Informatik, wie auch in der Mathematik, werden immer wieder Standard-Beweistechniken angewandt. Zu diesen Verfahren gehören u.a. die *vollständige Induktion*, *Widerspruchsbeweise* und insbesondere *Diagonalisierungstechniken*. Im folgenden Abschnitt wollen wir einige dieser Verfahren exemplarisch diskutieren.

Direkter Beweis einer Wenn-Dann Aussage Bei einem direkten Beweis schließen wir von der Hypothese A direkt auf die Schlussfolgerung B . Schematisch:

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{(A \rightarrow B)}$$

Diese Form des Beweisens wollen wir im Beweis des folgenden Satzes untersuchen.

2. Notationen und Grundbegriffe

2.32 Beispiel

Wir wollen zeigen: Wenn $m, n \in \mathbb{Z}$ beide ungerade sind, dann ist $m + n$ gerade.

Ein direkter Beweis dieser Wenn-Dann-Aussage ist der folgende:

Beweis: Zu zeigen ist $A \rightarrow B$ mit:

Aussage A = Die Zahlen $m, n \in \mathbb{Z}$ sind beide ungerade.
Aussage B = Die Zahl $m + n$ ist gerade.

- Wir nehmen A an.
- Also gibt es Zahlen $k, l \in \mathbb{Z}$, so dass $m = 2k + 1$ und $n = 2l + 1$ gilt.
- Dann folgt: $m + n = (2k + 1) + (2l + 1) = 2(k + l + 1)$.
- Also ist $m + n$ gerade (Aussage B).
- Also konnte unter der Annahme von A auch B gezeigt werden.

Also gilt nach der obigen Schlussregel $A \rightarrow B$. □

Indirekter Beweis Im indirekten Beweis nimmt man an, die zu beweisende Aussage sei falsch. Dann benutzt man einige der Voraussetzungen (Prämissen) und andere als wahr bekannte Aussagen, die ja alle als Voraussetzungen verwendet werden können, und schließt dann, dass eine der Voraussetzungen oder eine als wahr bekannte Aussage falsch sein muss. Als Schlussfolgerung bleibt (bei richtigem Rechnen) nur noch übrig, dass die zuvor als falsch angenommene Aussage wahr sein muss.

2.33 Beispiel

Wir wollen zeigen, dass die Aussage $s := \frac{a+b}{2}$ ist mindestens so groß wie $\sqrt{a \cdot b}$ gilt. Als Prämissen setzen wir voraus: $p := \text{„}a \text{ ist positiv reell“}$ und $q := \text{„}b \text{ ist positiv reell“}$. Als bekanntes Zusatzwissen nehmen wir die Aussage $r := \text{„}c^2 \text{ ist nicht negativ für jede reelle Zahl } c\text{“}$. Die Struktur der zu beweisenden Aussage ist also: „ p und q und r implizieren s “, formal notiert als $p \wedge q \wedge r \rightarrow s$. Für den indirekten Beweis nehmen wir also an, dass s falsch und somit $\neg s$ wahr sei. Wir nehmen also die Gültigkeit von „ $\frac{a+b}{2}$ ist kleiner als $\sqrt{a \cdot b}$ “ an. Wenn wir nun einen Widerspruch ableiten können, ist Aussage s gezeigt.

Beweis (zu Beispiel 2.33): Aus $\neg s$, formalisiert durch $(a + b) < 2 \cdot \sqrt{a \cdot b}$, folgt $(a + b)^2 < 4 \cdot a \cdot b$ und weiter $a^2 + 2 \cdot a \cdot b + b^2 < 4 \cdot a \cdot b$, d.h. $a^2 + b^2 < 2 \cdot a \cdot b$, bzw. $a^2 + b^2 - 2 \cdot a \cdot b < 0$. Dies widerspricht aber der Tatsache, dass jedes Quadrat einer reellen Zahl positiv oder Null ist (Aussage r). Also kann $a^2 + b^2 - 2 \cdot a \cdot b = (a - b)^2$ nach Voraussetzung r nicht negativ sein und somit ist der Schluss $p \wedge q \wedge r \rightarrow s$ richtig! □

Verallgemeinern wir die Aussage aus Beispiel 2.33, so erhalten wir die Aussage, dass stets das arithmetische Mittel mindestens so groß wie das geometrische Mittel ist, formal:

$$\frac{1}{n} \cdot \sum_{i=1}^n x_i \geq \sqrt[n]{\prod_{i=1}^n x_i}.$$

Induktionsbeweise Sei $\phi(n)$ irgendeine Aussage, die für alle natürlichen Zahlen n gültig sein soll. Eine Möglichkeit dies nachzuweisen besteht darin, vollständige Induktion anzuwenden.

Angenommen, folgendes gilt:

1. Induktionsanfang: Die Eigenschaft gilt für $n = 0$, d.h. $\phi(0)$ ist wahr.
2. Induktionsschritt: Wenn die Eigenschaft für ein beliebiges (aber festes) n gilt, dann können wir zeigen, dass sie auch für $n + 1$ gilt. Formal:

$$\forall n \in \mathbb{N} : \phi(n) \rightarrow \phi(n + 1)$$

Induktionsprinzip: Wenn für ϕ sowohl der Induktionsanfang als auch der Induktionsschritt gilt, dann gilt $\phi(n)$ für alle natürlichen Zahlen.

2.34 Beispiel

Wir zeigen die Summenformel $\sum_{i=1}^n i = \frac{1}{2}n(n+1)$ mit Hilfe des Induktionsprinzips.

Beweis: Induktionsanfang: Die Eigenschaft gilt für $n = 0$, denn $\sum_{i=1}^0 i = 0$ und $\frac{1}{2}n(n+1) = 0$.

Induktionsschritt: Wir nehmen an, dass die Summenformel für ein beliebiges, aber festes n gilt: $\sum_{i=1}^n i = \frac{1}{2}n(n+1)$ (Induktionsannahme, IA).

Unter der Voraussetzung, dass die Induktionsannahme gilt, wollen wir die Summenformel für $n+1$ zeigen: $\sum_{i=1}^{n+1} i = \frac{1}{2}(n+1)(n+2)$. Dies können wir durch folgende Umformung zeigen:

$$\sum_{i=1}^{n+1} i = \left(\sum_{i=1}^n i \right) + (n+1) \stackrel{IA}{=} \frac{1}{2}n(n+1) + (n+1) = \left(\frac{1}{2}n + 1 \right)(n+1) = \frac{1}{2}(n+2)(n+1)$$

Nach dem Induktionsprinzip gilt die Aussage also für alle natürlichen Zahlen. □

Diagonalisierung Auf Georg Cantor geht das mathematische Beweisverfahren der *Diagonalisierung* zurück, mit dem er beweisen konnte, dass die Menge \mathbb{R} der reellen Zahlen **nicht** abzählbar ist. Da diese Technik speziell bei der Methode des indirekten Beweises auch später noch benutzt werden wird, sei sie hier an einem ersten Beispiel vorgeführt. Zuvor jedoch ein sicher bekanntes Beispiel aus der elementaren Mathematik als Anwendung und zur Verdeutlichung des Prinzips eines indirekten Beweises.

Das folgende Theorem wollen wir nun mittels Diagonalisierung beweisen.

2.35 Theorem

Die Menge aller abzählbar unendlichen Folgen natürlicher Zahlen aus \mathbb{N} ist nicht abzählbar.

Beweis: Sei F_{abz} die Menge aller abzählbar unendlichen Folgen natürlicher Zahlen. Wenn wir nun annehmen, dass F_{abz} eine abzählbare Menge sei, so gibt es eine Abzählung, d.h., eine Abbildung $g : \mathbb{N} \rightarrow F_{\text{abz}}$ unter der $g(i)$ für jedes $i \in \mathbb{N}$ eine abzählbar unendliche Folge natürlicher Zahlen ist. Für jede dieser Folgen $g(i)$ gibt es also eine eigene Abzählung $f_i, i \in \mathbb{N}$ der Folgenglieder. Dies können wir uns als eine unendliche Matrix vorstellen, in der in der i -ten Zeile die Elemente der Folge $g(i)$ von links nach rechts entsprechend ihrer Abzählung $f_i(\mathbb{N})$ eingetragen sind, wie in Abbildung 2.2:

2. Notationen und Grundbegriffe

i	$g(i)$	0	1	2	3	4	5	\dots	n	\dots
0	$g(0)$	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	$f_0(4)$	$f_0(5)$	\dots	$f_0(n)$	\dots
1	$g(1)$	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	$f_1(5)$	\dots	$f_1(n)$	\dots
2	$g(2)$	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	$f_2(5)$	\dots	$f_2(n)$	\dots
3	$g(3)$	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	$f_3(5)$	\dots	$f_3(n)$	\dots
4	$g(4)$	$f_4(0)$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	$f_4(5)$	\dots	$f_4(n)$	\dots
5	$g(5)$	$f_5(0)$	$f_5(1)$	$f_5(2)$	$f_5(3)$	$f_5(4)$	$f_5(5)$	\dots	$f_5(n)$	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots	\vdots	\dots
n	$g(n)$	$f_n(0)$	$f_n(1)$	$f_n(2)$	$f_n(3)$	$f_n(4)$	$f_n(5)$	\dots	$f_n(n)$	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\dots	\vdots	\ddots

Abbildung 2.2.: Angenommene Abzählung aller abzählbaren Zahlenfolgen über \mathbb{N}

Wenn wir nun die neue Folge F_{diagonal} definieren durch

$$F_{\text{diagonal}} := \langle f_0(0) + 1, f_1(1) + 1, f_2(2) + 1, f_3(3) + 1, f_4(4) + 1, \dots \rangle, \quad (2.1)$$

so ist diese *offensichtlich* abzählbar! Sie müsste also in der angenommenen Abzählung $g : \mathbb{N} \rightarrow F_{\text{abz}}$ aller abzählbaren Folgen als $g(j)$ für ein $j \in \mathbb{N}$ vorkommen. Dann müsste jedoch definitionsgemäß, lt. Gleichung (2.1), für das Diagonalelement $f_j(j)$ gerade $f_j(j) = f_j(j) + 1$ gelten, was sicher unmöglich ist. Also kann die Annahme, F_{abz} sei eine abzählbare Menge, nicht stimmen. \square

Die gleiche Beweistechnik lässt sich beim Beweis des folgenden Theorems zur Abzählbarkeit der Potenzmenge zu einer gegebenen Menge anwenden.

2.36 Theorem

Die Potenzmenge 2^M einer abzählbaren Menge M ist genau dann abzählbar, wenn M eine endliche Menge ist.

Beweis: In Theorem 2.36 sind zwei Aussagen versteckt, die beide bewiesen werden müssen:

- (a) Wenn M *nicht* endlich ist, so ist die Menge 2^M aller Teilmengen von M nicht abzählbar.
- (b) Wenn M endliche Menge ist, so ist 2^M abzählbar.

zu (a): Sei M eine abzählbare unendliche Menge. Die surjektive Abbildung $f : \mathbb{N} \rightarrow M$ der Abzählung der Elemente von M . Für jedes $i \in \mathbb{N}$ bezeichnet also $f(i)$ ein bestimmtes Element von M . O.B.d.A. dürfen wir annehmen, dass in dieser Abzählung von M keines der Elemente doppelt auftritt, dass also f sogar eine Bijektion ist! Falls nun die Potenzmenge 2^M ebenfalls abzählbar ist, so gibt es auch für diese Menge eine Abzählung $g : \mathbb{N} \rightarrow 2^M$. Wir definieren die Diagonalmenge $D := \{f(j) \mid f(j) \notin g(j), j \in \mathbb{N}\}$. Offensichtlich ist $D \subseteq M$ und also $D \in 2^M$. Daher muss es eine Zahl $k \in \mathbb{N}$ mit $D = g(k)$ geben. Auf Grund der Definition der Menge D bedeutet dies nun aber, dass $f(k) \in D$ genau dann gilt, wenn $f(k) \notin g(k)$, also genau dann, wenn $f(k) \notin D$ gilt. Dies ist offensichtlich ein Widerspruch und somit kann 2^M nicht abzählbar sein.

zu (b): Sei $M := \{m_0, m_1, \dots, m_{k-1}\}$ eine endliche Menge. Wie auf Seite 12 festgestellt, hat die Potenzmenge von M gerade $2^{|M|}$ viele Elemente. Somit ist auch die Menge 2^M endlich und damit per Definition abzählbar.

□

Im vorangegangenen Beweis wurde für die endlichen Teilmengen der Menge M in Teil (b) keine konkrete Abzählung konstruiert. Statt dessen haben wir ein früheres Ergebnis verwendet. Auch die Angabe einer Abzählung von 2^M wäre eine Möglichkeit für diesen Teilbeweis gewesen. Dies ist in unserem Fall jedoch viel aufwändiger. Der Vollständigkeit halber wollen wir diese Alternative trotzdem präsentieren.

Beweisalternative zu Theorem 2.36 (b):

Für $k = 0$ ist $M = \emptyset$ und auch $2^M = \emptyset$, also per Definition abzählbar. Die leere Menge ist auch in jeder Potenzmenge enthalten. Für $k \geq 1$ definieren wir nun beispielsweise die Abzählfunktion $f : \mathbb{N} \rightarrow 2^M$ durch

$$f(x) := \begin{cases} \{m_{i_0}, \dots, m_{i_j}\} & \text{falls } x = \sum_{l=0}^j m_{i_l} \cdot 2^{i_l} \\ \emptyset & \text{sonst} \end{cases}$$

□

In der *konstruktiven* Beweisalternative wird eine $(k-1)$ -stellige Binärzahl⁷ als *Kodierung* für jede Teilmenge von M verwandt. Für $0 \leq l \leq k-1$ ist die l -te Stelle (von rechts) der Binärdarstellung 1 genau dann, wenn das Element $m_l \in M$ in der kodierten Teilmenge enthalten ist. Zum Beispiel kodiert die Binärdarstellung 10010 gerade die Menge $\{m_1, m_4\}$. Für alle Zahlen x , die sich nicht auf diese Weise als Kodierung einer Teilmenge von M interpretieren lassen, wird $f(x)$ die leere Menge, die ja in jeder Potenzmenge enthalten ist! Die so definierte totale Funktion f ist surjektiv, aber nicht bijektiv, da jede nichtleere endliche Teilmenge von M eindeutig durch ein endliches *Binärwort* aus den Werten der charakteristischen Funktion dargestellt wird, die leere Menge aber unendlich oft als Bild vorkommt.

2.5. Strukturen

Unter den vielen möglichen Relationen auf einer Menge sind die **Verknüpfungen** besonders hervorzuheben. Letztere sind Vorschriften, die zwei Elementen jeweils ein drittes zuordnen und werden in der Regel als **Operationen** bezeichnet. Solche (zweistelligen) Verknüpfungen können für eine Menge M z.B. als Abbildungen $g : M \times M \rightarrow M$ geschrieben werden, meistens wird für sie jedoch die Infixschreibweise verwendet. Mengen und darauf definierte Verknüpfungen sind uns von Gruppen, Ringen, Körpern oder anderen Algebren oder Strukturen bekannt.

2.37 Definition

Eine Menge A mit einer oder mehreren darin ausgezeichnete Relationen, Operationen und Konstanten nennt man eine **Struktur**. Eine Struktur wird notiert als ein Tripel (A, F, P) , wobei die Mengen folgende Bedeutung haben:

A ist die nicht-leere **Trägermenge** ($A \neq \emptyset$), auch Individuenbereich genannt, die oft mit den nullstelligen Funktionen identifiziert wird.

F ist eine **Klasse von Funktionen** oder Abbildungen.

P ist eine **Klasse von Prädikaten**, also Funktionen f mit $\text{range}(f) = \{w, f\}$

⁷siehe Abschnitt 2.8 für eine formale Definition des Begriff der Binärdarstellung.

2. Notationen und Grundbegriffe

Falls in der Struktur keine Funktionen vorhanden sind, also $F = \emptyset$ ist, so heißt die Struktur **relationale Struktur** bzw. **Relationalstruktur** oder auch **Relationensystem**.

Falls $P = \emptyset$ ist, so nennt man die Struktur eine **algebraische Struktur** oder kurz **Algebra**.

Die Bedingung $A \neq \emptyset$ in Definition 2.37 ist nötig, da sonst die Aussage „ $\forall x \in A : p(x)$ “ impliziert $\exists x \in A : p(x)$ “, nicht gültig wäre.

Bisher haben wir im Wesentlichen nur relationale Strukturen behandelt, z.B., partielle Ordnungen, Äquivalenzrelationen oder Graphen. In der Informatik begegnet uns die **Halbgruppe** besonders häufig; sie ist die wohl einfachste und grundlegendste algebraische Struktur.

2.38 Definition

Sei H eine Menge und $\odot : H \times H \rightarrow H$ eine Abbildung (Verknüpfung) für die das Assoziativgesetz gilt, d.h. $\forall a, b, c \in H : (a \odot b) \odot c = a \odot (b \odot c)$, dann heißt (H, \odot) **Halbgruppe**. Wenn klar ist, welche Verknüpfung gemeint ist, schreiben wir auch oft nur H statt (H, \odot) .

2.39 Definition

Eine Halbgruppe (H, \odot) heißt **Monoid** genau dann, wenn es ein Element $e_H \in H$ gibt, so dass $e_H \odot m = m \odot e_H = m$ für jedes $m \in H$ gilt. Das Element $e_H \in H$ nennt man das **neutrale Element** von (H, \odot) . Oft wird der Index H bei e_H weggelassen und nur e geschrieben.

2.40 Beispiel

$(\mathbb{N} \setminus \{0\}, +)$ ist eine Halbgruppe, während $(\mathbb{N}, +)$ ein Monoid mit der Null als neutralem Element ist.

$(\{0, 1\}, \cdot)$ ist ein endliches Monoid mit dem neutralen Element 1, wenn die Ziffern 0 und 1 als natürliche Zahlen aufgefaßt werden und die Operation \cdot als gewöhnliche Multiplikation interpretiert wird.

Die Menge aller injektiven Abbildungen einer Menge M in sich selbst, ist bezüglich der Hintereinanderausführung \circ ein Monoid, für das die Identität Id_M das neutrale Element ist. Sind f und g Abbildungen, so ist die neue Abbildung $f \circ g$, gesprochen „ f vor g “, erklärt durch:

$$(f \circ g)(x) = g(f(x)).$$

Dies ist äquivalent zur Komposition von Relationen. Es entspricht $(f \circ g)$ also gerade der Schreibweise $(f \cdot g)$, wenn f und g als Relationen aufgefaßt werden!

2.41 Definition

Für Teilmengen U, V einer Halbgruppe oder eines Monoides (H, \odot) sei die Menge $U \cdot V := \{u \odot v \mid u \in U, v \in V\}$ das **Komplexprodukt** von U und V .

Mit dem Komplexprodukt zweier Teilmengen einer Halbgruppe, wird also stillschweigend die Halbgruppenoperation auf die elementweise Verknüpfung übernommen!

Damit wird die Potenzmenge von H zu einer Halbgruppe $(2^H, \cdot)$, oder einem Monoid mit dem Einselement $\{e_H\}$, falls dieses in (H, \odot) existiert.

2.42 Definition

Eine Teilmenge M von (H, \odot) heißt gegenüber dem Produkt \odot abgeschlossen genau dann, wenn $\forall a, b \in M : a \odot b \in M$ gilt.

2.43 Definition

Sei (H, \odot) eine Halbgruppe (bzw. $(H \cup \{e_H\}, \odot)$ ein Monoid) dann heißt $M \subseteq H$ eine **Unterhalbgruppe** (bzw. **Untermonoid**) genau dann, wenn (M, \odot) selbst eine Halbgruppe (bzw. Monoid) ist. Das heißt, M ist gegenüber \odot abgeschlossen und enthält als Untermonoid auch das neutrale Element e_H .

2.44 Definition

Eine Abbildung h von einer Halbgruppe (H, \odot) auf eine andere Halbgruppe (G, \otimes) wird genau dann **Homomorphismus** genannt, wenn $\forall x, y \in H : h(x \odot y) = h(x) \otimes h(y)$ gilt. Ein Homomorphismus ist also eine strukturerhaltende Abbildung, die außer den Elementen auch noch die Operationen auf einander abbildet.

Ein Homomorphismus $f : (H, \odot) \rightarrow (G, \otimes)$ kurz $f : H \rightarrow G$ heißt

1. **Epimorphismus** genau dann, wenn f surjektiv ist
2. **Monomorphismus** genau dann, wenn f injektiv ist
3. **Isomorphismus** genau dann, wenn f bijektiv ist
4. **Endomorphismus** genau dann, wenn $H = G$ ist
5. **Automorphismus** genau dann, wenn f Isomorphismus ist und $H = G$ gilt.

Für $M \subseteq H$ definieren wir nun, ähnlich wie in Definition 2.16 für Relationen, den transitiven Abschluss unter Komplexproduktbildung.

2.45 Definition

Für eine Teilmenge $M \subseteq H$ einer Halbgruppe (H, \odot) (bzw. eines Monoides $(H \cup \{e_H\}, \odot)$) seien der **transitive Abschluss** M^+ sowie der **transitive und reflexive Abschluss** M^* wie folgt erklärt:

$$M^+ := \bigcup_{i \geq 1} M^i \text{ mit } M^1 := M \text{ und } M^{i+1} := M^i \cdot M$$

$$M^* := M^+ \cup \{e_H\}$$

wobei e_H das neutrale Element des Monoides ist.

- M^+ heißt **von M erzeugte Unterhalbgruppe**.
- M^* heißt **von M erzeugtes Untermonoid**.
- M heißt **Erzeugendensystem** von M^+ bzw. von M^* . Ist das Erzeugendensystem M endlich, so heißen M^+ bzw. M^* **endlich erzeugt**.
- M^+ bzw. M^* heißt **von M frei erzeugt** genau dann, wenn für $\forall i, k, r \in \mathbb{N} \ \forall m_i, n_i \in M \setminus \{e_H\}$ aus $m_1 \odot m_2 \odot \dots \odot m_k = n_1 \odot n_2 \odot \dots \odot n_r$ stets $k = r$ und $m_i = n_i$ folgt.

Es ist zu beachten, dass in obiger Darstellung die Mengen M^i das i -fache Komplexprodukt bezeichnen und nicht das cartesische Produkt! Diese Bedeutung der Exponenten wird später meistens bei Wortmengen, also Teilmengen von Σ^* für ein Alphabet Σ verwendet. Die Definitionen hierzu finden sich in Abschnitt 2.6.

2. Notationen und Grundbegriffe

Die Darstellung eines Elementes eines von M frei erzeugten Monoides M^* ist stets eindeutig. Außer dem Assoziativgesetz gilt keine andere Beziehung zwischen den Erzeugenden, es ist *frei von zusätzlichen Relationen*.

Ähnlich wie in Theorem 2.26 für Relationen gezeigt, ist M^+ der Durchschnitt aller M enthaltenden Unterhalbgruppen.

Eine von Mathematikern oft benutzte, äquivalente Definition des freien Monoides ist folgende, die auf andere Weise seine herausragende Rolle deutlich macht:

Ein Monoid M^* heißt genau dann frei von M erzeugt, wenn es zu jedem Homomorphismus $f : M \rightarrow A$ für ein Monoid A , jeweils genau einen Homomorphismus $h : M^* \rightarrow A$ gibt, so dass das Diagramm in Abbildung 2.3 kommutativ ist, d.h., dass $i \circ h = f$ gilt. Die Abbildung i ist dabei die sogenannte natürliche Einbettung von M in M^* (als Teilmenge).

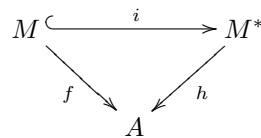


Abbildung 2.3.: Ein kommutierendes Diagramm

Hieraus folgt: Aus einer *frei wählbaren* Interpretation der Symbole eines Alphabetes Σ in dem beliebigen Monoid A (durch f) erhält man auf eindeutige Weise die Interpretation aller Wörter aus Σ^* .

2.6. Wortmengen

In den vorangegangenen Abschnitten haben wir uns mit allgemeinen Begriffen zu Mengen, Relationen und Funktionen beschäftigt. Wir wollen nun eine spezielle Klasse von Mengen definieren: die Wortmengen. Mengen von Wörtern sind immer dort von besonderer Wichtigkeit, wenn die Objekte, mit denen gearbeitet werden soll, kodiert werden müssen. Wir hatten bereits ein solches Beispiel kennengelernt: Binärwörter. Sie können als eine Kodierung der natürlichen Zahlen angesehen werden, genau wie auch die uns geläufige Dezimaldarstellung. Allen Kodierungen gemein ist die Tatsache, dass nur eine **endliche Menge von Zeichen** zur Bildung der Darstellung erlaubt ist (Bei der Binärdarstellung nur 0 und 1, bei der Dezimaldarstellung die zehn Ziffern 0 bis 9). Eine solche Darstellung werden wir im allgemeinen als ein **Wort** über einem endlichen **Alphabet** bezeichnen.

2.46 Definition

Ein **Alphabet** ist eine total geordnete endliche Menge von unterschiedlichen **Zeichen** (oder **Symbolen**), vergl. DIN 44 300.

Besonders in der Mathematik und der Logik bezeichnet man gelegentlich auch unendliche Mengen von Symbolen als Alphabete, allerdings sind dies dann stets abzählbare, total geordnete Mengen von Zeichen. Sofern nicht ausdrücklich anders erwähnt, wollen wir unter einem Alphabet jedoch immer eine endlichen Zeichenvorrat verstehen.

Tabelle 2.1.: ASCII – Code

Spalte	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(8	H	X	h	x
9	HT	EM)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[k	{
C	FF	FS	,	<	L	\	l	
D	CR	GS	-	=	M]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Ein in der Informatik häufig verwendetes Alphabet ist der sogenannte ASCII-Code (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), ein 7-bit Code, dessen achtes Bit in der Byte-Darstellung als Prüfbit, oder zur Erweiterung des Zeichensatzes verwendet wird. Es ist dies eine Liste von 128 Schrift- und Steuerzeichen, die als Hexadezimalzahlen kodiert sind: In der 16×8 Matrix (siehe Tabelle 2.1) wird die Spaltennummer j als Hexadezimalziffer *vor* die Hexadezimalziffer i der Zeilennummer gesetzt und dann die Zeichenkette ji zur Basis 16 interpretiert. Dadurch ergibt sich eine lineare Ordnung der Zeichen entsprechend ihrer Codenummern.

Einerseits kann der ASCII-Code als ein 128 Zeichen umfassendes Alphabet angesehen werden, andererseits basiert die Hexadezimaldarstellung der ASCII-Zeichen auf dem 16 zeichn umfassenden Alphabet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, A, B, C, D, E, F\}$. Diese beiden Alphabete sollten nicht verwechselt werden!

Für das Symbol "A" ergibt sich gerade $[41]_{16} = 65$, für "EOT" (*end of transmission*) $[4]_{16} = 4$, für "SP" (*space*) $[20]_{16} = 32$ und für "DEL" (*delete*) entsprechend $[7F]_{16} = 127$. Und es ergibt sich die Reihenfolge: $\dots \prec A \prec B \prec \dots \prec Z \prec \dots \prec a \prec \dots \prec z \prec \dots$.

In der Regel wird die Ordnung bei Alphabeten als implizit gegeben betrachtet, und jeder endliche Zeichenvorrat als Alphabet bezeichnet. Falls wir die Ordnung der Symbole bezeichnen möchten, wählen wir dazu das Zeichen \prec .

Um aus einem schon bekannten Wort w über dem Alphabet $\{A, B, \dots, Z\}$ ein neues zu konstruieren, kann man einzelne Symbole davor, dazwischen oder dahinter schreiben; und man kann auch zwei be-

2. Notationen und Grundbegriffe

kannte Wörter⁸ aneinanderschreiben. Aus den Wörtern „S“, (auch ein einzelnes Symbol ist ein Wort!) „HAUS“, „BOOT“ und „BESITZER“ kann man so die Wörter BOOTS, BOOTSHAUS, HAUSBOOT, HAUSBESITZER, BOOTSBESITZER oder auch HAUSBOOTBESITZER erzeugen. Diese Hintereinanderschreibung von Zeichenketten bezeichnet man als **Konkatenation**, und mit ihr als assoziativer Operation zwischen den Zeichenketten (Wörtern) ist eine Struktur, hier eine Halbgruppe definiert. Wenn wir an einer bestimmten Stelle **kein** Wort notieren wollen, aber diese Stelle dennoch bezeichnen müssen, so benötigen wir dafür ein spezielles Symbol (ähnlich der Null bei den Zahlen). Wir nennen dieses das *leere Wort* und notieren es als λ (*kleines lambda*). In vielen englischsprachigen Büchern findet man als Bezeichnung für das leere Wort ϵ (*epsilon*).

Das leere Wort λ ist niemals in einem Alphabet enthalten! Es handelt sich ja gerade um dasjenige Wort, welches ohne jegliches Symbol aus dem Alphabet gebildet wird.

2.47 Definition

Für ein Alphabet Σ , ist Σ^* das **freie Monoid** mit der **Konkatenation** oder Hintereinanderschreibung der einzelnen Zeichen als Monoidoperation, und dem leeren Wort λ als neutralem Element. Für $w \in \Sigma^*$ schreiben wir w^k anstelle von $\underbrace{ww \cdots w}_k$.

Jede Menge $L \subseteq \Sigma^*$ heißt **formale Sprache**.

$L := \{w \mid w \in \{a, b\}^* \wedge (w \text{ enthält das Symbol } a \text{ doppelt so oft wie das Symbol } b)\}$ ist zum Beispiel eine recht einfache formale Sprache mit umständlicher Beschreibung. Auch die Menge $\{w \mid w \in \{a, b, c\}^* \wedge (w \text{ beginnt mit dem Symbol } a)\}$ ist eine einfache formale Sprache mit unendlich vielen Elementen.

Einige Eigenschaften von Zeichenketten werden wir bei der Beschreibung und Spezifikation von formalen Sprachen benötigen, um solche umständlichen Definitionen vermeiden zu können:

2.48 Definition

- Zu jedem Wort $w \in \Sigma^*$ und jedem Symbol $x \in \Sigma$ bezeichne $|w|_x$ die **Häufigkeit des Vorkommens** des Symbols x im Wort w .
- Mit $|w|$ wird die **Länge des Wortes** w bezeichnet, d.h. $|w| := \sum_{x \in \Sigma} |w|_x$.
- Wir schreiben $u \sqsubseteq v$, wenn es ein Wort $w \in \Sigma^*$ gibt, so dass $v = uw$ ist. Das Wort u wird **Präfix** von v genannt.
- Das Wort u wird **echtes Präfix** von v genannt, wenn $u \sqsubseteq v$ und weder $u = v$ noch $u = \lambda$ gilt. Wir notieren dies als $u \subset v$.
- Gilt $z = uvw$ für Wörter $u, v, w, z \in \Sigma^*$, so ist v ein **Teilwort** von z .

2.49 Beispiel

Für jedes Wort w gilt $\lambda \sqsubseteq w$. Ebenso ist das leere Wort Teilwort eines jeden beliebigen Wortes!

Sei $\Sigma := \{L, R\}$, dann ist für $w := LLRLRLRLR$ gerade $|w| = 8$, $|w|_L = 5$, und $|w|_R = 3$.

⁸Hier wird bewusst „Wörter“ und *nicht* „Worte“ verwendet, denn es sind Zeichenketten und keine gesprochenen Worte oder bedeutungsvolle Begriffe!

Sei $D_1 := \{w \in \Sigma^* \mid |w|_L = |w|_R \text{ und } \forall u \in \Sigma^* : (u \sqsubseteq w) \rightarrow (|u|_L \geq |u|_R)\}$. Diese (formale) Sprache wird (*einseitige*) *Dyck-Sprache* über einem Klammepaar genannt, mit dem Symbol L als öffnender und R als schließender Klammer.

(D_1, \cdot) ist selbst ein Monoid, weil auch das leere Wort λ dazugehört und mit $u, v \in D_1$, stets auch $w := u \cdot v = uv \in D_1$ gilt, d.h. $D_1^* = D_1$ ist. Dieses Untermonoid kann jedoch niemals von einer endlichen Menge frei erzeugt werden! Den Beweis für diese Tatsache bleiben wir hier jedoch schuldig.

Betrachten wir $M := \{L, LR, RL\} \subsetneq \Sigma^*$ und bilden M^* , so ist auch M^* ein Untermonoid von Σ^* , jedoch ist dieses von M nicht frei erzeugt. Das Element $LRL \in M^*$ hat zwei verschiedene Zerlegungen in einzelne Erzeugende: $LRL = L \cdot RL = LR \cdot L$.

Neben der Konkatination, benutzen wir eine große Zahl weiterer Operationen auf Wörtern, von denen wir hier nicht alle sofort ohne ihren Zusammenhang zu möglichen „An- und Verwendungen“ definieren wollen. Zu den wichtigen gehören zum Beispiel die folgende, die wir als weiteres Beispiel einer rekursiven Definition angeben.

2.50 Definition

Sei Σ ein Alphabet, dann ist für jedes Wort $w \in \Sigma^*$ das **Spiegelwort** w^{rev} definiert durch:

$$(ux)^{\text{rev}} := xu^{\text{rev}} \quad \text{für} \quad x \in \Sigma, u \in \Sigma^*, \quad \text{und} \quad \lambda^{\text{rev}} := \lambda.$$

Für $L \subseteq \Sigma^*$ wird die Spiegelwortabbildung erweitert durch: $L^{\text{rev}} := \{w^{\text{rev}} \mid w \in L\}$.

2.51 Beispiel

- $\text{REGAL}^{\text{rev}} = \text{LAGER}$,
- $\text{MARKTKRAM}^{\text{rev}} = \text{MARKTKRAM}$ und
- $\text{RENTNER}^{\text{rev}} = \text{RENTNER}$.

Ein Wort, für das $w^{\text{rev}} = w$ gilt, nennt man **Palindrom**.

Aus der rekursiven Definition der Spiegelwörter ergeben sich auch hier wieder induktive Beweise. Für das folgende einfache Lemma soll der letzte ausführliche Beweis dieser Art vorgeführt werden.

2.52 Lemma

Für jedes Alphabet Σ gilt: $\forall u, v \in \Sigma^* : (uv)^{\text{rev}} = v^{\text{rev}}u^{\text{rev}}$.

Beweis: Wir verwenden vollständige Induktion über die Länge von v :

Induktions-Basis: $|v| = 0$, also ist $v = \lambda$ und $(uv)^{\text{rev}} = (u\lambda)^{\text{rev}} = u^{\text{rev}} = \lambda u^{\text{rev}} = \lambda^{\text{rev}} u^{\text{rev}} = v^{\text{rev}} u^{\text{rev}}$.

Induktions-Annahme: $|v| = n$, impliziert $(uv)^{\text{rev}} = v^{\text{rev}}u^{\text{rev}}$.

Induktions-Schritt: Sei $|v| = n + 1$, dann ist $v = wx$ für ein $x \in \Sigma$ und es gilt $w \in \Sigma^*$ mit $|w| = n$.

$(uv)^{\text{rev}} = (u(wx))^{\text{rev}}$	da $v = wx$
$= ((uw)x)^{\text{rev}}$	Konkatination ist assoziativ
$= x(uw)^{\text{rev}}$	Definition von $((uw)x)^{\text{rev}}$
$= x(w)^{\text{rev}}(u)^{\text{rev}}$	Induktionsannahme
$= (wx)^{\text{rev}}(u)^{\text{rev}}$	Definition von $(wx)^{\text{rev}}$
$= (v)^{\text{rev}}(u)^{\text{rev}}$	da $v = wx$.

□

Wir wollen nun einige Ordnungen auf Wortmengen definieren. Wir betrachten zunächst ein Beispiel:

2.53 Beispiel

Sei $w_1 \ll w_2$ für Wörter $w_1, w_2 \in \Sigma^*$ genau dann, wenn $|w_1| \leq |w_2|$ ist. Offensichtlich ist (Σ^*, \ll) keine totale Ordnung, denn Wörter gleicher Länge lassen sich mit \ll nicht vergleichen!

Um alle Wörter untereinander vergleichbar zu machen, kann man jedoch die sogenannte *lexikographische Ordnung* als Erweiterung der Ordnung \prec auf dem zugrundeliegenden Alphabe, also (Σ, \prec) , betrachten. Wir definieren diese Ordnung rekursiv:

2.54 Definition

Sei (Σ, \prec) ein mit \prec linear geordnetes Alphabet, dann ist die **lexikographische Erweiterung** $\prec^{\text{lex}} \subseteq \Sigma^* \times \Sigma^*$ von \prec definiert durch:

1. $\lambda \prec^{\text{lex}} w$ für alle $w \in \Sigma^*$.
2. $\forall x, y \in \Sigma \forall u, v \in \Sigma^+ : (xu \prec^{\text{lex}} yv)$ genau dann, wenn $\begin{cases} (x \prec y) \text{ oder} \\ (x = y) \text{ und } (u \prec^{\text{lex}} v) \end{cases}$

$\preceq^{\text{lex}} := \prec^{\text{lex}} \cup \text{Id}_{\Sigma^*}$ nennen wir **lexikographische Ordnung** auf Σ^* .

Gestützt auf diese rekursive Definition lassen sich nun einige Eigenschaften leicht induktiv beweisen.

2.55 Lemma

Sei (Σ, \prec) ein linear geordnetes Alphabet, dann gilt:

- a) $\forall v \in \Sigma^*, w \in \Sigma^+ : v \prec^{\text{lex}} vw$
- b) $\forall u, v \in \Sigma^* : (u \preceq^{\text{lex}} v)$ impliziert $(u \sqsubseteq v)$ oder $(\forall w_1, w_2 \in \Sigma^* : uw_1 \prec^{\text{lex}} vw_2)$

Beweis:

zu a): Wir beweisen (*): $v \prec^{\text{lex}} vw$ für $w \neq \lambda$ durch vollständige Induktion über die Länge des Präfixes v .

Induktions-Basis: Für $|v| = 0$ ist $v = \lambda$ und (*) gilt nach 1. der Definition.

Induktions-Annahme: Wir nehmen an, dass (*) für alle v mit $|v| = k$ gilt.

Induktions-Schritt: Jedes $v \in \Sigma^*$ mit $|v| = k + 1$ hat die Form $v = xu$ für ein $x \in \Sigma$. Nach 2. der Definition von \prec^{lex} gilt aber $xu \prec^{\text{lex}} xuw$ immer dann, wenn $u \prec^{\text{lex}} uw$ zutrifft. Dies gilt mit $|u| = k$ aber wegen der Induktionsannahme. Folglich gilt (*) nun für alle $v \in \Sigma^*$.

zu b): Nach a) ist die Implikation „ $u \preceq^{\text{lex}} v$ impliziert $u \sqsubseteq v$ “ richtig, wenn u tatsächlich ein Präfix von v ist. Sollte das nicht der Fall sein, so gibt es einen gemeinsamen (eventuell den trivialen: λ) Präfix $w \sqsubseteq u$ und $w \sqsubseteq v$ und zwei Symbole $x, y \in \Sigma$ mit $wx \sqsubset u$, $wy \sqsubset v$ und $x \prec y$, so dass $wx \prec^{\text{lex}} wy$. Entsprechend 3. der Definition kommt es dann nicht mehr darauf an, wie die echten Präfixe wx und wy von u und v weiter nach rechts verlängert werden, so dass $\forall w_1, w_2 \in \Sigma^* : uw_1 \prec^{\text{lex}} vw_2$ gilt. Die Implikation ist daher gültig.

□

2.56 Beispiel

Es gilt unter der üblichen linearen Ordnung der Symbole des lateinischen Alphabetes folgende lexikographische Anordnung:

$$AB \prec^{\text{lex}} FBI \prec^{\text{lex}} KLAUSUR \prec^{\text{lex}} OB \prec^{\text{lex}} PI \prec^{\text{lex}} UNI \prec^{\text{lex}} ZOO$$

In der deutschen Sprache ist die dem Alphabet unterliegende Ordnung \prec bezüglich der Umlaute nicht eindeutig definiert: Im Duden wird „ä“ wie „a“ behandelt, während es im Telefonbuch wie „ae“ einsortiert wird. In keinem Fall sind die Umlaute in die totale Ordnung \prec des Alphabets integriert!

Die lexikographische Striktordnung \prec^{lex} ist total, und besitzt kein Supremum und keine maximalen Elemente! Die Kette $a \prec^{\text{lex}} aa \prec^{\text{lex}} aaa \prec^{\text{lex}} \dots$ kann unendlich fortgesetzt werden und jedes Element der ebenfalls unendlichen Kette $b \prec^{\text{lex}} bb \prec^{\text{lex}} bbb \prec^{\text{lex}} \dots$ ist bezüglich \prec^{lex} größer als jedes beliebige Element a^i der ersten unendlichen, aufsteigenden Kette. Die in Definition 2.54 eingeführte Relation \ll eignet sich nicht als Grundlage für eine Abzählung aller Wörter, sofern das zugrundeliegende Alphabet mindestens zwei Symbole enthält. Dann ist es nämlich nicht möglich, ein Verfahren, d.h. eine Abbildung $g: \mathbb{N} \rightarrow \Sigma^*$, anzugeben, mit dem die Wörter aus Σ^* , mit $|\Sigma| \geq 2$, so abgezählt werden, dass stets $g(i) \prec^{\text{lex}} g(i+1)$ gilt! Um das zu erreichen wird in der Regel eine andere Ordnung auf Σ^* definiert, die wir **lexikalisch** nennen. In ihr werden die Wörter zuerst nach ihrer Länge verglichen (kürzere Wörter sind bzgl. der lexikalischen Ordnung kleiner als längere) und nur Wörter gleicher Länge werden noch lexikographisch geordnet.

2.57 Definition

Sei (Σ, \prec) ein mit \prec linear geordnetes Alphabet, dann ist die **lexikalische Erweiterung** $\prec^{\text{lg-lex}}$ von \prec definiert durch:

$w_1 \prec^{\text{lg-lex}} w_2$ gelte für beliebige Wörter $w_1, w_2 \in \Sigma^*$ genau dann, wenn

$$\text{entweder: } |w_1| < |w_2|, \quad \text{oder: } |w_1| = |w_2| \quad \wedge \quad w_1 \prec^{\text{lex}} w_2.$$

$\preceq^{\text{lg-lex}} := \prec^{\text{lg-lex}} \cup \text{Id}_{\Sigma^*}$ heißt **lexikalische Ordnung** auf Σ^* .

2.58 Beispiel

Es gilt unter der üblichen linearen Ordnung der Symbole des lateinischen Alphabetes folgende lexikalische Anordnung:

$$AB \prec^{\text{lg-lex}} OB \prec^{\text{lg-lex}} PI \prec^{\text{lg-lex}} FBI \prec^{\text{lg-lex}} UNI \prec^{\text{lg-lex}} ZOO \prec^{\text{lg-lex}} KLAUSUR$$

Mit dieser Definition ist es nun ein Leichtes, ein Verfahren zur sukzessiven Erzeugung aller Wörter von Σ^* für ein beliebiges aber festes Alphabet Σ zu definieren: In Schritt (1) beginnt man mit $k := 0$, notiert das kleinste Wort λ und geht zu Schritt (2). In Schritt (2) erhöht man k um eins und geht zu Schritt (3). In Schritt (3) notiert man die $|\Sigma|^k$ unterschiedlichen Wörter der Länge k in lexikographisch aufsteigender Reihenfolge. Dann fährt man wieder mit Schritt (2) fort.

Weniger leicht ist es vielleicht, eine bijektive Abzählfunktion zu formulieren, die auf Grund der Existenz des eindeutigen Erzeugungsverfahrens ja existieren muss: Zur gegebenen Zahl $i \in \mathbb{N}$ läßt man das angegebene Verfahren (beginnend mit dem minimalen Wort bezüglich $\prec^{\text{lg-lex}}$) solange laufen, bis das i -te Wort erzeugt wurde. Den eindeutig bestimmten Index i eines beliebigen Wortes kann man umgekehrt

2. Notationen und Grundbegriffe

mit dem Verfahren der Abzählung ebenfalls gewinnen: Man erzeugt sukzessive die Wörter mit dem Verfahren solange, bis das vorliegende Wort erreicht wird. Die Anzahl der erzeugten Wörter muss dabei nur mitgezählt werden, um den Index zu ermitteln. Wir werden später noch ein besseres Verfahren kennenlernen.

Verallgemeinerung auf Tupel von reellen Zahlen Oftmals benötigt man Abzählungen und Ordnungen auf m -Tupeln (Vektoren) aus \mathbb{R}^m . Ähnlich zur lexikographischen (und auch zur lexikalischen) Ordnung auf Σ^* definiert man Ordnungen $<^{\text{lex}} \subseteq \mathbb{R}^m$ (bzw. $<^{\text{lg-lex}} \subseteq \mathbb{R}^m$) für Tupel von reellen Zahlen:

2.59 Definition

Sei $m \in \mathbb{Z}^+$ dann ist die **lexikographische Erweiterung** $<^{\text{lex}} \subseteq \mathbb{R}^m$ von $< \subseteq \mathbb{R}$ definiert durch:

$v_1 <^{\text{lex}} v_2$ gelte für beliebige $v_1, v_2 \in \mathbb{R}^m$ genau dann, wenn

$$\exists k \in \{1, \dots, m\} : (\forall i : (1 \leq i < k) \rightarrow (v_1(i) = v_2(i)) \wedge (v_1(k) < v_2(k))).$$

$\leq^{\text{lex}} := <^{\text{lex}} \cup \text{Id}_{\mathbb{R}^m}$ nennen wir **lexikographische Ordnung** auf \mathbb{R}^m .

Darauf aufbauend auch für Vektoren die entsprechende lexikalische Variante:

2.60 Definition

Sei $m \in \mathbb{Z}^+$ dann ist die **lexikalische Erweiterung** $<^{\text{lg-lex}} \subseteq \mathbb{R}^m$ von $< \subseteq \mathbb{R}$ definiert durch:

$v_1 <^{\text{lg-lex}} v_2$ gelte für beliebige $v_1, v_2 \in \mathbb{R}^m$ genau dann, wenn

$$\text{entweder: } \sum_{i=1}^m v_1(i) < \sum_{i=1}^m v_2(i) \quad \text{oder: } \left(\sum_{i=1}^m v_1(i) = \sum_{i=1}^m v_2(i) \right) \wedge (v_1 <^{\text{lex}} v_2).$$

$\leq^{\text{lg-lex}} := <^{\text{lg-lex}} \cup \text{Id}_{\mathbb{R}^m}$ nennen wir **lexikalische Ordnung** auf \mathbb{R}^m .

In beiden Fällen, wie sonst auch üblich, gilt Gleichheit von m -Tupeln nur bei komponentenweiser Gleichheit! Auch sonst ist es gebräuchlich, eine auf \mathbb{R} definierte Relationen $\ll \subseteq \mathbb{R}$ so auf \mathbb{R}^m zu übertragen, dass $v_1 \ll v_2$ für beliebige $v_1, v_2 \in \mathbb{R}^m$ genau dann gilt, wenn $v_1(i) \ll v_2(i)$ für jede Komponente $i \in \{1, \dots, m\}$ zutrifft. Damit wird jede partielle Ordnung auf $\ll \subseteq \mathbb{R}$ zu einer partiellen Ordnung auf \mathbb{R}^m . Falls $\ll \subseteq \mathbb{R}$ totale Ordnung ist, so ist deren komponentenweise Erweiterung auf \mathbb{R}^m in der Regel nur noch partiell und ausschließlich im Falle $m = 1$ (trivialer Weise) ebenfalls total.

Für die Wörter der Form $a^i b^j$ gilt bei $a \prec b$ die Beziehung $a^i b^j \prec^{\text{lg-lex}} a^r b^s$ genau dann, wenn $(r, s) <^{\text{lg-lex}} (i, j)$ zutrifft.

Die Tupel $(i, j) \in \mathbb{N}^2$ in der zweidimensionalen, unendlichen Aufstellung nach Tabelle 2.2 werden, wie in Tabelle 2.3 angedeutet, entsprechend der Ordnung $<^{\text{lg-lex}}$ abgezählt.

2.7. Paar- und Tupelfunktionen

Häufig kommt es vor, dass man eine Abzählung oder Nummerierung einer (unendlichen) Objektmenge M benötigt, z.B. um eine systematische Betrachtung *aller* Objekte zu ermöglichen.⁹ **Paarfunktionen**

⁹Eine Nummerierung ist immer eine surjektive Abbildung der Form $\mathbb{N} \rightarrow M$.

Tabelle 2.2.: Tabelle der Tupel aus $\mathbb{N} \times \mathbb{N}$

i \ j	0	1	2	3	4	5	6	7	8	...
0	(0, 0)	(0, 1)	(0, 2)	(0, 3)	(0, 4)	(0, 5)	(0, 6)	(0, 7)	(0, 8)	...
1	(1, 0)	(1, 1)	(1, 2)	(1, 3)	(1, 4)	(1, 5)	(1, 6)	(1, 7)	(1, 8)	...
2	(2, 0)	(2, 1)	(2, 2)	(2, 3)	(2, 4)	(2, 5)	(2, 6)	(2, 7)	(2, 8)	...
3	(3, 0)	(3, 1)	(3, 2)	(3, 3)	(3, 4)	(3, 5)	(3, 6)	(3, 7)	(3, 8)	...
4	(4, 0)	(4, 1)	(4, 2)	(4, 3)	(4, 4)	(4, 5)	(4, 6)	(4, 7)	(4, 8)	...
5	(5, 0)	(5, 1)	(5, 2)	(5, 3)	(5, 4)	(5, 5)	(5, 6)	(5, 7)	(5, 8)	...
6	(6, 0)	(6, 1)	(6, 2)	(6, 3)	(6, 4)	(6, 5)	(6, 6)	(6, 7)	(6, 8)	...
...

sind Spezialfälle von Nummerierungen, nämlich Nummerierungen von Paaren aus \mathbb{N}^2 . Sie können auf einfache Weise durch geeignete Schachtelung zu sogenannten **Tupelfunktionen** erweitert werden. Somit brauchen wir anstelle aller Funktionen mit Vorbereich \mathbb{N}^m und Nachbereich \mathbb{N}^n nur noch einfache Zahlenfunktionen der Art $f : \mathbb{N} \rightarrow \mathbb{N}$ zu betrachten, da alle anderen Funktionen durch Komposition mit entsprechenden Tupelfunktionen konstruiert werden können.¹⁰

Natürlich ist es leicht, ein Verfahren zum sukzessiven Erzeugen des jeweils nächst größeren Tupels in \mathbb{N}^2 aufzuschreiben. Weit schwieriger ist es jedoch, eine wirkliche Bijektion zwischen \mathbb{N} und \mathbb{N}^2 mit geschlossenen Formeln anzugeben.

Auf den Begründer der Mengenlehre, Georg Cantor (1845 – 1918), geht das im Folgenden beschriebene Verfahren mit der Cantorsche Paarfunktion $\text{pair} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ zurück. Sie erzeugt gerade die Abzählung aus Tabelle 2.3. Diese Funktion ist jedoch nicht die einzige Art, eine Bijektion zwischen den Mengen \mathbb{N}^2 und \mathbb{N} anzugeben. Im Allgemeinen ist eine Funktion zu finden, welche ähnliche Eigenschaften, wie die spezielle Paarfunktion pair hat. Die benötigten Eigenschaften werden in der folgenden Definition festgelegt.

2.61 Definition

Eine Funktion $\text{pair} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, nennt man **Paarfunktion** (*pairing function*), wenn:

1. pair surjektiv ist und $\text{pair}(i, j)$ effektiv bestimmbar ist, und
2. zwei gleichfalls berechenbare Funktionen $\pi_1 : \mathbb{N} \rightarrow \mathbb{N}$ und $\pi_2 : \mathbb{N} \rightarrow \mathbb{N}$ existieren, für die:
 $\pi_1(\text{pair}(i, j)) = i$, $\pi_2(\text{pair}(i, j)) = j$ und $\text{pair}(\pi_1(k), \pi_2(k)) = k$ gilt.

Betrachten wir die Ordnung $<^{\text{lg-lex}} \subseteq \mathbb{N}^2$ und die Tupel aus \mathbb{N}^2 in aufsteigender Reihenfolge bezüglich $<^{\text{lg-lex}}$. Wie in Abbildung 2.3 werden wir an der Stelle (i, j) , also am Schnittpunkt von i -ter Zeile und j -ter Spalte der nach unten offenen Anordnung, genau die Zahl $\text{pair}(i, j)$ eintragen, die bei Abzählung der Tupel (beginnend bei 0 (*Null*) für das Tupel $(0, 0)$) entsprechend dieser Ordnung entsteht.

Für das k -te Tupel (i, j) gilt die Gleichung:

¹⁰Die hierzu notwendigen Eigenschaften von Tupelfunktionen, wie die Berechenbarkeit, werden wir hier einfach voraussetzen. Sie werden Gegenstand der Vorlesung F3 sein.

2. Notationen und Grundbegriffe

Tabelle 2.3.: Abzählung der Tupel aus $\mathbb{N} \times \mathbb{N}$ entsprechend $<^{\text{lg-lex}}$:

j i	0	1	2	3	4	5	6	7	...
0	0	1	3	6	10	15	21	28	...
1	2	4	7	11	16	22	29	...	
2	5	8	12	17	23	30	...		
3	9	13	18	24	31	...			
4	14	19	25	32	...				
5	20	26	33	...					
6	27	34	...						
7	35	...							
⋮	...								

$$k = \sum_{r=0}^{i+j} r$$

Die Gültigkeit dieser Gleichung kann folgendermaßen gezeigt werden:

Bezeichnet man die Nebendiagonale (*counter diagonal*) in der $k := \text{paar}(i, j)$ steht mit $cd(k)$, so gilt offensichtlich $cd(\text{paar}(i, j)) = i + j$. In der Nebendiagonalen mit der Nummer n findet man stets genau $n + 1$ Elemente, von denen jedes genau um 1 größer ist als das Vorangegangene. Daher ist $\text{paar}(0, j) = \sum_{r=0}^j r$, und für dieses und die weiteren Elemente $0 \leq t \leq i$ derselben Nebendiagonalen j gilt dann:

$\text{paar}(t, j - t) = \sum_{r=0}^j r + t$. Also ist die Funktion $\text{paar} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, definiert durch

$$\text{paar}(i, j) := \frac{(i + j)(i + j + 1)}{2} + i$$

ein Kandidat für eine Paarfunktion, wenn wir auch noch ihre Umkehrung mit Hilfe der Funktionen π_1 und π_2 angeben können.

Hierbei ist natürlich eine geschlossene Form erwünscht und nicht nur ein rekursives Bestimmungs- oder iteratives Erzeugungsverfahren (vergl. [Gruska])! Wie schon bemerkt, ist die Komponentensumme $\pi_1(\text{paar}(i, j)) + \pi_2(\text{paar}(i, j)) = i + j$ auf den Nebendiagonalen stets konstant so groß wie der maximale Zeilen- oder Spaltenindex dieser Nebendiagonalen. Wenn nun eine Zahl $z := \text{paar}(i, j)$ gegeben ist, aber weder die Nummer $cd(z)$ der Nebendiagonalen zu z , noch die Größen i oder j , so wäre $\text{paar}(0, i + j)$ die kleinste Zahl in dieser Nebendiagonalen $cd(z)$. Es ist dies die Zahl

$$k := \text{paar}(0, i + j) = \frac{(i + j)(i + j + 1)}{2} = \frac{cd(k)(cd(k) + 1)}{2}.$$

Durch Auflösen nach $cd(k)$ erhält man:

$$\text{paar}(0, i + j) = \frac{cd(k)(cd(k) + 1)}{2},$$

$$\begin{aligned}
2 \cdot \text{paar}(0, i+j) &= cd(k)(cd(k)+1), \\
2 \cdot \text{paar}(0, i+j) + \frac{1}{4} &= (cd(k))^2 + cd(k) + \frac{1}{4} \\
&= (cd(k) + \frac{1}{2})^2, \\
\sqrt{2 \cdot \text{paar}(0, i+j) + \frac{1}{4}} - \frac{1}{2} &= cd(k).
\end{aligned}$$

Es ist nun die Abbildung paar monoton, d.h.:

$$(i_1, j_1) <^{\text{lg-lex}} (i_2, j_2) \quad \text{impliziert} \quad \text{paar}(i_1, j_1) < \text{paar}(i_2, j_2).$$

Ebenfalls monoton ist die Abbildung, die $k \in \mathbb{N}$ den Wert $\sqrt{2k + \frac{1}{4}} - \frac{1}{2}$ zuordnet. Man rechnet leicht nach, dass

$$\text{paar}(i+j, 0) + 1 = \text{paar}(0, i+j+1)$$

die kleinste Zahl auf der nächst höheren Nebendiagonale ist, so dass (für $i, j \neq 0$):

$$\begin{aligned}
cd(\text{paar}(0, i+j)) &= \sqrt{2 \cdot \text{paar}(0, i+j) + \frac{1}{4}} - \frac{1}{2} \\
&< \sqrt{2 \cdot \text{paar}(i, j) + \frac{1}{4}} - \frac{1}{2} \\
&< \sqrt{2 \cdot \text{paar}(i+j, 0) + \frac{1}{4}} - \frac{1}{2} \\
&< \sqrt{2 \cdot (\text{paar}(i+j, 0) + 1) + \frac{1}{4}} - \frac{1}{2} \\
&= \sqrt{2 \cdot \text{paar}(0, i+j+1) + \frac{1}{4}} - \frac{1}{2} \\
&= cd(\text{paar}(0, i+j+1)) \\
&= cd(\text{paar}(i, j)) + 1
\end{aligned}$$

Daher ergibt sich eine geschlossene Formel zur Berechnung der Nebendiagonalen in der sich eine gegebene Zahl k befindet:

$$cd(k) = \left\lfloor \sqrt{2k + \frac{1}{4}} - \frac{1}{2} \right\rfloor.$$

Daraus kann über $k = \text{paar}(\pi_1(k), \pi_2(k)) = \frac{cd(k)(cd(k)+1)}{2} + \pi_1(k)$ zunächst $\pi_1(k)$, und aus $cd(k) = \pi_1(k) + \pi_2(k)$ sofort $\pi_2(k)$ ermittelt werden:

$$\begin{aligned}
\pi_1(k) &= k - \frac{cd(k)(cd(k)+1)}{2} \\
&= k - \frac{1}{2} \cdot \left\lfloor \sqrt{2 \cdot \text{paar}(i, j) + \frac{1}{4}} - \frac{1}{2} \right\rfloor \cdot \left\lfloor \sqrt{2 \cdot \text{paar}(i, j) + \frac{1}{4}} + \frac{1}{2} \right\rfloor \\
\pi_2(k) &= cd(k) - \pi_1(k)
\end{aligned}$$

2. Notationen und Grundbegriffe

Mit einer Paarfunktion $\text{pair} : \mathbb{N}^2 \rightarrow \mathbb{N}$ kann man auch **Tupelfunktionen** $\pi^{(k)}$ für höhere Dimensionen $k \in \mathbb{N}$ erklären:

2.62 Definition

Die **k -Tupelfunktion** $\pi^{(k)} : \mathbb{N}^k \rightarrow \mathbb{N}$ für $k > 2$ wird zurückgeführt auf die übliche Paarfunktion pair durch:

$$\pi^{(k)}(x_1, x_2, \dots, x_k) := \text{pair}(x_1, \pi^{(k-1)}(x_2, \dots, x_k)) = \underbrace{\text{pair}(x_1, \dots, \text{pair}(x_{k-1}, x_k))}_{(k-1)\text{-mal}} \dots$$

Die zu einer k -Tupelfunktion zugehörigen Projektionen auf die einzelnen Komponenten, die sogenannten **k -Komponentenfunktionen**, werden mit $\pi_i^{(k)}$ bezeichnet und sind für $i \in \{1, \dots, k\}$ ebenfalls bestimmbar.

2.63 Beispiel

Sei $\vec{x} := (2, 5, 7, 3)$ ein Quadrupel (4-Tupel) aus \mathbb{N}^4 . Mit $\pi^{(4)}(\vec{x})$ wird die natürliche Zahl

$$z := \text{pair}(2, \text{pair}(5, \text{pair}(7, 3)))$$

bezeichnet. Die dritte Komponente des ursprünglichen Tupels lässt sich nun mit der Komponentenfunktion $\pi_3^{(4)}$ zurückgewinnen:

$$\pi_3^{(4)}(z) = \pi_3^{(4)}(\pi^{(4)}(2, 5, 7, 3)) = 7$$

2.8. Nummerierungen von Wortmengen

Auf Kurt Gödel geht die Idee zurück, beliebige Zeichenketten durch natürliche Zahlen zu kodieren. Die Nummerierungen allein der Buchstaben, wie zum Beispiel beim ASCII-Code, geben ja noch keine Vorschrift zum Abzählen aller Wörter.

Gödelisierung Solch eine Zuordnung von Zahlen zu Wörtern nützt in vielen Fällen nur dann etwas, wenn das Ursprungsobjekt der Kodierung aus dieser auch wieder hergestellt werden kann. Daher wird die Gödelnummerierung der Wortmenge über einem Alphabet als injektive Funktion definiert.

Wir verwenden für diese Nummerierungen das griechische Symbol ν (sprich: nü) mit einem Index, der die jeweils gemeinte Nummerierung bezeichnet.

2.64 Definition

Sei Σ ein endliches Alphabet. Eine Funktion $\nu_\Sigma : \Sigma^* \rightarrow \mathbb{N}$ heißt **Gödelisierung**, wenn gilt:

1. $\forall u, v \in \Sigma^* : u \neq v$ impliziert $\nu_\Sigma(u) \neq \nu_\Sigma(v)$, ν_Σ ist also injektiv.
2. Aus jedem Argument $w \in \Sigma^*$ kann $\nu_\Sigma(w)$ in endlich vielen Schritten effektiv bestimmt werden.
3. Für jedes $n \in \mathbb{N}$ kann in endlich vielen Schritten effektiv bestimmt werden, ob es ein $w \in \Sigma^*$ gibt, für das $\nu_\Sigma(w) = n$ gilt.
4. Wenn es ein $w \in \Sigma^*$ mit $\nu_\Sigma(w) = n$ gibt, dann kann w effektiv aus n konstruiert werden.

Es ist also bei Gödelisierungen keine Bijektion zwischen Σ^* und \mathbb{N} gefordert! Es gibt verschiedene Möglichkeiten spezielle Gödelisierungen anzugeben, von denen wir schon die Cantorsche Paarfunktion kennengelernt haben.

Die Paarfunktionen pair , paar und $\pi^{(k)}$ sind Gödelisierungen nur für Zahlentupel über \mathbb{N} mit fester Dimension. Wenn wir allerdings endliche Zahlenfolgen oder Zeichenketten variabler Länge kodieren möchten, so taugen diese Abbildungen herzlich wenig.

Kurt Gödel verwendete als erster die Kodierung über die eindeutige Zahlenrepräsentation mit Hilfe von Primzahlen. Diese Kodierung $\nu_{\mathbb{N}-seq}$ ordnet zunächst jeder endlichen Zahlenfolge

$$\alpha_1, \alpha_2, \dots, \alpha_m \text{ mit } \alpha_i \in \mathbb{N}$$

ihre sogenannte Gödelnummer

$$\nu_{\mathbb{N}-seq}(\alpha_1, \alpha_2, \dots, \alpha_m) := \prod_{i=1}^m p_i^{\alpha_i + [i=m]} - 1$$

zu, wobei p_i die i -te Primzahl der Folge 2, 3, 5, 7, 11, ... bezeichnet.

Da $\alpha_i = 0$ möglich sein soll, wird nur der Exponent α_m der letzten und höchsten Primzahl p_m in dem Produkt um 1 erhöht! Hier macht sich die Iverson'sche Notation wieder nützlich. Damit alle Sequenzen über \mathbb{N} als Zahlen aus $\mathbb{N} \setminus 0$ kodiert werden können, musste am Ende noch 1 abgezogen werden. Dadurch ist die leere Folge ohne ein einziges Element als 0 kodiert! Als Beispiel eine Liste der ersten Werte für kurze Folgen u über \mathbb{N} :

Folge u	$\nu_{\Sigma}(u)$
0	1
0,0	2
1	3
0,0,0	4
1,0	5
0,0,0,0	6
2	7
0,1	8
0,0,1	9
...	...

Einem Wort $w = x_{i_1} x_{i_2} \dots x_{i_m}$ über dem Alphabet $\Sigma := \{x_0, x_1, \dots, x_{n-1}\}$ wird dann die eindeutig bestimmte, natürliche Zahl $\nu_{prim}(w) = \nu_{\mathbb{N}-seq}(i_1, i_2, \dots, i_m)$ zugeordnet. Dies führt zu einer ersten Gödelisierung von Zeichenketten:

2.65 Definition

Sei $\Sigma := \{x_0, x_1, \dots, x_{n-1}\}$ ein Alphabet, dann ist $\nu_{prim} : \Sigma^* \rightarrow \mathbb{N}$ definiert durch:

1. $\nu_{prim}(\lambda) := 0$,
2. $\nu_{prim}(w) := \prod_{j=1}^m p_j^{i_j + [j=m]} - 1$ für Wörter der Form $w = x_{i_1} x_{i_2} \dots x_{i_m}$ mit $x_{i_j} \in \Sigma$.

2. Notationen und Grundbegriffe

Aus $n \in \mathbb{N}$ erhält man $\nu_{prim}^{-1}(n)$ durch fortgesetztes Dividieren.

Mit $\Sigma := \{x_0, x_1, x_2\}$ erhalten wir so $\nu_{prim}(x_2x_1x_0x_1) = 2^2 \cdot 3^1 \cdot 5^0 \cdot 7^2 - 1 = 587$, $\nu_{prim}(x_1x_2x_0x_0) = 2^1 \cdot 3^2 \cdot 5^0 \cdot 7^1 - 1 = 125$, $\nu_{prim}(x_0) = 2^1 - 1 = 1$ und $\nu_{prim}(x_0x_0) = 2^0 \cdot 3^1 - 1 = 2$.

Da jede natürliche Zahl eine eindeutige Zerlegung als Produkt von Primzahlen besitzt, hat auch jedes $n \in \mathbb{N}$ ein Urbild unter der Umkehrfunktion ν_{prim}^{-1} . Die Gödelisierung ν_{prim} ist also eine Bijektion zwischen Σ^* und \mathbb{N} .

Als weitere Möglichkeit kann man aus der lexikalischen Ordnung der Zeichenketten ebenfalls eine eindeutige Gödelnummer $\nu_{lg-lex}(w)$ definieren.

2.66 Definition

Für das geordnete Alphabet (Σ, \prec) , $\Sigma := \{x_1, x_2, \dots, x_n\}$ mit $x_1 \prec x_2 \prec \dots \prec x_n$ sei $\nu_{lg-lex} : \Sigma^* \rightarrow \mathbb{N}$ rekursiv definiert durch

1. $\nu_{lg-lex}(wx_i) := n \cdot \nu_{lg-lex}(w) + i$, falls $w \in \Sigma^*$, $x_i \in \Sigma$,
2. $\nu_{lg-lex}(\lambda) := 0$.

Man zeigt leicht durch Induktion:

2.67 Lemma

Für $\Sigma := \{x_1, x_2, \dots, x_n\}$ und $x_{i_j} \in \Sigma$ gilt:

$$\nu_{lg-lex}(x_{i_m}x_{i_{m-1}} \cdots x_{i_0}) = \sum_{j=0}^m i_j \cdot n^j.$$

2.68 Beispiel

Für $\Sigma := \{a, b, c\}$ mit $a \prec b \prec c$ ist mit dieser Kodierung dann $\nu_{lg-lex}(bca) = 28$.

Die Gödelisierung ν_{lg-lex} ist tatsächlich ebenfalls eine Bijektion zwischen Σ^* und \mathbb{N} . Die Umkehrung kann am einfachsten durch einen Algorithmus angegeben werden, der für ein geordnetes Alphabet (Σ, \prec) bei Eingabe einer beliebigen Zahl $n \in \mathbb{N}$ dasjenige Wort $w \in \Sigma^*$ erzeugt, welches in der Abzählung als n -tes vorkommt. Tatsächlich interessiert nur die Reihenfolge der Indizes der Symbole im Alphabet $\Sigma := \{x_1, x_2, \dots, x_b\}$, die als b -adische Zahl (siehe Definition 2.71) aufgefaßt werden kann.

Stellensysteme für die Zahlendarstellung Man findet einen verwandten Algorithmus bei den Definitionen der Stellensysteme für natürliche Zahlen! Dieser ist in Algorithmus 2.70 dargestellt. Auf den zwei wichtigsten Notationen natürlicher Zahlen, den b -nären (b -ary) oder den b -adischen (b -adic) Darstellungen zur Basis (*base*, *radix*) b beruhen viele Kodierungen. Sie sind auch wegen der technischen Gegebenheiten bei realen Computern und elektronischen Schaltungen von besonderer Bedeutung.

2.69 Definition

Die **b -näre Darstellung** benutzt die Symbole der Ziffernmenge $B := \{0, 1, \dots, b-1\}$. Ist $b = 2$, so spricht man von einer **binären** Zahlendarstellung. Eine Zahl $n \in \mathbb{N}$ wird zur Basis $b = |B|$ durch eine Folge $a_k a_{k-1} \cdots a_0$ von Symbolen $a_i \in \{0, 1, \dots, b-1\}$ notiert, wenn $n = \sum_{i=0}^k a_i \cdot b^i$ gilt. Wir notieren dies wie üblich durch $[a_k a_{k-1} \cdots a_0]_b = n$.

Wie man aus einer Binärzahl $w \in \{0,1\}^*$ die Zahl $[w]_2 \in \mathbb{N}$ gewinnt ist nach der Definition natürlich einfach. Umgekehrt gewinnt man die Binärdarstellung einer Zahl $n \in \mathbb{N}$ durch fortgesetztes Dividieren durch 2 und Notieren der Reste, wie im Algorithmus 2.70 für die b -nären Darstellungen allgemein dargestellt.

2.70 Algorithmus (b -näre Darstellung einer Zahl $n \in \mathbb{N}$)

Sei $B := \{0, 1, \dots, b-1\}$ ein Alphabet von b Ziffern und $n \in \mathbb{N}$ die Eingabe für den Algorithmus.

```

begin
  repeat          (* wiederhole alles was vor until steht,
                    bis die Bedingung dahinter erfüllt ist *)
     $m := \lfloor \frac{n}{b} \rfloor$ ;  $r := n - m \cdot b$ ;
    schreibe  $r$  links neben eventuell schon Geschriebenes;
     $n := m$ 
  until
     $n := 0$ 
  end (* repeat *)
end. (* Algorithmus *)

```

2.71 Definition

Die **b -adische Darstellung** benutzt die Symbole der Ziffernmenge $B := \{1, 2, \dots, b\}$. Ist $b = 2$, so spricht man von einer **dyadischen** Zahlendarstellung. Eine natürliche Zahl $n \in \mathbb{N}, n \neq 0$ wird zur Basis $b = |B|$ durch eine Folge $a_k a_{k-1} \dots a_0$ von Symbolen $a_i \in \{1, \dots, b\}$ notiert, wenn $n = \sum_{i=0}^k a_i \cdot b^i$ gilt. Wir notieren dann $[a_k a_{k-1} \dots a_0]_{b\text{-adic}} = n$.

Von einer **monadischen** Darstellung spricht man, wenn $B = \{1\}$ ist. Statt der Ziffer 1 wird oft jedes beliebige Symbol z , nicht notwendig eine Ziffer, benutzt. Mit $b = 1$ wird dann eine natürliche Zahl $n \in \mathbb{Z}^+$ durch die Zeichenkette $1^n = \underbrace{11 \dots 1}_n$ dargestellt.

Will man die Null anders als mit dem leeren Wort λ darstellen, so vereinbart man in der Regel, dass $n \in \mathbb{N}$ durch $0^{n+1} = \underbrace{00 \dots 0}_{n+1}$ dargestellt wird. Das verwandte Alphabet ist dann $B = \{0\}$.
Wir bezeichnen dies als **modifizierte unäre** Darstellung.

Für das Alphabet $\Sigma := \{x_1, x_2, \dots, x_b\}$ könnte man eine b -adische Darstellung benutzen, und einem Wort $w = x_{i_1} x_{i_2} \dots x_{i_m}$ mit $x_{i_j} \in \Sigma$ die Zahl $\nu_{b\text{-adic}}(w) := \sum_{k=0}^{m-1} (i_{m-k}) b^k$ zuordnen. Mit dieser Definition hätte man nun eine eindeutige Gödelnummer für jedes Wort über Σ^* definiert, denn es gibt keine führenden Nullen in dieser Darstellung. Die b -adischen Darstellungen sind, anders als die b -nären, stets eindeutig! Sieht man etwas genauer hin, so stellt sich heraus, dass $\nu_{b\text{-adic}}(w) = \nu_{lg\text{-lex}}(w)$ ist, und erkennt die lexikalische Ordnung über Σ^* als Anordnung hinsichtlich der (durch die Hintereinanderschreibung der Indizes der in $w \in \Sigma^*$ verwendeten Symbole aus Σ) b -adisch repräsentierten Zahlen!

Es ergibt sich so für die natürlichen Zahlen bis zur acht die folgende Liste binärer und dyadischer Zahlendarstellungen, wobei die Binärzahlen stets ohne führende Nullen geschrieben wurden:

2. Notationen und Grundbegriffe

b -när	$n \in \mathbb{N}$	b -adisch
0	0	undef.
1	1	1
10	2	2
11	3	11
100	4	12
101	5	21
110	6	22
111	7	111
1000	8	112
\vdots	\vdots	\vdots

Mit Algorithmus 2.72 lassen sich die b -adischen Darstellungen von natürlichen Zahlen $n \in \mathbb{N}$ gewinnen, und damit auch die Wörter $w \in \Sigma^*$, die in der lexikalischen Ordnung $\prec^{\text{lg-lex}}$ zu einem geordneten Alphabet $(\{x_1, x_2, \dots, x_b\}, \prec)$ an n -ter Stelle stehen. Dieses Verfahren ähnelt stark dem Algorithmus 2.70 zur Umwandlung einer Zahl in ihre b -näre Darstellung.

2.72 Algorithmus (b-adische Darstellung einer natürlichen Zahl)

Sei $B := \{1, 2, \dots, b\}$ ein Alphabet von Ziffern und $n \in \mathbb{N}$ mit $n \neq 0$ die Eingabe für den Algorithmus.

```

begin
  repeat
     $m := \lfloor \frac{n}{b} \rfloor$ ;  $r := n - m \cdot b$ ;
    if  $r \neq 0$ 
      then
        begin
          schreibe  $r$  links neben eventuell schon Geschriebenes
           $n := m$ 
        end
      else
        begin
          schreibe  $b$  links neben eventuell schon Geschriebenes;
           $n := m - 1$ 
        end
      end (* if *)
    until
       $n := 0$ 
    end (* repeat *)
  end. (* Algorithmus *)

```

Würden wir das gleiche Prinzip für $w = x_{i_1}x_{i_2} \dots x_{i_m}$ mit $x_{i_j} \in \Sigma := \{0, 1, \dots, b-1\}$ die b -näre Darstellung für eine ähnlich definierte Funktion $g_{b\text{-ary}} : \Sigma^* \rightarrow \mathbb{N}$ verwenden, so gäbe es zwar zu jeder Zahl $n \in \mathbb{N}$ ein Wort $w \in \Sigma^*$ mit $g_{b\text{-ary}}(w) = n$, aber wegen der führenden Nullen wäre dieses nicht

eindeutig bestimmt. Die Abbildung $g_{b-ary} : \Sigma^* \rightarrow \mathbb{N}$ wäre zwar total aber nicht injektiv und wäre somit keine Gödelisierung. Man müsste in diesem Falle dann noch die führenden Nullen ausschließen und könnte erst so eine Bijektion zwischen $B \setminus \{0\} \cdot B^*$ und \mathbb{N} erklären. Hierbei ist es jedoch von Nachteil, dass nicht alle Wörter über dem Alphabet Σ auch verwendet werden dürfen.

In der Regel werden wir bei unseren Anwendungen die genaue Definition einer Gödelisierung selten wirklich benötigen. Es reicht in den meisten Fällen zu wissen, dass eine solche existieren muss. Daher werden wir davon abstrahieren und die gewählte Gödelisierung einheitlich bezeichnen.

2.73 Definition

Gödelisierungen werden von nun an immer als bijektive Abbildungen angesehen und – sofern das Alphabet Σ unmissverständlich festgelegt ist – durch $\langle \rangle : \Sigma^ \rightarrow \mathbb{N}$ notiert.*

2.74 Beispiel

Mit $\langle abbab \rangle$ wird diejenige natürliche Zahl dargestellt, die gemäß der beliebig gewählten, aber eindeutig festgesetzten Gödelisierung, die Gödelnummer von $abbab$ ist.

In den späteren Fällen wird $\mathbb{N} = \text{range}(\langle \rangle)$ oft sogar durch $\text{range}(\langle \rangle) := G^*$, für ein meist sehr kleines Alphabet G (in der Regel $G = \{0, 1\}$) ersetzt werden. Die lexikalische Ordnung auf G^* ist total, und definiert eine weitere Gödelisierung $\nu_{lg-lex} : G^* \rightarrow \mathbb{N}$, die dann der Abbildung $\langle \rangle$ nachgeschaltet werden kann, damit letztlich $\nu_{lg-lex} \circ \langle \rangle$ die verlangte Gödelisierung eines Wortes w in \mathbb{N} darstellt.

Will man Wortmengen, d.h. formale Sprachen oder kurz Sprachen, definieren, so gelingt das durch Aufzählung aller Elemente nur bei endlichen Mengen. Für unendliche Mengen ist natürlich eine endliche Beschreibung für diese Mengen, d.h. aller zu ihr gehörenden Elemente, nötig. In späteren Kapiteln werden dazu noch Kalküle, Grammatiken und auch die verschiedensten Typen von Automaten erklärt werden.

Für die am meisten verwendeten und wichtigen Wortmengen, die regulären Mengen, wird dazu gerne der endliche Automat verwendet.

3. Endliche Automaten und reguläre Mengen

Gewöhnlich wird unter einem Automaten oder einer Maschine ein mechanisches oder elektronisches Gerät verstanden, bei dem ausschließlich die Funktion ausschlaggebend für seine Beurteilung ist. Mechanische Geräte wie Hammer, Lupe oder Schreibmaschine werden bei komplexeren Funktionen von schlichten Werkzeugen unterschieden. Ähnliches findet man bei elektronischen Geräten, wie zum Beispiel bei der Unterscheidung zwischen Transistor, Solarzelle oder Computer. Im täglichen Leben finden wir an fast jeder Straßenecke Automaten: Zigarettenautomaten, Fahrkartenautomaten, Musikbox, Glücksspielautomat; aber auch zu Hause: Telefon, CD-Spieler, Fernseher oder Anrufbeantworter. Gemeinsam sind all diesen realen Maschinen folgende Charakteristika: **Eingabeinformationen** (z.B.: Tastendruck, Münzeinwurf, Wähllaktion, o.ä.), **interne Verarbeitung** ohne Rückgriff auf externe weitere Informationsquellen (z.B.: Prüfen der eingeworfenen Münzen, Berechnung des Fahrpreises auf Grund der Zielangabe, Summation der eingeworfenen Beträge) und **Ausgabeinformationen** (z.B.: Geldauswurf, Fahrkarte und Wechselgeld, Zigaretten, o.ä.). All diese Funktionen werden in der Regel unzweideutig stets in der gleichen Weise ausgeführt.

3.1. Deterministische endliche Automaten

Um nun von den nicht die Funktion betreffenden Aspekten wie Material, Farbe oder Herstellungspreis absehen zu können, wird in der Theoretischen Informatik von realen Bauweisen abstrahiert und nur Wesentliches beschrieben. Wir beginnen mit dem einfachsten Modell, dem **endlichen Automaten**. Dieser modelliert in der Regel eine Maschine, die ihre internen Zustände auf Grund von festgelegten Eingaben ändern kann und *keine Ausgabe* erzeugt. Deterministische endliche Automaten werden aber auch mit Ausgabefunktionen versehen und heißen dann nach ihren Erfindern Moore-Automat oder Mealy-Automat.

3.1 Definition

Ein **deterministischer endlicher Automat** (**DFA** für *deterministic finite automaton*) wird durch ein 5-Tupel $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ beschrieben, wobei

Z eine endliche Menge von **Zuständen** ist,

Σ ein endliches Alphabet von **Eingabesymbolen** ist,

$\delta : \subseteq Z \times \Sigma \longrightarrow Z$ ist die nicht notwendigerweise totale **Überföhrungsfunktion**,

$z_0 \in Z$ der **Startzustand** ist und

$Z_{\text{end}} \subseteq Z$ die Menge der **Endzustände** ist.

3. Endliche Automaten und reguläre Mengen

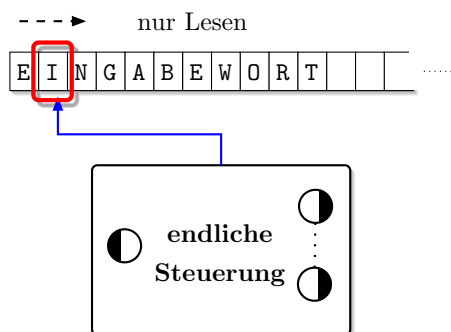


Abbildung 3.1.: Grundschemata des deterministischen endlichen Automaten

Zustände werden meist mit z_0, z_1, \dots oder mit p_i, q_j, \dots , etc. bezeichnet. Für Eingabesymbole werden üblicherweise Buchstaben vom Anfang des lateinischen Alphabetes verwendet, also a, b, c, \dots , o.ä.

Wie schon im einleitenden Beispiel gezeigt, veranschaulichen wir einen DFA durch seinen **Zustandsgraphen**. Dies ist ein Kanten bewerteter gerichteter Graph, dessen Knoten eineindeutig den Zuständen des DFA zugeordnet sind.

Da die Knotenmenge des Zustandsgraphen isomorph zu der Zustandsmenge des Automaten ist, wollen wir hier jeden Knoten als einen Kreis um die Zustandsbezeichnung zeichnen: (z) . Startzustände wer-

den hierbei als (z) und Endzustände als (z) gezeichnet. Eine mit dem Zeichen x beschriftete Kante $(z_1) \xrightarrow{x} (z_2)$ vom Knoten z_1 zum Knoten z_2 wird genau dann gezeichnet, wenn $\delta(z_1, x) = z_2$ gilt.

Einen DFA kann man sich als Maschine vorstellen, die ein Eingabeband besitzt auf dem in einzelne Felder jeweils Symbole $x_i \in \Sigma$ geschrieben sind, die in ihrer Folge von links nach rechts jenes Wort $w = x_1x_2 \dots x_n$ bilden, das als Eingabe des DFA dient. Um diese Eingabe verarbeiten zu können, besitzt der DFA einen Lesekopf, der auf dem Eingabeband jeweils ein Feld besuchen kann, dessen Beschriftung liest und in Abhängigkeit von diesem Symbol und dem momentanen Zustand entsprechend der Übergangsfunktion δ in den dadurch bestimmten Folgezustand wechselt. Der DFA startet dazu in seinem Startzustand z_0 , liest das Eingabewort von links nach rechts in der beschriebenen Weise und akzeptiert es, wenn er nach Lesen des letzten Symbols des Eingabewortes in einem Endzustand ist.

Diese informale Beschreibung wird nun mathematisch präziser, also formal, formuliert werden.

3.2 Definition

Zu einem gegebenen DFA $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ definieren wir rekursiv die **erweiterte Überföhrungsfunktion** $\hat{\delta} : Z \times \Sigma^* \rightarrow Z$ durch:

$$\hat{\delta}(z, xw) := \hat{\delta}(\delta(z, x), w)$$

für alle Zustände $z \in Z$, alle Symbole $x \in \Sigma$ und alle Wörter $w \in \Sigma^*$, sowie

$$\forall z_i \in Z : \hat{\delta}(z_i, \lambda) := z_i.$$

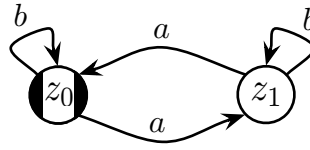


Abbildung 3.2.: Beispiel-DFA

Die von dem DFA A **akzeptierte Sprache** ist die Menge

$$L(A) := \{w \in \Sigma^* \mid \widehat{\delta}(z_0, w) \in Z_{\text{end}}\}.$$

Offensichtlich kann ein DFA auch die leere Menge \emptyset akzeptieren. Dazu muss nur die Menge Z_{end} der Endzustände als leere Menge definiert werden, was ja nicht ausgeschlossen wurde!

Die Funktion $\widehat{\delta}$ ist tatsächlich eine Erweiterung der Funktion δ , denn für alle Zustände $z \in Z$ und alle Symbole $x \in \Sigma$ gilt ja $\widehat{\delta}(z, x) = \delta(z, x)$. Ferner gilt offensichtlich

$$\widehat{\delta}(z, x_1 x_2 \cdots x_n) = \delta(\cdots \delta(\delta(z, x_1), x_2) \cdots, x_n).$$

Es ist üblich, anstelle der erweiterten Funktion $\widehat{\delta}$ etwas unpräzise nur δ zu notieren, sofern dadurch keine Verwirrung entsteht. Auch wir wollen diese Notation künftig verwenden.

3.3 Beispiel

Betrachten wir den Zustandsgraphen aus Abbildung 3.2. Es stellt einen DFA dar, der die Menge aller Wörter $w \in \{a, b\}^*$ erkennt, die eine gerade Anzahl von Symbolen a enthalten. Auch das leere Wort λ ist ein solches Wort und wird von dem Automaten akzeptiert, ohne den Startzustand zu verlassen. Dies ist möglich, da der Startzustand gleichzeitig ein Endzustand ist.

Bedenken Sie bei der Betrachtung eines endlichen Automaten immer genau, ob das Akzeptieren des leeren Wortes λ gewünscht ist oder nicht!

Neben der allgemeinsten Form des deterministischen endlichen Automaten (DFA) nach Def. 3.1 gibt es spezielle Modelle, die besonderen Einschränkungen unterworfen sind. Zum Beispiel war in der Definition des DFA nicht gefordert worden, dass es mindestens einen Endzustand geben muss, oder es zu jedem Eingabesymbol in jedem Zustand auch ein definierter Übergang existiert: Weder die Überföhrungsfunktion δ , noch folglich deren Erweiterung $\widehat{\delta}$, musste eine totale Funktion sein!

3.4 Definition

Ein DFA $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ heißt:

1. **vollständig** (Abk.: vDFA) genau dann, wenn für jedes $(p, x) \in Z \times \Sigma$ ein $q \in Z$ existiert, so dass $q = \delta(p, x)$ ist.
2. **initial zusammenhängend** (Abk.: izDFA) genau dann, wenn zu jedem Zustand $p \in Z$ ein Wort $w \in \Sigma^*$ existiert, mit $p = \widehat{\delta}(z_0, w)$.

3. Endliche Automaten und reguläre Mengen

Zwei verschiedene DFA, A und B , heißen genau dann **äquivalent**, wenn sie die gleiche Sprache akzeptieren, d.h. $L(A) = L(B)$ ist.

In vollständigen DFA's ist die Übergangsfunktion tatsächlich total, also eine Abbildung. Die Frage, ob es zu jedem DFA einen äquivalenten vollständigen DFA oder einen äquivalenten initial zusammenhängenden DFA gibt, werden wir in Kürze beantworten.

Nachdem wir gemäß Definition 2.47 jede Teilmenge von Σ^* als *formale Sprache* bezeichnen, jede Menge von Mengen in der Regel als *Klasse von Mengen*, wollen wir nun unter den Klassen von formalen Sprachen sogenannte „Familien von Sprachen“, kurz Sprachfamilien, kennzeichnen:

3.5 Definition

Eine Klasse \mathcal{L} von formalen Sprachen wird **Sprachfamilie** genannt, wenn folgende Bedingungen erfüllt sind:

1. $\mathcal{L} \neq \emptyset$.
2. Es existiert ein abzählbar unendliches Alphabet Γ , so dass für jede Sprache $L \in \mathcal{L}$ ein endliches Alphabet $\Sigma \subseteq \Gamma$ existiert mit $L \subseteq \Sigma^*$.
3. $\exists L \in \mathcal{L} : L \neq \emptyset$.

3.6 Beispiel

Offenbar ist die Klasse *aller* Wortmengen **keine** Sprachfamilie, da es für diese unendlich vielen Sprachen kein gemeinsames endliches Alphabet Σ gibt. Zum Beispiel gibt es für die unendliche, abzählbare Menge Γ kein endliches Alphabet Σ mit $\Gamma \subseteq \Sigma$. Die Klasse aller endlichen Mengen wie auch die Klasse 2^{Σ^*} aller Sprachen über dem Alphabet Σ sind offensichtlich Sprachfamilien, jedoch ohne sonderlich nutzbare Struktur. Alle PROLOG-Programme bilden eine Sprachfamilie, wenn jedes einzelne PROLOG-Programm als eine Zeichenkette über dem ASCII-Zeichensatz aufgefasst wird (auch die Wortzwischenräume „blanks“ sind einzelne Zeichen!).

3.7 Definition

Die Mengen von Wörtern, die von einem DFA akzeptiert (erkannt) werden können, nennt man **reguläre Mengen** und bezeichnet die gesamte Familie derselben mit $\mathcal{R}eg$. Mit $\mathcal{A}kz(\Sigma)$ wird die Familie aller jener Sprachen $L \subseteq \Sigma^*$ bezeichnet, die von deterministischen endlichen Automaten mit dem Eingabe-Alphabet Σ akzeptiert werden können. Mithin gilt:

$$\mathcal{A}kz(\Sigma) = \{L \subseteq \Sigma^* \mid L = L(A) \text{ für einen DFA } A\}$$

und

$$\mathcal{R}eg = \bigcup_{\substack{\Sigma \text{ ist endl.} \\ \text{Alphabet}}} \mathcal{A}kz(\Sigma).$$

Nach Definition 3.4 können wir auch spezielle Typen von deterministischen endlichen Automaten verwenden, wissen bisher aber noch nicht, ob die Beschränkung auf vDFA oder izDFA vielleicht das Akzeptieren einiger regulärer Mengen verhindert. Dass dem zum Glück nicht so ist, werden wir im Verlauf der nächsten Untersuchungen noch feststellen.

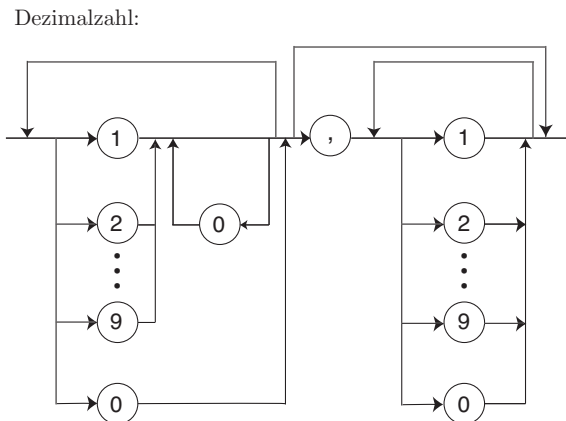


Abbildung 3.3.: Syntaxdiagramme

3.2. Nichtdeterministische endliche Automaten

Syntaxdiagramme zu Programmiersprachen wie zum Beispiel PASCAL oder MODULA geben an, auf welche Weise ein korrekter Ausdruck entsprechend der Definition geschrieben werden kann. Dies Vorgehen ist wahlfrei oder nichtdeterministisch in dem Sinne, dass alles erlaubt ist, was nicht verboten wurde.

3.8 Beispiel

In Abbildung 3.3 wird die Syntax für Dezimalzahlen durch ein rekursionsfreies Syntaxdiagramm angegeben.

Die Interpretation dieses Diagramms geschieht auf folgende Weise: Links beginnend dürfen die Kanten, den Pfeilen folgend, gegangen werden, wobei die in den Kreisen (oder Ovalen) stehenden Symbole aufgeschrieben werden sollen, und am Ende das Diagramm rechts verlassen werden muss. An den Abzweigungen kann jeder beliebige Weg gewählt werden, und jede so notierbare Zeichenkette ist eine in der betreffenden Programmiersprache erlaubte Dezimalzahl, wobei lediglich eine eventuelle Obergrenze in der Zahl der Ziffern mit dem Syntaxdiagramm nicht formuliert wurde. Aufmerksame Betrachter(innen) finden zum Beispiel für die Ziffernfolge 2000 mehr als nur einen Weg durch obiges Diagramm: Die kleine Rückwärtsschleife mit der Ziffer „0“ kann nach Besuchen des Knotens \bigcirc dreimal oder nur zweimal durchlaufen werden. In letzterem Fall muss die dritte Null der Ziffernfolge 2000 – nach Benutzen des oberen Rückwegs nach ganz links – dann über den unteren Weg gewählt werden. Ganz diesem Beispiel entsprechend, kann man sich einen endlichen nichtdeterministischen Automaten als einen bewerteten, gerichteten Graphen vorstellen. Dieser besitzt extra ausgezeichnete Start- und Zielknoten, und beschreibt dann die Menge aller jener Zeichenketten, die durch Aneinanderketten aller Kantenbewertungen entstehen können, die entlang gerichteter Kanten auf einem Weg von einem beliebigen Startknoten zu einem ebenso beliebigen Zielknoten zu finden sind. So wie ein Taxifahrer in einer Großstadt im allgemeinen mehrere Möglichkeiten hat, seinen Kunden zum Ziel zu fahren, gibt es in solch einem gerichteten gerichteten Graphen ebenfalls mehrere, oft beliebig viele, Möglichkeiten von einem Start- zu einem Zielknoten zu gelangen. In vielen Fällen vereinfachen sich die Definitionen von formalen Sprachen auf diese Weise erheblich, auch wenn damit die anschaulichere Vorstellung vom endlichen Automaten als einer real

3. Endliche Automaten und reguläre Mengen

existierenden Maschine, aufgegeben werden muss.

Eine andere Vorstellung sieht einen nichtdeterministischen endlichen Automaten lediglich als eine Erweiterung des bekannten DFA. In dieser Erweiterung gibt die Überföhrungsfunktion bei Eingabe eines Symbols in einem Zustand möglicherweise mehrere Folgezustände an. Beide Vorstellungen haben ihre Vorteile und wir geben beide Definitionsvarianten zum Vergleich an, werden danach aber stets die „graphenorientierte“ Definition 3.10 bevorzugen.

3.9 Definition

Ein **nichtdeterministischer endlicher Automat (NFA)**, engl. *nondeterministic finite automaton*) wird spezifiziert werden durch ein Tupel $A := (Z, \Sigma, \delta, Z_{\text{start}}, Z_{\text{end}})$, wobei

Z eine endliche Menge von **Zuständen** ist,

Σ ein endliches Alphabet von **Eingabesymbolen** ist,

$\delta : Z \times \Sigma \longrightarrow 2^Z$ eine Abbildung ist, die **Überföhrungsfunktion** genannt wird,

$Z_{\text{start}} \subseteq Z$ die **Menge der Startzustände** ist und

$Z_{\text{end}} \subseteq Z$ die **Menge der Endzustände** ist,

Die Überföhrungsfunktion δ ist deshalb eine totale Funktion, weil dort wo die Überföhrungsfunktion eines DFA undefiniert wäre, hier als Bild die leere Menge \emptyset gewählt werden kann. Wie beim DFA auch, wird die Abbildung δ verallgemeinert, hier aber etwas anders, weil wir es hier mit Zustandsmengen zu tun haben.

$\widehat{\delta} : 2^Z \times \Sigma^* \longrightarrow 2^Z$ sei erklärt durch:

$$\widehat{\delta}(Z', xw) := \bigcup_{z \in Z'} \widehat{\delta}(\delta(z, x), w)$$

für alle $w \in \Sigma^*$, alle $x \in \Sigma$ und jede Teilmenge $Z' \subseteq Z$, sowie

$$\forall Z' \subseteq Z : \widehat{\delta}(Z', \lambda) := Z'.$$

$L(A) := \{w \in \Sigma^* \mid \widehat{\delta}(Z_{\text{start}}, w) \cap Z_{\text{end}} \neq \emptyset\}$ ist die von A akzeptierte Sprache.

Wenn wir diese Definition benutzen, so müssen wir einen NFA, der genau das Wort $w \in \Sigma^*$ akzeptieren soll, mit $|w| + 1$ Zuständen modellieren, auch wenn diesem Wort in dem Zustandsgraphen genau ein Pfad zugeordnet ist. Auch darf das leere Wort λ nicht als Kanteninschrift verwendet werden. All dies schränkt einfache Darstellungsmöglichkeiten ein. Wie wir sehen werden, ist dies nicht nötig, und die folgende Definition des nichtdeterministischen endlichen Automaten ist die allgemeinste in dieser Form. Sie orientiert sich mehr an dem Zustandsgraphen als an der funktionalen Sichtweise mit einer Überföhrungsfunktion.

3.10 Definition

Ein **nichtdeterministischer, endlicher Automat (NFA)**, engl. *nondeterministic finite automaton*) ist ein Tupel $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$, wobei die einzelnen Größen wie folgt erklärt sind:

Z ist eine endliche Menge von Zuständen,

$Z_{\text{start}} \subseteq Z$ ist die Menge der Startzustände,

$Z_{\text{end}} \subseteq Z$ ist die Menge der Endzustände,

Σ ist ein endliches Alphabet von Eingabesymbolen und

$K \subseteq Z \times \Sigma^* \times Z$ ist die endliche Relation (vergl. Def. 2.10) der Zustandsübergänge. Ein Element $(p, \lambda, q) \in K$ wird als λ -**Kante** bezeichnet. Ein NFA ohne λ -Kanten wird als λ -freier NFA bezeichnet.

Um mit diesen Automaten arbeiten zu können, benötigen wir noch die Begriffe der *Rechnung* und der *Erfolgsrechnung* sowie den daraus resultierenden Begriff der *akzeptierten Sprache* eines NFA.

3.11 Definition

Eine Kantenfolge $p := (z_0, x_1, z_1), (z_1, x_2, z_2), \dots, (z_{n-1}, x_n, z_n)$, mit $(z_{i-1}, x_i, z_i) \in K$, für $0 < i \leq n$, heißt **Rechnung** von A für das Wort $|p| := w := x_1 x_2 \dots x_n \in \Sigma^*$. Dies wird kurz notiert durch: $z_0 \xrightarrow{w} z_n$ und für jedes $w \in \Sigma^*$ ist \xrightarrow{w} eine Teilmenge von $Z \times Z$ und kann somit als eine Relation auf Z angesehen werden.

Eine **Erfolgsrechnung** für das Wort w , ist eine Rechnung, bei der $z_0 \in Z_{\text{start}}$ und $z_n \in Z_{\text{end}}$ ist, oder $w = \lambda$ und $z_0 = z_n$ mit $z_n \in Z_{\text{start}} \cap Z_{\text{end}}$.

$L(A) := \{w \in \Sigma^* \mid \exists z_0 \in Z_{\text{start}} : \exists z_n \in Z_{\text{end}} : z_0 \xrightarrow{w} z_n\}$ bezeichnet die von A **akzeptierte Sprache**.

Es werden also genau solche Wörter $w \in \Sigma^*$ akzeptiert, für die in dem Automaten A eine Erfolgsrechnung existiert. Das leere Wort wird immer akzeptiert, wenn – wie beim DFA – Anfangs- und Endzustand identisch sind; oder ein ausschließlich λ -Kanten enthaltender Pfad von einem Startzustand zu einem Endzustand im Zustandsgraphen des NFA existiert. Es ist also möglich, dass auch ein λ -freier NFA das leere Wort akzeptiert!

Einen speziellen nichtdeterministischen endlichen Automaten A kann man natürlich immer durch explizite Angabe der fünf Mengen $Z, \Sigma, K, Z_{\text{start}}$ und Z_{end} notieren, besser ist aber meist auch hier die Angabe des Zustandsgraphen, der ähnlich erklärt wird wie beim DFA. Bei kleinen Automaten ist dies in der Regel auch übersichtlicher. Automaten, die einer bestimmten Einschränkung genügen sind in manchen Fällen einfacher zu benutzen, besonders bei Beweisen oder beim Umsetzen in Algorithmen. Daher geben wir hier zwei besondere Klassen von nichtdeterministischen endlichen Automaten an.

3.12 Definition

Ein NFA $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ heißt

1. **buchstabierend** genau dann, wenn $K \subseteq Z \times \Sigma \times Z$ ist,
2. **λ -frei** genau dann, wenn $K \subseteq Z \times \Sigma^+ \times Z$ ist.

Zwei NFA, A und B , heißen genau dann **äquivalent**, wenn sie die gleiche Sprache akzeptieren, d.h. $L(A) = L(B)$ ist.

Da wir die Zustandsgraphen von endlichen Automaten nicht nur als einfache oder bequeme Darstellungen ansehen wollen, sondern bei deren Benutzung auch Analysen vornehmen wollen, werden wir die hier

3. Endliche Automaten und reguläre Mengen

benötigten mathematischen Begriffe einführen und erläutern. Wir schließen dabei an die mathematischen Grundbegriffe aus Kapitel 2 an.

Relationen bilden ein wichtiges und anschauliches Hilfsmittel, nicht nur bei der Darstellung von endlichen Automaten, sondern auch bei der Beschreibung und Formulierung von anderen Modellen oder Algorithmen. Jede binäre Relation lässt sich als gerichteter Graph auffassen. Algorithmen für Graphen können umgekehrt auch oft bei Fragen zu Relationen sinnvoll genutzt werden.

3.13 Definition

Sei R eine binäre Relation auf $A \times B$. Der **Graph** $G(R)$ der Relation ist der **gerichtete Graph** (directed graph) $G(R) := (V, E)$ mit der Menge der **Knoten** (Ecken, vertices) $V := A \cup B$ und der Menge $E := R \subseteq V \times V$ der **gerichteten Kanten** (edges). Die Kante $(a, b) \in E$, notiert als $a \rightarrow b$, wird also genau dann gezeichnet, wenn $(a, b) \in R$ gilt. Notiert wird ein gerichteter Graph $G = (V, E)$ oft durch seine **Adjazenzmatrix** $C_G \in \{0, 1\}^{A \times B}$. Dies ist eine Matrix, deren Zeilen, mit den Elementen des Vorbereichs von R , also A , und deren Spalten, mit den Elementen des Nachbereichs von R , also B , bezeichnet werden. Die einzelnen Matrizenelemente $C_G(x, y)$ sind definiert durch:

$$\forall x \in A : \forall y \in B : C_G(x, y) := [(x, y) \in E].$$

Mit dieser Notation, kann die Komposition von Relationen (vergl. Def. 2.10) durch ein modifiziertes Produkt der Adjazenzmatrizen gebildet werden. Wir benutzen das gewöhnliche Matrizenprodukt, nur interpretieren wir die verwendeten Operationen Summe und Multiplikation anders!

3.14 Definition

Seien $A \in \{0, 1\}^{m \times n}$ und $M_2 \in \{0, 1\}^{n \times r}$ spezielle Matrizen über \mathbb{Z} , die wir in diesem Zusammenhang als **boolesche Matrizen** bezeichnen wollen, da die Werte 0 und 1 in ihren Komponenten wie boolesche Wahrheitswerte benutzt werden sollen. Das **boolesche Matrizenprodukt** von A und B ist für jedes Element $C(i, j)$, $1 \leq i \leq m$, $1 \leq j \leq r$ für $C := A \otimes B$ definiert durch:

$$C(i, j) := \max \left\{ \min \left\{ A(i, k), B(j, k) \right\} \mid 1 \leq k \leq n \right\}.$$

Statt $x + y$ in \mathbb{R} bilden wir also $\max\{x, y\}$ und anstelle von $x \cdot y$ in \mathbb{R} bilden wir $\min\{x, y\}$. Interpretiert man die Zahlen 0 und 1 als Wahrheitswerte (1 für *wahr* und 0 für *falsch*), so entspricht die modifizierte Summe (max) über den Werten 0 und 1 nun gerade dem logischen Junktoren \vee („oder“/„/“) und das Produkt (min) dem logischen Junktoren \wedge („und“/„/“). Daher heißt diese Multiplikation eben auch boolesches Matrizenprodukt. Es passt zur Definition der Adjazenzmatrix $C_{G(R)}$ zum Graph der Relation R , mit

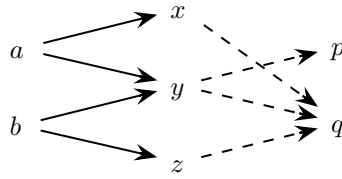
$$C_{G(R)}(x, y) = [(x, y) \in R] = \begin{cases} 1, & \text{falls } (x, y) \in R \\ 0, & \text{falls } (x, y) \notin R. \end{cases}$$

Mit dieser Darstellung erhalten wir für die Komposition $R_1 \cdot R_2$ der Relationen $R_1 \subseteq A \times B$ und $R_2 \subseteq B \times C$ dann $C_{G(R_1)} \otimes C_{G(R_2)} = C_{G(R_1 \cdot R_2)}$.

Ein kleines Beispiel soll dies veranschaulichen:

3.15 Beispiel

Seien $A := \{a, b\}$, $B := \{x, y, z\}$, $C := \{p, q\}$ und $R_1 \subseteq A \times B$, $R_2 \subseteq B \times C$ gegeben durch $R_1 := \{(a, x), (a, y), (b, y), (b, z)\}$ und $R_2 := \{(x, q), (y, p), (y, q), (z, q)\}$. Die Graphen der Relationen sind in Abbildung 3.4 dargestellt.


 Abbildung 3.4.: Die Graphen von R_1 (durchgehend) und R_2 (gestrichelt) aus Beispiel 3.15

Mit den Adjazenzmatrizen $C_{G(R_1)} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$ und $C_{G(R_2)} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ errechnet man leicht, dass deren boolesches Produkt gerade die Adjazenzmatrix $C_{G(R_1 \cdot R_2)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$ zum Graphen der Relationskomposition $R_1 \cdot R_2$ ergibt!

Wichtig bei dieser Auffassung ist, dass der Graph einer Relation keine doppelten Kanten zwischen den gleichen Punkten besitzt, denn eine Relation ist eine Menge von Paaren, und kann diese nicht mehrfach enthalten. Das wird erst möglich, wenn wir die einzelnen Kanten von einander unterscheiden können, was durch deren **Beschriftung** oder **Bewertung** geschieht. In nachfolgender Definition 3.16 wird ausdrücklich eine beliebige Menge X von möglichen Bewertungen oder Kantenbeschriftungen gestattet, damit wir diese Kanten bewerteten Graphen möglichst flexibel handhaben können.

3.16 Definition

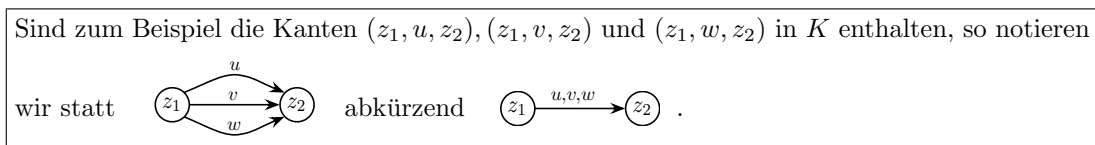
Sei $K \subseteq A \times X \times B$ eine Relation „von A nach B “ mit Kantenbewertungen aus der Menge Σ . Der **kantenbewertete gerichtete Graph** zu K ist spezifiziert durch das Tripel $G_K := (V, X, K)$ mit Knotenmenge $V := A \cup B$. Eine bewertete Kante $(a, x, b) \in K$ von G_K wird als $a \xrightarrow{x} b$ gezeichnet.

Wie beim DFA ist auch die Knotenmenge des Zustandsgraphen eines NFA isomorph zu seiner Zustandsmenge und jeder Knoten wird wieder als Kreis um die Zustandsbezeichnung gezeichnet. Start- und Endzustände werden in der gleichen Weise dargestellt wie beim Zustandsgraphen eines DFA's. Eine formale Definition können wir nun mit Hilfe des Graphen zu der Relation K eines NFA geben:

3.17 Definition

Sei $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ ein NFA, dann ist der Zustandsgraph zu A der Graph $G_K := (Z, \Sigma^*, K)$.

Da von einem Zustand p in einem nichtdeterministischen endlichen Automaten nun mehrere verschiedene Kanten mit unterschiedlichen Wörtern als Beschriftung zum selben Zustand q führen können, wird eine vereinfachte Darstellung bevorzugt:



3. Endliche Automaten und reguläre Mengen

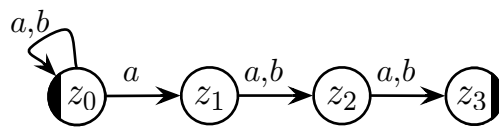


Abbildung 3.5.: NFA, der alle Wörter akzeptiert, die an drittletzter Position ein a haben

3.18 Beispiel

In Abbildung 3.5 ist der Zustandsgraph eines NFA, der genau diejenigen Wörter $w \in \{a, b\}^*$ akzeptiert, die an drittletzter Position das Symbol a enthalten. Dieser endliche Automat akzeptiert also z.B. das Wort $abaaba$, *nicht* aber das Wort $abaabaa$.

Da jeder DFA ein spezieller NFA ist, kann natürlich jede reguläre Menge von einem NFA akzeptiert werden. Zu fragen ist jedoch, ob es Sprachen gibt, die ein NFA akzeptieren kann, die nicht regulär sind und wie überhaupt gezeigt werden kann, ob es überhaupt eine formale Sprache gibt, die nicht regulär ist und wie man dieses eventuell beweisen könnte! Wir werden dies Antwort hier nicht schuldig bleiben, und sogar mehrere, jedoch sicher nicht alle, Möglichkeiten zu deren Beantwortung darstellen.

3.3. Endliche Automaten mit Ausgabe

Neben den im vorigen Abschnitt definierten endlichen Automaten, die zum *Akzeptieren* von Wörtern bzw. Sprachen verwendet werden, werden wir hier solche definieren, die ihre Eingaben in passende Ausgaben *transformieren*. Wir unterscheiden dabei *Mealy*- und *Moore-Automaten* hinsichtlich der Weise, wie die Ausgabe bestimmt wird. Beide Modelle sind deterministische endliche Automaten mit Ausgabe (*finite state machines*), die *ohne* spezifizierte Endzustandsmenge auskommen, denn *jede* Eingabe über dem Eingabealphabet soll eine eindeutig spezifizierte Ausgabe bewirken. Die Bezeichnung „Maschine“ (*machine*) wird anstelle von „Automat“ in der Regel immer dann benutzt, wenn es sich um *deterministische Automaten* handelt!

Beim Moore-Automaten wird ein einzelnes Symbol als Ausgabe jedesmal dann ausgegeben, wenn ein Zustand erreicht (oder durchlaufen) wird; die Ausgabe richtet sich dabei nur nach dem jeweiligen Zustand. Typische reale Automaten von diesem Typ sind die bekannten Fahrkartenautomaten. Dort kommt es nur darauf an welche Taste man gedrückt hat und wieviel Geld eingeworfen wurde, nicht aber, in welcher Reihenfolge die Münzen für den zu zahlende Betrag eingegeben wurden.

Beim Mealy-Automaten wird ein einzelnes Symbol als Ausgabe jeweils bei einer Transition, d.h. einem Zustandsübergang, getätigt. Ein Beispiel dafür war der Kugelautomat aus Kapitel 1. Auch die sogenannte *gsm* (*generalized sequential machine*) und der allgemeine Transduktor mit oder ohne akzeptierende Endzustände (*a-transducer*, *transducer*) arbeiten nach diesem Prinzip, können aber in jedem Schritt beliebige Zeichenketten ausgeben. Der folgende Transduktor ist tatsächlich eine *gsm* aber kein Mealy-Automat.

3.19 Beispiel

Der Transduktor aus Abbildung 3.6 transformiert jedes Eingabewort der Form $w_1cw_2c\dots cw_n$, mit $w_i \in$

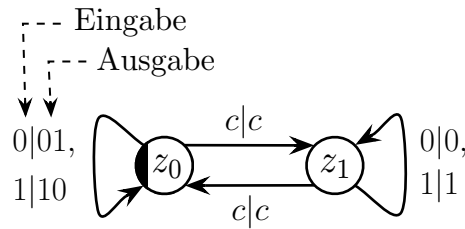


Abbildung 3.6.: Ein Transduktor

$\{0, 1\}^*$, in das entsprechende Ausgabewort der Form $\varphi(w_1)cw_2c\varphi(w_3)cw_4 \dots \varphi(w_{n-1})cw_n$, falls n gerade ist, bzw. $\varphi(w_1)cw_2c\varphi(w_3)cw_4 \dots w_{n-1}c\varphi(w_n)$, falls n ungerade ist. Hierbei wird $\varphi : \{0, 1\}^* \rightarrow \{0, 1\}^*$ als Homomorphismus mit $\varphi(0) = 01, \varphi(1) = 10$ benutzt.

3.20 Definition

1. Ein **Moore-Automat** wird beschrieben durch $M = (Z, \Sigma, \Gamma, \delta, \rho, q_0)$, wobei $A := (Z, \Sigma, \delta, z_0, Z)$ ein vollständiger DFA, Γ das **Ausgabealphabet** und $\rho : Z \rightarrow \Gamma$ die **Ausgabefunktion** ist.
2. Die **Ausgabe** von M für das Eingabewort $w := x_1x_2 \dots x_n$, mit $x_i \in \Sigma$, ist definiert durch das Wort $\rho(q_0)\rho(q_1) \dots \rho(q_n)$ wobei q_0, q_1, \dots, q_n diejenige Folge von Zuständen ist, für die $\delta(q_i, x_{i+1}) = q_{i+1}, 0 \leq i \leq n$ gilt, also $(q_0, x_1, q_1), (q_1, x_2, q_2), \dots, (q_{n-1}, x_n, q_n)$ eine Rechnung von A für die Eingabe $x_1x_2 \dots x_n$ ist.

Eine spezielle Endzustandsmenge gibt es bei Moore-Automaten nicht: jeder Zustand darf als Endzustand betrachtet werden, und da der Moore-Automat vollständig ist, kann jedem Wort aus Σ^* eine Ausgabe zugeordnet werden!

3.21 Beispiel

Wenn wir zwei Binärzahlen addieren wollen so schreiben wir diese gewöhnlich untereinander und beginnen von rechts die einzelnen Stellen zu addieren. Dabei ist die Summe von 1 und 1 eben die Binärzahl 10 und man notiert 0 mit 1 als Übertrag. Die Summe $\begin{pmatrix} 11101 \\ 1011 \end{pmatrix}$ liefert dann das Ergebnis 101000.

Wollen wir dieses mit einem Moore-Automaten bewerkstelligen, so müssen wir jeweils zwei übereinanderstehende Eingabeziffern mit den Symbolen $\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ und $\begin{pmatrix} 1 \\ 1 \end{pmatrix}$ als eine Folge von kombinierten Eingabesymbolen kodieren und diese von rechts nach links einzeln der Maschine zuführen.

Die Summe $\begin{pmatrix} 11101 \\ 1011 \end{pmatrix}$ wird also kodiert als $\begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ und entspricht dann der umgekehrten Eingabesequenz: $\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, die für einen eventuellen Übertrag am rechten Ende um das Symbol $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ verlängert wurde. Die Ausgabe an den Zuständen muss der letzten Stelle der jeweils errechneten einfachen Summe entsprechen, während mit dem Folgezustand der zu addierende Übertrag gespeichert werden muss.

Abbildung 3.7 zeigt einen Moore-Automaten mit dem Eingabealphabet $\{\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix}\}$. Die Eingabe beginnt mit dem letzten Bitpaar der Addition und endet mit Eingabe eines zusätzlichen Paares $\begin{pmatrix} 0 \\ 0 \end{pmatrix}$ für den letzten Übertrag.

Die Ausgabe ist bei z_i gerade $i \bmod 2$, also bei den beiden Zuständen z_0 und z_2 jeweils 0 und bei den beiden Zuständen z_1 und z_3 jeweils eine 1.

3. Endliche Automaten und reguläre Mengen

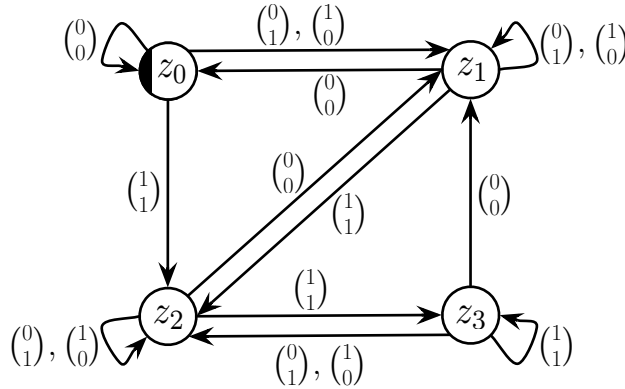


Abbildung 3.7.: Ein Moore-Automat als Serienaddierer

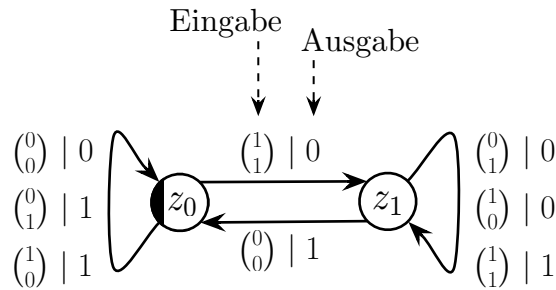


Abbildung 3.8.: Ein Mealy-Automat als Serienaddierer

3.22 Definition

1. Ein **Mealy-Automat** wird beschrieben durch $M = (Z, \Sigma, \Gamma, \delta, \rho, q_0)$, wobei $A := (Z, \Sigma, \delta, q_0, Z)$ ein vollständiger DFA, Γ das **Ausgabealphabet** und $\rho : Z \times \Sigma \rightarrow \Gamma$ die **Ausgabefunktion** ist.
2. Die **Ausgabe** von M für das Eingabewort $w := x_1 \dots x_n$, mit $x_i \in X$, ist definiert durch das Wort $\rho(q_0, x_1)\rho(q_1, x_2) \dots \rho(q_{n-1}, x_n)$ wobei q_0, q_1, \dots, q_{n-1} eine Folge von Zuständen ist, für die $\delta(q_i, x_{i+1}) = q_{i+1}, 0 \leq i < n-1$ gilt, also $(q_0, x_1, q_1), (q_1, x_2, q_2), \dots, (q_{n-1}, x_n, q_n)$ eine Rechnung von A für die Eingabe $x_1 x_2 \dots x_n$ ist.

3.23 Beispiel (Mealy-Automat als Serienaddierer)

Abbildung 3.8 zeigt einen Mealy-Automaten, also einen DFA mit Ausgabe, der einen Serienaddierer modelliert.

Mealy- und Moore-Automaten sind deterministische Maschinen, für die kein Endzustand spezifiziert wurde oder nötig gewesen wäre. Zu jedem Wort $w \in \Sigma^*$ gibt es eine eindeutige Übersetzung und diese wird durch den deterministischen Automaten funktional definiert. Mealy-Automaten werden oft bezüglich ihrer Ausgabefunktion zu $\rho : Z \times X \rightarrow Y^*$ erweitert. Solche Maschinen werden *gsm* (*generalized sequential machine*) genannt. Man definiert sogar die Erweiterung zu NFA's mit Ausgaben vom

Mealy-Typ, den sogenannten akzeptierenden Transduktoren, deren Studium wir hier nicht mehr weiter verfolgen wollen.

Das folgende Ergebnis über akzeptierende endliche Automaten ist sehr wichtig, da es deutlich macht, dass mit der Definition eines *nichtdeterministischen* gegenüber der des *deterministischen* endlichen Automaten, keine größere Klasse von definierbaren Sprachen entsteht.

3.24 Theorem

Jede von einem NFA akzeptierte Menge kann auch von einem initial zusammenhängenden, vollständigen DFA akzeptiert werden und ist daher regulär.

Um dieses erstaunliche Theorem auf einfache Weise beweisen zu können ist es hilfreich, vorher spezielle Umformungen an einem vorgegebenen beliebigen, nichtdeterministischen endlichen Automaten vorzunehmen. So kann ein NFA eine oder mehrere λ -Kanten besitzen, die in deterministischen endlichen Automaten ja gar nicht erlaubt waren. Man kann diese nicht einfach entfernen, ohne die akzeptierte Sprache zu verändern, aber man kann einen beliebigen NFA zunächst derart umformen, dass er danach immer noch die gleiche Sprache akzeptiert jedoch keine λ -Kanten mehr besitzt. Wir nennen einen NFA ohne λ -Kanten nach Definition 3.10 kurz λ -frei.

3.25 Theorem

Zu jedem NFA gibt es einen äquivalenten λ -freien NFA.

Beweis: Sei $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ ein gegebener NFA, der möglicherweise λ -Kanten besitzt. Ein neuer NFA $B := (Z, \Sigma, K', Z_{\text{start}}, Z'_{\text{end}})$ ohne λ -Kanten wird wie folgt erklärt:

$$K' := \{(z, w, z''') \in Z \times \Sigma^* \times Z \mid \exists (z', w, z'') \in K : \\ w \neq \lambda \wedge z \xrightarrow{\lambda} z' \wedge z'' \xrightarrow{\lambda} z''' \text{ (in } A)\}$$

$$\text{und } Z'_{\text{end}} := Z_{\text{end}} \cup \{z \in Z_{\text{start}} \mid \exists z' \in Z_{\text{end}} : z \xrightarrow{\lambda} z'\}.$$

Wir zeigen die Gleichheit von $L(A)$ und $L(B)$ durch den Beweis der gegenseitigen Inklusion:

$L(B) \subseteq L(A)$ gilt, weil (z, w, z') nur dann in K' ist, wenn im Zustandsgraphen von A ein Pfad von z nach z' existiert, auf dem das Wort w gelesen wird. Die Definition von Z'_{end} sichert, dass jeder Erfolgspfad in B auch einer in A ist.

$L(A) \subseteq L(B)$ gilt aus folgender Überlegung: Zunächst folgt aus $\lambda \in L(A)$ auch $\lambda \in L(B)$, denn dann ist einer der Startzustände von B auf Grund der Definition von Z'_{end} auch zugleich Endzustand.

Weiter ist klar, dass jede Kante $(z', w, z'') \in K$ mit $w \neq \lambda$ auch eine Kante von K' ist!

Nun lässt sich jeder Erfolgspfad p in A schreiben als:

$$z_1 \xrightarrow{\lambda} z'_1 \xrightarrow{w_1} z_2 \xrightarrow{\lambda} z'_2 \xrightarrow{w_2} z_3 \xrightarrow{\lambda} \dots \xrightarrow{w_{n-1}} z_n \xrightarrow{\lambda} z'_n \xrightarrow{w_n} z_{n+1} \xrightarrow{\lambda} z'_{n+1}$$

Hierbei ist $z'_i \xrightarrow{w_i} z_{i+1}$ gerade der Zustandsübergang mit einer Kante (z'_i, w_i, z_{i+1}) von A , mit $w_i \neq \lambda$. Wenigstens eine solche Kante muss in einem Pfad p mit $|p| \neq \lambda$ ja vorkommen. Nach

3. Endliche Automaten und reguläre Mengen

Definition von K' ist aber jetzt auch $(z_i w_i, z_{i+1}) \in K'$ für jedes $1 \leq i < n$ sowie $(z_n, w_n, z'_{n+1}) \in K'$. Folglich gilt in dem Automaten B auch $z_1 \xrightarrow[p]{*} z'_{n+1}$, was wegen $z_1 \in Z_{\text{start}}$ und $z_{n+1} \in Z'_{\text{end}}$ offensichtlich ein Erfolgspfad in B ist.

□

Obwohl der Beweis oben gegeben ist, können wir damit noch nicht ganz zufrieden sein, denn eigentlich wissen wir doch gar nicht, auf welche Weise wir den neuen, λ -freien Automaten nun im speziellen Fall *effektiv* konstruieren können. Für die Konstruktion der neuen Kantenmenge K' ist es offensichtlich nötig, festzustellen, ob für je zwei beliebige Zustände z und z' die Beziehung $z \xrightarrow{\lambda^*} z'$ gilt. Nun ist die Relation

$\xrightarrow{\lambda^*} \subseteq Z \times Z$ gerade die reflexive, transitive Hülle der Relation $\xrightarrow{\lambda} := \{(p, q) \mid (p, \lambda, q) \in K\}$, und letztere kann sofort aus dem Zustandsgraphen von A abgelesen werden: es sind genau die λ -Kanten dieses Automaten.

Zur effektiven Bestimmung des reflexiven, transitiven Abschlusses R^* einer endlichen zweistelligen Relation R (vergl. Def.2.16), also der Menge aller Wege zwischen den Knoten des gerichteter Graphen $G(R)$, taugt die Darstellung nach Theorem 2.26 natürlich nichts, denn es wäre ja ein unendlicher Durchschnitt zu bilden. Nach Theorem 2.25 wissen wir, dass es ausreicht von der Relation $\xrightarrow{\lambda}$ nur endlich viele

Kompositionen $\xrightarrow{\lambda^k}$ zu bilden um deren transitive Hülle $\xrightarrow{\lambda^+}$ zu erhalten. Die dort benutzte Oberschranke ist jedoch etwas groß, und es stellt sich die Frage, ob es nicht bessere Verfahren gibt.

In dem gerichteten Graphen $G(R)$ beschreibt die reflexive transitive Hülle R^* von R alle gerichteten Pfade in diesem Graphen. Die Lösung dieser Frage ist auch an anderen Stellen von großem Interesse, und daher wird in der Veranstaltung F3 das Verfahren von Warshall zur Bestimmung des reflexiven, transitiven Abschlusses $\xrightarrow{*}$ einer endlichen zweistelligen Relation $\longrightarrow \subseteq A \times A$ behandelt.

Wir benutzen an dieser Stelle eine konstruktive Variante von Theorem 2.25, die eine recht gute Komplexität besitzt.

Sei $R \subseteq A \times A$ eine endliche Relation auf der endlichen Menge $A, n := |A|$. (In der hier gesuchten Anwendung wird $R := \xrightarrow{\lambda}$ mit $A := Z$ sein). Mit den Definitionen von $R^{i+1} := R^i \cdot R$ und $R^0 := Id_A$ gilt nun

$$\begin{aligned} R^* &= \bigcup_{i \geq 0} R^i \\ &= \bigcup_{i=0}^{n-1} R^i = Id_A \cup R \cup R^2 \cup R^3 \cup \dots \cup R^{n-1} \\ &= (Id_A \cup R)^{n-1}, \end{aligned}$$

wobei letztere Gleichung leicht mit Induktion zu verifizieren ist. Man beachte hierbei, dass mit R^i hier **nicht** das i -fache kartesische Produkt gemeint wurde sondern die i -fache Komposition! Warum der Exponent nicht größer als $n - 1$ gewählt werden muss sieht man so ein:

Es gilt doch $(x, y) \in R^i$ genau dann, wenn es in dem Graphen $G(R)$ einen Pfad vom Knoten x zum Knoten y gibt. Wenn es aber solch eine Pfad gibt, so existiert immer auch einer mit höchstens $n =$

$|A|$ verschiedenen Knoten, also mit höchstens $n-1$ Kanten. (Alle Schleifen können weggelassen werden, wodurch sich jeder Pfad sukzessive (rekursiv) verkürzen lässt.)

Damit braucht zur Bestimmung der Relation $\xrightarrow{\lambda^*}$ nur das Produkt $(Id_Z \cup \xrightarrow{\lambda})^{|Z|-1}$ gebildet werden!

Für diese Aufgabe gibt es eine sehr effiziente Methode, die linear in der Größe $|A|$ arbeitet! Diese und weitere Methoden zu Design und Analyse von Algorithmen werden später im Studium, z.B. in der Veranstaltung F3 („Berechenbarkeit und Komplexität“) aber auch im Hauptstudium noch ausführlicher behandelt werden.

Bis jetzt können wir also zu jedem NFA einen äquivalenten λ -freien NFA konstruieren, aber dieser ist noch immer nicht deterministisch, ja noch nicht einmal buchstabierend.

3.26 Theorem

Zu jedem λ -freien NFA gibt es einen äquivalenten buchstabierenden NFA.

Beweis: Zu jeder Kante $k := (z, w, z') \in K$ des NFA $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ mit $|w| \geq 2$ werden $|w| - 1$ neue (Zwischen-)Zustände z_{k_i} definiert. Die Kante $k = (z, w, z')$ mit $w = x_1 x_2 \cdots x_n$ wird dann ersetzt durch die Kanten der Menge

$$\{(z, x_1, z_{k_1}), (z_{k_1}, x_2, z_{k_2}), \dots, (z_{k_{n-2}}, x_{n-1}, z_{k_{n-1}}), (z_{k_{n-1}}, x_n, z')\}.$$

□

Nach diesen Umformungen eines NFA erhält man also mit Sicherheit einen äquivalenten λ -freien, buchstabierenden NFA. Dieser braucht aber nicht vollständig zu sein, da in manchen Zuständen nicht für jedes Eingabesymbol $x \in X$ eine herausführende Kante existieren muss. Will man auch dieses gewährleisten, so fügt man einen weiteren Zustand z_s als *Senke* hinzu, und ergänzt den Automaten durch vorher fehlende Kanten, die in diese Senke führen. Im Zustand z_s gibt es für jedes Symbol $x \in X$ eine Kante (z_s, x, z_s) . Bei der Umformung in einen deterministischen Automaten mit der folgende Konstruktion des sogenannten Potenzautomaten wird aber ohnehin ein vollständiger äquivalenter DFA entstehen, so dass dieses Verfahren nicht detaillierter dargestellt werden braucht.

Nach diesen Vorbereitungen ist es jetzt einfacher möglich, den Beweis von Theorem 3.24 zu formulieren:

Beweis (von Theorem 3.24): Sei $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ ein buchstabierender, nicht notwendig vollständiger, NFA. Wir konstruieren den sog. **Potenzautomaten** zu A , dessen Zustände als Namen gerade die Teilmengen von Z erhalten, und der ein vollständiger deterministischer endlicher Automat ist. (Wer damit besser umgehen kann, mag sich die Zustandsmenge so vorstellen, als stünde die Teilmenge $Q \subseteq Z$ für den Zustand z_Q einer Menge $\{z_P \mid P \subseteq Z\}$.) Dieser vDFA B werde wie folgt erklärt: $B := (2^Z, \Sigma, \delta, z_0, Z'_{\text{end}})$ mit $Z'_{\text{end}} := \{M \in 2^Z \mid M \cap Z_{\text{end}} \neq \emptyset\}$ und $z_0 := Z_{\text{start}}$, d.h., die Menge der Startzustände von A bildet nun den einzigen Startzustand z_0 von B . Die Überfunktionsabbildung δ ist für alle $M \in 2^Z$ und jedes $x \in \Sigma$ definiert durch:

$$\delta(M, x) := \bigcup_{z \in M} \{z' \in Z \mid (z, x, z') \in K\}.$$

Zu jeder akzeptierenden Rechnung von B auf einem Wort w gibt es wegen der Definition von δ ebenfalls einen Erfolgspfad von A auf w . Andererseits findet sich auch für jeden Erfolgspfad von A eine akzeptierende Rechnung in B . Die in A besuchten Zustände kommen dabei in den einzelnen Zuständen des Pfades von B , die ja selbst Teilmengen von Z sind, in der gleichen Reihenfolge vor. □

3. Endliche Automaten und reguläre Mengen

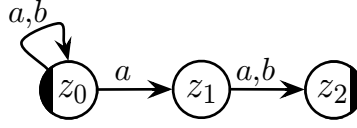


Abbildung 3.9.: Der NFA akzeptiert alle Wörter, die an vorletzter Stelle ein a haben

Der Potenzautomat „merkt“ sich im Zustandsnamen, in welchen Zuständen der nichtdeterministische endliche Automat sich nach derselben Sequenz von gelesenen Symbolen befinden könnte. Er akzeptiert ein Wort genau dann, wenn der ursprüngliche NFA mit diesem Wort in einen Endzustand hätte gelangen können, d.h. genau dann, wenn es im ursprünglichen Automaten einen Erfolgspfad für dieses Wort gibt.

3.27 Beispiel

Wir konstruieren mit der Potenzautomatenkonstruktion zu dem in Abbildung 3.9 gezeichneten NFA $A := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ einen äquivalenten vDFA B . Für den gegebenen NFA A seien die Bestandteile erklärt durch:

$$\begin{aligned}
 Z &:= \{z_0, z_1, z_2\}, \\
 \Sigma &:= \{a, b\}, \\
 K &:= \{(z_0, a, z_0), (z_0, b, z_0), (z_0, a, z_1), (z_1, a, z_2), (z_1, b, z_2)\}, \\
 Z_{\text{start}} &:= \{z_0\}, \\
 Z_{\text{end}} &:= \{z_2\} \text{ und } .
 \end{aligned}$$

Sein Zustandsgraph ist in Abbildung 3.9 dargestellt. Nach Konstruktion sind die Bestandteile von $B := (Z', \Sigma, \delta, z_0, Z'_{\text{end}})$ gegeben durch:

$$\begin{aligned}
 Z' &:= 2^Z = \left\{ \emptyset, \{z_0\}, \{z_1\}, \{z_2\}, \{z_0, z_1\}, \{z_0, z_2\}, \{z_1, z_2\}, \{z_0, z_1, z_2\} \right\}, \\
 \Sigma &:= \{a, b\}, \\
 Z'_{\text{end}} &:= \left\{ \{z_2\}, \{z_0, z_2\}, \{z_1, z_2\}, \{z_0, z_1, z_2\} \right\}
 \end{aligned}$$

Die graphische Darstellung des Potenzautomaten B und δ sind dem (nicht zusammenhängenden) Zustandsgraphen aus Abbildung 3.10 zu entnehmen.

Wie man in Abbildung 3.10 sehen kann, muss der Potenzautomat nicht initial zusammenhängend sein. Der rechte Teil der Abbildung zeigt einen vom Startzustand aus nicht erreichbaren Teilgraphen, der folglich für die Akzeptierung von Wörtern nicht relevant ist.

Natürlich kann man getrost alle diejenigen Komponenten eines vDFA weglassen, die nicht vom Startzustand aus erreicht werden können. Entsprechend muss dann die Übergangsfunktion auf die, eventuell kleinere, neue Zustandsmenge eingeschränkt werden. An der endlichen Darstellung des Zustandsgraphen kann man dies sofort sehen, falls der vDFA eine überblickbare Größe besitzt. Wir wollen hier

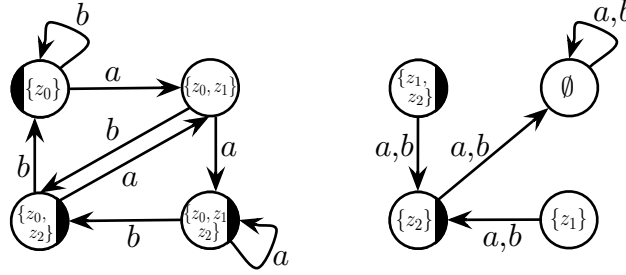


Abbildung 3.10.: Potenzautomat zu dem NFA aus Abbildung 3.9

ein einfaches, wenn auch nicht das effizienteste, Verfahren vorstellen, wie man in dem Zustandsgraphen eines, nicht notwendig vollständigen, DFA die initial zusammenhängende Komponente bestimmen kann. Wenn wir in Zukunft vom Potenzautomaten sprechen, meinen wir üblicherweise nur den initial zusammenhängenden Teil des Potenzautomaten nach dem Beweis von Theorem 3.24.

3.28 Algorithmus (initiale Zusammenhangskomponente eines vDFA)

Sei $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein beliebiger DFA. Wir berechnen schrittweise die induktiv definierten Mengen $M_i \subseteq Z$, beginnend bei M_0 , gemäß:

$$M_i := \begin{cases} M_{i-1} \cup \bigcup_{\substack{z \in M_{i-1} \\ x \in \Sigma}} \delta(z, x) & \text{für } i > 0 \\ \{z_0\} & \text{für } i = 0 \end{cases}$$

Offensichtlich ist jeder Zustand $z \in M_i$, $i \in \mathbb{N}$, mit einer passenden Eingabe $w \in \Sigma^*$, mit $|w| \leq i$, vom Startzustand aus erreichbar. Wegen $M_{i-1} \subseteq M_i \subseteq Z$ und der Endlichkeit der Zustandsmenge Z muss es einen Index $k \leq |Z|$ geben, für den $M_{k+1} = M_k$ ist (mit jeder Erhöhung des Indexes i um 1 kann möglicherweise nur ein weiterer Zustand in die Menge M_i aufgenommen werden).

Das Konstruktionsverfahren endet, wenn das erste mal $M_k = M_{k+1}$ wird, was spätestens bei $k = |Z| - 1$ der Fall sein wird. Die Menge M_k enthält bei Abschluss des Verfahrens also genau die von z_0 aus erreichbaren Zustände im DFA A .

Nach der gegebenen Konstruktion des Potenzautomaten, hat der äquivalente DFA exponentiell mehr Zustände als der zu Beginn bekannte NFA. Dass dies keine Schwäche dieser Konstruktion ist, sondern in einigen Fällen tatsächlich so viele Zustände für den DFA nötig sind, zeigt das folgende Resultat von Theorem 3.29.

3.29 Theorem

Zu jeder natürlichen Zahl $n \geq 2$ gibt es einen buchstabierenden, λ -freien NFA A_n mit genau n Zuständen, so dass jeder äquivalente vollständige DFA mindestens $2^n - 1$ Zustände besitzt.

Die Automaten A_n haben die Gestalt aus Abbildung 3.11.

3. Endliche Automaten und reguläre Mengen

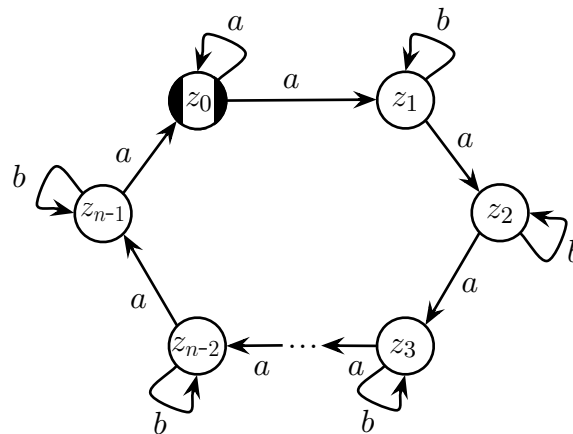


Abbildung 3.11.: Der Automat A_n hat genau n Zustände.

Den Beweis dieser Aussage findet man als Übungsaufgabe in [Floyd&Beigel]. Wir werden später mit weniger Aufwand das ähnliche Theorem 3.50 beweisen.

Statt eines Beweises überlegen Sie bitte, welche Menge solch ein NFA A_n beschreibt! An den Schwierigkeiten, diese Menge mit Worten exakt zu beschreiben, merken wir, dass eine andere Methode für die Beschreibung regulärer Mengen praktisch sein kann. In der Literatur findet man die Definition regulärer Mengen nicht nur durch endliche Automaten, sondern auch mit Hilfe von **rationalen Ausdrücken**. Rationale Ausdrücke werden deshalb auch oft als **reguläre Ausdrücke** bezeichnet. Den formalen Beweis für Äquivalenz zu endlichen Automaten wird Theorem 3.34 geben.

3.4. Rationale Ausdrücke

Im Folgenden wird nun ein weiteres, besonders typisches Beispiel einer *induktiven Definition* gegeben, bei der neu zu definierende Objekte durch schon bekannte erklärt werden (Definition der Syntax) und parallel dazu auch die Bedeutung (Semantik) definiert wird.

3.30 Definition

Sei Σ ein endliches Alphabet, dann sind **rationale Ausdrücke** über Σ genau die mit den folgenden Regeln erzeugbaren Gebilde:

1. \emptyset ist ein rationaler Ausdruck, der die (leere) Menge $M_{\emptyset} := \emptyset$ beschreibt.
2. Für jedes $a \in \Sigma$ ist a ein rationaler Ausdruck, der die Menge $M_a := \{a\}$ beschreibt.
3. Sind A und B rationale Ausdrücke, welche die Mengen M_A und M_B beschreiben, dann ist auch
 - $(A + B)$ ein rationaler Ausdruck, der die Menge $M_A \cup M_B$ beschreibt,
 - $A \cdot B$ ein rationaler Ausdruck, der die Menge $M_A \cdot M_B$ beschreibt, und
 - $(A)^*$ ein rationaler Ausdruck, der die Menge M_A^* beschreibt.

$(A)^+$ ein rationaler Ausdruck, der die Menge M_A^+ beschreibt.

4. Nur die mit den Regeln 1.) bis 3.) erzeugbaren Gebilde sind rationale Ausdrücke.

Es bezeichne $\text{Rat}(\Sigma)$ die Familie aller mit rationalen Ausdrücken beschreibbaren Teilmengen von Σ^* , und $\text{Rat} := \bigcup_{\Sigma \text{ ist endl. Alphabet}} \text{Rat}(\Sigma)$.

Um Klammern zu sparen, dürfen neben den äußersten Klammern auch weitere weggelassen werden, wobei die Prioritätenreihenfolge $()^*$ vor \cdot , und \cdot vor $+$ für Eindeutigkeit sorgt. Damit steht $a \cdot b^* + a$ eindeutig für $(a \cdot (b)^* + a)$ und anstelle von $(c + (a + b))$ notieren wir $(a + b + c)$ als Beschreibung der Menge $\{a, b, c\}$. Es gelten die Assoziativgesetze für $+$ und \cdot genauso wie die für die Vereinigung oder das Komplexprodukt, weshalb wir diese Klammern und den Punkt \cdot meist ebenfalls weglassen können! Ebenfalls behandeln wir rationale Ausdrücke grundsätzlich wie Namen, bzw. Bezeichnungen der Mengen die sie beschreiben. Wir erlauben folgerichtig Notationen wie $babaa \in (a, b)^*$ anstelle von $babaa \in M_{(a,b)^*}$. In einigen Büchern wird in rationalen Ausdrücken auch Λ als Beschreibung der Menge $\{\lambda\}$ benutzt. Dass dies hier nicht geschieht, ist kein entscheidender Verlust, denn \emptyset^* beschreibt die gleiche Menge!

Rationale Ausdrücke werden in vielen Programmen zur Datei- oder Textverarbeitung benutzt, um Suchverfahren flexibler als nur mit sogenannten „wildcards“ vornehmen zu können. Manche Betriebssysteme tun dies auf der Ebene der Dateiverwaltung und ebenfalls einige Textverarbeitungsprogramme. Im UNIX-Betriebssystem kennt man zum Beispiel das Programm **egrep** (*extended global regular expression print*), was wir hier nur knapp und beispielhaft vorstellen wollen. Wer schon einmal mit dem UNIX-System gearbeitet hat, kennt vielleicht auch die Varianten „grep“ und „fgrep“.

Die Kommandosyntax für egrep ist:

egrep $\langle \text{regausdruck} \rangle \langle \text{dateiname} \rangle$.

Die Befehle zur Beschreibung der rationalen Ausdrücke müssen zwangsläufig anders dargestellt werden, und es gilt dabei:

rationaler Ausdruck	egrep-Notation
Zeilenanfang	\wedge
Zeilenende	$\$$
c	c
(Gesamtalphabet:) Σ	\cdot
$r + \emptyset^*$	$r?$
r^*	r^*
r^+	r^+
$r + s$	$r s$
$r \cdot s$	rs
(r)	(r)

Wenn man z.B. den Befehl

3. Endliche Automaten und reguläre Mengen

```
egrep ^Th...i.$ /usr/dict/duden.
```

eingibt, so würden die Wörter **Thermik**, **Theorie** und **Thespis** gefunden und ausgedruckt. Wir wollen auf weitere Details und Feinheiten dieser Syntax hier nicht eingehen!

Für die Beschreibung von regulären Mengen scheinen sich die rationalen Ausdrücke geradezu anzubieten. So können wir z.B. die Menge $L_{a??}$ aller Wörter $w \in \{a, b\}^*$, bei denen das Symbol a an drittletzter Position vorkommt einfach beschreiben: $L_{a??}$ wird durch $(a, b)^*a(a, b)(a, b)$ oder knapper durch $(a, b)^*a(a, b)^2$ beschrieben.

Wir werden im folgenden Abschnitt zeigen, dass es sich tatsächlich bei den Operationen, welche in der Definition der rationalen Ausdrücke verwendet werden, um Operationen handelt, gegen die die Familie der von endlichen Automaten abgeschlossen ist. Weitere Abschlusseigenschaften werden wir im Abschnitt 3.9 kennenlernen.

3.5. Einfache Abschlusseigenschaften

Es ist einfach zu sehen, dass eine Menge mit nur einem einzelnen Wort als Element und mithin auch jede endliche Menge von Wörtern durch einen rationalen Ausdruck beschrieben werden kann. Wenn wir nun beweisen könnten, dass die Vereinigung, das Produkt und die Sternbildung (Kleenesche Hülle) regulärer Mengen wieder reguläre Mengen sind, so wüßten wir, dass jeder rationale Ausdruck eine reguläre Menge beschreibt, die alternativ auch von einem endlichen Automaten akzeptiert werden könnte.

3.31 Theorem

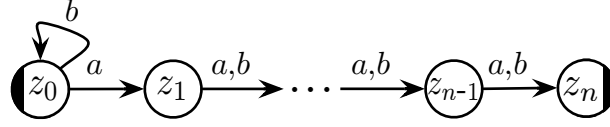
Mit $L_1 \in \mathcal{A}kz(\Sigma_1)$ und $L_2 \in \mathcal{A}kz(\Sigma_2)$ ist auch $L_1 \cup L_2 \in \mathcal{A}kz(\Sigma_1 \cup \Sigma_2)$.

Beweis (Konstruktion eines NFA für $L_1 \cup L_2$): Seien zwei vDFA's $A := (Z_1, \Sigma_1, \delta_1, z_{1,0}, Z_{1,\text{end}})$ und $B := (Z_2, \Sigma_2, \delta_2, z_{2,0}, Z_{2,\text{end}})$ mit disjunkten Zustandsmengen gegeben, für die gilt $L_1 = L(A)$ und $L_2 = L(B)$.

Der NFA $C_{A \cup B} := (Z_3, \Sigma_3, K_3, Z_{3,0}, Z_{3,\text{end}})$ für $L_1 \cup L_2$ ist gegeben durch:

$$\begin{aligned} Z_3 &:= Z_1 \uplus Z_2, \\ \Sigma_3 &:= \Sigma_1 \cup \Sigma_2, \\ Z_{3,\text{start}} &:= \{z_{1,0}, z_{2,0}\}, \\ Z_{3,\text{end}} &:= Z_{1,\text{end}} \cup Z_{2,\text{end}} \text{ und} \\ K_3 &:= \{(p_1, x, \delta_1(p_1, x)) \mid p_1 \in Z_1, x \in \Sigma_1\} \cup \{(p_2, x, \delta_2(p_2, x)) \mid p_2 \in Z_2, x \in \Sigma_2\}. \end{aligned}$$

Dieser Automat ist zwar nichtdeterministisch und im Allgemeinen nicht vollständig, aber schon buchstabiierend. Wie es möglich ist, daraus einen äquivalenten vDFA zu konstruieren haben wir mit der Potenzautomatenkonstruktion von Theorem 3.24 schon gesehen. Es ist offensichtlich, dass zu jeder Erfolgsrechnung in A (bzw. in B) eine entsprechende Erfolgsrechnung in $C_{A \cup B}$ existiert, denn mit der Wahl des Startzustandes wird entschieden, ob ein Wort der Menge L_1 oder L_2 akzeptiert werden soll. Umgekehrt, entspricht einer Erfolgsrechnung in $C_{A \cup B}$, die in einem der Endzustände von A (bzw. von B) endet, die entsprechende Erfolgsrechnung in A (bzw. in B). \square

Abbildung 3.12.: Ein NFA für die Sprache $\{b\}^* \cdot \{a\} \cdot \{a, b\}^{n-1}$ **3.32 Theorem**

Mit $L_1 \in \text{Rat}(\Sigma_1)$ und $L_2 \in \text{Rat}(\Sigma_2)$ ist auch $L_1 \cdot L_2 \in \text{Rat}(\Sigma_1 \cup \Sigma_2)$.

Beweis (Konstruktion eines buchstabierenden NFA's für $L_1 \cdot L_2$): Seien $A := (Z_1, \Sigma_1, \delta_1, z_{1,0}, Z_{1,\text{end}})$ und $B := (Z_2, \Sigma_2, \delta_2, z_{2,0}, Z_{2,\text{end}})$ vollständige DFA's mit $Z_1 \cap Z_2 = \emptyset$ und $L_1 = L(A)$ und $L_2 = L(B)$. Ein buchstabierender NFA $C_{A \cdot B}$ für das Produkt $L_1 \cdot L_2$ ist $C_{A \cdot B} := (Z_1 \cup Z_2, \Sigma_1 \cup \Sigma_2, K, z_{1,0}, Z_{2,\text{end}})$ mit $K := K_1 \cup K_2 \cup K_3$ wobei $K_1 := \{(p_1, x, \delta_1(p_1, x)) \mid x \in \Sigma_1, p_1 \in Z_1\}$ die zum vDFA A gehörende Kantenmenge, $K_2 := \{(p_2, x, \delta_2(p_2, x)) \mid x \in \Sigma_2, p_2 \in Z_2\}$ die zum vDFA B gehörende Kantenmenge und $K_3 := \{(p_1, x, \delta_2(z_{2,0}, x)) \mid x \in \Sigma_2, p_1 \in Z_{1,\text{end}}\}$ die beide Automaten verbindenden, λ -freien Kanten sind. \square

Bei obiger Konstruktion wird der NFA $C_{A \cdot B}$ in der Regel kein vollständiger DFA sein, so z.B. wenn $\Sigma_1 \cap \Sigma_2 = \emptyset$ gilt. Ist $\Sigma_1 = \Sigma_2$, so ist es leider zudem nötig, den entstandenen NFA in einen deterministischen umzuformen. Es kann dabei sogar geschehen, dass für vDFA's A und B mit $|Z_1| = m$ und $|Z_2| = n$ jeder äquivalente DFA für $L(A) \cdot L(B)$ mindestens 2^{m+n} Zustände haben muss! Als illustratives, aber diese Aussage nicht beweisendes, Beispiel betrachten wir den Automaten $A := (\{z_0\}, \{a, b\}, \delta, z_0, \{z_0\})$ mit $\delta(z_0, a) := \delta(z_0, b) := z_0$ und $L(A) = \{a, b\}^*$. Für B wählen wir den Automaten Abbildung 3.12, mit $L(B) = \{b\}^* \cdot \{a\} \cdot \{a, b\}^{n-1}$. Es ergibt sich $L(A) \cdot L(B) = \{a, b\}^* \cdot \{a\} \cdot \{a, b\}^{n-1}$. Von dieser Menge werden wir nach dem Beweis von Theorem 3.50 wissen, dass sie kein vDFA mit weniger als 2^n Zuständen akzeptiert!

Wenn wir für die Sternbildung ebenfalls einen DFA konstruieren müssen, so gelingt dies auf einfache Weise auch nicht ohne die übliche Umformung eines buchstabierenden NFA in den äquivalenten DFA:

3.33 Theorem

Mit $L \in \text{Akz}(\Sigma)$ ist auch $L^* \in \text{Akz}(\Sigma)$.

Beweis (Konstruktion eines buchstabierenden NFA's für L^*): Sei $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein vDFA mit $L = L(A)$, so wird ein buchstabierender NFA C_{A^*} definiert durch $C_{A^*} := (Z \uplus \{p\}, \Sigma, K, \{p\}, \{p\})$ mit $K := K_1 \cup K_2 \cup K_3$, wobei $K_1 := \{(z, x, \delta(z, x)) \mid x \in \Sigma, z \in Z\}$ die zum vDFA A gehörende Kantenmenge, sowie $K_2 := \{(p, x, \delta(z_0, x)) \mid x \in X\}$ und $K_3 := \{(z, x, p) \mid x \in \Sigma, \delta(z, x) \in Z_{\text{end}}\}$ die neuen Nicht- λ -Kanten als Verbindungen mit dem neuen Zustand sind. Der neue Zustand ist notwendig, damit das leere Wort akzeptiert werden kann! \square

Wir werden im folgenden Theorem zeigen, dass auch umgekehrt *jede* reguläre Menge durch einen rationalen Ausdruck beschrieben werden kann. Daher unterscheidet sich die Familie aller Sprachen, die durch rationale Ausdrücke definierbar sind, nicht von denen, die durch endliche Automaten akzeptiert werden können.

3. Endliche Automaten und reguläre Mengen

3.34 Theorem (Satz von Kleene, 1951)

Für jedes Alphabet Σ gilt $\mathcal{A}kz(\Sigma) = \mathcal{R}at(\Sigma)$ und folglich $\mathcal{R}eg = \mathcal{R}at$.

Beweis: Der Beweis erfolgt wiederum in zwei Schritten, in denen die Inklusion einer der beiden Familien in der jeweils anderen gezeigt wird.

$\mathcal{R}at(\Sigma) \supseteq \mathcal{A}kz(\Sigma)$: Wir definieren zu jedem vollständigen DFA $A = (Z, \Sigma, \delta, z_1, Z_{\text{end}})$ die reguläre Menge $L(A)$, zu der ein rationaler Ausdruck leicht gebildet werden kann. Dazu seien für alle $0 \leq k \leq |Z|$ und alle $1 \leq i, j \leq |Z|$ mit $Z := \{z_1, \dots, z_n\}$ folgende Mengen definiert:

$$R_{i,j}^k := \left\{ w \in \Sigma^* \mid \begin{array}{l} \delta(z_i, w) = z_j \text{ und für } w = uv \text{ mit } u \neq \lambda \neq v \\ \text{folgt aus } \delta(z_i, u) = z_r \text{ stets } r \leq k \end{array} \right\}.$$

Die Menge $R_{i,j}^k$ enthält alle Wörter, die auf Pfaden vom Zustand z_i zum Zustand z_j über die Menge $\{z_1, \dots, z_k\}$ von Zwischenzuständen gelesen werden können. Folglich ist

$$R_{i,j}^0 = \{x \in \Sigma \cup \{\lambda\} \mid \delta(z_i, x) = z_j \text{ oder } (x = \lambda) \wedge (i = j)\}.$$

Wir zeigen nun die Gültigkeit der folgenden Gleichung:

$$R_{i,j}^k = \begin{cases} \{x \mid \delta(z_i, x) = z_j \text{ oder } (x = \lambda) \wedge (i = j)\}, & \text{falls } k = 0 \\ R_{i,j}^{k-1} \cup R_{i,k}^{k-1} \cdot (R_{k,k}^{k-1})^* \cdot R_{k,j}^{k-1}, & \text{falls } k \geq 1. \end{cases}$$

Induktions-Basis: $R_{i,j}^0$ ist in der Definition wie in der ersten der zwei Gleichungen angegeben.

Induktions-Schritt: Für $k = m \geq 0$ sei für die Mengen $R_{i,j}^m$ die Rekursionsformel bewiesen. und wir zeigen, dass sie dann auch für die Mengen $R_{i,j}^{m+1}$ gilt. Jeder Pfad von z_i nach z_j , der nur Knoten der Menge $\{z_1, \dots, z_m, z_{m+1}\}$ durchläuft, kann in Stücke zerlegt werden, in denen nur Knoten der Menge $\{z_1, \dots, z_m\}$ durchlaufen werden, die aber stets beim Knoten z_{m+1} aneinander gesetzt sind, sofern dieser überhaupt vorkommt. Kommt z_{m+1} nicht vor, so gehört das auf diesem Pfad gelesene Wort zur Menge $R_{i,j}^m$. Kommt z_{m+1} wenigstens einmal auf dem Pfad vor, so hat dieser eine Zerlegung in der folgenden Art:

$$z_i \xrightarrow[u]{*} z_{m+1} \xrightarrow[v_1]{*} z_{m+1} \xrightarrow[v_2]{*} z_{m+1} \xrightarrow[v_3]{*} \dots \xrightarrow[v_r]{*} z_{m+1} \xrightarrow[w]{*} z_j$$

Die Pfade mit den Wörtern $u, v_1, v_2, \dots, v_r, w$ durchlaufen nun nur noch Zustände der Menge $\{z_1, \dots, z_m\}$.

Offensichtlich gilt: $u \in R_{i,m+1}^m, \forall 1 \leq i \leq r : v_i \in R_{m+1,m+1}^m$ und $w \in R_{m+1,j}^m$. Dies zeigt zunächst die Inklusion: $R_{i,j}^{m+1} \subseteq R_{i,j}^m \cup R_{i,m+1}^m \cdot (R_{m+1,m+1}^m)^* \cdot R_{m+1,j}^m$. Die Umkehrung dieser Inklusion ist leicht ersichtlich.

Für die oben definierten Mengen $R_{i,j}^k$ gilt nun offensichtlich $L(A) = \bigcup_{z_j \in Z_{\text{end}}} R_{1,j}^n$, wobei z_1 der Startzustand des Automaten A war.

Setzt man nun die Rekursionsformel solange in $R_{1,j}^n$ ein, bis die Anfangsmengen $R_{i,j}^0$ erreicht sind, so erhält man eine endliche Beschreibung der Menge $L(A)$, die leicht in einen rationalen Ausdruck umgeschrieben werden kann.

$\text{Rat}(\Sigma) \subseteq \text{Akz}(\Sigma)$: Wir benutzen die Induktive Definition der rationalen Ausdrücke und zeigen zuerst, dass die einfachen Basismengen durch endliche Automaten dargestellt werden können. Dann werden schon bekannte Konstruktionen an NFA's entsprechend der Zusammenfügung schon bestehender rationaler Ausdrücke zu Neuen durchgeführt:

1. \emptyset ist ein rationaler Ausdruck, der die leere Menge beschreibt und wird von dem DFA $C_\emptyset := (\{z_0\}, \Sigma, \delta, z_0, \emptyset)$ akzeptiert. (C_\emptyset hat ein beliebiges Eingabealphabet, jedoch keinen Endzustand).
2. Für jedes $a \in \Sigma$ ist a ein rationaler Ausdruck, der die Menge $\{a\}$ beschreibt und diese wird von dem DFA $C_a := (\{z_0, z_1\}, \{a\}, \delta, z_0, \{z_1\})$ mit $\delta(z_0, a) := z_1$ akzeptiert.
3. Sind A und B rationale Ausdrücke, und $C_A := (Z_A, X_A, K_A, Z_{A,\text{start}}, Z_{A,\text{end}})$ sowie $C_B := (Z_B, X_B, K_B, Z_{B,\text{start}}, Z_{B,\text{end}})$ NFA's mit disjunkten Zustandsmengen, welche die durch die rationalen Ausdrücke dargestellten Mengen M_A und M_B akzeptieren, dann akzeptiert der NFA
 - a) $C_{A \cup B}$ von Theorem 3.31 die durch $(A + B)$ beschriebene Menge $M_A \cup M_B$.
 - b) $C_{A \cdot B}$ von Theorem 3.32 die durch $A \cdot B$ beschriebene Menge $M_A \cdot M_B$.
 - c) C_{A^*} von Theorem 3.33 die durch den rationalen Ausdruck $(A)^*$ beschriebene Menge M_A^* .
 - d) $C_{A^+} := (Z_A, X_A, K_A \cup \{(p, \lambda, q) \mid p \in Z_{A,\text{end}}, q \in Z_{A,\text{start}}\}, Z_{A,\text{start}}, Z_{A,\text{end}})$ die durch den rationalen Ausdruck $(A)^+$ beschriebene Menge M_A^+ .
4. Damit kann für jeden rationalen Ausdruck r ein endlicher Automat konstruiert werden, der die von diesem beschriebene Menge M_r akzeptiert.

Hiermit ist der Beweis von Theorem 3.34 abgeschlossen, denn wir haben gezeigt, wie für einen beliebigen NFA ein äquivalenter rationaler Ausdruck erzeugt werden kann und dass umgekehrt zu jedem rationalen Ausdruck auch ein äquivalenter endlicher Automat existiert. \square

Wir können nun die rationalen Ausdrücke als einfache syntaktische Darstellung von regulären Mengen benutzen, ohne sofort endliche Automaten konstruieren zu müssen. Wir könnten sogar ohne weitere Nachteile die Syntax der rationalen Ausdrücke um solch Notationen erweitern wie: $(A)^n$ für jedes $n \in \mathbb{N}$, $(A \cap B)$ oder $\overline{(A)}$. Wir werden im Folgenden solche erweiterten rationalen Ausdrücke freizügig benutzen!

3.6. Ein Anwendungsbeispiel

Es folgt nun ein etwas größeres Anwendungsbeispiel für die Nutzung von endlichen Automaten mit Ausgabe, welche verbunden wird mit einer rekursiven Definition der benötigten Zustände. Wir wollen das Problem der Suche und Identifizierung von endlich vielen Schlüsselwörtern in einem Text mit Hilfe eines endlichen (Moore-)Automaten lösen. Die hier gewählte Darstellung unterscheidet sich von den in Lehrbüchern üblichen dadurch, dass wir auf eine mögliche Implementierung, d.h. einer Umsetzung und Verwirklichung in einem realen Programm, weniger Wert legen als auf die prinzipielle Struktur. Unsere Lösung ist eine rekursive Variante des Verfahrens von Donald E. Knuth, James H. Morris und Vaughn R. Pratt (1977).

3. Endliche Automaten und reguläre Mengen

Problemstellung: Ein fortlaufender Text soll daraufhin untersucht werden, ob gewisse „Schlüsselwörter“ (*keys*) darin enthalten sind, und ausgegeben werden, um welche es sich handelt und an welchen Stellen diese im Text auftreten. Diese Schlüsselwortsuche soll natürlich auf beliebige Texte angewendet und ebenfalls auch mit wechselnden Schlüsselwörtern erneut gestartet werden können. Das bedeutet, dass als Eingabe des zu entwerfenden Algorithmusses nur die endliche Menge $K \subseteq \Sigma^*$ der Schlüsselwörter verwendet werden kann, nicht aber der Text $t \in \Sigma^*$, auf den das Verfahren ja angewendet werden soll.

Ob in einem Text $t \in \Sigma^*$ überhaupt eines der Schlüsselwörter der Menge $K \subseteq \Sigma^*$ vorkommt, könnte man feststellen, in dem man die Frage „ $\{t\} \cap \Sigma^* \cdot K \cdot \Sigma^* \neq \emptyset$?“ beantwortet. Da die Mengen $\{t\}$ und $\Sigma^* \cdot K \cdot \Sigma^*$ offensichtlich regulär sind, könnte man mit Hilfe des Beweises von Theorem 3.34 eine NFA konstruieren, der diese Menge akzeptiert. Für diesen NFA müsste man dann noch feststellen, ob dieser die leere Menge akzeptiert oder nicht. Niemand würde wirklich in dieser Weise vorgehen, da der Zeitaufwand immens wäre! Außerdem würden wir auf diese Weise nicht feststellen können, an welchen Stellen im Text t welche Schlüsselwörter vorkommen und unsere Aufgabe wäre nur unzureichend gelöst.

Für jede endliche Menge $K \subseteq \Sigma^*$ von Schlüsselwörtern konstruieren wir nun einen Moore-Automaten A der $\Sigma^* \cdot K \cdot \Sigma^*$ mit den passenden Ausgaben versieht direkt:

3.35 Definition

Sei Σ eine Alphabet und $K \subseteq \Sigma^*$ eine endliche Menge von Schlüsselwörtern. Der **Moore-Automat** A_K zur **Schlüsselwortsuche** $A_K := (Z, \Sigma, \Gamma, \delta, \rho, z_\lambda)$ ist erklärt durch:

$$\begin{aligned} Z &:= \{z_v \mid \exists w \in K : v \sqsubseteq w\}, \\ \forall z_v \in Z : \forall x \in \Sigma : \delta(z_v, x) &:= \begin{cases} z_{vx} & \text{wenn } z_{vx} \in Z \text{ ist,} \\ \delta(z_u, x) & \text{wenn } z_{vx} \notin Z, \text{ und } v = yu, y \in \Sigma, \\ z_\lambda & \text{sonst.} \end{cases} \\ \rho(z_w) &:= \begin{cases} \text{„Schlüsselwörter } v_1, v, \dots, v_k \text{ wurden gefunden!“,} \\ \text{falls } \forall i \in \{1, \dots, k\} : (v_i \in K) \wedge \exists u : w = uv_i \text{ (} v_i \text{ ist suffix von } w \text{).} \\ \lambda, \text{ d.h. keine Ausgabe, sonst.} \end{cases} \end{aligned}$$

Wir wollen die Definition des Moore-Automaten zur Schlüsselwortsuche anhand eines Beispiels verdeutlichen.

3.36 Beispiel

Betrachten wir folgende Menge von Schlüsselwörtern. $K := \{der, erde\} \subseteq \{d, e, r\}^*$, so erhalten wir die Zustandsmenge $Z = \{z_\lambda, z_d, z_{de}, z_{der}, z_e, z_{er}, z_{erd}, z_{erde}\}$ und folgenden Moore-Automaten. Man kann zeigen, dass der hier definierte Moore-Automat zur Textsuche der kleinste mit dieser Eigenschaft ist. Er hat $|Z| = 1 + \sum_{v \in K} |v|$ viele Zustände und kann schnell in $c \cdot |\Sigma| \cdot |Z|$ Schritten konstruiert werden, in dem man die Kanten bei z_λ beginnend bestimmt. Ein weiterer Vorteil dieses Automaten ist, dass er bei der Suche nach den Schlüsselwörtern jedes Zeichen des Textes t nur genau einmal liest und ohne Rücksprünge in Realzeit arbeitet. Zudem kann dieses Verfahren leicht für ähnliche Probleme abgewandelt werden.

In Abbildung 3.13 sind die offensichtlichen Ausgaben in den Zuständen z_{erde} und z_{der} nicht eingezeichnet worden.

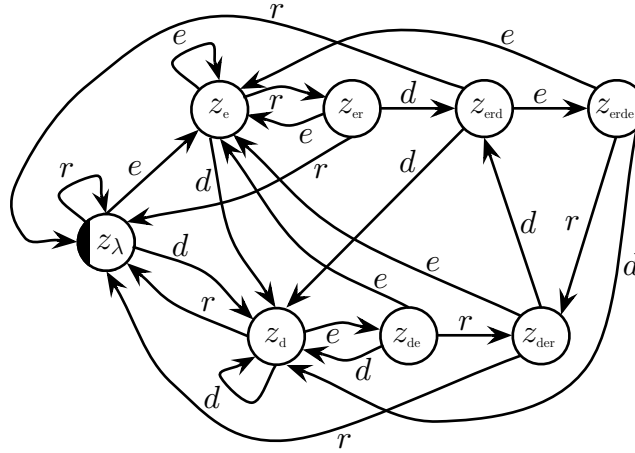


Abbildung 3.13.: Ein Moore-Automat zur Schlüsselwortsuche für die Wörter „erde“ und „der“

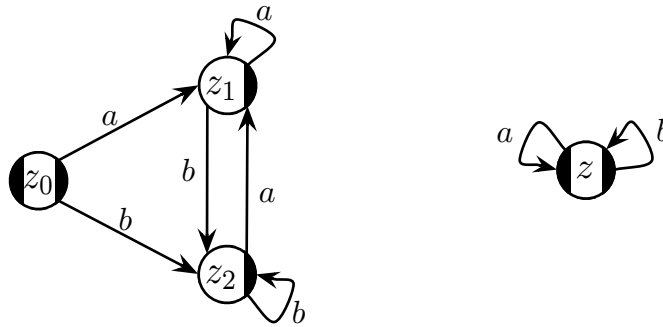


Abbildung 3.14.: Zwei äquivalente Automaten

3.7. Äquivalenzbegriffe und minimale Automaten

Betrachtet man die beiden in Abbildung 3.14 abgebildeten vDFA's, so stellt sich schnell heraus, dass die beiden endlichen Automaten beide die gleiche Sprache akzeptieren, nämlich $\{a, b\}^*$.

Wir wollen jetzt daran gehen, die Frage zu beantworten, wie ein kleinster äquivalenter DFA aussieht, ob er immer konstruiert werden kann und ob es vielleicht davon unterschiedliche geben kann. Vorher erklären wir den Begriff „Transitionsmonoid“ und eine sehr praktische Notation.

Für einen vollständigen DFA $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ gibt es zu jedem Zustand $z \in Z$ und jedem Wort $w \in \Sigma^*$ genau einen Folgezustand z' mit $z \xrightarrow{*w} z'$. Jedes $w \in \Sigma^*$ induziert für diesen Automaten also eine Abbildung: $f_w : Z \rightarrow Z$ mit $f_w(z) := \delta(z, w)$.

3. Endliche Automaten und reguläre Mengen

3.37 Definition

Zu einem gegebenen vDFA $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ schreiben wir $(z)^w$ für den Zustand $\delta(z, w)$, der auf eindeutige Weise von z aus bei Eingabe von w erreicht wird.

Die Menge $F_A := \{(\)^w \mid w \in \Sigma^*\}$ der Abbildungen $(\)^w : Z \rightarrow Z$, mit denen die Zustandsübergänge des vDFA's A beschrieben werden, bilden zusammen mit der Komposition von Funktionen das endliche **Transitionsmonoid** (F_A, \circ) zu A .

Wieso ist die Menge F_A aller Abbildungen $(z)^w : Z \rightarrow Z$ für jeden vollständigen DFA A endlich? Wieviele Abbildungen dieser Art gibt es?

Es gilt offensichtlich $((z)^u)^v = (z)^{uv}$, was die Komposition $(f_v \circ f_u)(z) = f_v(f_u(z))$ in der üblicheren Notation darstellt. Weiter ist $(z)^\lambda = z$ für jeden Zustand $z \in Z$, d.h. stets ist f_λ neutrales Element des Transitionsmonoides.

Ein DFA A definiert in natürlicher Weise eine rechtsinvariante Äquivalenzrelation R_A , deren Index der Zahl der nicht überflüssigen Zustände von A gleicht. Wir wiederholen hier die aus der Mathematik bekannten Definitionen zur besseren Lesbarkeit.

3.38 Definition

Eine Relation $R \subseteq A \times A$ heißt **Äquivalenzrelation** genau dann, wenn R reflexiv, symmetrisch und transitiv ist.

Ist $R \subseteq A \times A$ eine Äquivalenzrelation auf A , so heißt die Menge

$$[a]_R := \{b \in A \mid (a, b) \in R\}$$

die **Äquivalenzklasse** von a . Jedes $b \in [a]$ ist ein **Repräsentant** von $[a]_R$. Man schreibt gewöhnlich nur $[a]$, wenn deutlich ist, welche Relation gemeint ist.

Die Menge aller Äquivalenzklassen notieren wir als $A/R := \{[a]_R \mid a \in A\}$.

Als **Index** der Äquivalenzrelation R auf A bezeichnet man die Zahl der verschiedenen Äquivalenzklassen, die R hat. Wir notieren dies durch: $\text{Index}(R) := |A/R|$.

Eine Äquivalenzrelation R auf einer Menge A zerlegt diese in disjunkte Teilmengen und spezifiziert dadurch eine **Partition** von A .

Wir formulieren dies als Theorem:

3.39 Theorem

Ist $R \subseteq A^2$ eine Äquivalenzrelation, so bilden die Äquivalenzklassen bezüglich R eine Partition von A .

Beweis: Zunächst zeigen wir $A = \bigcup_{a \in A} [a]_R$. Sei dazu $x \in A$ beliebig, so gilt xRx wegen der Reflexivität von R . Mithin gilt $x \in [x]_R$, erst recht also $x \in \bigcup_{a \in A} [a]_R$. Folglich gilt $A \subseteq \bigcup_{a \in A} [a]_R$. Umgekehrt ist $[a]_R \subseteq A$ auf Grund der Definition von $[a]_R$.

Zu zeigen ist nun noch, dass für alle $[a]_R, [b]_R \in A/R$ entweder $[a]_R = [b]_R$ oder $[a]_R \cap [b]_R = \emptyset$ gilt. Seien $a, b \in A$ mit $[a]_R \cap [b]_R \neq \emptyset$. Also muss es ein $c \in [a]_R \cap [b]_R$ geben, für das dann aRc und bRc gilt. Sei nun

$d \in [a]_R$ ein beliebiges Element, so gilt dRa , aRc und cRb , wobei die Symmetrie von R ausgenutzt wird. Aus der Transitivität von R folgt nun dRb und $d \in [b]_R$. Also ist $[a]_R \subseteq [b]_R$. Durch Vertauschen der Symbole a und b in den letzten beiden Sätzen gewinnt man mit der gleichen Begründung die Inklusion $[b]_R \subseteq [a]_R$, so dass aus $[a]_R \cap [b]_R \neq \emptyset$ tatsächlich $[a]_R = [b]_R$ folgt. \square

Zwei Äquivalenzklassen einer Äquivalenzrelation sind also entweder identisch oder disjunkt! Speziell die Identität ist eine symmetrische, reflexive, und transitive Relation, also eine Äquivalenzrelation. Selbstverständlich ist die Identitätsrelation auch antisymmetrisch. Aber im Allgemeinen sind Äquivalenzrelationen gerade nicht antisymmetrisch!

3.40 Beispiel

Betrachten Sie die folgende Relation: $\equiv_4 \subseteq (\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\})^2$ mit $a \equiv_4 b$ genau dann, wenn $b - a$ ein Vielfaches von 4 ist. Dass \equiv_4 eine Äquivalenzrelation ist, kann leicht nachgeprüft werden. Aber sicherlich ist \equiv_4 nicht antisymmetrisch, denn es gilt $[1]_{\equiv_4} = [5]_{\equiv_4} = [9]_{\equiv_4}$, also $1 \equiv_4 5$ und $5 \equiv_4 1$ aber nicht $1 = 5$.

Oft werden Äquivalenzrelationen (genauso wie Kongruenzrelationen) mit dem (meist noch mit einem Index versehenen) Symbol \equiv bezeichnet, aber auch das mit R indizierte Gleichheitszeichen $=_R$ ist nicht ungebräuchlich.

3.41 Definition

Eine Äquivalenzrelation \equiv hat **endlichen Index**, wenn die Zahl $\text{ind}(\equiv)$ der Äquivalenzklassen endlich ist.

Eine Äquivalenzrelation $\equiv \subseteq M \times M$ heißt genau dann **rechtsinvariant**, wenn für jedes Element m der Halbgruppe (M, \circ) aus $m_1 \equiv m_2$ stets $(m_1 \circ m) \equiv (m_2 \circ m)$ folgt.

Nach dieser wichtigen Definition einer Äquivalenzrelation wollen wir einen kleinen Exkurs zu den Grenzen der regulären Sprachen unternehmen, um eine Anwendung dieses Zweiges der Theorie aufzuzeigen.

3.7.1. Exkurs: Grenzen der regulären Sprachen

Am Ende dieses Abschnitts werden wir eine Sprache kennenlernen, von der wir zeigen, dass sie nicht regulär ist. Dazu müssen wir noch einige Begriffe einführen.

3.42 Definition

Sei $A = (Z, \Sigma, \delta, \{z_0\}, Z_{\text{end}})$ ein DFA, dann ist die Äquivalenzrelation $R_A \subseteq \Sigma^* \times \Sigma^*$ erklärt durch:

$$u R_A v \text{ genau dann, wenn } (z_0)^u = (z_0)^v$$

Sei $L \subseteq \Sigma^*$ eine reguläre Menge, dann ist die **Nerode-Äquivalenz** oder **syntaktische Rechtskongruenz** $R_L \subseteq \Sigma^* \times \Sigma^*$ erklärt durch:

$$u R_L v \text{ genau dann, wenn } \forall w \in \Sigma^* : (uw \in L \text{ genau dann, wenn } vw \in L).$$

Es ist leicht zu sehen, dass sowohl R_A als auch R_L eine rechtsinvariante Äquivalenzrelation ist.

Das folgende Ergebnis ist als **Satz von Nerode** bekannt und zeigt, dass R_L von endlichem Index ist:

3. Endliche Automaten und reguläre Mengen

3.43 Theorem (Myhill & Nerode, 1957/1958)

Die folgenden Aussagen sind äquivalent:

1. $L \subseteq \Sigma^*$ ist regulär.
2. L ist Vereinigung von Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation mit endlichem Index.
3. Die Nerode-Äquivalenz R_L hat endlichen Index.

Beweis: Wir führen einen Ringschluss durch, indem wir zunächst zeigen, dass Aussage 2. aus 1. folgt. Dann schließen wir von 2. auf 3. und zeigen dann noch, dass 1. aus 3. folgt. Damit ist dann die Äquivalenz der drei Aussagen gezeigt.

1. \rightarrow 2.: Sei $A = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein nach Theorem 3.24 existierender DFA, der $L \subseteq \Sigma^*$ akzeptiert, und R_A die Relation auf Σ^* gemäß Definition 3.42 mit: uR_Av genau dann, wenn $(z_0)^u = (z_0)^v$. L ist die Vereinigung derjenigen Klassen, die zu Endzuständen gehören, d.h. $L = \{w \in \Sigma^* \mid (z_0)^w \in Z_{\text{end}}\}$, und da das Transitionsmonoid endlich ist, hat auch R_A nur endlich viele Klassen. R_A ist rechtsinvariant, da im DFA die mit einem Wort $w \in \Sigma^*$ aus $z \in Z$ erreichten Zustände $(z)^w$ stets eindeutig sind.
2. \rightarrow 3.: Sei R die nach Voraussetzung existierende rechtsinvariante Äquivalenzrelation von endlichem Index. Wir zeigen, dass R_L eine Vergrößerung von R ist, d.h. xRy impliziert xR_Ly . Damit bestehen die Äquivalenzklassen bezüglich R_L aus der Vereinigung einiger Äquivalenzklassen bezüglich R . Folglich gilt $\text{ind}(R_L) \leq \text{ind}(R)$ und die Nerode-Äquivalenz R_L ist von endlichem Index. Sei also xRy . Wegen der Rechtsinvarianz von R folgt $xzRyz$ für jedes $z \in \Sigma^*$. Da nach Voraussetzung jede Äquivalenzklasse von R entweder ganz zu L gehört oder kein Wort aus L enthält, gehören entweder beide Wörter xz und yz zu L oder beide nicht zu L . Damit ist aber xR_Ly offensichtlich.
3. \rightarrow 1.: Aus der Nerode-Relation konstruieren wir einen endlichen Automaten $A_{R_L} := (Z_{R_L}, \Sigma, \delta, Z_{\text{end}})$ der L erkennt, wie folgt: Jeder der endlich vielen Äquivalenzklassen $[w]$ für $w \in \Sigma^*$ wird der Zustand $z_{[w]}$ zugeordnet, so dass die Mengen Z_{R_L} und Σ^*/R_L zueinander isomorph sind. Der Anfangszustand ist $z_{[\lambda]}$ und als Endzustände werden diejenigen Zustände $z_{[w]}$ gewählt, für die $w \in L$ ist. Diese Menge ist wohl definiert, denn aus der Definition von R_L folgt, dass entweder alle Wörter einer Äquivalenzklasse $[w]_{R_L}$ in L enthalten sind oder aber gar keines. Auch die Definition der Übergangsfunktion δ durch

$$\forall z_{[w]} \in Z_{R_L} : \forall x \in X : \delta(z_{[w]}, x) := z_{[wx]}$$

ist wohldefiniert, denn mit $[w_1] = [w_2]$ gilt $w_1R_Lw_2$ und wegen der Rechtsinvarianz von R_L folgt auch $w_1xR_Lw_2x$ und somit $[w_1x] = [w_2x]$ für jedes $x \in X$. Die Folgezustände von $z_{[v]} \in \Sigma^*/R_L$ bei Eingabe von $w \in \Sigma^*$ ergeben sich als: $(z_{[v]})^w := z_{[vw]}$ und mit $(z_{[\lambda]})^w = z_{[w]} \in Z_{\text{end}}$ wird jedes Wort $w \in L$ von dem neuen Automaten akzeptiert. Als Tupel notiert schreibt sich der eben definierte endliche Automat als $A_{R_L} := (Z_{R_L}, X, \delta_{R_L}, z_{[\lambda]}, Z_{\text{end}})$ mit:

$$\begin{aligned} Z_{R_L} &:= \{z_{[w]} \mid [w] \in \Sigma^*/R_L\}, \\ \delta_{R_L} &: Z \times X \longrightarrow Z \text{ mit } \delta(z_{[w]}, x) := z_{[wx]} \text{ und} \\ Z_{\text{end}} &:= \{z_{[w]} \mid w \in L\}. \end{aligned}$$

Dieser endliche Automat ist ein initial zusammenhängender und vollständiger DFA!

□

Als Anwendung von Theorem 3.43 wir nun zeigen, dass eine einfache Sprache nicht regulär ist:

3.44 Korollar

Die Sprache $DUP := \{a^i b^i \mid i \in \mathbb{N}\}$ ist nicht regulär.

Beweis: Betrachtet man einige (nicht alle, aber unendlich viele!) Äquivalenzklassen von R_L , so sieht man

$$[ab] = DUP = \{a^i b^i \mid i \in \mathbb{N}\}$$

$$[a^2 b] = \{a^{i+1} b^i \mid i \in \mathbb{N}\}$$

$$[a^3 b] = \{a^{i+2} b^i \mid i \in \mathbb{N}\}$$

⋮

$$[a^k b] = \{a^{i+k-1} b^i \mid i \in \mathbb{N}\}$$

⋮

Man sieht, dass $[a^i b] \neq [a^j b]$ für $i \neq j$ gilt: $a^i b \not\equiv_{R_L} a^j b$, denn mit $w := b^{i-1}$ gilt $a^i b w \in DUP$ aber $a^j b w \notin DUP$. Somit sind also die unendlich vielen Klassen $[a^k b]$ der Nerode-Äquivalenz R_L paarweise verschieden, und letztere daher nicht von endlichem Index. □

3.7.2. Minimale Automaten

Wie im wirklichen Leben, sind wir auch in der theoretischen Informatik daran interessiert, möglichst knappe Darstellungen zu verwenden. Zum Beispiel würde jeder Programmierer eine Sequenz von Anweisungen der Form „ $X := X+1$ “, „ $X := X-1$ “, „ $X := X+1$ “, „ $X := X-1$ “ einfach aus einem Programm streichen, da sie a m Ergebnis der Berechnung nichts ändert! Hier wollen wir insbesondere überflüssige Komponenten bei der Beschreibung von Sprachen vermeiden und suchen also nach möglichst kleinen Automaten für die zu beschreibende Sprache. Ein Beispiel für unterschiedlich große Automaten, welche dieselbe Sprache akzeptieren, war in Abbildung 3.14 auf Seite 75 gegeben.

3.45 Definition

Zu jedem endlichen Automaten (NFA) $A = (Z, \Sigma, K, Z_{start}, Z_{end})$ heiße die Menge $L(z) := \{w \in \Sigma^* \mid z \xrightarrow[w]{*} z', z' \in Z_{end}\}$ die **Leistung** des Zustandes z .

Zwei Zustände von A heißen genau dann **äquivalent**, wenn $L(z) = L(z')$ ist. Wir notieren dies in der Form $z \equiv z'$. Nicht äquivalente Zustände, $z \not\equiv z'$, nennen wir **unterscheidbar**.

A heißt genau dann **reduziert**, wenn keine zwei verschiedenen Zustände äquivalent sind.

3.46 Definition

Ein endlicher Automat heißt genau dann **minimal**, wenn er

1. vollständig,

3. Endliche Automaten und reguläre Mengen

2. *initial zusammenhängend*,
3. *deterministisch und*
4. *reduziert ist*.

Diese Definition ist insofern merkwürdig, als bisher doch völlig unklar ist, wieso die Bezeichnung *minimal* gerechtfertigt ist. Wir werden also zunächst zeigen, dass kein vollständiger DFA, der zu einem minimalen Automaten äquivalent ist, weniger Zustände als dieser haben kann. Damit ist die Bezeichnung *minimal* gerechtfertigt. Sodann werden wir sehen, dass es (im Wesentlichen) nur *genau einen* minimalen DFA geben kann, der die gleiche Sprache akzeptiert.

3.47 Theorem

Ein vollständiger DFA $A = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ist genau dann minimal, wenn kein äquivalenter, vollständiger DFA weniger Zustände besitzt.

Beweis: Es sind zwei Richtungen zu zeigen:

„ \Rightarrow “: Wenn kein äquivalenter, vollständiger DFA weniger Zustände als A besitzt, so muss A entsprechend Definition 3.46 minimal sein.

„ \Leftarrow “: Wenn A minimal ist und $B = (Z', X, \delta', z'_0, Z'_{\text{end}})$ ein zu A äquivalenter vollständiger DFA, dann gilt $|Z| \leq |Z'|$.

zu „ \Rightarrow “: Wir zeigen die logisch äquivalente Kontraposition (A nicht minimal \rightarrow es gibt äquivalenten vollständigen DFA B mit weniger Zuständen) der behaupteten Implikation. Wenn $A = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ *nicht* minimal ist, so kann das nach Definition 3.46 nur daran liegen, dass der vollständige DFA A entweder nicht initial zusammenhängend ist (2. der Def.), oder äquivalente Zustände besitzt (4. der Def.). Falls A nicht initial zusammenhängend ist, so gibt es mindestens einen Zustand, der von z_0 aus nicht erreicht wird, also entfernt werden kann ohne die akzeptierte Sprache zu verändern. Der neue Automat wäre ein äquivalenter vollständiger DFA mit weniger Zuständen. Falls A nicht reduziert ist, so gibt es zwei verschiedene, aber äquivalente Zustände $p, q \in Z$. Identifiziert man diese Zustände miteinander, so erhält man wieder einen äquivalenten vollständigen DFA mit kleinerer Zustandsmenge.

zu „ \Leftarrow “: Wir werden eine surjektive Abbildung $g : Z' \rightarrow Z$ finden, womit dann alles gezeigt ist.

Es sei für jedes $z' \in Z'$ $g(z') := (z_0)^w$, falls in B $z'_0 \xrightarrow{*}_w z'$ für irgend ein Wort w gilt. Wir zeigen:

(i) g ist wohldefiniert, d.h tatsächlich eine Abbildung.

(ii) g ist surjektiv.

zu (i): Seien $w, v \in \Sigma^*$ mit $w \neq v$ derart, dass in B sowohl $z'_0 \xrightarrow{*}_w z'$ als auch $z'_0 \xrightarrow{*}_v z'$ gilt. Wir müssen einsehen, dass dann auch $(z_0)^w$ und $(z_0)^v$ in A der gleiche Zustand ist:

Zunächst ist $L(z') = L((z_0)^w) = L((z_0)^v)$, denn für ein Wort wu gilt $wu \in L(A)$ mit $L(A) = L(B) = L(z'_0)$ genau dann, wenn $u \in L((z_0)^w)$ gdw. $u \in L((z_0)^v)$ gdw. $u \in L(z')$ ist (A und B sind beide vollständige DFA's!). Somit sind also $(z_0)^w$ und $(z_0)^v$ im minimalen DFA

A äquivalente Zustände, müssen also identisch sein. Damit ist die Definition von g mittels $g(z') := (z_0)^w$, falls $z'_0 \xrightarrow[w]{*} z'$ in B möglich ist, unabhängig von der Wahl des Wortes w und mithin eindeutig.

zu (ii): Es ist zu zeigen, dass es zu jedem Zustand $z \in Z$ ein $z' \in Z'$ gibt, so dass $g(z') = z$ ist.

Da A als minimaler Automat initial zusammenhängend ist, gibt es zu jedem $z \in Z$ wenigstens ein Wort $w \in \Sigma^*$ mit $z = (z_0)^w$. Da B vollständig sein sollte, gibt es in B für dieses w dann immer auch einen Zustand $z' \in Z'$ mit $z' = (z'_0)^w$ und also gilt nach Definition gerade $g(z') = (z_0)^w = z$. Dies liefert die verlangte Surjektivität von g . □

Als nächstes wollen wir zeigen, dass minimale DFA *eindeutig* bestimmt sind, d.h. je zwei äquivalente minimale DFA sind modulo Namensgebung der Zustände identisch.

3.48 Theorem

Je zwei äquivalente, minimale DFA sind isomorph, d.h., sie besitzen bis auf die Bezeichnungen der Zustände das gleiche Zustandsdiagramm.

Beweis: Seien A_1 und A_2 mit $A_i := (Z_i, X, \delta_i, z_{0,i}, Z_{\text{end},i}), i = 1, 2$ zwei minimale Automaten die die gleiche Sprache $L := L(A_1) = L(A_2)$ akzeptieren. Da ein minimaler endlicher Automat gemäß seiner Definition 3.46 ein vollständiger und deterministischer endlicher Automat ist, muss nach Theorem 3.47 zunächst $|Z_1| = |Z_2|$ gelten. Um nun eine Bijektion $f : A_1 \rightarrow A_2$ zwischen den Zustandsgraphen herzustellen, definieren wir $f(z_{0,1}) := z_{0,2}$, $Z_{\text{end},2} := \{f(z) \mid z \in Z_{\text{end},1}\}$ und $\forall x \in X, z \in Z_1 : f(\delta_1(z, x)) := \delta_2(f(z), x)$. Die Abbildung f bildet also jede Kante des Zustandsgraphen von A_1 auf eine Kante des Zustandsgraphen von A_2 ab. Tatsächlich ist f ein Isomorphismus zwischen den Kantenmengen. Wenn $(z_{0,1}, w_1, z_{1,1}), (z_{1,1}, w_2, z_{2,1}), \dots, (z_{n-1,1}, w_n, z_{n,1})$, mit $(z_{i-1,1}, w_i, z_{i,1})$ eine Erfolgsrechnung in A_1 ist, so ist auch $(f(z_{0,1}), w_1, f(z_{1,1})), \dots, (f(z_{n-1,1}), w_n, f(z_{n,1}))$ eine Erfolgsrechnung in A_2 und umgekehrt. □

Der minimale endliche Automat ist weiterhin (bis auf Isomorphie) identisch mit dem initial zusammenhängenden, vollständigen DFA A_{R_L} der im Beweis zu Theorem 3.42, im Schritt **3.** \rightarrow **1.**, konstruiert wurde.

3.49 Korollar

Der im Beweis zu Theorem 3.43 konstruierte endliche Automat A_{R_L} ist ein minimaler DFA.

Beweis: Sei $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein beliebiger vollständiger DFA mit $L := L(A)$. Nach Theorem 3.43, Beweisschritt 1. \rightarrow 2., existiert eine rechtsinvariante Äquivalenzrelation R_A und L ist endliche Vereinigung jener Äquivalenzklassen **mod** R_A die zu den Endzuständen von A gehören. Damit folgt $\text{ind}(R_A) \leq |Z|$. Nach dem Beweis von 2. \rightarrow 3. folgt, dass R_L eine Vergrößerung von R_A ist, mithin $\text{ind}(R_L) \leq \text{ind}(R_A)$ gilt. Aus dem Beweisschritt 3. \rightarrow 1. folgt, dass $\text{ind}(R_L) = |Z_{R_L}|$ und also $|Z_{R_L}| \leq |Z|$ ist. □

Wir können nun mit Fug und Recht von *dem* minimalen endlichen Automaten sprechen, wohl wissend, dass nach den Theoremen 3.29 und 3.50 *nicht*deterministische endliche Automaten logarithmisch viel kleiner sein können! Dies zeigen wir im Beweis zu Theorem 3.50

3. Endliche Automaten und reguläre Mengen

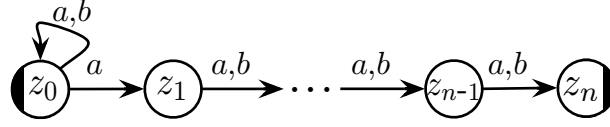


Abbildung 3.15.: Ein NFA für die Sprache $L_n := \{w \in \{a, b\}^* \mid \text{das } n\text{-letzte Symbol in } w \text{ ist } a\}$

3.50 Theorem

- (i) Es gibt einen λ -freien, buchstabierenden NFA mit $n + 1$ Zuständen, der die Sprache $L_n := \{w \in \{a, b\}^* \mid \text{das } n\text{-letzte Symbol in } w \text{ ist ein } a\}$ akzeptiert.
- (ii) Jeder DFA, der die Menge L_n akzeptiert hat 2^n Zustände.

Beweis:

zu (i): Der NFA aus Abbildung 3.15 akzeptiert die Sprache L_n .

zu (ii): Sei $B = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein DFA mit $L(B) = L_n$ und seien $w_1, w_2 \in \{a, b\}^n$ verschiedene Wörter. Wir zeigen $(z_0)^{w_1} \neq (z_0)^{w_2}$: Da w_1 und w_2 verschieden sind, gibt es mindestens einen Buchstaben, an dem sich die beiden Wörter unterscheiden, d.h., es gibt Wörter $u_1, u_2, v \in \{a, b\}^*$ mit o.B.d.A. $w_1 = u_1av$ und $w_2 = u_2bv$. Für $v' \in \{a, b\}^*$ mit $|v'| := n - |v| - 1$ gilt nun 1.): $w_1v' \in L_n$ und 2.): $w_2v' \notin L_n$, denn der n -letzte Buchstabe von $w_1v' = u_1avv'$ ist ein a aber der von $u_2bv v'$ nicht. Wäre nun aber $(z_0)^{w_1} = (z_0)^{w_2} = z'$, so hieße dies doch $(z')^{v'} \in Z_{\text{end}}$ wegen 1.) und gleichzeitig auch $(z')^{v'} \notin Z_{\text{end}}$ wegen 2.). Dies ist aber ein Widerspruch, folglich kann die Annahme $(z_0)^{w_1} = (z_0)^{w_2}$ nicht richtig gewesen sein. Da es 2^n paarweise verschiedene Wörter der Länge n in $\{a, b\}^*$ gibt, folgt die Existenz von ebenso vielen verschiedenen Zuständen im Automaten B .

□

Auf die gleiche Weise können wir nun auch Theorem 3.29 beweisen.

Beweis (zu Theorem 3.29): Sei $K_n := L(A_n)$ die vom NFA A_n akzeptierte Sprache. Es ist leicht zu sehen, dass neben dem leeren Wort λ kein anderes Wort der Länge kleiner als n akzeptiert wird und alle nicht leeren Wörter $w \in K_n$ die Form $w = uav$ mit $v \in \{a, b\}^{n-1}$ besitzen.

□

Ein minimaler DFA besitzt keine äquivalenten Zustände. Daher reicht es aus, in einem gegebenen initial zusammenhängenden vollständigen deterministischen endlichen Automaten äquivalente Zustände zu identifizieren, um einen äquivalenten und minimalen DFA mit einer neuen Zustandsmenge zu konstruieren. Jeder Zustand in diesem neuen Automaten ist die Äquivalenzklasse der Zustände gleicher Leistung des alten Automaten.

3.51 Definition

Sei $A = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein DFA, und für jeden Zustand $z \in Z$ seine Äquivalenzklasse $[z] := \{z' \mid z' \equiv z\}$ die Menge aller Zustände mit gleicher Leistung (vgl. Def. 3.41). Der **Äquivalenzautomat** A' zu A ist wie folgt erklärt: $A' := (Z', \Sigma, \delta', [z_0], Z'_{\text{end}})$, wobei $Z' := \{[z] \mid z \in Z\}$, $Z'_{\text{end}} := \{[z] \mid z \in Z_{\text{end}}\}$ und $\delta'([z_1]_A, x) = [z_2]_A$ genau dann, wenn $\delta(p, x) = q$ für irgendwelche Zustände $p \in [z_1]$ und $q \in [z_2]$ gilt.

3.52 Theorem

Der Äquivalenzautomat A' zu einem initial zusammenhängenden vollständigen DFA A ist minimal und akzeptiert die gleiche Sprache.

Beweis: Aus Definition 3.41 folgt sofort, dass die Endzustandsmenge wohldefiniert ist, denn es impliziert $z \equiv z'$ entweder $z, z' \in Z_{\text{end}}$ oder $z, z' \notin Z_{\text{end}}$. Die neue Übergangsfunktion δ' ist wohldefiniert, weil aus $z \equiv z'$ für jedes $a \in X$ stets $(z)^a \equiv (z')^a$ folgt. Unter Rückgriff auf Definition 3.41 sieht man dies leicht:

- (1) $z \equiv z'$ impliziert $L(z) = L(z')$ oder
- (2) $\forall w \in \Sigma^* : (z)^w \in Z_{\text{end}}$ genau dann, wenn $(z')^w \in Z_{\text{end}}$. Also auch
- (3) $\forall a \in X, \forall w \in \Sigma^* : (z)^{aw} \in Z_{\text{end}}$ genau dann, wenn $(z')^{aw} \in Z_{\text{end}}$ und es folgt
- (4) $(z)^a \equiv (z')^a$.

Es ist noch zu zeigen, dass A und A' die gleiche Sprache akzeptieren:

Sei dazu $w \in \Sigma^*$ und z_1, z_2, \dots, z_n die von A beim Lesen von w durchlaufene Zustandsfolge. Nach der Definition von A' , speziell K' , durchläuft A' die Zustandsfolge $[z_1], [z_2], \dots, [z_n]$. A akzeptiert w , genau dann, wenn $z_n \in Z_{\text{end}}$ ist. Der Äquivalenzklassenautomat akzeptiert w genau dann, wenn $[z_n] \in Z'_{\text{end}}$ ist. Entsprechend der Definition gilt aber $z_n \in Z_{\text{end}}$ genau dann, wenn $[z_n] \in Z'_{\text{end}}$ ist.

Schließlich muss gezeigt werden, dass der Äquivalenzautomat minimal ist:

Dies folgt aus Korollar 3.49, wenn wir zeigen, dass der Äquivalenzautomat höchstens $\text{ind}(R_L)$ viele Zustände besitzt. Dies folgt nun aber schon, wenn im Äquivalenzautomaten zwei Wörter w_1 und w_2 mit $w_1 R_L w_2$ stets zum gleichen Zustand $[(z_0)^{w_1}] = [(z_0)^{w_2}]$ führen, oder formaler: $\forall w_1, w_2 \in \Sigma^* : (w_1 R_L w_2) \rightarrow ((z_0)^{w_1} \equiv (z_0)^{w_2})$.

$$\begin{aligned}
 w_1 R_L w_2 &\rightarrow \forall v \in \Sigma^* : w_1 v \in L \leftrightarrow w_2 v \in L \\
 &\rightarrow \forall v \in \Sigma^* : (z_0)^{w_1 v} \in Z_{\text{end}} \leftrightarrow (z_0)^{w_2 v} \in Z_{\text{end}} \\
 &\rightarrow \forall v \in \Sigma^* : ((z_0)^{w_1})^v \in Z_{\text{end}} \leftrightarrow ((z_0)^{w_2})^v \in Z_{\text{end}} \\
 &\rightarrow (z_0)^{w_1} \equiv (z_0)^{w_2} \\
 &\rightarrow [(z_0)^{w_1}] = [(z_0)^{w_2}]
 \end{aligned}$$

□

Bei Kenntnis der Äquivalenzklassen bezüglich \equiv kann der Äquivalenzklassenautomat in einer Zeit proportional zum Produkt $|Z| \cdot |X|$ konstruiert werden. Schwieriger scheint die Bestimmung der Relation \equiv zu sein, denn gemäß der Definition müssen immer alle Zustände und die unendliche Menge aller Wörter in Σ^* betrachtet werden.

Man kann den minimalen DFA zu jedem gegebenen DFA mit n Zuständen in $c \cdot n \cdot \log n$ Schritten konstruieren. Die hier dargestellte einfachere Methode (siehe z.B. [Wegener] oder [Hopcroft&Ullman]) arbeitet mit etwas schlechteren $d \cdot n^2$ Schritten. Die jeweiligen Konstanten c und d hängen dabei stets auch von der Größe des verwendeten Eingabealphabets Σ ab. Man macht sich bei diesem Verfahren zu Nutze, dass es leichter ist, die Nichtäquivalenz zweier Zustände z und z' zu bezeugen, denn dazu benötigt man nur ein sogenanntes **z und z' unterscheidendes Wort** $w \in \Sigma^*$ mit $(z)^w \in Z_{\text{end}}$ und $(z')^w \notin Z_{\text{end}}$ oder umgekehrt, mit $(z)^w \notin Z_{\text{end}}$ und $(z')^w \in Z_{\text{end}}$.

3.53 Algorithmus (Bestimmung äquivalenter Zustände eines vDFA)

Eingabe: $A = (Z, \Sigma, K, z_0, Z_{\text{end}})$ ein DFA mit $Z = \{z_1, z_2, \dots, z_n\}$.

3. Endliche Automaten und reguläre Mengen

Ausgabe: Markierung aller Zustandspaarmengen $\{z_i, z_j\}$ mit $z_i \not\equiv z_j$ und $i < j$.

Ein gerichteter Graph $G_A := (V, E)$ mit $V := \{v \mid v \in 2^Z \text{ und } |v| = 2\}$ wird als Datenstruktur verwendet. Die Knoten in V sind also Paarmengen $\{z_i, z_j\}$ von unterschiedlichen Zuständen und können gefärbt werden. Zu Beginn ist kein Knoten von G_A gefärbt. Im weiteren Verlauf werden gerichtete Kanten entgegen der Richtung der Kanten im DFA gezogen oder entfernt und ungefärbte Knoten weiß oder schwarz gefärbt. Die geschwärzten Knoten geben dann jeweils Paare von inäquivalenten Zuständen an.

Initialisierung: Jeder Knoten $\{z_i, z_j\}$, für den entweder ($z_i \in Z_{\text{end}}$ und $z_j \notin Z_{\text{end}}$) oder aber ($z_i \notin Z_{\text{end}}$ und $z_j \in Z_{\text{end}}$) ist, wird schwarz gefärbt. D.h., das die Zustände z_i und z_j unterscheidende Wort ist λ .

Falls sich im weiteren Verlauf der Konstruktion irgendwann z_i und z_j als inäquivalent erweisen, so können alle Knoten, die von dem geschwärzten Knoten $\{z_i, z_j\}$ aus über noch zu definierende Kanten erreichbar sind, ebenfalls als inäquivalent betrachtet und daher auch schwarz gefärbt werden.

Färbekonstruktion: Solange noch ungefärbte Knoten in dem Graphen existieren, tue man folgendes:

Schritt 1: Wähle einen ungefärbten Knoten $\{p, q\}$ und färbe diesen weiß.

Schritt 2: Für jedes $a \in \Sigma$, für das $(p)^a \neq (q)^a$ ist, überprüfe, ob der Knoten $\{(p)^a, (q)^a\}$ schwarz markiert ist.

wenn ja, so wird auch $\{p, q\}$ schwarz markiert und alle von diesem Knoten über Kanten erreichbaren noch nicht schwarzen Knoten ebenfalls. Danach werden alle Kanten, die im Knoten $\{p, q\}$ enden, entfernt.

wenn nein, so zeichne eine Kante $\{(p)^a, (q)^a\} \rightarrow \{p, q\}$.

Ergebnis: Wenn der Knoten $\{p, q\}$ am Ende weiß ist, so gilt $p \equiv q$.

Wenn der Knoten $\{p, q\}$ am Ende schwarz ist, so gilt $p \not\equiv q$

Beweis der Korrektheit von Algorithmus 3.53:

1. Zunächst beweisen wir die Termination des Algorithmus:

Die „Solange“-Schleife terminiert, denn jeder Knoten wird höchstens einmal weiß und höchstens einmal schwarz gefärbt.

2. Zum Beweis der Korrektheit zeigen wir:

(i) Falls $\{p, q\}$ schwarz ist, dann folgt $p \not\equiv q$, und

(ii) falls $p \not\equiv q$ ist, so ist der Knoten $\{p, q\}$ am Ende schwarz gefärbt.

Es ist festzuhalten, dass aus der Inäquivalenz zweier Zustände $p \not\equiv q$ stets die Existenz eines p und q unterscheidenden Wortes, eines sogenannten Zeugen (*witness*) $w \in \Sigma^*$ folgt.

zu (i): Wir führen eine Induktion über die Zahl d der Durchläufe der „Solange“-Schleife (Schritte 1 und 2) durch.

Induktions-Basis: Für $d = 0$ zeigt die Initialisierung, dass $\{p, q\}$ geschwärzt ist, weil λ das unterscheidende Wort ist.

Induktions-Annahme: Sei (i) richtig für alle in den ersten Schleifendurchläufen geschwärzten Knoten.

Induktions-Schritt: Im nächsten Durchlauf der Schritte 1 und 2 wird $\{p, q\}$ geschwärzt, falls einer der folgenden Fälle zutrifft:

direkte Färbung: $\exists x \in \Sigma : \{(p)^x, (q)^x\}$ war schon schwarz, d.h., es gibt einen Zeugen $w \in \Sigma^*$, der $(p)^x$ und $(q)^x$ unterscheidet. Mithin unterscheidet das Wort xw die beiden Zustände p und q und es gilt $p \neq q$.

indirekte Färbung: Ein anderer Knoten $\{s, t\}$ wird direkt geschwärzt und der Knoten $\{p, q\}$ ist auf einem Pfad von $\{s, t\}$ erreichbar, so dass daher $\{p, q\}$ geschwärzt wird. Nach der Definition des Ziehens neuer Kanten in Schritt 2 gibt es ein Wort $v \in \Sigma^*$ mit $s = (p)^v$ und $t = (q)^v$. Da nun aber $\{s, t\}$ nur dann geschwärzt wurde, wenn es ein s und t unterscheidendes Wort w gab, gilt für dieses w dann, dass genau einer der Zustände $(s)^w$ und $(t)^w$ ein Endzustand ist. Folglich ist auch genau einer der Zustände $((p)^v)^w$ und $((q)^v)^w$ ein Endzustand. Das p und q unterscheidende Wort ist also vw .

Damit ist (i) durch Induktion gezeigt.

zu (ii): Für jedes $p \neq q$ gibt es ein kürzestes Wort w , welches p und q unterscheidet. Wir wählen Induktion über die Länge k dieser Wörter.

Induktions-Basis: Gilt $|w| = 0$, so muss λ das unterscheidende Wort sein. Da der endliche Automat ein DFA war, kann nur genau einer der beiden Zustände ein Endzustand sein und in diesem Fall wurde der Knoten $\{p, q\}$ bei der Initialisierung geschwärzt. Einmal geschwärzte Knoten werden aber auch nicht wieder anders gefärbt.

Induktionsannahme: Alle Knoten $\{s, t\}$, bei denen s und t durch Wörter mit Längen bis zu k unterscheidbar waren, wurden irgendwann geschwärzt.

Induktionsschritt: Sei $p \neq q$ unterscheidbar mit dem kürzestem Wort $v = aw$ mit $|v| = k + 1$ und $a \in X$, $w \in \Sigma^*$. Dann unterscheidet w die Zustände $(p)^a$ und $(q)^a$ und der Knoten $\{(p)^a, (q)^a\}$ wird wegen der Induktionsannahme irgendwann geschwärzt werden. Irgendwann wird auch $\{p, q\}$ unter den noch nicht gefärbten Knoten ausgewählt. Wenn dann für irgend ein $x \in \Sigma$ ($x = a$ ist dabei möglich) der Knoten $\{(p)^x, (q)^x\}$ schon schwarz ist, so wird in direkter Färbung auch sogleich $\{p, q\}$ geschwärzt. Falls jedoch $\{(p)^x, (q)^x\}$ noch für kein $x \in \Sigma$ schwarz ist, dann werden die Kanten $\{(p)^x, (q)^x\} \rightarrow \{p, q\}$ für jedes $x \in \Sigma$ gezeichnet. Da nach der Induktionsannahme irgendwann der Knoten $\{(p)^a, (q)^a\}$ geschwärzt wird, wird zu diesem Zeitpunkt auch $\{p, q\}$ in indirekter Färbung geschwärzt.

Damit ist die Korrektheit des Färbealgorithmus gezeigt. \square

3.54 Theorem

Der Färbealgorithmus zum Auffinden äquivalenter Zustände arbeitet in $c \cdot |\Sigma| \cdot |Z|^2$ Schritten, wobei c eine Konstante ist.

Beweisskizze: Jeder Knoten wird in Schritt 1 maximal einmal betrachtet. Jedesmal werden in Schritt 2 dazu $|\Sigma|$ Knoten $\{(p)^x, (q)^x\}$ untersucht. Für jeden dieser Knoten kann $\{p, q\}$ und alle von hier aus

3. Endliche Automaten und reguläre Mengen

erreichbaren Knoten, das sind maximal $|Z|$ viele, besucht werden um diese zu schwärzen oder um Kanten zu ziehen. Insgesamt ergibt sich die Zahl von höchstens $c \cdot |X| \cdot |Z|^2$ Besuchen von einzelnen Knoten. Genauere Beweise finden sich in der Literatur. \square

Auf Grund diese Ergebnisses ist der minimale DFA in höchstens quadratischer Zeit aus einem DFA konstruierbar.

Bei besserer Programmierung kann jedoch ein Algorithmus formuliert werden, der das Finden äquivalenter Zustände sogar in nur $c \cdot |Z| \cdot \log(|Z|)$ vielen Schritten ermöglicht. Wir wollen an dieser Stelle noch nicht auf diese Methode des *Dynamischen Programmierens* eingehen. Dazu verweisen wir auf andere Veranstaltungen, zum Beispiel auch auf die Vorlesung F3 („Berechenbarkeit und Komplexität“) in diesem Zyklus.

3.8. Das Pumping-Lemma

Die Frage, ob alle Mengen regulär sind, muss sofort verneint werden! Eine spezielle Menge, die Sprache DUP hatten wir in Korollar 3.44 bereits kennengelernt und als nicht-regulär nachgewiesen.

Um von einer beliebig vorgegebenen Menge von Wörtern nachzuweisen, dass es keinen endlichen Automaten geben kann, der sie akzeptiert, gibt es jedoch noch weitere, einfacher anzuwendende Methoden. Als wichtiges Hilfsmittel dient hier das sogenannte „uvw-Theorem“ (*pumping lemma*).

3.55 Theorem (Pumping-Lemma der regulären Sprachen, *uvw*-Theorem)

Zu jeder regulären Menge $R \in \text{Reg}$ gibt es eine Zahl $n \in \mathbb{N}$, so dass jedes Wort $z \in R$ mit $|z| \geq n$ zerlegt werden kann in die Form $z = uvw$ mit:

- (i) $|uv| \leq n$
- (ii) $|v| \geq 1$ und
- (iii) $\{u\}\{v\}^*\{w\} \subseteq R$

Beweis: Sei $A = (Z, \Sigma, K, z_0, Z_{\text{end}})$ ein DFA der R akzeptiert, d.h.: $L(A) = R$. Wähle $n := |Z|$. Sei nun $z = x_1 x_2 \dots x_m \in R$, mit $m \geq n$ und $x_i \in \Sigma$, ein beliebiges Wort der Sprache. Dann sind doch die $m + 1$ Zustände $z_0, (z_0)^{x_1}, (z_0)^{x_1 x_2}, \dots, (z_0)^{x_1 x_2 \dots x_m}$ alle von z_0 aus erreichbar und werden auf diesem Erfolgspfad für die Akzeptierung von z besucht. Da aber nur maximal $n \leq m$ viele davon verschieden sein können, muss nach dem Schubfachprinzip (vergl. 2.19) spätestens mit dem $(n + 1)$ -ten, ein Zustand in dieser Folge zweimal vorgekommen sein.

Sei also für $0 \leq i < j \leq n$ gerade $(z_0)^{x_1 x_2 \dots x_i} = (z_0)^{x_1 x_2 \dots x_j}$, dann finden wir mit $u := x_1 x_2 \dots x_i$, $v := x_{i+1} x_{i+2} \dots x_j$ und $w := x_{j+1} x_{j+2} \dots x_m$ gerade die verlangte Zerlegung von z , die (i) und (ii) erfüllt. Dass auch (iii) gilt, sieht man leicht ein, denn mit $z' := (z_0)^u$ gilt doch $z' = (z')^\lambda = (z')^v = (z')^{vv} = \dots$. Also werden alle Wörter der Form $u \cdot v^i \cdot w$, für jedes $i \geq 0$, von A ebenfalls akzeptiert. \square

Da jede reguläre Menge auch von einem minimalen DFA akzeptiert wird, kann für die Zahl n im *uvw*-Theorem die Anzahl der Zustände des minimalen Automaten, der $R \in \text{Reg}$ akzeptiert, angenommen werden.

Obwohl die Folgerung des uvw -Theorems (d.h. die Aufteilbarkeit der „langen“ Wörter mit den angegebenen Eigenschaften) für alle regulären Mengen gilt, gibt es auch nicht-reguläre Mengen, für die diese Aussagen gelten. Daher kann mit Hilfe des uvw -Theorems nicht bewiesen werden, dass eine Menge tatsächlich regulär ist, sondern nur umgekehrt, dass sie *nicht* regulär sein kann!

3.56 Beispiel (Anwendung des uvw -Theorems)

Wir wollen mit Hilfe des uvw -Theorems zeigen, dass die Sprache $DUP := \{a^n b^n \mid n \in \mathbb{N}\}$ nicht regulär ist: Wäre DUP regulär, so gäbe es eine Zahl $k \in \mathbb{N}$, für die die Eigenschaften des Pumping-Lemmas zuträfen (genauer: für die die Konklusion des Pumping Lemmas zuträfe). Sei also $z := a^k b^k \in DUP$ eines der Wörter mit $|z| > k$. Jede Zerlegung von z als $z = uvw$ mit $|uv| \leq k$ bedeutet $v = a^m$ für ein $m \in \mathbb{N}$ mit $1 \leq m \leq k$. Damit müßte aber auch das Wort $a^{k-m} b^k$ ein Element der Sprache DUP sein, was nicht stimmt. Da alle Schritte korrekt aus der Annahme abgeleitet wurden, und dann ein Widerspruch entstand, kann nur die Annahme falsch gewesen sein. Damit ist $DUP \notin Reg$ mit einer weiteren Methode nachgewiesen.

Die im uvw -Theorem gefolgerten Eigenschaften treffen zum Beispiel auf die Sprache $L := \{c^k a^l b^m \mid k, l, m \in \mathbb{N} : k = 0 \text{ oder } l = m\}$ zu. Der Versuch, mit dem uvw -Theorem zu beweisen, dass L nicht regulär ist muss deshalb fehlschlagen. Wenn nämlich $z \in L$ den Buchstaben „c“ nicht enthält, so gilt dies auch für alle Wörter $uv^i w$, ganz gleich welche Zerlegung gewählt wurde.

Enthält das Wort z das Symbol „c“, so wählen wir eine Zerlegung mit $u = \lambda$, $v = c$ und w als geeignetem Suffix. Alle Wörter der Form $uv^i w$ sind dann wieder Elemente der Menge L . Also kommt kein Widerspruch zu der Annahme L sei regulär zustande, obwohl L eben nicht regulär ist, wie wir später auf Seite 91 zeigen werden.

Alle bisher definierten Klassen von Sprachen waren in Sinne von Definition 3.5 Sprachfamilien. Operationen, die man auf Sprachen anwendet, wie z.B. den mengentheoretischen Durchschnitt kann man zwar auch auf die Sprachfamilien anwenden, da diese ja selbst Mengen sind, jedoch ist dies im allgemeinen nicht das Ziel. Offensichtlich ergeben sich dabei wenig neue Gesichtspunkte: $Rat \cap Reg = Rat$ ist bekannt, denn jede rationale Menge ist auch eine reguläre Menge.

3.9. Operatoren auf Sprachfamilien

Was wir eigentlich betrachten wollen, ist hier eher der Durchschnitt der Sprachen aus den beiden Familien. Man bezeichnet in der AFL-Theorie (*theory of abstract formal languages*) dann auch korrekterweise diese Operationen auf den Sprachen als **Operatoren** auf den Sprachfamilien, um so den Unterschied deutlich zu machen. Man einigte sich auf folgende Notation:

3.57 Definition

Für Sprachfamilien \mathcal{L}_1 und \mathcal{L}_2 sei

$$\begin{aligned}\mathcal{L}_1 \vee \mathcal{L}_2 &:= \{L \mid \exists L_1 \in \mathcal{L}_1, \exists L_2 \in \mathcal{L}_2 : L = L_1 \cup L_2\} \\ \mathcal{L}_1 \wedge \mathcal{L}_2 &:= \{L \mid \exists L_1 \in \mathcal{L}_1, \exists L_2 \in \mathcal{L}_2 : L = L_1 \cap L_2\} \\ \mathcal{L}_1^* &:= \{L^* \mid L \in \mathcal{L}_1\}\end{aligned}$$

3. Endliche Automaten und reguläre Mengen

Obschon damals noch nicht so notiert, zeigten wir $\text{Reg} \vee \text{Reg} = \text{Reg}$ in Theorem 3.31 und offensichtlich gilt nach Theorem 3.33 auch $\text{Reg}^* := \{L^* \mid L \in \text{Reg}\} = \text{Reg}$, wobei wir den Operator $*$ genauso bezeichnen können wie den Stern-Abschluss (Kleene'sche-Hülle) einer formalen Sprache. Letzteres Vorgehen nennt man auch **kanonische Erweiterung** einer Operation.

Neben diesen offensichtlichen Eigenschaften regulärer Mengen sind jedoch auch besonders die mengentheoretische Komplementbildung (bzw. der Differenzbildung) und der Durchschnitt bedeutsam.

3.58 Theorem

Mit $L \in \text{Akz}(\Sigma)$ ist auch $\bar{L} := \Sigma^* \setminus L \in \text{Akz}(\Sigma)$.

Beweis: Sei $A := (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein vollständiger DFA mit $L = L(A)$. Den vDFA $C_{\bar{A}}$ mit $L(C_{\bar{A}}) = \bar{L}$ erhält man sofort als $C_{\bar{A}} := (Z, \Sigma, \delta, z_0, Z \setminus Z_{\text{end}})$. Da A vollständig war, ist nun jede der nicht akzeptierenden Rechnungen von A eine Erfolgsrechnung in $C_{\bar{A}}$ und jede Erfolgsrechnung von A ist in $C_{\bar{A}}$ nicht mehr akzeptierend. \square

Da nach den Gesetzen von de Morgan $A \cap B = \overline{\bar{A} \cup \bar{B}}$ gilt, wissen wir nach dem Vorangegangenen (siehe: Theorem 3.31) auch, dass der Durchschnitt zweier regulärer Mengen wieder regulär sein muss. Die Hintereinanderausführung der bisher kennengelernten Umformungen der endlichen Automaten wäre allerdings zur Konstruktion eines Automaten $C_{A \cap B}$ mit $L(C_{A \cap B}) = L(A) \cap L(B)$ sehr zeitaufwendig und unbefriedigend. Wir wollen daher die wichtige Konstruktion des Produktautomaten vorstellen.

3.59 Theorem

Mit $L_1 \in \text{Akz}(\Sigma_1)$ und $L_2 \in \text{Akz}(\Sigma_2)$ ist auch $L_1 \cap L_2 \in \text{Akz}(\Sigma_1 \cap \Sigma_2)$.

Beweis (Konstruktion eines vDFA für $L_1 \cap L_2$): Seien $A := (Z_1, \Sigma_1, \delta_1, z_{1,0}, Z_{1,\text{end}})$ und $B := (Z_2, \Sigma_2, \delta_2, z_{2,0}, Z_{2,\text{end}})$ vDFA's mit $L_1 = L(A)$ und $L_2 = L(B)$. Der vDFA $C_{A \cap B} := (Z_3, \Sigma_3, \delta_3, z_{3,0}, Z_{3,\text{end}})$ mit $L(C_{A \cap B}) = L_1 \cap L_2$ wird **Produktautomat** genannt, weil dessen Zustandsmenge das kartesische Produkt der Zustandsmengen Z_1 und Z_2 ist. Seine Bestandteile sind folgendermaßen spezifiziert:

$$\begin{aligned} Z_3 &:= Z_1 \times Z_2, \\ \Sigma_3 &:= \Sigma_1 \cap \Sigma_2, \\ \delta_3((z_1, z_2), x) &:= (\delta_1(z_1, x), \delta_2(z_2, x)) \text{ für alle Zustände } (z_1, z_2) \in Z_3, \\ z_{3,0} &:= (z_{1,0}, z_{2,0}), \\ Z_{3,\text{end}} &:= Z_{1,\text{end}} \times Z_{2,\text{end}}. \end{aligned}$$

Es ist offensichtlich, dass zu jeder Erfolgsrechnung für ein Wort $w \in L_1 \cap L_2$ Erfolgsrechnungen in A und in B existieren, für die in der ersten und in der zweiten Komponente die entsprechende Erfolgsrechnung in $C_{A \cup B}$ existiert. Umgekehrt, entspricht einer Erfolgsrechnung in $C_{A \cup B}$, die im Zustand (z_1, z_2) mit $z_1 \in Z_{1,\text{end}}$ und $z_2 \in Z_{2,\text{end}}$ endet, sowohl eine Erfolgsrechnung in A als auch eine ebensolche in B . \square

3.60 Definition

Eine Sprachfamilie \mathcal{L} ist **abgeschlossen unter einem Operator** $o : 2^X \times 2^X \longrightarrow 2^X$ genau dann, wenn $o(L_1, L_2) \in \mathcal{L}$, für alle $L_1, L_2 \in \mathcal{L}$ gilt.

Wir nennen in diesem Fall den Operator o einen **Abschlußoperator** der Familie \mathcal{L} und sagen, dass die Familie \mathcal{L} gegenüber o abgeschlossen ist.

Wir hatten bisher gesehen, dass die Familie $\mathcal{R}eg$ der regulären Mengen abgeschlossen ist gegenüber den Operatoren: Vereinigung (\vee), (Komplex-)Produkt, Sternbildung, Komplement, Mengendifferenz und Durchschnitt (\wedge). Das Wechseln des Alphabets durch Austausch einzelner Symbole ist offensichtlich auch keine Schwierigkeit. Man ersetze die entsprechenden Symbole in dem die Menge beschreibenden rationalen Ausdruck. Diese Idee wird nun mehrfach verallgemeinert, indem für einzelne Symbole nun ganze Sprachen *substituiert* werden:

3.61 Definition

Eine **Substitution** ist ein Homomorphismus $s : \Sigma^* \longrightarrow 2^{\Gamma^*}$, wobei Σ und Γ Alphabete sind und s folgende Eigenschaften besitzt:

1. Für jedes $a \in \Sigma$ ist $s(a) \subseteq \Gamma^*$ definiert.
2. $s(\lambda) := \{\lambda\}$.
3. $\forall u, v \in \Sigma^* : s(uv) = s(u) \cdot s(v)$

Ist $s(a)$ regulär (bzw. endlich) für jedes $a \in \Sigma$, dann heißt s eine **reguläre** bzw. eine **endliche** Substitution.

Für eine Sprache $L \subseteq \Sigma^*$ werde s kanonisch (in natürlicher Weise) erweitert durch:

$$s(L) := \bigcup_{w \in L} s(w)$$

3.62 Beispiel

Für $s : \{0, 1\}^* \longrightarrow 2^{\{a, b\}^*}$ mit $s(0) := \{a\}$, $s(1) := \{b\}^*$ ergibt sich $s(0^*(0+1)1^*) = a^*(a+b^*)(b^*)^* = a^*b^*$, wobei wir die Mengen durch nur leicht vereinfachte, rationale Ausdrücke beschrieben haben.

3.63 Theorem

Die Familie der regulären Mengen ist gegenüber regulären Substitutionen abgeschlossen.

Beweis: Jede reguläre Menge R wird durch einen rationalen Ausdruck dargestellt. Ersetzt man nun jedes Symbol a in diesem Ausdruck durch den rationalen Ausdruck der $s(a)$ beschreibt, so ergibt sich ein rationaler Ausdruck für $s(R)$. \square

Substitutionen ersetzen einzelne Symbole durch Wortmengen, also in der Regel sogar durch unendlich viele Wörter, unter denen auch das leere Wort λ vorkommen darf. Die Umkehrung, nämlich längere (Teil-)Wörter in einer Zeichenkette auf einzelne Symbole zu verkürzen, scheint zunächst nicht so leicht möglich. Mathematisch wird dies durch die Umkehrung eines Homomorphismus, also der Anwendung eines inversen Homomorphismus geleistet.

3.64 Definition

Seien Σ und Γ endliche Alphabete sowie $h : \Sigma^* \longrightarrow \Gamma^*$ ein Homomorphismus. Die zu h inverse Relation, genannt **inverser Homomorphismus**, kann auch als totale Funktion $h^{-1} : \Gamma^* \longrightarrow 2^{\Sigma^*}$. Hierbei definiert h^{-1} für jedes $w \in \Gamma^*$ die Menge aller möglichen Urbilder von w durch:

$$h^{-1}(w) := \{v \in \Sigma^* \mid \exists w \in \Gamma^* : h(v) = w\}.$$

3. Endliche Automaten und reguläre Mengen

Diese Menge kann daher auch leer sein. Wie auch bei Homomorphismen und Substitutionen wird der inverse Homomorphismus kanonisch auf formale Sprachen erweitert:

$$h^{-1}(L) := \bigcup_{w \in L} h^{-1}(w)$$

3.65 Beispiel (inverse Homomorphismen)

Wir definieren zwei Homomorphismen und betrachten die Anwendung jeweils des inversen Homomorphismus auf (a) einzelne Wörter und (b) auf Sprachen. In (b) wird die kanonische Erweiterung in der üblichen Form verwendet.

(a) Sei $h : \{a, b, c\}^* \rightarrow \{x, y\}^*$ definiert durch

$$\begin{aligned} a &\mapsto xyx, \\ b &\mapsto xy, \\ c &\mapsto yx. \end{aligned}$$

Dann ist $h^{-1}(xy) = \{b\}$, $h^{-1}(xx) = \emptyset$ und $h^{-1}(xyxyx) = \{ac, ba\}$.

(b) Sei $f : \{a, b, c\}^* \rightarrow \{x, y\}^*$ definiert durch

$$\begin{aligned} a &\mapsto x, \\ b &\mapsto y, \\ c &\mapsto \lambda. \end{aligned}$$

Dann ist $h^{-1}(\{\lambda\}) = \{c\}^*$, $h^{-1}(\{y\}) = \{c\}^*\{b\}\{c\}^*$ und $h^{-1}(\{x, y\}^*) = \{a, b, c\}^*$.

3.66 Theorem

Sei $L \in \mathcal{A}kz(\Gamma)$ und $h : \Sigma^* \rightarrow \Gamma^*$ ein Homomorphismus, dann ist $h^{-1}(L) \in \mathcal{A}kz(\Sigma)$.

Beweis: Sei $L = L(A)$ für einen vDFA $A = (Z, \Gamma, \delta_1, z_0, Z_{\text{end}})$. Wir konstruieren einen DFA $B = (Z, \Sigma, \delta_2, z_0, Z_{\text{end}})$, der bei Eingabe eines Symbols x aus Σ den vDFA A auf der Eingabe $h(x)$ simuliert. Dazu wird δ_2 definiert durch: $\forall (z, x) \in Z \times \Sigma : \delta_2(z, x) := (z)^{h(x)}$, wobei in B der mit $h(x)$ in A erreichte Zustand $(z)^{h(x)}$ eingenommen wird. Wenn also in A ein Wort $v \in (h(\Sigma))^*$ akzeptiert wird, so wird jedes $u \in \Sigma^*$ mit $h(u) \in (h(\Sigma))^*$ auch von B akzeptiert. Die Umkehrung ist ebenso einfach. \square

Wir hatten in Kapitel 2 (Def. 2.50) die Operation der Spiegelwortbildung w^{rev} kennengelernt, um z.B. Palindrome definieren zu können und um den Zusammenhang zwischen rekursiven Definitionen und induktiven Beweisen zu illustrieren. Wir werden sehen, dass die Familie $\mathcal{R}eg$ auch gegenüber der Spiegelwortabbildung abgeschlossen ist:

3.67 Theorem

Für jede reguläre Menge $L \in \mathcal{R}eg$ gilt $L^{\text{rev}} \in \mathcal{R}eg$,

Beweis: Sei $L = L(A)$ für einen vDFA $A = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$. Wir konstruieren den buchstabierenden NFA $A_{\text{rev}} = (Z, \Sigma, K, Z_{\text{end}}, Z_{\text{start}})$ mit $K := \{(q, x, p) \mid \delta(p, x) = q\}$. Es ist offensichtlich, dass jeder Erfolgspfad in A_{rev} einem rückwärts von einem End- zu einem Startzustand in A gegangener Erfolgspfad entspricht. \square

Die in den Theoremen 3.63, 3.66 und 3.67 behandelten Operationen stellen also der Familie \mathcal{Reg} weitere Abschlusseigenschaften zur Verfügung!

Eine oftmals sehr einfache Methode, von gewissen Mengen zu zeigen, dass sie nicht regulär sind, geht nach folgendem Schema vor:

Zunächst wird angenommen, dass die fragliche Menge regulär sei. Dann benutzt man Eigenschaften regulärer Mengen, um durch Transformationen mit Abschluss-Operatoren solche Sprachen zu erhalten, von denen man schon weiß, oder für die man vielleicht leichter zeigen kann, dass diese nicht regulär sind. Sind sie dieses nicht, so kann die Ausgangsmenge auch nicht regulär sein, denn wäre sie es, so doch auch alle Mengen, die aus dieser mit den Abschlusseigenschaften erzeugt wurden.

3.68 Beispiel

- (a) Wir zeigen, dass die Menge $C := \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$ nicht regulär ist, wie folgt: Setze $D := C \cap \{a\}^* \{b\}^*$. Mit $C \in \mathcal{Reg}$ wäre auch $D \in \mathcal{Reg}$, es ist aber $D = \text{DUP} = \{a^n b^n \mid n \in \mathbb{N}\}$ bekanntlich nicht regulär.
- (b) Wir sahen, dass das uvw -Theorem nicht direkt benutzbar war, um zu beweisen, dass die Sprache $L := \{c^k a^l b^m \mid k, l, m \in \mathbb{N} : k = 0 \text{ oder } l = m\}$ nicht regulär ist. Wir definieren nun die Menge $E := L \cap \{c\}^* \{a\}^* \{b\}^*$, die wegen Satz 3.59 regulär wäre, wenn L dies ist. Wenn wir nun auf E den Homomorphismus $h : \{a, b, c\}^* \rightarrow \{a, b\}^*$ anwenden, der das einzige „c“ löscht und auf den anderen Symbolen die Identität bleibt, so gilt doch $h(E) = \text{DUP} \notin \mathcal{Reg}$ und folglich auch $L \notin \mathcal{Reg}$!

Neben den bisher kennengelernten werden wir nun noch eine weitere Abschlusseigenschaft der Sprachfamilie der regulären Mengen vorstellen.

3.69 Definition

Für Mengen $K, L \subseteq \Sigma^*$ ist der **Rechtsquotient** $K/L := \{u \in \Sigma^* \mid \exists v \in L : uv \in K\}$ erklärt.

3.70 Theorem

1. Sind L_1 und L_2 reguläre Mengen, so ist auch L_1/L_2 eine reguläre Menge.
2. Ein DFA der diese Menge akzeptiert kann effektiv aus den üblichen Spezifikationen für die regulären Mengen L_1 und L_2 konstruiert werden.

Beweis: Sei $A = (Z, \Sigma, \delta, z_0, Z_{\text{end}})$ ein DFA der L_1 akzeptiert. In diesem ändern wir nun nur noch die Menge der Endzustände zu $Z'_{\text{end}} := \{z \in Z \mid \exists v \in L_2 : (z)^v \in Z_{\text{end}}\}$. Diese neue Menge lässt sich aber effektiv ausrechnen, denn für jeden Zustand $z \in Z$ ist die Leistung $L(z)$ regulär und mit ihr auch die Menge $L(z) \cap L_2$. Ob dieser Durchschnitt nun aber ein Element enthält oder nicht, lässt sich mit dem später genannten Theorem 3.74 entscheiden. \square

Es sollte hier bemerkt werden, dass 1. in Theorem 3.70 auch noch richtig ist, wenn die Sprache L_1 regulär ist, jedoch L_2 eine beliebige Menge von Wörtern ist. Diese braucht dann also nicht mehr regulär zu sein! Der definierte endliche Automat ist immer noch funktionsfähig und leistet das Gewünschte. Es ist jedoch denkbar, dass diesen Automaten niemand wirklich hinschreiben kann, weil die Frage $L(z) \cap L_2 \neq \emptyset$ nicht für jede Sprache L_2 effektiv lösbar ist. Wir werden in späteren Veranstaltungen des Studiums, wie unter anderem in der Vorlesung F3 („Berechenbarkeit und Komplexität“), solche Mengen noch explizit kennenlernen.

3.10. Entscheidungsprobleme

Nach der Konstruktion von endlichen Automaten und dem Umgang mit diesen stellen sich viele Probleme als wichtig heraus. Zum Beispiel kann es nötig sein, zu erfahren, ob ein vorgegebenes Wort von diesem akzeptiert wird. Diese Frage muss zum Beispiel in einer ersten Phase der Syntaxanalyse imperativer (auch: zustandsorientierter) Programme, der lexikalischen Analyse vom sogenannten „scanner“ entschieden werden.

Für die Untersuchung in der theoretischen Informatik ist es praktisch, Probleme und Problem-Klassen stets in der gleichen Weise zu notieren. Wir werden diese hier im Folgenden vorstellen:

Definition (〈 Name oder Bezeichnung des Problems 〉)

Eingabe: 〈 Auflistung der verwendbaren Daten 〉

Frage: 〈 Zu gewinnende neue Information 〉

Die Frage nach einem Entscheidungsverfahren für eine reguläre Menge R , also nach einem Algorithmus, der zu jedem vorgegebenem Wort verkündet, ob dieses in der Menge R enthalten ist oder nicht, wird als das (**spezielle**) **Wortproblem** der Menge R bezeichnet. Die Spezifikation der Menge R können wir uns beliebig denken, z.B. durch einen NFA oder einen rationalen Ausdruck, denn darauf kommt es nicht an. Es muss nur ein Algorithmus *existieren*, der das gewünschte leistet, konstruieren brauchen wir diesen aus der gegebenen Sprachspezifikation von R nicht! In der verabredeten Schreibweise wird das (spezielle) Wortproblem also wie folgt spezifiziert:

3.71 Definition (Wortproblem für eine reguläre Menge L)

Eingabe: Ein Wort $w \in \Sigma^*$

Frage: Gilt $w \in L$?

Als Algorithmus wählen wir einfach den DFA der R akzeptiert. Der zeitliche Aufwand zur Lösung dieses Problems ist damit nicht wesentlich größer als die Zeit, jedes einzelne Symbol des Wortes w zu lesen und in der Übergangstafel des DFA nachzusehen. Der Aufwand ist also linear in der Länge des Wortes w , wenn davon ausgegangen wird, dass jeder einzelne Schritt eines Zustandsübergangs konstant viel Zeit benötigt.

Auch wenn diese Art der Problemstellung in solch einem einfachen Fall etwas künstlich erscheint, hat sie doch in späteren Fällen unbestreitbare Vorteile. Zum Beispiel dann, wenn wir ein Verfahren suchen, das nicht nur für diesen einen Automaten das Wortproblem entscheidet, sondern sogar für jeden beliebigen vorgelegten Automaten anwendbar ist. Diese Frage wird als das **allgemeine Wortproblem** bezeichnet und ist formulierbar als:

3.72 Definition (Allgemeines Wortproblem für λ -freie NFA's)

Eingabe: Ein NFA $A = (Z, \Sigma, K, z_0, Z_{end})$ und ein Wort $w \in \Sigma^*$

Frage: Gilt $w \in L(A)$?

Das allgemeine Wortproblem für NFA's ließe sich natürlich dadurch lösen, dass zuerst der NFA in einen äquivalenten DFA umgewandelt wird, und danach in diesem nachgesehen wird, ob der Pfad bei Eingabe von w in einen Endzustand führt. Das ist wegen der möglichen Größe des DFA nicht zu empfehlen.

In vielen Fällen hilft es, wenn die Konstruktion des Potenzautomaten nur insoweit durchgeführt wird, wie es die möglichen Pfade in dem λ -freien NFA entlang des Wortes $w := w_1w_2w_3 \dots w_n$ erfordern.

Aber auch das sieht zunächst noch recht unangenehm aus, verlangt das Verfahren doch, dass von der Menge $M_i := \{q \in Z \mid \exists p \in Z_{\text{start}} : p \xrightarrow[v]{*} q, \text{ für } v = w_1w_2 \dots w_i\}$ aller in i Schritten aus der Menge Z_{start} erreichbaren Zustände, im nächsten Schritt die folgende Menge $M_{i+1} := \{q \in Z \mid \exists p \in M_i : p \xrightarrow[w_{i+1}]{*} q\}$ konstruiert werden muss. Das benötigt in der Regel jedesmal $|M_i| \cdot |X|$ viele Schritte, also in der Summe im schlechtesten denkbaren Fall, wenn nämlich in jedem Schritt eine neue Teilmenge von Z erzeugt wird, höchstens $|Z|^n$. Man betrachte die Automaten der Theoreme 3.29 und 3.50 und entsprechend gewählte Wörter.

Letztlich können diese Mengen jedoch nie mehr als $|Z|$ Elemente enthalten, so dass sich diese bei Wörtern mit $|w| > |Z|$ nicht mehr vergrößern und jeder Übergang höchstens $|Z|^2$ Schritte zum Nachsehen in der Übergangstabelle benötigt.

3.73 Definition (Leerheitsproblem (emptiness problem) für endliche Automaten)

Eingabe: Ein DFA oder ein NFA A

Frage: Ist $L(A) = \emptyset$?

3.74 Theorem

Das Leerheitsproblem für endliche Automaten kann in linearer Zeit entschieden werden.

Beweis: Gegeben sei ein NFA $A = (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$. $L(A) = \emptyset$ genau dann, wenn es in dem Zustandsgraphen von A von keinem der Anfangszustände einen Pfad zu einem Endzustand gibt. Das kann mit Tiefensuche (*depth-first*) und Zurücksetzen (*backtracking*) leicht getestet werden.

3.75 Algorithmus

Im Verlauf des Verfahrens werden Zustände ROT gefärbt. Zu Beginn ist noch kein Zustand gefärbt.

0. Man beginne bei einem der Startzustände z_0 und markiere diesen ROT.
1. Sei z_1 der aktuelle ROT gefärbte Zustand und $\{z_{1.1}, \dots, z_{1.k}\}$ die Menge der Folgezustände von z_1 . (Beachte $k \leq |Z|$).
2. Gibt es unter den Folgezuständen des aktuellen Zustands z_1 einen Endzustand, so gilt $L(A) \neq \emptyset$ und der Algorithmus stoppt. Andernfalls wähle man einen ungefärbten Folgezustand $z_{1.i}$ von z_1 zum aktuellen Zustand (das ist der Schritt der Tiefensuche) und färbt diesen ROT. Gehe zu 1.
Gibt es keinen ungefärbten Folgezustand von z_1 , so gehe man zu dem Vorgänger von z_1 (das ist das Zurücksetzen) von dem man diesen Zustand auf dem Hinweg erreicht hatte (dieser ist schon auf dem Hinweg ROT gefärbt worden) und gehe zu 1.

Jede Kante des Zustandsgraphen wird auf diese Weise höchstens zweimal durchlaufen und dabei ein Baum aller vom Ausgangszustand erreichbaren Zustände aufgebaut. Wird das Verfahren für alle Startzustände durchgeführt, sind alle Zustände ROT gefärbt und die Antwort erhalten. Der Gesamtaufwand wird durch die Anzahl der existierenden Kanten linear begrenzt und die Größe des endlichen Automaten ist ebenfalls linear in der Anzahl der Kanten. \square

Das Äquivalenzproblem für DFAs ist gegeben durch:

3. Endliche Automaten und reguläre Mengen

3.76 Definition (Äquivalenzproblem für DFAs)

Eingabe: Zwei DFAs $A = (Z_A, \Sigma_A, \delta_A, z_{A,0}, Z_{A,end})$ und $B = (Z_B, \Sigma_B, \delta_B, z_{B,0}, Z_{B,end})$

Frage: Gilt $L(A) = L(B)$?

3.77 Theorem

Das Äquivalenzproblem für DFAs ist in Zeit $|\Sigma_A \cap \Sigma_B| \cdot |Z_A| \cdot |Z_B|$ entscheidbar.

Beweis: Für $L_1 := L(A)$ und $L_2 := L(B)$ gilt doch $L_1 = L_2$ genau dann, wenn sowohl $L_1 \cap \overline{L_2} = \emptyset$ als auch $L_2 \cap \overline{L_1} = \emptyset$ ist. Ein vervollständigter DFA kann sofort in einen DFA umgewandelt werden, der das Komplement des ersten akzeptiert. Dazu müssen nur alle ehemaligen Endzustände zu gewöhnlichen Zuständen modifiziert und umgekehrt alle Zustände, die keine Endzustände waren, nun in Endzustände gewandelt werden. Dies geschieht mit linearem Aufwand. Den Durchschnittsautomaten $C_{A \cap B} := (Z_A \times Z_B, \Sigma_C = \Sigma_A \cap \Sigma_B, \delta_C, (z_{A,0}, z_{B,0}), Z_{C,end})$ hatten wir schon im Beweis zu Theorem 3.59 kennengelernt. Seine Konstruktion verlangt die Bestimmung aller Einträge der Übergangsfunktion δ_C mit $|\Sigma_C| \cdot |Z_A| \cdot |Z_B|$ vielen Argumenten in entsprechend vielen Schritten. \square

3.78 Definition (Universalitätsproblem für DFA's)

Eingabe: Ein DFA $A = (Z, \Sigma, \delta, z_0, Z_{end})$

Frage: Ist $L(A) = \Sigma^*$?

3.79 Theorem

Das Universalitätsproblem für deterministische endliche Automaten ist in linearer Zeit entscheidbar.

Beweis: Gegeben sei ein DFA $A = (Z, \Sigma, \delta, z_0, Z_{end})$. Es gilt $L(A) = \Sigma^*$ genau dann, wenn es in dem Zustandsgraphen von A von dem Anfangszustand keinen Pfad zu einem Zustand gibt, der nicht Endzustand ist und wenn außerdem für jeden Zustand und jedes Eingabesymbol ein Nachfolgezustand definiert ist. Das kann leicht festgestellt werden, denn dann muss jeder Zustand ein Endzustand sein und jeden Zustand müssen $|\Sigma|$ Kanten verlassen! \square

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

J.W. Backus führte 1959 eine Notation ein, die er zusammen mit P. Naur 1961 benutzte, um die Syntax der Programmiersprache ALGOL 60 zu formulieren. Schon 1955 hatte der Linguist Noam Chomsky die sogenannten semi-Thue-Systeme, die Axel Thue 1914 als allgemeine Ersetzungssysteme für Zeichenketten definiert hatte, abgewandelt, um damit Grammatiken für natürliche Sprachen zu definieren. Seine kontextfreie Grammatik ist die Grundlage der Backus/Naur-Notation, die wir hier als Beispiel bei der Definition eines Listentyps anwenden. Kontextfreie Grammatiken erlauben eine präzise Spezifikation der zulässigen syntaktischen Konstrukte und geben zugleich eine Möglichkeit für eine strukturelle Analyse der generierten Sätze oder Wörter. Damit sind natürliche Interpretationen oder sogar eingeschränkte Semantikzuordnungen möglich. Dass diese für die Analyse natürlicher Sprache nicht ausreichen, bedeutet keinen großen Nachteil, denn für regelbasierte Syntaxdefinitionen, wie die der meisten Programmiersprachen, sind diese Grammatiken ein nützliches Spezifikationsmittel.

4.1 Beispiel

```
⟨type list⟩ ::= ⟨simple variable⟩ | ⟨simple variable⟩, ⟨type list⟩  
⟨simple variable⟩ ::= ⟨variable identifier⟩  
⟨variable identifier⟩ ::= ⟨identifier⟩  
⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩⟨letter⟩ | ⟨identifier⟩⟨digit⟩  
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
```

Hierbei bedeutet die Schreibweise „ $\langle Name \rangle$ “ eine metalinguistische Variable mit der Bezeichnung *Name*. Das Zeichen „ $::=$ “ wird gelesen als „soll sein“ oder „wird erklärt durch“. Der senkrechte Strich „|“ wird als Zeichen für die Alternative verwendet, während alles andere für sich selbst steht. Leichter zu lesen sind in der Regel die Syntaxdiagramme, die wir ohne Rekursion schon in Kapitel 3 gesehen haben. In Büchern zur Definition und Verwendung von Programmiersprachen, wie z.B. PASCAL oder MODULA, begegnen uns Syntaxdiagramme wie die aus Abbildung 4.1. Beim Lesen der Beschriftungen auf den Pfaden vom Anfang eines Syntaxdiagramms zum Ausgang muss der Inhalt eines jeden Kreises oder ovalen Feldes notiert werden, während die rechteckigen Kästchen bedeuten, dass an diesen Stellen an den Anfang des in dem Rechteck mit Namen bezeichneten Syntaxdiagramms gesprungen werden muss. Dieses wird dann solange interpretiert, bis der Ausgang erreicht wird, um dann an der Aussprungstelle des „aufrufenden“ Diagramms fortzufahren. Auf diese Weise können Mengen von Wörtern spezifiziert werden, die, wie wir sehen werden, nicht mehr regulär sind.

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

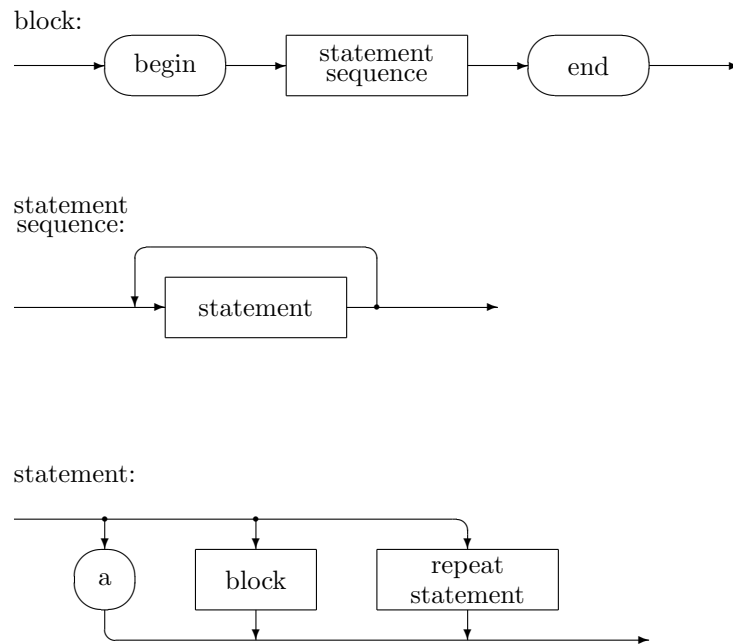


Abbildung 4.1.: Drei Syntaxdiagramme

4.1. Grammatiken

In der theoretischen Informatik hat sich anstelle der Backus/Naur-Notation eine kürzere und vielleicht einfacher lesbare Notation eingebürgert. Bevor wir eine formale Definition für kontextfreie Grammatiken geben, wollen wir jedoch zunächst den allgemeineren Begriff einer Grammatik definieren, um dann die kontextfreien Grammatiken als einen Spezialfall zu betrachten.

4.2 Definition (Grammatik)

Eine **Grammatik** wird beschrieben durch ein Quadrupel $G = (V_N, V_T, P, S)$ mit:

V_N ist ein endliches Alphabet von **Nonterminalen** (auch: metalinguistischen Variablen, Hilfszeichen oder syntaktischen Kategorien).

V_T ist ein mit V_N disjunktes endliches Alphabet von **Terminalen** (auch: Terminalsymbolen). Es muss also $V_T \cap V_N = \emptyset$ gelten. Das Gesamtalphabet wird mit $V := V_T \uplus V_N$ bezeichnet.

$P \subseteq V^* \setminus V_T^* \times V^*$ ist eine endliche Menge von **Produktionen** (oder: Regeln).

$S \in V_N$ ist das **Startsymbol**.

Eine Regel $(u, v) \in P$ wird üblicherweise als $u \rightarrow v$ notiert. Es ist zu beachten, dass nur eine einzige Bedingung an die Form einer Regel in einer allgemeinen Grammatik gestellt wird, nämlich die Bedingung, dass das zu ersetzende Teilwort mindestens ein Nonterminalsymbol enthält! Es können also mittels der Regeln einer allgemeinen Grammatik ganze Teilwörter durch andere Teilwörter ersetzt werden. Der Spezialfall der kontextfreien Grammatik, dem für die Spezifikation und Analyse der Syntax

von Programmiersprachen eine besondere Bedeutung zukommt, sieht hingegen nur vor, dass einzelne Nonterminale durch Wörter ersetzt werden können.

Im Unterschied zur kontextfreien Grammatik kann bei der allgemeinen Grammatik die Ersetzung eines Nonterminals also von den benachbarten Symbolen abhängen. Der Kontext, in dem die Ersetzung stattfindet ist also entscheidend! Aus diesem Grund erhielten die *kontextfreien Grammatiken* ihren Namen.

4.3 Definition (kontextfreie Grammatik)

Eine **kontextfreie Grammatik** (CFG, context-free grammar) ist eine Grammatik $G := (V_N, V_T, P, S)$, für die einschränkend gilt:

$P \subseteq V_N \times V^*$ ist eine endliche Menge von **Produktionen** (oder: Regeln).

Eine kontextfreie Regel $(A, w) \in P$ wird üblicherweise als $A \rightarrow w$ notiert. Wir bezeichnen $A \rightarrow \lambda$ als **λ -Produktion**.

Gilt $P \subseteq V_N \times V^+$, so besitzt G keine λ -Produktionen und heißt **λ -frei**.

Wir benötigen nun noch eine **Ableitungsrelation** zwischen Wörtern, um die Erzeugung von (abgeleiteten) Wörtern aus dem Startsymbol beschreiben zu können. Diese Ableitungsrelation wird – analog zu der erweiterten Zustandsüberföhrungsfunktion bei DFAs bzw. der Übergangsrelation bei NFAs – als eine Erweiterung der Produktionen definiert.

4.4 Definition (Ableitungsrelation)

Die **einschrittige Ableitung** eines Wortes v aus einem Wort u mittels der Produktionen der Grammatik G wird notiert als $u \xRightarrow{G} v$. Dabei ist die Relation $\xRightarrow{G} \subseteq V^* \times V^*$ für alle $u, v \in V^*$ definiert durch:

$$u \xRightarrow{G} v$$

genau dann, wenn

$$\exists u_1, u_2 \in V^* : \exists w_l \in V^* V_N V^* : \exists w_r \rightarrow w_l \in P : u = u_1 w_l u_2 \wedge v = u_1 w_r u_2$$

Als **Ableitungsrelation** wird die reflexive, transitive Hölle der Relation \xRightarrow{G} bezeichnet, die als

$$\xRightarrow{*}_G \subseteq V^* \times V^* \text{ geschrieben wird.}$$

Aus der vorangegangenen Definition ist leicht ersichtlich, dass ein Teilwort in einem Wort u , welches als linke Seite einer Produktion vorkommt, durch die rechte Seite dieser Produktion ersetzt werden kann.

Für kontextfreie Grammatiken vereinfacht sich die Definition der *einschrittigen Ableitungsrelation* aus Definition 4.4 folgendermaßen:

$$u \xRightarrow{G} v$$

genau dann, wenn

$$\exists u_1, u_2 \in V^* : \exists A \in V_N : \exists A \rightarrow w \in P : u = u_1 A u_2 \wedge v = u_1 w u_2$$

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

Es ist zu beachten, dass für jedes Wort $w \in V^*$, selbst wenn keine Produktion anwendbar ist, wegen der Reflexivität der Ableitungsrelation stets $w \xRightarrow{*}_G w$ gilt. Eine Zeichenkette $u \in V^*$, mit $S \xRightarrow{*}_G u$ nennt man **Satzform**, aber letztendlich interessant sind nur die Satzformen über dem Terminalalphabet. Wir definieren daher die von einer (kontextfreien) Grammatik erzeugte Sprache als die Menge aller aus dem Startsymbol erzeugbaren Terminalwörter:

4.5 Definition (erzeugte Sprache)

Sei $G = (V_N, V_T, P, S)$ eine Grammatik. $L(G) := \{w \in V_T^* \mid S \xRightarrow{*}_G w\}$ ist die von G generierte oder erzeugte Sprache.

Die Äquivalenz zweier endlicher Automaten hatten wir im vorangegangenen Kapitel über die Gleichheit der akzeptierten Sprachen definiert. Die folgende Definition führt einen analogen Äquivalenzbegriff auch für Grammatiken ein:

4.6 Definition (Äquivalenz von Grammatiken)

Zwei Grammatiken G_1 und G_2 heißen genau dann **äquivalent**, wenn sie die gleiche Sprache erzeugen, also wenn $L(G_1) = L(G_2)$ gilt.

Um im Folgenden wieder Aussagen über die Mächtigkeit der kontextfreien Sprachen machen zu können und diese mit anderen Sprachfamilien vergleichen zu können, führen wir die Familie der kontextfreien Sprachen auf naheliegende Weise ein:

4.7 Definition (Familie der kontextfreien Sprachen)

Die Familie aller kontextfreien Sprachen ist $\mathcal{Cf} := \{M \mid \exists \text{ CFG } G : M = L(G)\}$.

Eine kontextfreie Grammatik G wird üblicherweise einfach durch Angabe der einzelnen Produktionen spezifiziert. Dabei verabreden wir, dass die Elemente von V_N stets Großbuchstaben sind, während die Terminale durch Kleinbuchstaben bezeichnet werden. Die Mengen V_N und V_T ergeben sich eindeutig aus der vollständigen Spezifikation von P . Wird das Startsymbol ausnahmsweise nicht mit S bezeichnet, so muss dies vermerkt werden.

4.8 Beispiel

Wollen wir die Grammatik $G := (V_N, V_T, P, S)$ mit $P = \{S \rightarrow aSb, S \rightarrow \lambda\}$ notieren, so schreiben wir sie meist kurz als $S \rightarrow aSb \mid \lambda$. Der senkrechte Strich kann dabei als „oder“ gelesen werden und stellt ein Trennsymbol dar, so dass wir es hier immernoch mit zwei Regeln zu tun haben.

G erzeugt die nicht reguläre Sprache $L(G) = \text{DUP} = \{a^n b^n \mid n \in \mathbb{N}\}$.

Beispiel 4.9 zeigt, dass die leere Menge auch als erzeugte Sprache von kontextfreien Grammatiken vorkommen kann.

4.9 Beispiel

Betrachten wir die Grammatik G mit folgenden Produktionen, so stellen wir fest, dass $L(G) = \emptyset$ ist:

$$S \rightarrow AS \mid BA, \quad A \rightarrow BAB \mid SB, \quad B \rightarrow b$$

Offensichtlich sind hier alle Hilfszeichen überflüssig oder nutzlos.

Abgesehen von der Möglichkeit, die leere Menge durch eine kontextfreie Grammatik zu spezifizieren, hat Beispiel 4.9 gezeigt, dass möglicherweise nicht jedes Nonterminal und jede Regel einer Grammatik auch tatsächlich etwas zur erzeugten Sprache beiträgt. Solche überflüssigen Bestandteile sind natürlich unerwünscht, da sie das Verständnis wie auch den effizienten Umgang mit CFGs behindern. Selbst wenn man es selbst in der Hand hat, eine Grammatik anzugeben, die keine überflüssigen Bestandteile aufweist, so kann es durch die Anwendung von standardisierten Verfahren auf diese Grammatik passieren, dass überflüssige Symbole oder Regeln nachträglich (durch einen Algorithmus) eingefügt werden. Wir wollen deshalb zunächst formal definieren, wann eine kontextfreie Grammatik in diesem Sinne keine überflüssigen Bestandteile enthält und werden dann erwartungsgemäß sehen, dass das Entfernen dieser Symbole, etc. nicht zu einer Veränderung der erzeugten Sprache führt.

4.10 Definition (reduzierte Grammatik)

Eine kontextfreie Grammatik $G := (V_N, V_T, P, S)$ heisst genau dann **reduziert**, wenn gilt:

1. Jedes Nonterminal ist **produktiv**, d.h. $\forall A \in V_N$ gilt $\exists w \in V_T^* : A \xRightarrow{*}_G w$.
2. Jedes Nonterminal ist **erreichbar**, d.h. $\forall A \in V_N$ gilt $\exists u, v \in V^* : S \xRightarrow{*}_G uAv$.

Theorem 4.11 zeigt, dass zu allen kontextfreien Grammatiken außer denen, welche die leere Sprache erzeugen, eine äquivalente reduzierte CFG konstruiert werden kann.

4.11 Theorem

Zu jeder CFG G mit $L(G) \neq \emptyset$ kann effektiv eine äquivalente reduzierte CFG G' konstruiert werden.

Beweis: Die Konstruktion besteht aus zwei separaten Teilen:

Teil 1 entfernt Symbole (und Produktionen), so dass danach alle verwendeten Nonterminale stets auf Terminalwörter abgeleitet werden können, d.h. G enthält danach nur noch produktive Nonterminalsymbole.

Teil 2 entfernt danach alle Symbole, die vom Startsymbol aus nicht erreichbar sind.

Teil 1 der Konstruktion: Zu gegebener CFG $G := (V_N, V_T, P, S)$ definieren wir Mengen $M_i \subseteq V$ durch: $M_0 := V_T$, $M_{i+1} := M_i \cup \{A \in V_N \mid \exists w \in M_i^* : A \rightarrow w \in P\}$. Da für jedes $i \in \mathbb{N}$ $M_i \subseteq V$ endlich ist, und stets $M_i \subseteq M_{i+1}$ gilt, gibt es einen Index k , so dass $M_k = M_{k+1}$ ist. M_k enthält dann offensichtlich nur produktive Symbole.

Nun bilden wir die neue CFG $G' = (V'_N, V'_T, P', S') := (M_k \cap V_N, V_T, P \cap (M_k \times M_k^*), S)$ mit $L(G') = L(G)$.

Teil 2 der Konstruktion: Nun werden alle erreichbaren Symbole in G' bestimmt: Man setze $M_0 := \{S'\}$ und $M_{i+1} := M_i \cup \{B \in V'_N \mid \exists A \in M_i \exists u, v \in V'^* : A \rightarrow uBv \in P'\}$.

Wieder gilt $\exists k \in \mathbb{N} : M_k = M_{k+1}$ und M_k ist die Menge aller erreichbaren Nonterminale in G' . Man definiert nun $G'' := (V''_N, V''_T, P'', S'')$, indem von G' nur noch solche Produktionen beibehalten werden, deren rechte und linke Seiten nur mit erreichbaren Nonterminalen gebildet werden. Unter Umständen kann die Menge der dabei erreichbaren Terminalzeichen verkleinert werden.

□

Das Verfahren aus dem Beweis zu Theorem 4.11 wird falsch, wenn zuerst Schritt 2 und danach Schritt 1 angewendet wird, wie das folgende Beispiel zeigt: In der CFG mit den Produktionen $S \rightarrow AB|\lambda$, $A \rightarrow a$ sind alle Nonterminale erreichbar und keines würde in Schritt 2 gestrichen werden. Da das Symbol B jedoch unproduktiv ist, würden damit dann im nachfolgenden Schritt die Produktionen $S \rightarrow \lambda$ und $A \rightarrow a$ entstehen. Da beide Grammatiken aber nur λ als einziges Terminalwort erzeugen, ist die zweite davon nicht reduziert.

4.2. Chomsky-Normalform kontextfreier Grammatiken

Für die meisten Beweise und Konstruktionen mit kontextfreien Grammatiken sind Normalformen praktisch. Reduzierte Grammatiken können bereits als einer Normalform genügend angesehen werden, für die weitere Arbeit mit kontextfreien Grammatiken erweist sich aber die sogenannte Chomsky-Normalform als besonders wichtig.

4.12 Definition (Chomsky-Normalform)

Eine kontextfreie Grammatik $G := (V_N, V_T, P, S)$ ist in **Chomsky-Normalform** (CNF) genau dann, wenn alle Produktionen in P von der Form $A \rightarrow BC$ oder $A \rightarrow a$ für $A, B, C \in V_N$ und $a \in V_T$ sind.

Offensichtlich können kontextfreie Grammatiken in Chomsky-Normalform niemals das leere Wort λ erzeugen. Daher ist es praktisch, jede gewöhnliche CFG so umzuformen, dass $S \rightarrow \lambda$ die einzige Produktion ist, die das leere Wort erzeugt, falls dieses in der erzeugten Sprache überhaupt vorkommt. Eine solche Grammatik wollen wir als *erweiterte Chomsky-Normalform* (eCNF) bezeichnen. Wir werden dies in den Beweis des nächsten Theorems integrieren, in dem gezeigt wird, dass tatsächlich zu jeder kontextfreien Grammatik eine äquivalente Grammatik in erweiterter Chomsky-Normalform existiert.

4.13 Theorem

Zu jeder CFG G kann effektiv eine äquivalente CFG $G' := (V'_N, V'_T, P', S')$ konstruiert werden, für die folgendes gilt:

Falls $\lambda \notin L(G)$, so ist G' in Chomsky-Normalform. Gilt $\lambda \in L(G)$, so ist $S' \rightarrow \lambda$ die einzige λ -Produktion in P' und die Produktionen in $P' \setminus \{S' \rightarrow \lambda\} \subseteq V'_N \times (V'_N \cdot V'_N \cup V'_T)$ sind in Chomsky-Normalform. Ausserdem kommt S' auf keiner rechten Seite einer Regel aus P' vor.

Beweis (Konstruktion einer Chomsky-Normalform-Grammatik): Sei $G := (V_N, V_T, P, S)$ eine beliebige CFG.

Der Beweis besteht aus sechs Schritten, in denen die ursprüngliche Grammatik in jeweils äquivalente Grammatiken G_1 bis G_6 umgeformt wird, von denen die letzte die gewünschte Eigenschaft hat. Wir benennen zunächst die Schritte und beschreiben danach die jeweilige Konstruktion:

1. λ -frei-machen
2. Reduzieren
3. Kettenregeln entfernen

4. Ersetzen langer Terminalregeln
5. Verkürzen zu langer Regeln
6. Wiederherstellen der ursprünglichen Sprache durch evtl. Hinzunahme einer λ -Regel

1. Schritt: Es werden die λ -Produktionen entfernt: Für jedes $A \in V_N$ entscheiden wir, ob $A \xRightarrow{*}_G \lambda$ gilt. Dies ist leicht mit bereits bekannten Methoden möglich. Wir bestimmen die Menge $V_\lambda \subseteq V_N$ von Nonterminalen, aus denen u.a. auch das leere Wort abgeleitet werden kann. Dazu setzen wir:

$$\begin{aligned} M_0 &:= \{A \in V_N \mid A \longrightarrow \lambda \in P\} \text{ und} \\ M_{i+1} &:= M_i \cup \{A \in V_N \mid \exists w \in M_i^* : A \longrightarrow w \in P\} \end{aligned}$$

Es existiert ein Index $k \in \mathbb{N}$ mit $M_k = M_{k+1}$ und es ist $V_\lambda := M_k$ die Menge die gesucht wurde. Wir definieren nun die endliche Substitution:

$$\begin{aligned} \sigma : V^* \longrightarrow 2^{V^*} \quad \text{durch} \quad \sigma(A) &:= \{A\}, \text{ falls } A \in (V \setminus V_\lambda), \text{ und} \\ \sigma(A) &:= \{\lambda, A\} \text{ falls } A \in V_\lambda \end{aligned}$$

Damit ist dann $\sigma(u) = \{v \mid v \text{ geht aus } u \text{ dadurch hervor, dass einige (alle, keines) der Symbole der Menge } V_\lambda \text{ in } u \text{ gestrichen werden}\}$. Zum Beispiel gilt dann für $V_\lambda := \{A, B\}$: $\sigma(aABAbC) = \{aABAbC, aBAbC, aABbC, aAAbC, aAbC, aBbC, abC\}$. Als neue Grammatik erhalten wir nach dem ersten Schritt zunächst eine CFG G_1 , welche die Sprache $L(G_1) = L(G) \setminus \{\lambda\}$ erzeugt:

Es sei $G_1 := (V_N, V_T, P_1, S)$ mit $P_1 := \{A \longrightarrow v \mid v \neq \lambda \wedge v \in \sigma(w) \text{ für } A \longrightarrow w \in P\}$. Damit sind in P_1 also keine der λ -Produktionen von G mehr enthalten, und $\lambda \notin L(G_1)$ ist offensichtlich. Den Beweis für die Behauptung $L(G_1) \supset L(G) \setminus \{\lambda\}$ liefert eine Induktion über die Länge der Ableitungen in G , wobei wir hier die folgende noch allgemeinere Aussage (4.1) beweisen, da sie leichter zu zeigen ist:

$$\forall A \in V_N : A \xRightarrow{*}_G w \wedge w \in V_T^+ \text{ impliziert } A \xRightarrow{*}_{G_1} w \quad (4.1)$$

Induktionsbasis: Für Ableitungen der Länge 1 in G gilt mit $A \xRightarrow{*}_G w \wedge w \in V_T^+$ stets $A \longrightarrow w \in P$ und damit auch $A \longrightarrow w \in P_1$, da $P \subseteq P_1$ nach Definition gilt.

Induktionsschritt: Jede Ableitung $A \xRightarrow{*}_G w$ der Länge $k+1$ für ein nicht leeres Wort $w \in V_T^+$ hat

die Form $A \xRightarrow{*}_G \Sigma_1 \Sigma_2 \dots \Sigma_r \xRightarrow{*}_G w$, wobei für $1 \leq i \leq r$ Ableitungen der Form $\Sigma_i \xRightarrow{*}_G w_i$ mit $w = w_1 w_2 \dots w_r$ existieren. Alle diese Ableitungen haben eine Länge kleiner als k , so dass wir für sie die Aussage (4.1) als bewiesen annehmen wollen. Falls nun $w_i \neq \lambda$ ist, so ist auch $\Sigma_i \xRightarrow{*}_{G_1} w_i$ erfüllt. Falls jedoch $w_i = \lambda$ gilt, so ist $\Sigma_i \in V_\lambda$. Durch Streichen dieser Σ_i in $\Sigma_1 \Sigma_2 \dots \Sigma_r$, erhält man ein Wort $v \neq \lambda$, für das $A \longrightarrow v \in P_1$ ist. Aus dieser ersten Produktion zusammen mit den Ableitungen $\Sigma_i \xRightarrow{*}_{G_1} w_i$ erhält man die gesuchte Ableitung für $A \xRightarrow{*}_{G_1} w$.

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

Die Umkehrung $L(G_1) \subseteq L(G) \setminus \{\lambda\}$ ist leichter einzusehen, denn alle Produktionen in G_1 entstammen gültigen Ableitungen in G .

2. Schritt: G_1 ist möglicherweise nicht reduziert, so dass wir in diesem Schritt eine reduzierte, äquivalente CFG $G_2 := (V_{N_2}, V_{T_2}, P_2, S)$ aus G_1 mit dem Verfahren aus dem Beweis zu Theorem 4.11 erhalten.

3. Schritt: In G_2 kann es nun noch sogenannte **Kettenregeln** geben, also Produktionen der Form $A \rightarrow B \in V_N \times V_N$, die in der Chomsky-Normalform nicht vorkommen dürfen.

Wir definieren die Relation $\ll \subseteq V_N \times V_N$ folgendermaßen: $A \ll B$ gilt genau dann, wenn $A \rightarrow B \in P_2$. Mit \ll^* sei die reflexive transitive Hülle von \ll bezeichnet, die nach den früheren Verfahren bestimmt werden kann. Nun wird im dritten Schritt die Grammatik $G_3 := (V_{N_3}, V_{T_3}, P_3, S)$ definiert, indem $V_{N_3} := V_{N_2}$, $V_{T_3} := V_{T_2}$ und die neue Produktionenmenge $P_3 := \{A \rightarrow w \mid w \notin V_{N_3} \wedge \exists B \rightarrow w \in P_2 \wedge A \ll^* B\}$ anstelle von P_2 gesetzt wird. Das Startsymbol darf das alte S bleiben, denn sollte die reduzierte Grammatik kein einziges Wort mehr erzeugen, dann wurde $L(G_2) = \emptyset$ schon durch $P_2 = \emptyset$ erreicht. Die Gleichheit von $L(G_3)$ und $L(G_2)$ zeigt man leicht auch formal.

4. Schritt Da Terminalsymbole bei Grammatiken in Chomsky-Normalform nur durch Produktionen der Form $A \rightarrow a$ generiert werden dürfen, ersetzen wir in allen rechten Seiten, deren Länge größer 1 ist, von Produktionen in P_3 jedes Terminal a durch ein neues Nonterminal $\langle a \rangle$ und erweitern P_3 zu P_4 durch Hinzunahme der Produktionen $\langle a \rangle \rightarrow a$. Man erhält so im vierten Schritt $G_4 := (V_{N_4}, V_{T_3}, P_4, S)$.

5. Schritt Da nun aber in P_4 noch Produktionen $A \rightarrow w$ mit $|w| > 2$ vorhanden sein können, wird im fünften Schritt eine weitere Konstruktion nötig. Wir definieren neue Nonterminale $\langle v \rangle$ für jedes echte Präfix v mit $|v| \geq 2$ einer rechten Seite einer Produktion in P_4 . Zusammen mit neuen Produktionen erhalten wir so:

$$\begin{aligned} G_5 &:= (V_{N_5}, V_{T_3}, P_5, S) \text{ mit} \\ V_{N_5} &:= \{\langle v \rangle \mid \exists u \neq \lambda : \exists A \rightarrow w \in P_4 : w = vu \wedge |v| \geq 2\} \cup V_{N_4} \text{ und} \\ P_5 &:= \{A \rightarrow \langle v \rangle x \mid A \rightarrow w \in P_4 : |w| \geq 3 \wedge w = vx \wedge x \in V_{N_4}\} \cup \\ &\quad \{\langle v \rangle \rightarrow \langle u \rangle y \mid \langle u \rangle, \langle v \rangle \in (V_{N_5} \setminus V_{N_4}) \wedge y \in V_{N_4} \wedge v = uy\} \cup \\ &\quad \{\langle v \rangle \rightarrow xy \mid \langle v \rangle \in (V_{N_5} \setminus V_{N_4}) \wedge x, y \in V_{N_4} \wedge v = xy\} \cup \\ &\quad \{A \rightarrow w \mid A \rightarrow w \in P_4 \wedge |w| \leq 2\} \end{aligned}$$

G_5 ist nun eine Grammatik in Chomsky-Normalform und erzeugt dieselbe Sprache, wie die Ursprungsgrammatik G , falls $\lambda \notin L(G)$ gilt. Anderenfalls muss noch ein letzter Schritt hinzukommen, in dem aus der CNF-Grammatik G_5 eine eCNF-Grammatik konstruiert wird.

6. Schritt Im ersten Schritt wurde die Menge V_λ zur Ausgangsgrammatik G bestimmt. Dort konnte also die Frage „ $\lambda \in L(G)$?“ auf die Frage „ $S \in V_\lambda$?“ reduziert, und damit entschieden werden. Gilt also $\lambda \in L(G)$, so konstruieren wir G_6 , indem wir zu den Nonterminalen von G_5 ein neues Startsymbol S_{neu} , sowie die neuen Produktionen $S_{\text{neu}} \rightarrow \lambda$ und $\{S_{\text{neu}} \rightarrow w \mid \exists S \rightarrow w \in P_5\}$ zu P_5 hinzufügen.

□

Der Beweis des vorigen Theorems gestattet folgendes Korollar (Folgerung), welches hier noch einmal hervorgehoben werden soll:

4.14 Korollar

Es gibt einen Algorithmus, der zu jeder vorgelegten kontextfreien Grammatik G entscheidet, ob $\lambda \in L(G)$ ist.

Beweis: Man konstruiere die Menge V_λ wie im ersten Schritt des Beweises von Theorem 4.13. Dann ist $\lambda \in L(G)$ genau dann, wenn $S \in V_\lambda$ ist. \square

4.3. Lineare Grammatiken und reguläre Mengen

Bevor wir an die Konstruktion weiterer kontextfreier Grammatiken gehen, wollen wir zeigen, dass jede reguläre Menge eine spezielle kontextfreie Sprache ist. Die Grammatiken, die solche regulären Mengen erzeugen, besitzen eine einfache Struktur: Alle Produktionen enthalten in dem Wort, das die rechte Seite bildet, stets nur ein Nonterminal und dieses immer nur als erstes oder stets nur als letztes Zeichen.

4.15 Definition (lineare Grammatik)

Eine kontextfreie Grammatik $G := (V_N, V_T, P, S)$ heißt

linear, falls $P \subseteq V_N \times (V_T^* \cdot V_N \cdot V_T^* \cup V_T^*)$,

linkslinear, falls $P \subseteq V_N \times (V_N \cdot V_T^* \cup V_T^*)$, und

rechtslinear, falls $P \subseteq V_N \times (V_T^* \cdot V_N \cup V_T^*)$ ist.

Eine CFG wird genau dann **einseitig linear** genannt, wenn sie entweder linkslinear oder rechtslinear ist.

In einigen Büchern werden rechtslineare Grammatiken kurz als **reguläre Grammatiken** bezeichnet. Von Noam Chomsky wurden diese Grammatiken als **Typ-3-Grammatiken** bezeichnet. Wir behalten aber im Folgenden die definierte Notation bei.

4.16 Theorem

Eine Sprache $R \subseteq \Sigma^$ ist regulär genau dann, wenn es eine einseitig lineare Grammatik G gibt, die sie erzeugt, d.h., wenn $L(G) = R$ gilt.*

Beweis: Sei $R \in \text{Reg}$ definiert durch einen DFA $A_1 = (Z, \Sigma, K, z_0, Z_{\text{end}})$, dann werde eine rechtslineare CFG $G_R := (V_N, V_T, P, S)$ erklärt durch:

$$\begin{aligned} V_N &:= \{[z] \mid z \in Z\} \\ V_T &:= \Sigma \\ S &:= [z_0] \\ P &:= \{[z] \longrightarrow a[z'] \mid (z, a, z') \in K\} \cup \\ &\quad \{[z] \longrightarrow a \mid \exists z' \in Z_{\text{end}} : (z, a, z') \in K\} \cup \\ &\quad \{[z_0] \longrightarrow \lambda \mid z_0 \in Z_{\text{end}}\} \end{aligned}$$

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

Es ist leicht zu sehen, dass zu jedem Erfolgspfad in A die entsprechende Ableitung in G_R gefunden werden kann, und umgekehrt. Wenn wir $P' := \{[z] \rightarrow [z']a \mid \exists(z, a, z') \in K\} \cup \{[z] \rightarrow a \mid \exists z' \in Z_{\text{end}} : (z, a, z') \in K\}$ als linkslineare Regelmenge verwenden, dann generiert G'_R gerade $L(G'_R) = R^{\text{rev}}$. Da $L^{\text{rev}} \in \text{Reg}$ nach Theorem 3.3.67 für jede Sprache $L \in \text{Reg}$, kann $R = (R^{\text{rev}})^{\text{rev}}$ auch von einer linkslinearen CFG generiert werden.

Zu zeigen ist jetzt nur noch die Umkehrung, dass nämlich zu jeder rechtslinearen, (bzw. linkslinearen) CFG $G := (V_N, V_T, P, S)$ ein endlicher Automat konstruiert werden kann, der $L(G)$ akzeptiert. Wir zeigen dies hier nur für die rechtslinearen Grammatiken, denn ist eine CFG G linkslinear, so ersetzen wir vorab jede Produktion $A \rightarrow Ba$ ($A, B \in V_N, a \in V_T$) durch die rechtslineare Produktion $A \rightarrow aB$ und erhalten die rechtslineare CFG G' mit $L(G') = L(G)^{\text{rev}}$. Wenn dann zu $L(G')$ ein endlicher Automat A' konstruiert wurde, kann dieser nach Theorem 3.3.67 in denjenigen umgeformt werden, der $L(A')^{\text{rev}} = L(G)$ akzeptiert. Dazu definieren wir einen NFA $A_2 := (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$ durch

$$\begin{aligned} Z &:= \{z_A \mid A \in V_N\} \cup \{z_\lambda\} \\ \Sigma &:= V_T \\ Z_{\text{start}} &:= \{z_S\} \\ Z_{\text{end}} &:= \{z_\lambda\} \\ K &:= \{(z_Q, u, z_R) \mid Q, R \in V_N \wedge u \in V_T^* \wedge Q \rightarrow uR \in P\} \cup \\ &\quad \{(z_Q, u, z_\lambda) \mid Q \in V_N \wedge u \in V_T^* \wedge Q \rightarrow u \in P\} \end{aligned}$$

Auch hier ist es offensichtlich, dass zu jeder Ableitung eines Wortes $w \in V_T^*$ ein entsprechender Erfolgspfad im NFA gehört, und nur solche Wörter akzeptiert werden, die auch von der Grammatik generiert werden können. \square

4.4. Ableitungen und Ableitungsbäume

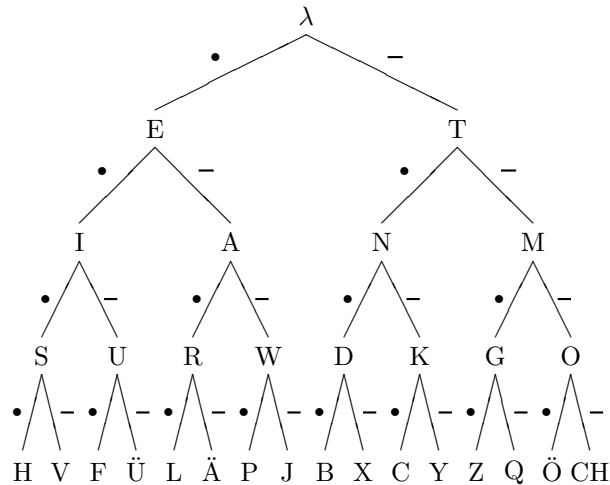
Zur Analyse von Wörtern einer kontextfreien Sprache ist es nötig, deren syntaktische Struktur zu bestimmen, denn in dieser sind die syntaktischen Kategorien durch die Nonterminale beschrieben. Diese Strukturen sind in dem sogenannten Ableitungsbaum des untersuchten Wortes sehr einprägsam dargestellt. Wir geben daher hier zunächst die nötigen Definitionen, bevor wir uns in Kapitel 6 den Analyseverfahren widmen.

In der Graphentheorie ist ein Baum jeder ungerichtete Graph ohne Kreise. Ein Baum heißt **lokal endlich**, wenn alle seine Knoten stets nur endlich viele direkte Nachbarn besitzen. In der Informatik verwenden wir häufig Bäume mit speziellen zusätzlichen Eigenschaften.

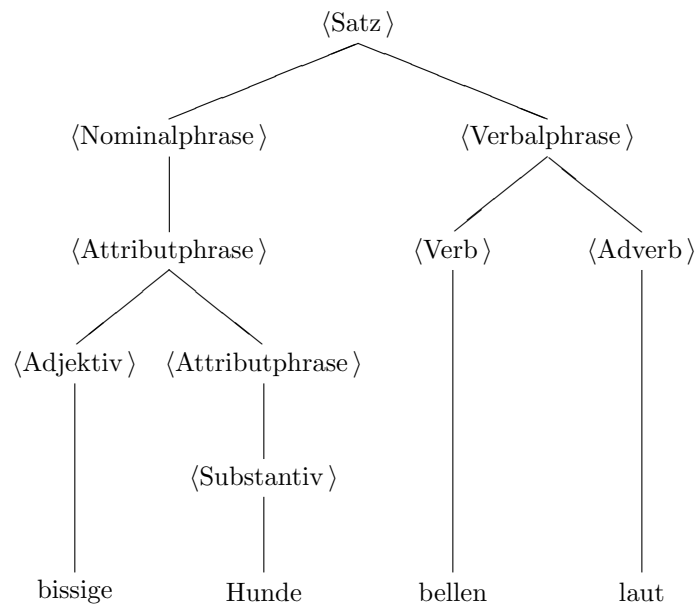
4.17 Definition (Baum)

Unter einem **Baum** verstehen wir im Folgenden einen gerichteten, geordneten, zyklensfreien und knotenmarkierten Graphen mit einem ausgezeichneten Knoten, der keinen Vorgängerknoten besitzt, der sogenannten **Wurzel**. Die Wurzel wird gewöhnlich oben gezeichnet und die Richtung der Kanten wird entweder durch Pfeile oder durch die Anordnung gegeben. **Vorgänger** stehen über ihren **Nachfolgern**. Knoten ohne Nachfolger heißen **Blätter** und die Nachfolgerknoten sind von links nach rechts angeordnet. Die einzelnen Knoten werden mit Anschriften oder **Bezeichnungen** versehen, die aus einer beliebigen Menge gewählt werden können. **Kantenmarkierungen** können auf die gleiche Weise erlaubt sein.

Als Beispiel für einen Baum betrachten wir den Kodierungsbaum für den Morse-Code. Dies ist ein binärer Baum, bei dem die Kantenbezeichnungen auf dem Pfad von der Wurzel λ zu einem Knoten den Morsecode seiner (Knoten-)Bezeichnung angeben:



Die Zerlegung eines Satzes einer natürlichen Sprache gemäß der syntaktischen Kategorien zeigt folgendes Beispiel eines Ableitungsbaumes. Hier sind wie in der Backus/Naur-Notation die syntaktischen Kategorien in spitzen Klammern eingrahmt. Die Terminalsymbole sind hier Wörter der „natürlichen“ Sprache:



Ableitungsbäume zu kontextfreien Grammatiken, manchmal auch Syntaxbäume genannt, sind spezielle Bäume, bei denen die inneren Knoten auf spezielle Weise mit den Nonterminalen beschriftet werden und die Blätter mit den Terminalsymbolen der generierten Zeichenkette. Als Wurzel eines Ableitungsbaumes fungiert das Startsymbol S der kontextfreien Grammatik.

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

4.18 Definition (Ableitungsbaum)

Sei $G := (V_N, V_T, P, S)$ eine beliebige CFG, dann wird für jedes Symbol $A \in V$ ein **A-Ableitungsbaum** $B(A)$ wie folgt erklärt:

1. Jedes Blatt von $B(A)$ ist mit einem Symbol $a \in V_T \cup \{\lambda\}$ beschriftet.
2. Jeder Knoten von $B(A)$, der kein Blatt ist, ist mit einem Symbol $\Sigma \in V_N$, die Wurzel mit dem Symbol A beschriftet.
3. Ein (innerer) Knoten ist mit dem Symbol $\Sigma \in V_N$ genau dann beschriftet, wenn seine Nachfolgerknoten von links nach rechts mit $\Sigma_1, \Sigma_2, \dots, \Sigma_k$ beschriftet sind und $\Sigma \rightarrow \Sigma_1 \Sigma_2 \dots \Sigma_k$ eine Produktion von G ist.

Notiert werden knotenbeschriftete Wurzelbäume oft auch als Terme, wobei die Terme zu den obigen Ableitungsbäumen einer gegebenen CFG G dann wiederum durch eine kontextfreie Grammatik G_T folgendermaßen definiert werden können:

4.19 Definition (Termform für Ableitungsbäume)

Zu einer beliebigen CFG $G := (V_N, V_T, P, S)$ definieren wir die entsprechende kontextfreie Grammatik für die Ableitungsbäume von G in Termform $G_T := (V'_N, V'_T, P', S')$ durch:

$$\begin{aligned} V'_N &:= \{X' \mid X \in V_N\} \\ V'_T &:= V_N \cup V_T \cup \{\Lambda, (,), ;\} \\ P' &:= \{X' \rightarrow X(X''_1; X''_2; \dots; X''_k) \mid X \rightarrow X_1 X_2 \dots X_k \in P \text{ wobei} \\ &\quad X''_i = X'_i, \text{ falls } X_i \in V_N; X''_i = X_i, \text{ falls } X_i \in V_T, \text{ und} \\ &\quad X''_1 = \Lambda, \text{ falls } k = 1 \text{ und } X_1 = \lambda\} \end{aligned}$$

$L(G_T)$ ist nun genau die Menge aller derjenigen Terme, die Ableitungsbäume von G beschreiben.

4.20 Beispiel

Betrachte die CFG G_1 mit den folgenden Produktionen:

$$S \rightarrow SS \mid aSb \mid \lambda$$

Die Terme t_1 und t_2 sind zwei unterschiedliche Beschreibungen von Ableitungsbäumen für das gleiche Wort $aabb$:

$$\begin{aligned} t_1 &:= S(S(a, S(a, S(\Lambda), b), b), S(\Lambda)) \text{ und} \\ t_2 &:= S(S(\Lambda), S(a, S(a, S(\Lambda), b), b)) \end{aligned}$$

Die von dieser Grammatik generierte Sprache $L(G_1)$ ist die **Dyck-Sprache** D_1 aller korrekt geklammerten Ausdrücke, bei denen a die öffnende und b die schließende Klammer bedeutet. Die gleiche Sprache kann auch durch die Grammatik G_2 mit den folgenden Produktionen erzeugt werden:

$$S \rightarrow aSbS \mid \lambda$$

Bei G_2 ist es nie möglich, dass ein Wort der Sprache zwei unterschiedliche Ableitungsbäume besitzt.

Die in Beispiel 4.20 für die Grammatik G_2 festgestellte Eigenschaft, für jedes Wort der erzeugten Sprache genau einen Ableitungsbaum zu haben, ist gerade für solche kontextfreie Grammatiken von Interesse, deren Semantik durch die Struktur des Ableitungsbaums bestimmt wird. Dies sind insbesondere auch jene Grammatiken, die die Grundlage für die Definition der Syntax von Programmiersprachen bilden. Daher ist der folgende Begriff von großer praktischer Bedeutung:

4.21 Definition (Mehrdeutigkeit)

Eine CFG G heißt **mehrdeutig** (*ambiguous*) genau dann, wenn es für mindestens ein Wort $w \in L(G)$ zwei verschiedene Ableitungsbäume gibt. Andernfalls heißt G **eindeutig**.

Eine kontextfreie Sprache $L \in \mathcal{Cf}$ heißt **eindeutig** (*unambiguous*) genau dann, wenn es eine eindeutige CFG G mit $L = L(G)$ gibt. Andernfalls heißt L **mehrdeutig**.

An einer konkreten Grammatik lässt sich nicht ohne weiteres feststellen, ob die erzeugte Sprache eindeutig ist, ob also eine äquivalente eindeutige Grammatik existiert.

Nicht nur, dass ein einzelnes Wort einer Grammatik eventuell mehr als einen Ableitungsbaum besitzt, es gibt in der Regel zu einem bestimmten Ableitungsbaum auch mehrere verschiedene Ableitungen. Diese unterscheiden sich alle nur in der Reihenfolge der jeweils ersetzten Nonterminale. Sicher ist jedoch, dass immer dann, wenn in der Ableitung das jeweils am weitesten links stehend Nonterminal als nächstes ersetzt wird, diese sogenannte Linksableitung für das Wort, zu dem der Ableitungsbaum gehört, durch diesen eindeutig bestimmt ist.

4.22 Definition (Linksableitung)

Eine Ableitung in einer CFG heißt **Linksableitung** (bzw. **Rechtsableitung**) genau dann, wenn in jedem Schritt der Ableitung das ersetzte Zeichen das am weitesten links (bzw. rechts) stehende Nonterminal ist. Die zugehörigen Ableitungsrelationen werden mit $\xRightarrow{\text{li}}$ und $\xRightarrow{\text{re}}$ bezeichnet.

Die Eindeutigkeit der Ableitungsbäume ist gleichbedeutend mit der Eindeutigkeit der Linksableitung jeder einzelnen Ableitung. Die Dyck-Sprache D_1 ist wegen der Existenz der eindeutigen CFG G_2 aus Beispiel 4.20 eine eindeutige Sprache. Wir werden später sehen, dass alle deterministischen kontextfreien Sprachen eindeutige Sprachen sind. Die Definition der Familie der deterministischen kontextfreien Sprachen werden wir mit Hilfe eines geeigneten Automatenmodells in Kapitel 5 geben. Es ist allerdings festzustellen, dass nicht alle kontextfreien Sprachen eindeutig sind. Wir formulieren dieses Ergebnis ohne Beweis:

4.23 Theorem

Die Sprache $L_M := \{a^r b^s c^t \mid \exists r, s, t \in \mathbb{N} : r = s \vee s = t\}$ ist kontextfrei aber nicht eindeutig.

Für die eindeutige Syntaxanalyse bei kontextfreien Sprachen ist es insbesondere bei der sogenannten „top-down“-Analyse wichtig, dass in der verwendeten Grammatik keine linksrekursive Produktion der Art $A \rightarrow Aw$ vorkommt.

4.24 Definition (Linksrekursion)

Jede Produktion $A \rightarrow w$, bei der das Nonterminal A auf der linken Seite vorkommt, nennen wir eine **A-Produktion**. Diese A-Produktion heißt **linksrekursiv**, wenn für die rechte Seite $w = Aw$ gilt.

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

Wir werden im Folgenden zeigen, dass jede kontextfreie Sprache von einer Grammatik generiert werden kann, die keine linksrekursive Produktion enthält.

4.25 Theorem

Zu jeder CFG G gibt es eine äquivalente CFG G' , deren Produktionen nicht linksrekursiv sind.

Beweis: Sei $G := (V_N, V_T, P, S)$ eine CFG und $A \rightarrow Au_1 \mid Au_2 \mid \dots \mid Au_r$ alle linksrekursiven A -Produktionen und $A \rightarrow v_1 \mid v_2 \mid \dots \mid v_s$ die verbleibenden A -Produktionen. Eine neue CFG $G' := (V'_N, V'_T, P', S')$ wird aus G gebildet durch Hinzufügen des neuen Nonterminals \bar{A} und Ersetzen aller linksrekursiven A -Produktionen von G durch die Produktionen: $A \rightarrow v_1\bar{A} \mid v_2\bar{A} \mid \dots \mid v_s\bar{A}$ und $\bar{A} \rightarrow u_1\bar{A} \mid u_2\bar{A} \mid \dots \mid u_r\bar{A} \mid u_1 \mid u_2 \mid \dots \mid u_r$. Dass eine beliebige Linksableitung $A \Rightarrow Au_{i_1} \Rightarrow Au_{i_2}u_{i_1} \Rightarrow \dots \Rightarrow Au_{i_p} \dots u_{i_2}u_{i_1} \Rightarrow v_ju_{i_p} \dots u_{i_2}u_{i_1}$ mit A -Produktionen von G , ersetzt werden kann durch $A \Rightarrow v_j\bar{A} \Rightarrow v_ju_{i_p}\bar{A} \Rightarrow \dots \Rightarrow v_ju_{i_p} \dots u_{i_2}\bar{A} \Rightarrow v_ju_{i_p} \dots u_{i_2}u_{i_1}$, ist offensichtlich. Die umgekehrte Transformation ergibt sich ebenso leicht. Wenn diese Konstruktion nun für alle Nonterminale A von G nacheinander in einer beliebig festgelegten Reihenfolge durchgeführt wird, erhalten wir die gewünschte äquivalente CFG ohne linksrekursive Produktionen. \square

Ausgehend von dieser Konstruktion definieren wir eine weitere Normalform für kontextfreie Grammatiken:

4.26 Definition (Greibach-Normalform)

Eine CFG $G := (V_N, V_T, P, S)$ ist in **Greibach-Normalform** genau dann, wenn $P \subseteq V_N \times V_T \cdot V_N^*$ ist.

Von Normalformen wird in der Regel verlangt, dass jedes Objekt aus einer hinreichend großen durch eine Eigenschaft charakterisierten Menge von Objekten in diese Form transformierbar ist. Wir zeigen, dass jede CFG effektiv in eine Greibach-Normalform transformiert werden kann, welche (mit Ausnahme des leeren Wortes) dieselbe Sprache erzeugt.

4.27 Theorem

Zu jeder CFG G kann effektiv eine CFG G' in Greibach-Normalform konstruiert werden, für die $L(G') = L(G) \setminus \{\lambda\}$ gilt.

Beweis (nach Sheila Greibach): Wir konstruieren G' in mehreren Schritten: Zunächst wird G in eine CFG G_1 umgeformt, die keine Kettenregeln besitzt, und bei der $S \rightarrow \lambda$ die einzige λ -Produktion ist, welches dann auch in keiner rechten Seite einer Produktion mehr vorkommt. Alle anderen Regeln sind in Chomsky-Normalform. D.h. G_1 ist in erweiterter Chomsky-Normalform.

Die nötigen Verfahren waren Bestandteil des Beweises von Theorem 4.13. Wir erhalten so die CFG $G_1 := (V_{N_1}, V_T, P_1, S_1)$. Nehmen wir nun an, dass o.B.d.A. $V_{N_1} := \{A_1, A_2, \dots, A_n\}$ und $S_1 := A_1$ gesetzt wird.

Wichtig für den Beweis ist die angenommene Ordnung auf dem Nonterminalalphabet.

Mit Algorithmus 4.28 werden wir durch sukzessive Veränderung der Regelmenge P_1 aus G_1 eine CFG $G_2 := (V_{N_2}, V_T, P_2, S_2)$ konstruieren, für die aus $A_i \rightarrow A_j w \in P_2$ stets $j > i$ folgt. Ein zweites Verfahren (Algorithmus 4.29) erzeugt dann aus G_2 die gesuchte CFG in Greibach-Normalform.

4.28 Algorithmus

```

begin
  for  $k := 1$  to  $n$  do
    for  $j := 1$  to  $k - 1$ 
      for jede Produktion der Form  $A_k \rightarrow A_j u$  do
        begin
          for alle Produktionen  $A_j \rightarrow v$  do
            füge neue Produktion  $A_k \rightarrow vu$  hinzu
            und entferne die Produktion  $A_k \rightarrow A_j u$ ;
          end
        end (* for Produktionen  $A_k \rightarrow A_j u$  *)
      end (* for  $j$  *)
    for jede Produktion der Form  $A_k \rightarrow A_k u$  do
      begin
        definiere neues Nonterminal  $B_k$  und ergänze neue Produktionen
         $B_k \rightarrow u$  und  $B_k \rightarrow uB_k$  und
        entferne die Produktion  $A_k \rightarrow A_k u$ 
      end;
      for jede Produktion der Form  $A_k \rightarrow u$ ,
      bei der  $u$  nicht mit dem Nonterminal  $A_k$  beginnt do
        füge die Produktion  $A_k \rightarrow uB_k$  hinzu
      end (* for Produktionen  $A_k \rightarrow u$  *)
    end (* for Produktionen  $A_k \rightarrow A_k u$  *)
  end (* for  $k$  *)
end (* Algorithmus *)

```

Die neu entstandene Regelmenge wird jetzt mit P_2 bezeichnet.

Der Aufruf der Schleife „**for** jede Produktion der Form $A_k \rightarrow u$ “ innerhalb der Schleife „**for** jede Produktion der Form $A_k \rightarrow A_k u$ **do**“ in Algorithmus 4.28 ist hier richtig aber in [Hopcroft&Ullman], auch in der deutschen Ausgabe, falsch!)

Es ist nicht schwer nachzuprüfen, dass folgendes gilt: Wenn $A_i \rightarrow w$ für $1 \leq i \leq n$ eine Produktion in P_2 ist, so beginnt w entweder mit einem Terminal oder einem Nonterminal A_j mit $j > i$, oder es handelt sich um die λ -Produktion $A_1 \rightarrow \lambda$. Insbesondere beginnen die rechten Seiten w der Produktionen $A_n \rightarrow w \in P_2$ stets mit einem Terminalsymbol! Weiter gilt für jede Produktion $B_i \rightarrow w' \in P_2$, dass w' entweder mit einem Terminal oder dem Nonterminal A_i beginnt. Fortgesetzte Substitution der Variablen A_k mit Index $k > j$, an erster Stelle einer rechten Seite von A_j -Produktionen durch rechte Seiten von A_k -Produktionen ergibt eine Situation, nach der jede rechte Seite einer A_k -Produktion mit einem Terminalsymbol beginnt. Dies wird durch Algorithmus 4.29 erreicht.

4.29 Algorithmus

Als Eingabe wird die mit Algorithmus 4.28 erzeugte CFG G_2 erwartet.

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

```

begin
  for  $k := n$  downto 2 do (*  $k$  wird schrittweise um 1 verringert *)
    for  $j := k - 1$  downto 1 do (*  $j$  wird schrittweise um 1 verringert *)
      Für jede Produktion  $A_k \rightarrow w \in P_2$  und jede Produktion
       $A_j \rightarrow A_k v \in P_2$  füge die Produktion  $A_j \rightarrow wv$  zu  $P_2$  hinzu und
      streiche danach  $A_j \rightarrow A_k v$  aus  $P_2$  heraus.
    end (* for  $j$  *)
  end (* for  $k$  *)
  Nenne die so erhaltene Regelmenge  $P_3$ 
end (* Algorithmus *)

```

Nach Anwendung von Algorithmus 4.29 beginnt also jede rechte Seite einer Produktion ungleich $A_1 \rightarrow \lambda$ in G' mit einem Terminalsymbol und $P_3 \setminus \{A_1 \rightarrow \lambda\}$ ist fast schon die gewünschte Produktionsmenge von G' . Da die rechte Seite w' einer B_i -Produktion $B_i \rightarrow w' \in P_2$ noch mit dem Nonterminal A_i beginnen kann, müssen diese für jedes $1 \leq i \leq n$ durch die entsprechenden Seiten der A_i -Produktionen ersetzt werden. So erhält man die Produktionsmenge P' und die gewünschte Grammatik ist konstruiert. \square

4.5. Das Pumping-Lemma der kontextfreien Sprachen

Bei den regulären Mengen hatten wir uns die Frage gestellt, ob es wohl auch formale Sprachen gibt, die nicht regulär sind. Diese Frage können wir nun natürlich auch für die kontextfreien Sprachen stellen und erhalten eine ähnliche Charakterisierung, wie bei den regulären Mengen. Wir beweisen auch für die kontextfreien Sprachen ein *Pumping-Lemma*, das sogenannte „*uvwxy*-Theorem“.

4.30 Theorem (Bar-Hillel, Perles, Shamir 1961)

Für jede kontextfreie Sprache $L \in \mathcal{Cf}$ gibt es eine Zahl $n \in \mathbb{N}$, so dass jedes Wort $z \in L$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ besitzt, für die folgendes gilt:

$$(i): |vx| \geq 1 \qquad (ii): |vwx| \leq n \qquad (iii): \forall i \geq 0 : uv^iwx^iy \in L$$

Beweis: Sei $L \in \mathcal{Cf}$ beliebig und $G := (V_N, V_T, P, S)$ eine CFG in Chomsky-Normalform für $L \setminus \{\lambda\}$. Setze $n := 2^k$, wobei $k = |V_N|$ ist. Betrachte den Ableitungsbaum für ein Wort $z \in L$ mit $|z| \geq n$. Bis auf die letzten Schritte, mit denen die Terminale erzeugt werden, ist dieser ein Binärbaum, d.h. jeder Knoten hat entweder zwei oder keinen Nachfolger. Die Zahl der Knoten im Ableitungsbaum mit genau einem Nachfolger ist genau $|z|$. Mit Induktion beweist man leicht, dass für jeden Binärbaum (ohne Knoten mit nur einem Nachfolger) folgende Beziehung gilt:

$$p \leq 2^q \tag{4.2}$$

wobei p die Zahl der Blätter bezeichnet und q die Anzahl der Kanten des längsten Pfades von der Wurzel zu einem Blatt. Also folgt, mit $|z| \geq 2^k$ sofort $2^q \geq |z| \geq 2^k$ oder $q \geq k$.

Da auf dem längsten Pfad mit q Kanten in dem Ableitungsbaum zu z genau $q + 1$ Knoten liegen, die mit Nonterminalen beschriftet sind, die Anzahl aller zur Verfügung stehenden Nonterminale k aber

4.5. Das Pumping-Lemma der kontextfreien Sprachen

Für CFG in CNF gilt:

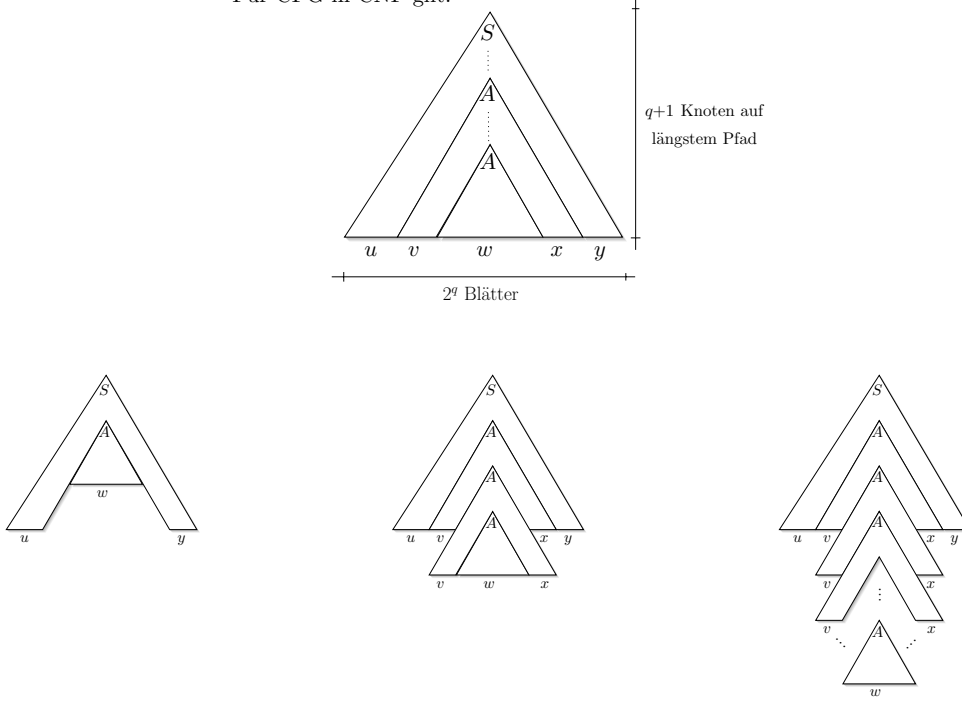


Abbildung 4.2.: Zur Aussage des Pumping-Lemma

kleiner als $q + 1$ ist, kommt also auf mindestens einem Pfad in dem Ableitungsbaum zu z mindestens ein Nonterminal doppelt vor. Wir wählen dasjenige Symbol, welches von den Blättern her gesehen sich zum ersten Mal wiederholt. Dieses sei hier mit A bezeichnet, und ist bei jedem Auftreten auf dem Pfad die Wurzel eines A -Ableitungsbaumes. Wegen der Wahl des zuletzt wiederholten Nonterminals A , ist sichergestellt, dass der Pfad bis zum vorletzten A höchsten k Kanten besitzt.

Der A -Ableitungsbaum vom vorletzten A erzeugt das Teilwort $vw x$ von z und der letzte A -Ableitungsbaum generiert das w . Wegen der Beziehung (4.2) folgt $|vw x| \leq n = 2^k$ und $|v x| \geq 1$ folgt daraus, dass der vorletzte A -Knoten zwei Nachfolger besitzt, von denen nur einer für den Pfad zum letzten A -Knoten benötigt wird. Es ist lediglich noch Eigenschaft (iii): $\forall i \geq 0 : uv^i wx^i y \in L$ zu zeigen. Der A -Ableitungsbaum am vorletzten A -Knoten kann wiederholt an der Position des letzten A -Knotens eingesetzt werden. Dadurch ergeben sich Ableitungen der Form:

$$S \xrightarrow{*}_G uAy \xrightarrow{*}_G uvAxy \xrightarrow{*}_G uvvAxy \xrightarrow{*}_G uv^iwx^iy$$

Aber auch $i = 0$ ist möglich, indem der zweite A -Ableitungsbaum beim vorletzten A -Knoten eingehängt wird: $S \xrightarrow{*}_G uAy \xrightarrow{*}_G uwy$.

Abbildung 4.2 veranschaulicht die Argumentation.

□

Da jede kontextfreie Sprache von einer kontextfreien Grammatik in Chomsky-Normalform erzeugt wird, kann für die Zahl n aus dem Pumping-Lemma die Anzahl $|V_N|$ der Nonterminale einer solchen Gram-

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

matik zugrundegelegt werden, so dass sich n als $2^{|V_N|}$ ergibt.

Obwohl die Folgerung des Pumping-Lemmas (d.h. die Aufteilbarkeit der „langen“ Wörter mit den angegebenen Eigenschaften) für alle kontextfreien Sprachen gilt, gibt es auch nicht-kontextfreie Mengen, für die diese Aussagen gelten. Daher kann mit Hilfe des Pumping-Lemmas nicht bewiesen werden, dass eine Menge tatsächlich kontextfrei ist, sondern nur umgekehrt, dass sie *nicht* kontextfrei sein kann!

4.31 Beispiel

Wir wenden das $uvwxy$ -Theorem an, um zu zeigen, dass folgende Sprachen nicht kontextfrei sein können: $L_1 := \{a^n b^n c^n \mid n \in \mathbb{N}\}$ und $L_2 := \{a^k b^{k^2} \mid k \in \mathbb{N}\}$. Der Beweis ist indirekt, d.h., wir nehmen an, L_1 (bzw. L_2) sei kontextfrei und werden dann einen Widerspruch herleiten, so dass wir diese Annahme widerrufen müssen.

Wenn $L_1 \in \mathcal{Cf}$, dann gilt die Folgerung aus dem $uvwxy$ -Theorem und es existiert eine Konstante $n \in \mathbb{N}$, so dass jedes $z \in L_1$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ besitzt mit den drei Eigenschaften des $uvwxy$ -Theorems. Wir wählen das Wort $z := a^n b^n c^n$. Es ist offensichtlich, dass bei keiner Aufteilung dieses Wortes in die Faktoren $uvwxy$ das Teilwort v die Form $a^i b^j$ oder $b^i c^j$ haben kann, weil dann sofort $uvvwxy \notin a^* b^* c^*$ wäre. Gleiches gilt für das Teilwort x und daher sind als mögliche Aufteilungen nur noch die Fälle denkbar, bei denen $v \in a^* + b^* + c^*$ wie auch $x \in a^* + b^* + c^*$ jeweils nur aus einem einzelnen Symbol gebildet werden. Da $|vx| \neq 0$ folgt daraus, dass beim Pumpen der Teilwörter v und x höchstens zwei, mindestens jedoch eines der zwei Teilwörter a^n oder c^n **nicht** verändert werden kann. (Wegen $|vwx| \leq n$ kann es nicht sein, dass a^n und c^n verändert werden, nicht jedoch b^n !) Also gibt es im Widerspruch zur Annahme keine passende Zerlegung von $z := a^n b^n c^n$ und $L_1 := \{a^n b^n c^n \mid n \in \mathbb{N}\} \notin \mathcal{Cf}$ folgt schlüssig.

Wenn $L_2 \in \mathcal{Cf}$, dann existiert $n \in \mathbb{N}$, so dass jedes $z \in L_2$ mit $|z| \geq n$ eine Zerlegung $z = uvwxy$ besitzt mit den Eigenschaften des $uvwxy$ -Theorems. Wir probieren es mit $z = a^n b^{n^2}$. Offensichtlich kann weder v noch x von der Form $a^p b^p$ mit $p \geq 1$ sein, denn dann wäre das Wort $uvvwxy$ kein Teilwort mehr von $a^* b^*$, also auch nicht mehr in L . Ebenfalls kann nicht $v \in b^*$ gelten, denn dann wäre auch $x \in b^*$ und somit würde schon das Wort $wwy = a^n b^{n^2-r}$ für ein $r \geq 1$ nicht mehr in L liegen. Wäre nun $v = a^p$ für $p \geq 1$ so könnte für x entweder (i) $x = a^j$ oder (ii) $x = b^j$ gelten. Im Fall (i) bedeutet dies $wwy = a^{n-p-j} b^{n^2} \notin L$. Also kann höchstens (ii) gelten. Dann jedoch ergibt sich $\forall i \in \mathbb{N} : uv^i w x^i y = a^{n+i \cdot p} b^{n^2+i \cdot j}$. Für $i = n$ und $j \leq n$ ergibt sich auch hier ein Widerspruch und L_2 ist nicht kontextfrei.

4.6. Ein Entscheidbarkeitsresultat

So wie bei den regulären Mengen, interessiert auch für kontextfreie Grammatiken bzw. kontextfreie Sprachen das Wortproblem. Wir definieren es zunächst und diskutieren dann die Entscheidbarkeit dieses Problems.

4.32 Definition (Wortproblem für eine kontextfreie Sprache L)

Eingabe: Ein Wort $w \in \Sigma^*$

Frage: Gilt $w \in L$?

4.33 Theorem

Das spezielle Wortproblem für kontextfreie Sprachen ist entscheidbar, d.h. es gibt ein Verfahren, das zu einer durch eine CFG G spezifizierten Sprache $L = L(G)$ für jedes w feststellt, ob $w \in L$ gilt.

Beweis: O.B.d.A. sei $L = L(G)$ durch eine CFG $G = (V_N, V_T, P, S)$ in Greibach-Normalform gegeben, und $w \in V_T^*$. Da in einer Ableitung $S \xrightarrow{*}_G v$ mit jedem Schritt ein weiteres Terminal erzeugt wird, brauchen nur die endlich vielen Ableitungen der Länge $|w|$ daraufhin überprüft zu werden, ob eine darunter ist, die das Wort w generiert. \square

Das eben verwendete Verfahren erfordert, bezogen auf die Länge des Eingabewortes w und die Größe der Grammatik G , im allgemeinen exponentiellen Aufwand. Man kann aber auf diese Weise auch das allgemeine Wortproblem lösen, in dem man zu einer beliebigen vorgegebenen CFG zunächst deren Greibach-Normalform konstruiert und dann dieses Verfahren anwendet. Dieses ist aber allemal exponentiell im Aufwand, bezogen auf die Größe der gegebenen Eingabedaten. Besser, aber auch noch nicht optimal für das spezielle Wortproblem, ist das Verfahren von Cocke, Younger und Kasami. Dieses basiert auf der Idee, dass in einer Grammatik in Chomsky-Normalform ein Wort $w = x_1x_2 \dots x_n$ immer dann aus dem Nonterminal A abgeleitet werden kann, wenn Wörter $x_1x_2 \dots x_j$ und $x_{j+1}x_{j+2} \dots x_n$ aus den Nonterminalen B bzw. C abgeleitet werden können, und die Regel $A \rightarrow BC$ in der Grammatik existiert. Mit Methoden der dynamischen Programmierung gewinnt man so einen Algorithmus, der in Polynomzeit arbeitet.

4.34 Theorem

Das spezielle Wortproblem für kontextfreie Sprachen in Chomsky-Normalform ist in Polynomzeit lösbar.

Beweis: Es gibt einen Algorithmus, der für eine kontextfreie Grammatik $G = (V_N, V_T, P, S)$ in Chomsky-Normalform und ein Wort $w \in V_T^*$ in $c_1 \cdot |P| \cdot |w|^3 + c_2$ Schritten entscheidet, ob $w \in L(G)$ ist. Hierbei sind c_1 und c_2 Konstanten, die nicht von G abhängen.

4.35 Algorithmus (Cocke-Younger-Kasami)

Eingaben: Eine kontextfreie Grammatik $G = (V_N, V_T, P, S)$ in Chomsky-Normalform und ein Wort $w \in V_T^*$ mit $w = x_1x_2 \dots x_n$, für $x_i \in V_T$.

Für alle $i, j \in \{0, 1, 2, \dots, n\}$ mit $0 \leq i \leq j$ definieren wir Mengen $V_{i,j} \subseteq V_N$ durch $V_{i-1,j} := \{A \in V_N \mid A \xRightarrow{*} x_i x_{i+1} \dots x_j\}$, die sukzessive als die Elemente der Felder $V_{i,j}$ einer $(n+1) \times (n+1)$ -Matrix mit Elementen aus V bestimmt werden.

```

for  $i := 1$  to  $n$  do
     $V_{i-1,i} := \{A \in V_N \mid A \rightarrow x_i \in P\}$ 
end (* for  $i$  *)
Setze  $V_{0,0} := \emptyset$  und  $\forall 1 \leq i \leq n : V_{i,i} := \{x_i\}$ 
for  $m := 2$  to  $n$  do
    for  $i := 0$  to  $n - m$  do
         $j := i + m$ 
         $V_{i,j} := \{A \in V_N \mid A \rightarrow BC \in P : \exists k : i < k < j : B \in V_{i,k} \wedge C \in V_{k,j}\}$ 
    end (* for  $i$  *)
end (* for  $m$  *)

```

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

Am Ende ist $V_{0,n}$ bestimmt, und es gilt $S \in V_{0,n}$ genau dann, wenn $w \in L(G)$ ist. Das ist auch ohne formal strengen Beweis einzusehen, denn es ist ja für $w = x_1 x_2 \dots x_n$ gerade $S \Rightarrow w$ für $n = 1$, wenn $S \rightarrow x_1 \in P$ und bei $n \geq 2$, falls eine Produktion $S \rightarrow BC \in P$ existiert, so dass $B \in V_{0,k}$ und $C \in V_{k,n}$ für ein k gilt, d.h. Ableitungen $B \xRightarrow{*} x_1 x_2 \dots x_k$ und $C \xRightarrow{*} x_{k+1} x_{k+2} \dots x_n$ existieren. \square

Für eine feste Grammatik G arbeitet das CYK-Verfahren in $c_1 \cdot |w|^3 + c_2$ Schritten, wobei c_1 und c_2 Konstante sind, die nicht von G , sondern nur von der Implementation der einzelnen Schritte abhängen. Der CYK-Algorithmus läßt sich auch wie folgt in eine Matrixmultiplikationsaufgabe umdeuten.

Zu gegebener CFG $G = (V_N, V_T, P, S)$ und für Mengen $U, W \subseteq V_N$ werden die Operationen $*$ und $+$ definiert:

$$\begin{aligned} U * W &:= \{A \in V_N \mid \exists B \in U : \exists C \in W : A \rightarrow BC \in P\} \\ U + W &:= U \cup W \end{aligned}$$

Es gilt nun $(U_1 + U_2) * W = U_1 * W + U_2 * W$ und die Berechnung der Mengen $V_{i,j}$ geschieht wie folgt:

$$\begin{aligned} V_{i,j} &:= \sum_{k=1}^n V_{i,k} * V_{k,j} \text{ mit} \\ V_{i-1,i} &:= \{A \in V_N \mid A \rightarrow x_i \in P\} \text{ und} \\ V_{k,j} &:= \emptyset \text{ falls } k > j. \end{aligned}$$

Es gilt dann für $j - i \geq 2$ gerade:

$$V_{i,j} := \sum_{k=i+1}^{j-1} V_{i,k} * V_{k,j}$$

Das spezielle Wortproblem für allgemeine kontextfreie Sprachen wird häufig mit dem hier nicht vorgestellten Verfahren nach Earley (1970) entschieden. Dieses benötigt im schlechtesten Fall ebenfalls $c_1 \cdot |w|^3 + c_2$ Schritte, hat aber den Vorteil, dass dieses Verfahren für eindeutige kontextfreie Sprachen nur $c_1 \cdot |w|^2 + c_2$ Schritte benötigt und sogar häufig in Linearzeit arbeitet.

Unter Verwendung der schnellen Matrixmultiplikation von Valiant und Strassen ist dieses Problem sogar in $|w|^{2,7}$ zu lösen. Spätere Verbesserungen ergaben Laufzeiten proportional zu $|w|^{2,4\dots}$ und sogar zu $|w|^{2,36\dots}$. Da dieses Problem in den meisten praktischen Fällen gar nicht für beliebige kontextfreie Sprachen gelöst werden muss, ist die Frage nach dem schnellsten Algorithmus für dieses Problem aber ohnehin eher von theoretischem Interesse.

Die Syntax von Programmiersprachen basiert in der Regel auf eindeutigen oder gar deterministischen Sprachen, die durch weitere Einschränkungen letztlich sogar nicht mehr kontextfrei bleiben. Trotzdem ist das Herz jedes Compilers ein Analyseverfahren für spezielle, meist deterministische, kontextfreie Grammatiken. Für die Definition von deterministischen kontextfreien Sprachen sei wiederum auf Kapitel 5 verwiesen.

4.7. Wichtige Abschlusseigenschaften

Im Folgenden werden wir untersuchen, ob die Familie der kontextfreien Sprachen ähnlich gute Eigenschaften besitzt, wie die Familie der regulären Mengen.

4.36 Theorem

Die Familie \mathcal{Cf} ist abgeschlossen gegenüber den drei regulären Operatoren \vee , \cdot und $*$:

1. Vereinigung, d.h. $\mathcal{Cf} \vee \mathcal{Cf} \subseteq \mathcal{Cf}$
2. Komplexprodukt, d.h. $\mathcal{Cf} \cdot \mathcal{Cf} \subseteq \mathcal{Cf}$
3. Kleene'sche Hülle (Sternbildung), d.h. $\mathcal{Cf}^* \subseteq \mathcal{Cf}$

Beweis: Für $i \in \{1, 2\}$ seien kontextfreie Sprachen L_i gegeben durch $L_i := L(G_i)$ für $G_i := (V_{i,N}, V_{i,T}, P_i, S_i)$.

1. $L_1 \cup L_2 = L(G_3)$ für $G_3 := (V_{1,N} \uplus V_{2,N} \uplus \{S_3\}, V_{1,T} \cup V_{2,T}, P_3, S_3)$ mit $P_3 := P_1 \cup P_2 \cup \{S_3 \rightarrow S_1, S_3 \rightarrow S_2\}$
2. $L_1 \cdot L_2 = L(G_4)$ für $G_4 := (V_{1,N} \uplus V_{2,N} \uplus \{S_4\}, V_{1,T} \cup V_{2,T}, P_4, S_4)$ mit $P_4 := P_1 \cup P_2 \cup \{S_4 \rightarrow S_1 S_2\}$
3. $L_1^* = L(G_5)$ für $G_5 := (V_{1,N} \uplus \{S_5\}, V_{1,T}, P_5, S_5)$ mit $P_5 := P_1 \cup \{S_5 \rightarrow S_1 S_5, S_5 \rightarrow \lambda\}$

□

4.37 Theorem

Die Familie der kontextfreien Sprachen ist **nicht** abgeschlossen gegenüber folgenden Operatoren:

1. Durchschnittsbildung
2. Komplementbildung
3. Bildung einer Mengendifferenz

Beweis: Wir zeigen zunächst 1. durch Angabe eines Gegenbeispiels:

$L_1 := \{a^n b^n \mid n \in \mathbb{N}\}$ wie auch $L_2 := \{b^m c^m \mid m \in \mathbb{N}\}$ sind kontextfreie Sprachen, deren Grammatiken leicht konstruiert werden können. Wegen $\mathcal{Reg} \subseteq \mathcal{Cf}$ und Theorem 4.36 sind dann auch die Sprachen $L_3 := L_1 \cdot \{c\}^*$ und $L_4 := \{a\}^* \cdot L_2$ kontextfrei. Wäre die Familie \mathcal{Cf} gegenüber Durchschnittsbildung abgeschlossen, so wäre nun auch $L_5 := L_3 \cap L_4 = \{a^n b^n c^n \mid n \in \mathbb{N}\} \in \mathcal{Cf}$. L_5 ist nun aber identisch zu der Sprache L_1 aus Beispiel 4.31 auf Seite 112, für die wir bereits mit Hilfe des Pumping-Lemmas gezeigt haben, dass sie nicht kontextfrei ist. Somit ist die Menge L_5 also nicht kontextfrei und die Durchschnittsbildung kann keine Abschlusseigenschaft für \mathcal{Cf} sein.

2. folgt aus 1.:

$$\begin{aligned} \overline{L_5} &= \{a, b, c\}^* \setminus L_5 = L_6 \cup L_7 \cup L_8 \text{ mit} \\ L_6 &:= \{a^r b^s c^t \mid r \neq s, r, s, t \in \mathbb{N}\}, \\ L_7 &:= \{a^r b^s c^t \mid s \neq t, r, s, t \in \mathbb{N}\} \text{ und} \\ L_8 &:= \overline{a^* b^* c^*} \end{aligned}$$

4. Kontextfreie Sprachen und Chomsky-Typ-2-Grammatiken

Letztere Menge L_8 ist regulär und damit kontextfrei. L_6 wird mit nachstehenden Regeln erzeugt, bei denen alle Großbuchstaben Nonterminale sind:

$$\begin{array}{ll} S_6 \longrightarrow ABC, & S_6 \longrightarrow A'B'C, \\ A \longrightarrow aA, & A \longrightarrow a, \\ B \longrightarrow aBb, & B \longrightarrow \lambda, \\ A' \longrightarrow aA'b, & A' \longrightarrow \lambda, \\ B' \longrightarrow bB', & B' \longrightarrow b, \\ C \longrightarrow cC, & C \longrightarrow \lambda \end{array}$$

Es ist leicht nachzuprüfen, dass gilt:

$$AB \xRightarrow{*} a^r b^s c^t, \forall r, s, t \in \mathbb{N} \text{ mit } r > s \wedge A'B' \xRightarrow{*} a^r b^s c^t, \forall r, s, t \in \mathbb{N} \text{ mit } r < s$$

Eine Grammatik für L_7 ist ganz entsprechend zu bilden. Damit ist $\overline{L_5}$ als Vereinigung kontextfreier Sprachen nach Theorem 4.36 selbst kontextfrei, ihr Komplement jedoch nicht, womit 2. vollständig gezeigt ist.

Da die Komplementbildung eine spezielle Mengendifferenz ist, folgt 3. sofort aus 2. \square

Obwohl der Durchschnitt zweier kontextfreier Sprachen in der Regel nicht kontextfrei ist, kann eine etwas schwächere, aber dennoch sehr bedeutende, Abschlusseigenschaft bezüglich der Durchschnittsbildung gezeigt werden. Dieses häufig verwendete Resultat besagt, dass die Familie der kontextfreien Sprachen gegenüber Durchschnittsbildung mit regulären Mengen abgeschlossen ist.

4.38 Theorem

Für $L \in \mathcal{Cf}$ und $R \in \mathcal{Reg}$ gilt $L \cap R \in \mathcal{Cf}$, kurz $\mathcal{Cf} \wedge \mathcal{Reg} \subseteq \mathcal{Cf}$.

Beweis: Sei $L = L(G)$ für eine CFG $G = (V_N, V_T, P, S)$ und $R = L(A)$ für einen NFA $A = (Z, \Sigma, K, Z_{\text{start}}, Z_{\text{end}})$. Wir konstruieren die neue CFG $G' := (V'_N, V'_T, P', S')$ mit den neuen Hilfszeichen $V'_N := \{[z, X, z'] \mid z, z' \in Z \text{ und } X \in V_N\} \cup \{S'\}$ und den R und L gemeinsamen Terminalzeichen $V'_T := \Sigma \cap V_T$.

O.B.d.A. nehmen wir an, dass G in Chomsky-Normalform vorliegt und definieren daraus die Regelmengemenge P' . Für jede Produktion $A \longrightarrow BC \in P$ seien folgende Produktionen in P' enthalten:

$$[z, A, z''] \longrightarrow [z, B, z'][z', C, z''] \in P' \text{ für alle } z, z', z'' \in Z$$

Weiterhin seien für jede Produktion $A \longrightarrow a$ in P folgende Produktionen in P' :

$$[z, A, z'] \longrightarrow a \in P' \text{ für alle } (z, a, z') \in K$$

Ausserdem werden als Startproduktionen zu P' folgende Produktionen hinzugenommen:

$$S' \longrightarrow [z, S, z'] \in P' \text{ für alle } z \in Z_{\text{start}} \text{ und } z' \in Z_{\text{end}}$$

Es gilt $L(G') = L \cap R$, denn in jeder abgeleiteten Satzform in G' bildet die erste Komponente des Nonterminals eine Folge von Zuständen, die mit dem Startzustand von A beginnt und in der zweiten Komponente des letzten Zeichens mit einem Endzustand endet. Genau dann, wenn die Kante (z, a, z')

in K vorkommt, kann das Terminalsymbol a aus $[z, A, z']$ erzeugt werden, so dass zu jedem generierten Wort $w \in L$ immer auch ein Erfolgspfad in A gehört. Dank der vielen Produktionen in G' wird auch jeder mögliche Erfolgspfad von A in den Satzformen erzeugt, und es folgt $L(G') = L \cap R$. \square

Wir betrachten nun als Beispiel für eine indirekte Anwendung des $uvwx$ -Theorems die Sprache $L := \{a^k b^l c^m d^n \mid k, l, m, n \in \mathbb{N} : k = 0 \text{ oder } l = m = n\}$. Der Versuch, mit dem Pumping-Lemma zu beweisen, dass L nicht kontextfrei ist, muss fehlschlagen, denn wenn $z \in L$ den Buchstaben a nicht enthält, so gilt dies auch für alle Wörter uv^iwx^iy , ganz gleich welche Zerlegung gewählt wurde.

Enthält das Wort z das Symbol a , so wählen wir eine Zerlegung mit $u = v = w = \lambda$, $x = a$ und y als geeignetem Suffix. Alle Wörter der Form uv^iwx^iy sind dann wieder Elemente der Menge L . Also kommt kein Widerspruch zu der Annahme L sei kontextfrei zustande. Trotzdem ist L nicht kontextfrei, und das läßt sich so beweisen:

Wäre L kontextfrei, so wäre auch $L \cap \{a\}\{b\}^*\{c\}^*\{d\}^* = \{ab^n c^n d^n \mid n \in \mathbb{N}\}$ eine kontextfreie Sprache (nach Theorem 4.38). Von letzterer Menge kann unter Anwendung des $uvwx$ -Theorems leicht gezeigt werden (siehe Beweis zu 1. von Theorem 4.37), dass sie nicht kontextfrei ist. Also konnte auch L dies nicht sein, obwohl das Pumping-Lemma nicht direkt anwendbar war.

Es sind schärfere Ausformulierungen des Pumping-Lemmas bekannt, mit denen es möglich ist, direkt zu zeigen, dass L nicht kontextfrei ist. Aber auch diese Varianten stellen keine *notwendige Bedingung* für die Eigenschaft einer Menge kontextfrei zu sein dar, d.h. die Folgerung des Pumping-Lemmas kann auch für Sprachen zutreffen, die nicht kontextfrei sind. In eigentlich allen Fällen kommt man mit dem hier verwendeten $uvwx$ -Theorems aus, wenn dazu noch einfache Abschlusseigenschaften der Familie \mathcal{Cf} , wie oben exemplarisch gezeigt, hinzugenommen werden.

5. Kellerautomaten

Die regulären Mengen hatten wir in Kapitel 3 als die von endlichen Automaten akzeptierten Mengen kennengelernt. Die kontextfreien Grammatiken hatten wir im vorangegangenen Kapitel verwendet, um die kontextfreien Sprachen zu definieren. Sie wurden ursprünglich 1955 – 1957 von N. Chomsky definiert und dienen seit 1958 als Basis für die Definition der **Syntax** (griech. *syntaxis*, Zusammenordnung, Lehre vom Satzbau) von Programmiersprachen. Damals war das noch ALGOL 58, für die M. Paul nach der Methode von K. Samelson (1955) einen Übersetzer mit einem *Kellerspeicher* für die Zuse Z22 schrieb. Die Bearbeitung von vollständig geklammerten arithmetischen Ausdrücken, mit einem nach dem Kellerprinzip arbeitenden Zwischenspeicher, wurde schon 1951 von H. Rutishauser (1918 – 1970) entwickelt. Seit 1959 durch F. L. Bauer und K. Samelson, formalisiert durch Oettinger 1961 sowie 1962 von D. Knuth und 1963 von M. P. Schützenberger, wurden *Kellerautomaten* für die Analyse von kontextfreien Sprachen verwendet. Wir betrachten den Kellerautomaten zunächst als Automatenmodell zum Akzeptieren beliebiger kontextfreier Sprachen. Erst in Kapitel 6 werden wir einen kleinen Einblick in die Methoden der deterministischen Syntaxanalyse geben.

5.1. Nichtdeterministische Kellerautomaten

In diesem Abschnitt führen wir den nichtdeterministischen Kellerautomaten als eine Erweiterung des nichtdeterministischen endlichen Automaten ein. Während der NFA nur in seinen (endlich vielen) Zuständen Informationen zwischenspeichern konnte, hat der Kellerautomat einen Keller- oder Stapelspeicher mit unbegrenzter Kapazität. Dieser Speicher hat die für Kellerspeicher üblichen Zugriffsbeschränkungen, d.h. es kann immer nur auf das zu oberst liegende Element lesend zugegriffen werden. Auch Schreibvorgänge sind nur in Form eines Anfügens am oberen Rand des Speichers möglich. Trotz dieser Einschränkungen werden wir schnell zu dem Schluß kommen, dass es sich hierbei tatsächlich um ein mächtigeres Automatenmodell handelt, als der endliche Automat es war.

Wir beginnen mit der formalen Definition des nichtdeterministischen Kellerautomaten und führen danach die wichtigsten Begriffe und Eigenschaften ein, die wir für einen Vergleich der Mächtigkeit mit anderen Automatenmodellen sowie Grammatiken benötigen.

Wie schon bei den endlichen Automaten ist auch hier zu beachten, dass ein *nichtdeterministischer* Kellerautomat durchaus auch *deterministisch* sein kann. Er *darf* aber im Gegensatz zur deterministischen Variante auch *nicht deterministisch* sein.

5.1 Definition (nichtdeterministischer Kellerautomat)

Ein **nichtdeterministischer Kellerautomat** (PDA für *push down automaton*) ist ein Tupel $A = (Z, \Sigma, \Gamma, K, Z_{start}, Z_{end}, \perp)$, wobei gilt:

5. Kellerautomaten

Z ist eine endliche Menge von Zuständen.

Σ ist ein endliches Alphabet von **Eingabesymbolen**.

Γ ist ein endliches Alphabet von **Kellersymbolen**.

$K \subseteq Z \times \Sigma^* \times \Gamma^* \times Z$ ist die endliche **Zustandsüberföhrungsrelation**.

$Z_{\text{start}} \subseteq Z$ ist die Menge der **Startzustände**.

$Z_{\text{end}} \subseteq Z$ ist die Menge der **Endzustände**.

$\perp \in \Gamma$ ist das **Kellerbodenzeichen** oder **Kellerbodensymbol**.

Wir stellen uns den Kellerautomaten folgendermaßen vor: Er besitzt eine endliche Steuerung, ein Eingabeband mit einem Lesekopf, der auf dem Eingabewort aus Σ^* nur von links nach rechts bewegt werden kann¹ und einen Kellerspeicher, auf dem Symbole aus dem Kelleralphabet Γ abgelegt werden können. Als Anschauung hierfür kann Abbildung 5.1 dienen.

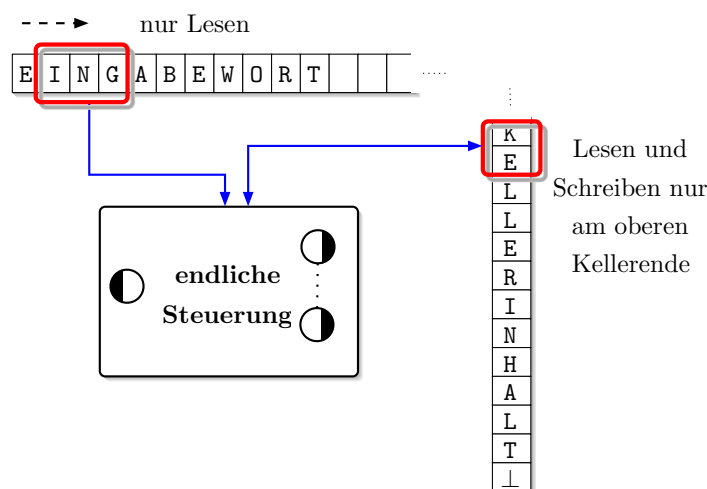


Abbildung 5.1.: Das Grundmodell eines Kellerautomaten

In jedem „Schritt“, den ein Kellerautomat macht, kann er – gemäß der Angabe in der Zustandsüberföhrungsrelation K – ein Symbol auf dem Eingabeband lesen, gleichzeitig ein Teilwort vom oberen Rand des Kellers und in Abhängigkeit dieser beiden Faktoren den Zustand in der endlichen Steuerung wechseln. Man kann sich dies so vorstellen, als hätte der Lesekopf ein sich automatisch anpassendes Fenster, dessen Breite von der Länge der für einen Zustandsübergang nötigen zu lesenden Zeichenkette bestimmt wird. Auf den Kellerinhalt kann – wie bereits gesagt – ausschließlich an seinem oberen Ende (*top*) zugegriffen werden. Dies geschieht wie beim Eingabeband ebenfalls mit Hilfe eines variablen Fensters, dessen Inhalt als ein Wort $v \in \Gamma^*$ dargestellt wird.

5.2 Notation

Wir beschreiben den Kellerinhalt durch ein Wort $v \in \Gamma^*$. Dabei verwenden wir die Konvention, dass das oberste Zeichen des Kellerinhaltes in v ganz am Anfang, d.h. links, steht.

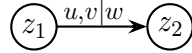
¹Dies entspricht genau der Vorstellung des Eingabewortes bei einem endlichen Automaten.

Das Kelleralphabet kann vom Eingabealphabet verschieden sein. Soll zum Beispiel nur die Anzahl des Vorkommens eines bestimmten Symbols zwischengespeichert werden, kann evtl. ein einziges Symbol im Kelleralphabet genügen. Auf jeden Fall enthält das Kelleralphabet aber ein spezielles Symbol, welches als einziges am Anfang einer jeden Rechnung den Keller belegt, das so genannte *Kellerbodensymbol* \perp . Es wird manchmal benötigt, um vor dem Einnehmen eines akzeptierenden Zustandes zu prüfen, ob der Keller „leer“ ist. Wie dies genau funktioniert, werden wir weiter unten genauer untersuchen.

Für die Angabe konkreter Kellerautomaten bedienen wir uns wiederum einer graphischen Automaten-darstellung. Abweichend von der Darstellung der endlichen Automaten, muss hier jedoch auch der Zugriff auf den Kellerspeicher an den Kanten notiert werden. Die Beschriftung folgt der Konvention aus Notation 5.3.

5.3 Notation

Zustandsüberführungen werden als beschriftete Kanten gezeichnet. Für $(z_1, u, v, w, z_2) \in K$ wird in einem Zustandsdiagramm die folgende Kante gezeichnet.



Die Arbeitsweise des PDA wird durch die Elemente der Zustandsübergangsrelation K , die auch als Einträge einer Tabelle aufgefasst werden können, festgelegt und hier zunächst für ein Element $(z_1, u, v, w, z_2) \in K$ informal beschrieben:

Falls sich der PDA in dem Zustand z_1 befindet, die Zeichenkette u unter dem Lesekopf steht und die Zeichenkette v den obersten Teil des Kellers bildet, wird der linke Rand des (variablen) Lesekopfes hinter die eben gelesene Eingabe gesetzt, der gelesene oberste Teil v des Kellerinhalts durch die Zeichenkette w ersetzt und dann der Zustand z_2 eingenommen.

Als *Konfiguration* wird im Allgemeinen eine vollständige Beschreibung eines bestimmten Systems bezeichnet. Die Systemstruktur wird dabei als bekannt vorausgesetzt, so daß die Konfiguration *dieses* Systems dann nur noch die aktuelle Speicherbelegung, inkl. der Belegung von Steuerregistern etc. umfassen muss. Um die Situation zu beschreiben, in der sich ein PDA befindet, benutzen wir ebenfalls so genannte Konfigurationen, die eine komplette Beschreibung des Automaten in der aktuellen Situation darstellen (*instantaneous description, ID*). Dazu gehört der aktuelle Zustand und die aktuelle Speicherbelegung genauso, wie die Kopfposition auf der Eingabe. Da ein Kellerautomat ein bereits gelesenes Symbol des Eingabewortes nicht ein zweites Mal besuchen kann, reicht für PDAs die folgende Definition der Konfiguration aus, in der statt der Kopfposition lediglich die verbleibende (Rest-)Eingabe notiert wird.

5.4 Definition (Konfiguration)

Eine **Konfiguration** des PDA $A = (Z, \Sigma, \Gamma, K, Z_{start}, Z_{end}, \perp)$ wird notiert als Element $k \in \Gamma^* \times Z \times \Sigma^*$.

In der Konfiguration $k := (v, z, w)$ beschreibt v den aktuellen Kellerinhalt, wobei das erste Symbol von v das oberste Kellerzeichen (*top*-Symbol), z der aktuelle Zustand des PDA's und w die noch zu lesende Eingabe ist. Hier enthält der passende Präfix von w gerade die Zeichen, die in dem Fenster unter dem Lesekopf sind. In der Anfangskonfiguration besteht das Kellerwort v nur aus dem Kellerbodensymbol,

5. Kellerautomaten

also $v = \perp$. Meist wählen wir für die Zustandsbezeichner Z und die Eingabe- und Kellersymbole $\Sigma \cup \Gamma$ disjunkte Mengen und schreiben die Konfiguration (v, z, w) kurz als Zeichenkette $vzw \in \Gamma^* \cdot Z \cdot \Sigma^*$. In diesem Fall trennt der aktuelle Zustandsbezeichner z das gerade auf dem Keller stehende Wort v von dem Teil der Eingabe, der noch zu bearbeiten ist (das ist die Zeichenkette w). In der Konfiguration vzw kann der PDA ein Teilwort vom oberen Rand des Kellers lesen (ein Präfix von v) und einige Symbole von der Eingabe lesen (ein Präfix von w) und in Abhängigkeit dieser beiden gelesenen Teilwörter den Zustand wechseln und ein Wort auf den Keller schreiben.

Wie bei den endlichen Automaten, wollen wir mit der folgenden Definition die (einschrittige) Übergangsrelation definieren und zu einer (mehrschrittigen) Überführungsrelation verallgemeinern.

5.5 Definition (Überführungsrelation)

Seien $u, u', v \in \Gamma^*$, $w, w' \in \Sigma^*$ und $z, z' \in Z$. Zwischen Konfigurationen eines Kellerautomaten A wird die **Überführungsrelation** (Rechenschritt-, Transitionsrelation) $\vdash \subseteq \Gamma^* \cdot Z \cdot \Sigma^* \times \Gamma^* \cdot Z \cdot \Sigma^*$ definiert durch:

$$uvzww' \vdash u'vz'w' \text{ genau dann, wenn } \exists(z, w, u, u', z') \in K$$

Wie üblich bezeichnet \vdash^* die reflexive, transitive Hülle von \vdash und wir schreiben \vdash_A^* , wenn andernfalls unklar ist, zu welchem PDA diese Überführungsrelation gehört.

Auch Kellerautomaten werden verwendet, um formale Sprachen zu definieren. Aus diesem Grunde müssen wir noch formal definieren, was hier die akzeptierte Sprache sein soll. Es gibt für Kellerautomaten zwei unterschiedliche Arten, Sprachen zu akzeptieren. Die erste entspricht derjenigen Art, die wir bereits bei endlichen Automaten eingeführt hatten. Sie wird durch das Erreichen eines Endzustandes beschrieben und beachtet nicht den dabei erreichten Kellerinhalt. Die zweite Variante hingegen beachtet nur den Kellerinhalt und wird so definiert, dass jedes Wort akzeptiert wird, für das es eine Rechnung existiert, die den Keller komplett leert. In diesem Fall darf nicht einmal mehr das Kellerbodensymbol im Kellerspeicher verbleiben. Beide Möglichkeiten wollen wir nun formal definieren.

5.6 Definition (mit Endzustand akzeptierte Sprache)

Die vom nichtdeterministischen PDA $A = (Z, \Sigma, \Gamma, K, Z_{\text{start}}, Z_{\text{end}}, \perp)$ mit **Endzustand akzeptierte Sprache** $L(A)$ ist

$$L(A) := \{w \in \Sigma^* \mid \exists z_0 \in Z_{\text{start}} : \exists z_e \in Z_{\text{end}} : \exists v \in \Gamma^* : \perp z_0 w \vdash^* v z_e\}$$

5.7 Definition (mit leerem Keller akzeptierte Sprache)

Die vom nichtdeterministischen PDA A mit **leerem Keller akzeptierte Sprache** $N(A)$ ist

$$N(A) := \{w \in \Sigma^* \mid \exists z_0 \in Z_{\text{start}} : \exists z \in Z : \perp z_0 w \vdash^* z\}$$

5.8 Beispiel

Der PDA aus Abbildung 5.2 akzeptiert mit leerem Keller die Sprache $N(A) = \{a^n b^n \mid n \in \mathbb{N}, n \geq 1\}$. Beachten Sie, dass dieser PDA keinen Endzustand benötigt!

Verwenden wir denselben PDA mit z' als einzigen Endzustand, so erhalten wir $L(A) = \{a^n b^m \mid n, m \in \mathbb{N} \wedge n \geq 1 \wedge n \geq m\}$.

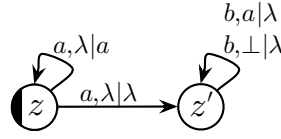


Abbildung 5.2.: Ein Beispiel-PDA ohne Endzustände

Analog zu den endlichen Automaten und Grammatiken definieren wir die Äquivalenz zweier Kellerautomaten über die Gleichheit der akzeptierten Sprachen. Wir verwenden hier die Akzeptierungsbedingung aus Definition 5.6. Wir werden später untersuchen, ob die Definitionen 5.6 und 5.7 zu unterschiedlichen Sprachfamilien führen.

5.9 Definition (Äquivalenz von Kellerautomaten)

Zwei Kellerautomaten A und B heißen **äquivalent** genau dann, wenn ihre mit Endzustand akzeptierten Sprachen gleich sind, d.h. $L(A) = L(B)$ gilt.

5.2. Deterministische Kellerautomaten und weitere Eigenschaften von PDAs

Wenn wir eine deterministische Arbeitsweise des Kellerautomaten voraussetzen wollen, so ist das variable Fenster des Lesekopfes und der ähnlich flexible Puffer zum Zugriff auf den obersten Kellerinhalt hinderlich. Ähnlich den endlichen Automaten definieren wir zunächst den fast-buchstabierenden Kellerautomaten, der sowohl auf der Eingabe als auch vom Kellerinhalt höchstens ein Symbol bei jedem Übergang lesen kann, aber auf dem Keller noch ein beliebig langes Wort schreiben darf kann.

5.10 Definition (Eigenschaften von Kellerautomaten)

Ein Kellerautomat $A = (Z, \Sigma, \Gamma, K, Z_{start}, Z_{end}, \perp)$ heißt

- **fast-buchstabierend** genau dann, wenn $K \subseteq Z \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\}) \times \Gamma^* \times Z$ gilt.
- **buchstabierend** genau dann, wenn $K \subseteq Z \times (\Sigma \cup \{\lambda\}) \times \Gamma \times \Gamma^* \times Z$ gilt.
- **deterministisch**, (Abk.: DPDA) genau dann, wenn die folgenden Bedingungen erfüllt sind:
 1. $|Z_{start}| = 1$
 2. A ist buchstabierend, d.h. $K \subseteq Z \times (\Sigma \cup \{\lambda\}) \times \Gamma \times \Gamma^* \times Z$
 3. Zu jedem $(z, x, X) \in Z \times (\Sigma \cup \{\lambda\}) \times \Gamma$ gibt es höchstens ein $(w, z') \in \Gamma^* \times Z$ mit $(z, x, X, w, z') \in K$.
 4. Falls $(z, \lambda, X, w, z') \in K$ ist, so gilt $\forall x \in \Sigma : \forall z'' \in Z : (z, x, X, w', z'') \notin K$.

Die Familie der von DPDA's mit Endzustand akzeptierten Sprachen wird mit $det\mathcal{Cf}$ bezeichnet.

Die Sprachfamilie $det\mathcal{Cf}$ bekommt einen eigenen Namen, da sich zeigen wird, dass sie sich von der Familie der von nichtdeterministischen Kellerautomaten akzeptierbaren Sprachen unterscheidet. Nach einigen

5. Kellerautomaten

einfachen Konstruktionen auf PDAs widmen wir uns dem Vergleich von kontextfreien Grammatiken und Kellerautomaten.

Es ist zu beachten, dass ein DPDA in jedem Schritt ein Symbol vom Keller lesen muss!

5.11 Theorem

Zu jedem PDA $A = (Z, \Sigma, \Gamma, K, Z_{\text{start}}, Z_{\text{end}}, \perp)$ kann effektiv ein äquivalenter fast-buchstabierender PDA $B := (Z', \Sigma, \Gamma', K', Z'_{\text{start}}, Z'_{\text{end}}, \perp)$ konstruiert werden.

Beweis: Zunächst sorgen wir dafür, dass die Eingabe immer durch Lesen von höchstens einem Symbol $a \in \Sigma$ geschieht. Der neu zu konstruierende äquivalente PDA $C := (Z'', \Sigma, \Gamma'', K'', Z''_{\text{start}}, Z''_{\text{end}}, \perp)$ liest die Eingabe Zeichen für Zeichen und speichert die jeweils gelesene Symbolkette in einem Zustand, der somit als Puffer benutzt wird. $Z'' := Z \cup \{[z, u'] \mid (z, u, v, w, z') \in K \wedge u' \in \Sigma^* \text{ ist echter Präfix von } u \text{ mit } |u'| \geq 1\}$. Jede Zustandsüberführung $k = (z, u, v, w, z') \in K$ mit

$$\exists n \geq 2 : \forall 1 \leq i \leq n : u_i \in \Sigma \wedge u = u_1 u_2 \dots u_n$$

wird ersetzt durch folgende neue Zustandsüberführungen:

$$(z, u_1, v, \lambda, [z', u_1]), ([z', u_1], u_2, \lambda, \lambda, [z', u_1 u_2]), \dots, ([z', u_1 \dots u_{n-1}], u_n, \lambda, w, z')$$

In K'' werden ansonsten alle Zustandsüberführungen von A übernommen, bei denen nur ein oder gar kein Eingabesymbol gelesen wird, bei denen also $|u| \leq 1$ ist. Es ist leicht zu sehen, dass $L(C) = L(A)$ gilt. Denselben Pufferungsmechanismus wendet man an, um auch auf dem Keller stets höchstens ein Symbol zu lesen und bei Bedarf zu ersetzen. Man erhält so den gewünschten PDA $B := (Z', \Sigma, \Gamma', K', Z'_{\text{start}}, Z'_{\text{end}}, \perp)$. Diesen Beweis mögen die Leser(innen) selbst ausführen. \square

5.3. Kellerautomaten und kontextfreie Sprachen

Bei endlichen Automaten hatten wir ein Eingabewort stets dann akzeptiert, wenn durch Lesen des gesamten Wortes ein Endzustand erreicht werden konnte, bei Kellerautomaten kennen wir nun aber zwei, möglicherweise verschiedene Akzeptierungsweisen: Mit leerem Keller wie in Beispiel 5.8 oder auch mit Endzustand, wie im folgenden Beweis von Theorem 5.12. Anders als dort jedoch sind im allgemeinen die mit leerem Keller akzeptierten Sprachen selbst bei äquivalenten Kellerautomaten unterschiedlich. Die Leser(innen) mögen ein konkretes Beispiel dafür selbst angeben.

5.12 Theorem

Zu jeder kontextfreien Grammatik $G = (V_N, V_T, P, S)$ kann effektiv ein Kellerautomat $A_G := (Z, \Sigma, \Gamma, K, Z_{\text{start}}, Z_{\text{end}}, \perp)$ konstruiert werden, für den $N(A_G) = L(A_G) = L(G)$ gilt.

Beweis: Sei $G = (V_N, V_T, P, S)$ eine beliebige CFG. Der hier konstruierte Kellerautomat $A_G := (Z, \Sigma, \Gamma, K, Z_{\text{start}}, Z_{\text{end}}, \perp)$ besitzt nur drei Zustände, $Z := \{z_0, z_1, z_2\}$, verwendet $\Sigma := V_T$, und benutzt das Gesamtalphabet von G zusammen mit dem Kellerbodenzeichen $\Gamma := V_N \cup V_T \cup \{\perp\}$ als Kellersymbole. Der PDA hält stets einen Teil der bei einer Linksableitung erzeugten Satzform im Keller. Zuerst wird das Startsymbol der CFG auf den Keller geschrieben und bei der weiteren Bearbeitung

das jeweils oberste Kellerzeichen entweder durch die rechte Seite einer passenden Produktion ersetzt oder, wenn dies ein Terminalzeichen ist, mit der Eingabe verglichen und bei Gleichheit gelöscht. Erst wenn dann die Eingabe vollständig gelesen wurde und das Kellerbodensymbol \perp wieder alleine auf dem Keller zu finden ist, wurde das Eingabewort abgeleitet. Dann erst kann in den Endzustand übergegangen und der Keller endgültig gelöscht werden. Der Zustandsgraph aus Abbildung 5.3 beschreibt den PDA vollständig.

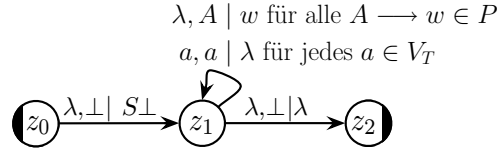


Abbildung 5.3.: Dieser PDA simuliert eine CFG

Ein formaler Beweis der Korrektheit dieser Konstruktion ist einfach und den Leser(inne)n überlassen. \square

Nachdem wir nun ein Maschinenmodell gefunden haben, mit dem jede kontextfreie Sprache, wenn auch nichtdeterministisch, akzeptiert werden kann, stellen sich zwei Fragen: „Erkennen diese Automaten ausschließlich kontextfreie Sprachen, oder akzeptieren sie sogar nicht kontextfreie Sprachen?“ und „Kann jede kontextfreie Sprache von einem DPDA akzeptiert werden?“. Auch wenn letztere Frage mit nein beantwortet werden muss, läßt sich zum Glück zeigen, dass Kellerautomaten nur kontextfreie Sprachen akzeptieren.

Als Lemma zeigen wir zunächst, dass jede von einem PDA mit Endzustand akzeptierbare Sprache auch von einem PDA mit leerem Keller akzeptiert werden kann. Dieser PDA kann sogar effektiv konstruiert werden.

5.13 Lemma

Zu jedem mit Endzustand akzeptierenden PDA $A := (Z, \Sigma, \Gamma, K, Z_{\text{start}}, Z_{\text{end}}, \perp)$ kann effektiv ein PDA $B := (Z', \Sigma, \Gamma', K', Z'_{\text{start}}, Z'_{\text{end}}, \perp)$ mit $L(A) = N(B)$ konstruiert werden.

Beweis: Nach Theorem 5.11 können wir o.B.d.A. davon ausgehen, dass $A := (Z, \Sigma, \Gamma, K, Z_{\text{start}}, Z_{\text{end}}, \perp)$ ein fast-buchstabierender PDA ist. Ein PDA $B := (Z', \Sigma, \Gamma', K', Z'_{\text{start}}, Z'_{\text{end}}, \perp)$ mit $L(A) = N(B)$ wird folgendermaßen konstruiert: Zunächst nehmen wir ein neues Symbol $\$$ hinzu, das als Kellerbodenzeichen benutzt wird und nicht gelöscht werden kann. Also $\Gamma' := \Gamma \uplus \{\$$. Dies ist nötig, damit kein leerer Keller erzeugt werden kann wenn die Eingabe nicht akzeptiert werden darf, da der ursprüngliche Automat ja möglicherweise bereits vor dem Akzeptieren im Endzustand den Keller leeren konnte.

Jeden der ehemaligen Startzustände von A erreicht man in B durch die zusätzlichen Zustandsübergänge der Menge $K_1 := \{(z_{\text{begin}}, \lambda, \perp, \perp \$, z) \mid z \in Z_{\text{start}}\}$. Dazu wird mit $Z' := Z \uplus \{z_{\text{begin}}, z_{\text{empty}}\}$ ein neuer Startzustand z_{begin} , sowie ein neuer Zustand z_{empty} zu Z hinzugefügt, damit mit neuen Zustandsübergängen der Menge $K_2 := \{(z_e, \lambda, \lambda, \lambda, z_{\text{empty}}) \mid z_e \in Z_{\text{end}}\}$ sowie „Schleifen“ der Menge $K_3 := \{(z_{\text{empty}}, \lambda, a, \lambda, z_{\text{empty}}) \mid a \in \Gamma'\}$ der Keller nur nach Akzeptieren mit Endzustand endgültig gelöscht werden kann. Wir erhalten $K' := K \cup K_1 \cup K_2 \cup K_3$.

Einmal in den Zustand z_{empty} gelangt, kann der neue PDA nichts anderes tun als den gesamten Keller zu leeren, ohne dabei weitere Eingabe zu lesen. Andererseits kann dieser Fall immer nur dann eintreten,

5. Kellerautomaten

wenn der PDA A das Eingabewort akzeptiert hatte. Da A ein fast-buchstabierender PDA war, ist auch B ein fast-buchstabierender PDA. \square

Der im Beweis zu Lemma 5.13 konstruierte PDA B akzeptiert zwar mit leerem Keller dieselbe Sprache, die der ursprüngliche PDA A mit Endzustand akzeptiert hatte. Trotzdem dürfen wir hier gemäß Definition 5.9 *nicht* von äquivalenten PDAs sprechen!

5.14 Theorem

Für jeden PDA A ist $L(A)$ eine kontextfreie Sprache.

Beweis: Sei $B := (Z', \Sigma, \Gamma', K', Z'_{\text{start}}, Z'_{\text{end}}, \perp)$ der gemäß Lemma 5.13 zu A existierende fast-buchstabierende PDA mit $L(A) = N(B)$.

Ein PDA $C := (Z'', \Sigma, \Gamma'', K'', Z''_{\text{start}}, Z''_{\text{end}}, \perp)$, der auf dem Keller stets ein Symbol ersetzen muss und somit buchstabierend ist, wird auf der Basis des PDA B definiert. Für die Konstruktion dieses Kellerautomaten wird für jeden Zustandsübergang $(z, x, \lambda, v, z') \in K'$ mit $x \in \Sigma \cup \{\lambda\}$ die Menge $K_4 := \{(z, x, X, vX, z') \mid X \in \Gamma'\}$ zu K'' hinzugefügt. Durch K_4 wird es dem PDA ermöglicht, die Wirkung der vormaligen Zustandsübergänge, für die nichts vom Keller gelesen werden musste, mit dem Lesen eines beliebigen Kellersymbols zu simulieren. Das gelesene Symbol wird auf jeden Fall sofort wieder auf den Keller geschrieben. Die übrigen Zustandsübergänge werden von K' übernommen, d.h. $K'' := K' \setminus \{(z, x, \lambda, v, z') \in K' \mid x \in \Sigma \cup \{\lambda\}\} \cup K_4$. Dadurch bleiben auch die Übergänge aus K' zum endgültigen Leeren des Kellers in K'' erhalten.

Offensichtlich gilt somit $N(C) = L(A)$ und K'' hat die Eigenschaften, die für das weitere Vorgehen, die sogenannte „Tripelkonstruktion“ einer kontextfreien Grammatik, benötigt werden.

Für die Tripelkonstruktion verwenden wir ein Nonterminalalphabet mit dem die Zustandsübergänge des PDA notiert werden können. Es wird durch die Regeln der kontextfreien Grammatik eine Zustandsfolge des Kellerautomaten „geraten“. In der Grammatik existiert genau dann eine Ableitung eines Terminalwortes w , wenn die in der Satzform geratene Zustandsfolge zu einer Erfolgsrechnung des PDA auf w passt.

$$\begin{aligned} V_N &:= \{S\} \cup \{[z, X, z'] \mid X \in \Gamma'', z, z' \in Z''\} \\ G_C &:= (V_N, V_T, P, S) \text{ mit } V_T := \Sigma \text{ sei die CFG zum PDA } C \text{ und} \\ P &:= \{S \rightarrow [z, \$, z'] \mid z \in Z''_{\text{start}}, z' \in Z''\} \cup \\ &\quad \{[z, X, z'] \rightarrow a \mid (z, a, X, \lambda, z') \in K'', a \in \Sigma \cup \{\lambda\}, X \in \Gamma'', z, z' \in Z''\} \cup \\ &\quad \{[z, X, z_k] \rightarrow a[z', B_1, z_1] \cdot [z_1, B_2, z_2] \cdot \dots \cdot [z_{k-1}, B_k, z_k] \mid k \geq 1, z_i \in Z'', \\ &\quad (z, a, X, B_1 \cdot B_2 \cdot \dots \cdot B_k, z') \in K''\} \text{ die zugehörige Produktionsmenge.} \end{aligned}$$

Man beweist per Induktion, dass für diese Grammatik $L(G) = N(C)$ gilt. Wer es nicht selbst versuchen möchte, kann Hinweise zum Beweis in der Literatur finden, z.B. in [Schöning], [Hopcroft&Ullman] oder auch in [Wegener]. \square

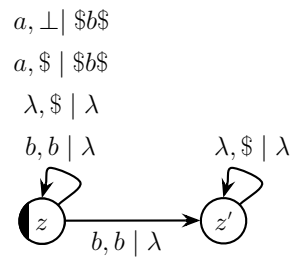


Abbildung 5.4.: PDA aus Beispiel 5.15

5.15 Beispiel

Betrachten wir den PDA aus Abbildung 5.4, so erfüllt dieser alle Bedingungen, die an den PDA C im Beweis von Theorem 5.14 gestellt wurden.

Wir können also den letzten Schritt der Konstruktion an diesem Beispiel durchführen. Als Nonterminalalphabet erhalten wir die Menge

$$V_N := \{S\} \cup \{[z_1, y, z_2] \mid z_1, z_2 \in \{z, z'\} \wedge y \in \{a, b, \perp, \$\}\}$$

und folgende Menge von Produktionen:

$$\begin{aligned} & \{ S \longrightarrow [z, \perp, z], \quad S \longrightarrow [z, \perp, z'], \\ & [z, b, z] \longrightarrow b, \quad [z, b, z'] \longrightarrow b, \\ & [z, \$, z] \longrightarrow \lambda, \quad [z', \$, z'] \longrightarrow \lambda \} \cup \\ & \{ [z, \$, z_3] \longrightarrow a[z, \$, z_1][z_1, b, z_2][z_2, \$, z_3] \mid z_1, z_2, z_3 \in \{z, z'\} \} \cup \\ & \{ [z, \perp, z_3] \longrightarrow a[z, \$, z_1][z_1, b, z_2][z_2, \$, z_3] \mid z_1, z_2, z_3 \in \{z, z'\} \}. \end{aligned}$$

Diese CFG ist offensichtlich nicht reduziert, denn die Nonterminale $[z', b, z']$, $[z', b, z]$ und $[z', \$, z]$ sind nicht terminierbar, können aber erreicht werden. Wenn alle Produktionen, die diese Hilfszeichen enthalten gestrichen werden, erhalten wir eine kleinere Grammatik mit folgenden Produktionen:

$$\begin{aligned} S &\longrightarrow [z, \$, z], \quad S \longrightarrow [z, \$, z'], \\ [z, b, z] &\longrightarrow b, \quad [z, b, z'] \longrightarrow b, \\ [z, \perp, z] &\longrightarrow a[z, \$, z][z, b, z][z, \$, z'] \\ [z, \perp, z] &\longrightarrow a[z, \$, z][z, b, z'] [z', \$, z'] \\ [z', \$, z'] &\longrightarrow \lambda, \quad [z, \$, z] \longrightarrow \lambda, \\ [z, \$, z] &\longrightarrow a[z, \$, z][z, b, z][z, \$, z] \\ [z, \$, z'] &\longrightarrow a[z, \$, z][z, b, z][z, \$, z'] \\ [z, \$, z'] &\longrightarrow a[z, \$, z][z, b, z'] [z', \$, z'] \end{aligned}$$

Weitere Umformungen dieser CFG ergeben die äquivalenten Grammatiken:

$$S \longrightarrow A, A \longrightarrow aAbA, A \longrightarrow ab, \text{ oder kürzer: } S \longrightarrow aSbS \mid ab.$$

5. Kellerautomaten

Das Verfahren der Tripelkonstruktion konnte zwar an diesem Beispiel illustriert werden, der im Beweis verwendete Automat aus dem vorangegangenen Lemma muss aber noch weiterreichende Bedingungen erfüllen!

Die zuletzt in Beispiel 5.15 erhaltene Grammatik ist eine eindeutige CFG für die bekannte Dyck-Sprache, d.h. die Sprache der wohlgeformten Klammerausdrücke mit dem Klammerpaar bestehend aus einer öffnenden Klammer a und einer schließenden Klammer b mit mindestens einem Klammerpaar. Formal ist die Dyck-Sprache in der nachstehenden Definition 5.16) erklärt.

5.16 Definition (Dyck-Sprache)

Die Dyck-Sprache D_1 ist definiert durch:

$$D_1 := \{w \in \{a, b\}^* \mid (w = uv) \rightarrow (|u|_a \geq |u|_b) \wedge (|w|_a = |w|_b)\}$$

Nach den bisher erzielten Resultaten können wir nun jede kontextfreie Sprache nicht nur durch eine kontextfreie Grammatik, sondern auch mit einem Kellerautomaten spezifizieren. Wie bei den kontextfreien Grammatiken, gibt es auch hier verschiedenen äquivalente Modelle bzw. Akzeptierungsbedingungen, was wir durch Theorem 5.17 ausdrücken.

5.17 Theorem

Folgende Aussagen sind äquivalent:

1. $L \in \mathcal{Cf}$
2. $L = L(A)$ für einen beliebigen PDA A
3. $L = L(A)$ für einen fast-buchstabierenden PDA A
4. $L = L(A)$ für einen buchstabierenden PDA A
5. $L = N(A)$ für einen fast-buchstabierenden PDA A

Beweis: Das Theorem folgt direkt aus Theorem 5.12, Lemma 5.13 und Theorem 5.14. □

In Theorem 5.17 werden Sprachfamilien miteinander verglichen. Die Aussage des Theorems ist, dass die Familie der durch kontextfreie Grammatiken erzeugbaren Sprachen dieselbe ist, wie die Familie der von Kellerautomaten mit Endzustand akzeptierbaren Sprachen. Außerdem sind die Familien der von buchstabierenden Kellerautomaten mit Endzustand und der von fast-buchstabierenden Kellerautomaten mit leerem Keller akzeptierbaren Sprachen ebenfalls äquivalent zu den vorgenannten Sprachfamilien. All diese Maschinenmodelle beschreiben also dieselbe Sprachfamilie, nämlich die kontextfreien Sprachen!

5.4. Deterministisch kontextfreie Sprachen

Bisher haben wir uns auf die Untersuchung von nichtdeterministischen Kellerautomaten konzentriert und festgestellt, dass diese genau die kontextfreien Sprachen akzeptieren. Ob auch die deterministischen Kellerautomaten dazu fähig sind, wollen wir nun beleuchten. Vom Standpunkt der Syntaxanalyse

von Programmiersprachen wäre solch ein Ergebnis von nicht unerheblichem Vorteil. Leider stellt sich jedoch heraus, dass die Familie detCf , der von deterministischen Kellerautomaten mit Endzustand akzeptierbaren (kontextfreien) Sprachen, eine *echte* Teilfamilie der Familie Cf ist! Für den Beweis dieser echten Inklusion werden wir einen Operator definieren, unter dessen Anwendung die deterministisch kontextfreien Sprachen abgeschlossen sind, nicht aber beliebige kontextfreie Sprachen.

5.18 Definition (präfixfrei)

Eine Sprache $L \subseteq \Sigma^*$ heißt **präfixfrei** genau dann, wenn

$$\forall w \in L : \forall v \in \Sigma^* : (wv \in L) \rightarrow (v = \lambda).$$

$\min(L) := \{w \in L \mid (w = uv \wedge u \in L) \rightarrow (v = \lambda)\}$ bezeichnet den **präfixfreien Anteil** der Sprache L .

5.19 Beispiel

Betrachten wir die Menge $M := \{a^m b^n \mid m, n \in \mathbb{N} : n \geq m\}$, so sehen wir leicht, dass $\min(M) = \{a^n b^n \mid n \in \mathbb{N}\}$ ist, denn es gibt für $n > m$ mindestens einen echten Präfix eines Wortes $a^m b^n$, der wieder als Wort der Menge M vorkommt. Also muss $n = m$ gelten.

Während bei den nichtdeterministischen PDA's die Akzeptierungsbedingung² nach Theorem 5.17 unerheblich war, ist dies bei der deterministischen Variante ein entscheidendes zusätzliches Kriterium. Theorem 5.20 wird als erster Schritt zum Beweis dieser Tatsache dienen.

5.20 Theorem

Eine deterministische kontextfreie Sprache $L \in \text{detCf}$ ist genau dann präfixfrei, wenn ein DPDA A existiert, der L mit leerem Keller erkennt, d.h. $L = N(A)$ gilt.

Beweis: Aus $L \in \text{detCf}$ folgt die Existenz eines DPDA A mit $L = L(A)$. Ist L zusätzlich präfixfrei, so können in A alle Kanten, die von einem Endzustand wegführen, in einen neuen Zustand „umgebogen“ werden, der nicht mehr verlassen werden kann, und in dem der gesamte Kellerinhalt gelöscht wird. Dies zeigt, dass $L = N(B)$ für den so veränderten DPDA B gilt. Ist andererseits $L = N(A)$ für einen DPDA, so ist natürlich $L \in \text{detCf}$ und der deterministische Automat A ist buchstabierend und kann bei leerem Keller keine weitere Bewegung mehr machen. \square

Aus dem obigen Resultat erhalten wir sofort als Korollar:

5.21 Korollar

Die präfixfreien deterministisch kontextfreien Sprachen bilden eine echte Teilfamilie der Familie detCf .

Beweis: Die Menge $\{a\}^* \{b\}^* = \{a^n b^m \mid m, n \in \mathbb{N}\}$ ist, da regulär, deterministisch kontextfrei, aber offensichtlich nicht präfixfrei. \square

Im vorigen Fall ist zu berücksichtigen, dass ein DPDA stets ein Symbol vom Keller lesen muss. Daher ist es nicht möglich, den endlichen Automaten ohne Benutzung des Kellers, der ja zu Beginn einer jeden Rechnung das Kellerbodensymbol \perp als Inhalt hat, als endliche Kontrolle eines DPDA zu verwenden und diesen dann mit leerem Keller akzeptieren zu lassen.

²gemeint sind die Akzeptierung mit Endzustand und die Akzeptierung bei leerem Keller

5.5. Abschlusseigenschaften der deterministisch kontextfreien Sprachen

In diesem Abschnitt befassen wir uns mit Abschlussoperatoren für die Familie der deterministisch kontextfreien Sprachen.

Der Beweis von Theorem 5.20 ließ schon erahnen, dass der präfixfreie Anteil einer deterministischen kontextfreien Sprache wieder deterministisch und kontextfrei ist. Theorem 5.22 zeigt, dass dies tatsächlich der Fall ist.

5.22 Theorem

Für $L \in \text{det Cf}$ ist auch $\min(L) \in \text{det Cf}$.

Beweis: Sei $L = L(A)$ für einen DPDA A , dann erhalten wir einen DPDA B für $\min(L)$ dadurch, dass wir in A alle Kanten entfernen, die von einem Endzustand wegführen. Diese Kanten dürfen nicht benutzt werden, um erneut in einen Endzustand zu geraten, denn $\min(L)$ ist präfixfrei, und wenn diese nie in einen Endzustand führen können, sind sie ohnehin nutzlos für das Akzeptieren eines Wortes. \square

Als Spezialfall eines Abschlusses gegenüber Durchschnittsbildung, zeigen wir nun den Abschluss der deterministisch kontextfreien Sprachen gegenüber Durchschnitt mit regulären Mengen.

5.23 Theorem

$\text{det Cf} \wedge \text{Reg} \subseteq \text{det Cf}$, d.h., die Familie der deterministischen kontextfreien Sprachen ist gegenüber Durchschnittsbildung mit regulären Mengen abgeschlossen.

Beweis: Da wir bisher noch keine grammatikalische Charakterisierung der deterministisch kontextfreien Sprachen kennen, können wir keinen dem Beweis von Theorem 3.70 entsprechenden Beweis verwenden. Es ist aber leicht, die endliche Kontrolle jedes DPDA mit einem deterministischen endlichen Automaten auf eine Weise parallel zu schalten, dass bei jedem Lesen eines Eingabesymbols sowohl die Transition des DPDA als auch die des DFA durchgeführt wird. Dies entspricht dem Vorgehen bei der Konstruktion eines Produktautomaten für den Durchschnitt zweier regulärer Sprachen.

$L \in \text{det Cf}$ werde von dem DPDA $A := (Z, X, Y, K, \{z_0\}, Z_{\text{end}})$ akzeptiert und $R \in \text{Reg}$ sei $L(B)$ für einen vDFA $B := (Z', X', \delta, z'_0, Z'_{\text{end}})$. Ähnlich der Konstruktion des Produktautomaten werde ein DPDA C mit $L(C) = L \cap R$ definiert durch:

$$C := (Z \times Z', X \cap X', Y, K', \{(z_0, z'_0)\}, Z_{\text{end}} \times Z'_{\text{end}})$$

mit der Kantenmenge K' definiert durch: $((p, p'), x, y, v, (q, q')) \in K'$ genau dann, wenn entweder

1. $(x \neq \lambda) \wedge ((p, x, y, v, q) \in K) \wedge (\delta(p', x) = q')$
oder
2. $x = \lambda \wedge ((p, x, y, v, q) \in K) \wedge (p' = q')$.

Das heisst, beide Automaten arbeiten synchron. Wenn der DPDA kein Symbol von der Eingabe liest, so wartet der vDFA in seinem Zustand bis zur nächsten Buchstabeneingabe. Akzeptiert wird, wenn beide Automaten akzeptieren. Da beide deterministisch waren ist deren „Produktkombination“ auch deterministisch!

□

Unter Verwendung der Theoreme 5.22 und 5.23 sind wir nun in der Lage, von gewissen kontextfreien Sprachen zu zeigen, dass diese *nicht* deterministisch sein können.

5.24 Definition (Palindrome)

Die Sprache aller Palindrome über dem Alphabet $\{a, b\}$ ist definiert durch:

$$PAL := \{w \in \{a, b\}^* \mid w = w^{rev}\}$$

5.25 Theorem

Die kontextfreie Sprache PAL ist eindeutig, aber nicht deterministisch.

Beweis: Dass PAL eindeutig kontextfrei ist folgt sofort aus der Existenz der eindeutigen CFG $G := (V_N, V_T, P, S)$ mit $L(G) = PAL$ mit den Produktionen

$$S \longrightarrow aSa \mid bSb \mid a \mid b \mid \lambda.$$

Wäre nun $PAL \in detCf$, dann wäre unter Anwendung von Theorem 5.23 auch die Menge $K := PAL \cap (ab)^+(ba)^+(ab)^+(ba)^+ \in detCf$. Wegen Theorem 5.22 müßte dann auch $\min(K) \in detCf$ gelten. Nun ergibt sich:

$$\min(K) = \{(ab)^m(ba)^n(ab)^n(ba)^m \mid 0 \leq n \leq m\}$$

und diese Sprache ist nicht einmal mehr kontextfrei, wie man mit dem $uvwxy$ -Theorem zeigen kann. Also kann PAL selbst nicht deterministisch kontextfrei gewesen sein. □

Die Idee, ein DPDA könnte beim Akzeptieren von z.B. $uu^{rev}uu^{rev} \in PAL$ nicht festlegen, ob bereits nach dem Lesen des ersten Teilwortes u die Mitte des Palindroms gefunden sei, oder erst nach uu^{rev} , macht das formal bewiesene Ergebnis plausibel, kann aber allein nicht als Beweis dienen!

Ein Beweis dafür, dass eine Sprache L von keinem DPDA akzeptiert wird, muss von jedem DPDA schlüssig zeigen, dass er nicht L akzeptiert. Nur zu zeigen, dass ein bestimmtes Vorgehen, d.h. ein möglicher Algorithmus, nicht zum Ergebnis führt, reicht nicht aus, denn es könnte ja ein anderes Vorgehen geben, welches nicht so offensichtlich ist, aber dennoch korrekt arbeitet.

In Beispiel 5.26 lernen wir eine weitere nicht deterministisch kontextfreie Sprache kennen und zeigen, dass ihr Komplement in Cf liegt. Da wir in Theorem 5.27 zeigen werden, dass in $detCf$ die Komplementbildung ein Abschlussoperator ist, kann diese Sprache aber auch nicht deterministisch kontextfrei sein!!

5.26 Beispiel

Betrachten wir die Sprache $COPY := \{w \in \{a, b\}^* \mid w = v \cdot v\}$. Wir können mit dem Pumping-Lemma für kontextfreie Sprachen zeigen, dass $COPY$ nicht kontextfrei ist. Es ist allerdings relativ leicht zu sehen, dass das Komplement $\overline{COPY} := \{a, b\}^* \setminus COPY$ kontextfrei ist.

Es gilt nämlich $\overline{COPY} = L(G) \cup R_{odd}$ für die CFG G mit folgenden 8 Produktionen:

$$\begin{aligned} S &\longrightarrow AB \mid BA \\ A &\longrightarrow CAC \mid a \\ B &\longrightarrow CBC \mid b \\ C &\longrightarrow a \mid b \end{aligned}$$

5. Kellerautomaten

und die reguläre Menge $R_{\text{odd}} := (aa + ab + ba + bb)^*(a + b)$.

Mit Hilfe der A -Produktionen werden Ableitungen $A \xRightarrow{*} uav$ erzeugt, bei denen die Wörter u und v die gleiche Länge haben. Entsprechendes gilt für die B -Produktionen, so dass ausgehend von A (bzw. von B) alle Wörter ungerader Länge mit dem Symbol a (bzw. b) in der Mitte erzeugt werden. Aus S können mithin nur Wörter gerader Länge in $\{a, b\}^*$ abgeleitet werden. Zu der Zeichenkette AB gibt es folgende Ableitungen:

$$AB \xRightarrow{*} u_a a v_a u_b b v_b$$

mit beliebigen Wörtern u_a, u_b, v_a und v_b , bei denen lediglich $|u_a| = |v_a|$, sowie $|u_b| = |v_b|$ sichergestellt ist. Da also $|v_a u_b| = |u_a| + |v_b| = |u_a v_b|$ gilt, steht bei einer vermuteten Zerlegung der abgeleiteten Zeichenkette $u_a a v_a u_b b v_b = w_1 w_2$ mit $|w_1| = |w_2|$ das Symbol a an der Position $|u_a| + 1$ in w_1 , während an der gleichen Position in w_2 das Symbol b zu finden ist. Daher gilt $L(G) = \{w \in \{a, b\}^* \mid w = w_1 w_2, |w_1| = |w_2|, w_1 \neq w_2\}$. Da kein Wort w mit ungerader Länge in $COPY$ vorkommt, die Wörter in $L(G)$ jedoch alle von gerader Länge sind, muss noch die reguläre Menge $R_{\text{odd}} = (aa + ab + ba + bb)^*(a + b)$ hinzugenommen werden. Da die Sprachfamilie \mathcal{Cf} gegenüber Vereinigung abgeschlossen ist, haben wir mit diesen Bemerkungen $\overline{COPY} \in \mathcal{Cf}$ gezeigt.

Mit Theorem 5.27 können wir zeigen, dass – wie auch $PAL - \overline{COPY}$ aus Beispiel 5.26 keine deterministisch kontextfreie Sprache sein kann.

5.27 Theorem

Die Familie der deterministisch kontextfreien Sprachen ist effektiv gegenüber Komplementbildung abgeschlossen, d.h. für $L \in \text{detCf}$ ist stets auch $\bar{L} \in \text{detCf}$.

Beweis (Idee): Grundsätzlich folgt man derselben Idee wie bei den endlichen Automaten, nämlich den Ansatz die Endzustände des DPDA mit denen aus der Menge $Z \setminus Z_{\text{end}}$ (also den Nicht-Endzuständen) zu vertauschen. Dies geht aber aus folgenden Gründen nicht so einfach wie zunächst angenommen, denn der DPDA kann ein Wort w aus unterschiedlichen Gründen ablehnen³:

1. Die Rechnung bricht ab, weil der Keller leer ist.
2. Das Wort w wird nicht vollständig gelesen, weil eine nötige Folgekonfiguration in der Rechnung nicht definiert ist.
3. Der DPDA gerät in eine Endlosschleife von Übergängen, ohne weitere Symbole von der Eingabe zu lesen.
4. Ein Wort kann auch mit einer Rechnung akzeptiert werden, bei der der DPDA ohne weiteres Lesen der Eingabe abwechselnd in End- und Nicht-Endzustände gerät.

In keinem der vier Fälle ändert das Vertauschen der Endzustände diese Situation, und diese Fälle müssen besonders bedacht und behandelt werden. Wir werden diesen Beweis nicht im Detail ausführen, sondern nur skizzieren, wie hierbei vorgegangen werden kann. In [Harrison] findet man dies auf zehn Druckseiten ausgeführt:

³Bei einem vDFA wurde ein Wort genau dann nicht akzeptiert, wenn es komplett gelesen wurde und danach kein Endzustand erreicht war. Eine andere Möglichkeit des Ablehnens gab es dort nicht!

Zu 1.: Neues Kellerbodenzeichen vor das alte setzen.

Zu 2.: Einführen eines neuen Zustandes p_{Senke} , aus dem keine Kante mehr herausführt. Bisher in anderen Zuständen nicht definierte Folgekonfigurationen führen jetzt alle in den Zustand p_{Senke} .

Zu 3.: Fall a): In der λ -Schleife wird der Keller nie kürzer, aber ein Endzustand kommt mindestens einmal vor. Wenn die Konfiguration xq den Beginn der λ -Schleife bedeutet, wird die Kante (q, λ, x, x', q') durch folgende neue Übergänge ersetzt: $\{(q, \lambda, x, x, p)\} \cup \{(p, b, y, y, p') \mid b \in \Sigma, y \in \Gamma\} \cup \{(p', b, y, y, p') \mid b \in \Sigma, y \in \Gamma\}$. Hierbei wird der neue Zustand p als neuer Endzustand definiert.

Fall b): Falls die Konfiguration xq der Beginn einer λ -Schleife ist, in der niemals ein Endzustand vorkommt, so wird die Kante (q, λ, x, x', q') durch die neuen Kanten $\{(q, \lambda, x, x, p')\} \cup \{(p', b, y, y, p') \mid b \in \Sigma, y \in \Gamma\}$ ersetzt.

Die Situation 4. sowie die Notwendigkeit, diese Fälle effektiv voneinander unterscheiden zu können, lassen sich nicht mehr kurz skizzieren, so dass auf die Literatur verwiesen werden muss. \square

Wir haben oben mehrfach gezeigt, dass eine eindeutige kontextfreie Sprache nicht deterministisch zu sein braucht. Wir können allerdings zeigen, dass zumindest umgekehrt jede deterministisch kontextfreie Sprache auch eindeutig sein muss.

Dieses Ergebnis ist für die Syntaxanalyse der auf kontextfreien Grammatiken basierenden Programmiersprachen von großer Bedeutung.

5.28 Theorem

Jede deterministische kontextfreie Sprache ist eindeutig.

Beweis (Idee): Wir benötigen die Tripelkonstruktion, um aus dem DPDA, der eine Sprache $L \in \text{detCf}$ akzeptiert, eine CFG zu erhalten. Diese Grammatik besitzt nun zu jeder akzeptierenden, eindeutig bestimmten, Rechnung des DPDA genau eine Linksableitung.

Da die Tripelkonstruktion bei einem PDA angewendet wird, der mit leerem Keller akzeptiert, werden wir die Sprache L um ein weiteres Endezeichen zu $L \cdot \{\phi\}$ ergänzen, wodurch die Sprache $L \cdot \{\phi\}$ präfixfrei ist und von einem DPDA mit leerem Keller akzeptiert werden kann.

Die Details sind mit dieser Beweisidee nicht vollständig ausgeführt. Der Beweis wird gemäß Abbildung 5.5 in einzelne Schritte zerlegt. \square

Nun haben wir die wichtigsten Eigenschaften der kontextfreien Sprachen anhand von kontextfreien Grammatiken und anhand von Kellerautomaten behandelt. Im nächsten Kapitel werden wir uns der praktischen Anwendung für die Syntaxanalyse zuwenden.

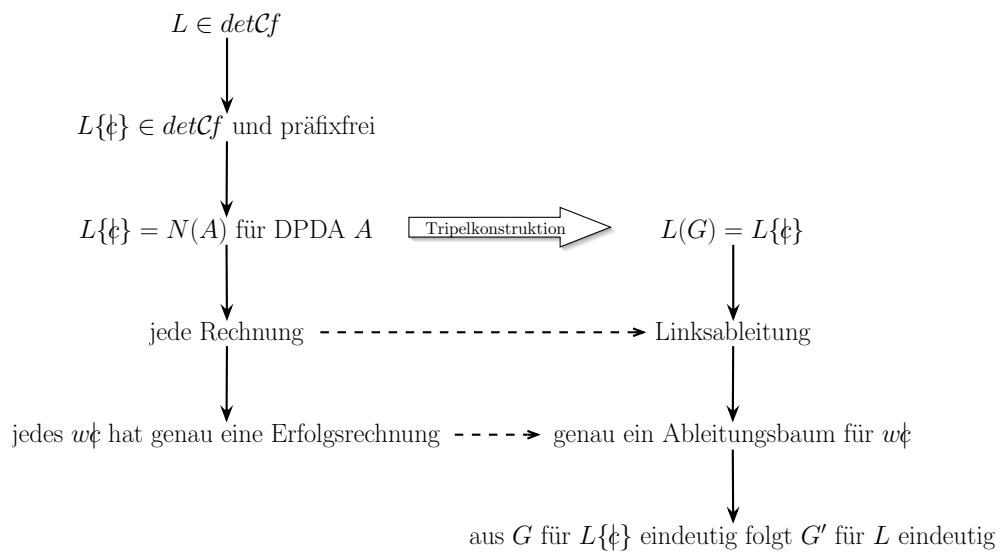


Abbildung 5.5.: Die Beweisidee zu Theorem 5.28

6. Syntaxanalyse kontextfreier Sprachen

In diesem Kapitel soll hauptsächlich die Anwendung von Ergebnissen der theoretischen Informatik auf die grundsätzlichen Methoden bei der Übersetzung von deklarativen (imperativen) Programmiersprachen untersucht werden. Aber auch auf die wesentlichsten Aufgaben und einige Techniken des Übersetzerbaus wird eingegangen. Vollständigkeit im Sinne einer Vorlesung zu Compilerbau ist hier nicht beabsichtigt und auch nicht möglich, dazu sind weiterführende Veranstaltungen nötig.

Die Aufgabe eines **Übersetzers** ist es, eine Programmiersprache bedeutungsäquivalent in eine andere Sprache zu transformieren. Meist sind die (Quell-)Programme in einer höheren Programmiersprache geschrieben und sollen in eine maschinenorientierte (Ziel-)Sprache übersetzt werden, damit der in der Quellsprache formulierte Algorithmus auf einem Rechner ausgeführt werden kann. Solche Übersetzer werden in der Regel als **Compiler** bezeichnet. Da ein Computer in der Regel nur Maschinencode ausführen kann, ist eine zweite Übersetzung aus der maschinenorientierten Zwischensprache nötig, die aber oft einfacher zu bewerkstelligen ist. Zu den einzelnen Techniken der Codegenerierung werden wir in diesem Skript nichts ausführen, die Methoden der Syntaxanalyse werden den größten Raum einnehmen.

Ein Compiler übersetzt das Quellprogramm (*source code*) in ein ausführbares Zielprogramm (*target code*) vollständig, bevor dieses ausgeführt wird. Ein Vorteil dieses Verfahrens ist es, dass der übersetzte Zielcode wiederverwendet werden kann.

Bei einigen Programmiersprachen ist es möglich oder sogar vorteilhaft, nur jeweils den gerade auszuführenden Befehl zu übersetzen und zur Ausführung bringen zu lassen. Solche Programme bezeichnet man als **Interpreter** (*interpreter*). Ein Vorteil ist hier die Möglichkeit zur Programmänderung während der Laufzeit. Ein nicht zu unterschätzender Nachteil ist die wiederholt nötige Übersetzung gleicher Anweisungen, jedesmal wenn diese durchlaufen werden. Auch ist die Wiederverwendung des einmal ausgeführten Programms nur als Quellcode möglich.

6.1. Aufbau eines Compilers

Compiler arbeiten meist nach dem in den folgenden Abschnitten skizzierten Schema und sollen bei ihrer Übersetzung grundsätzlich die dort skizzierten Aufgaben erfüllen, die jeweils an einem kleinen Beispiel illustriert werden.

6.1.1. Lexikalische Analyse

Das Quellprogramm wird analysiert. Es werden Symbole eingelesen, überflüssige Leerzeichen unterdrückt, die Korrektheit sprachinterner Schlüsselworte geprüft und diese durch interne Symbole (*Token*)

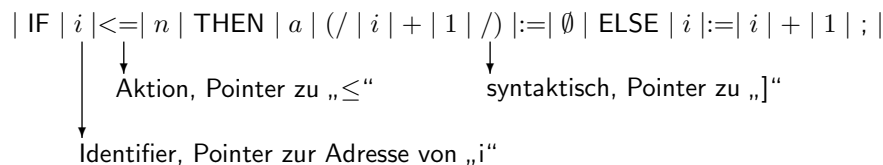
6. Syntaxanalyse kontextfreier Sprachen

ersetzt. Dabei werden Pointer zu weiteren Informationen wie *Wert*, *Tokentyp* und *ASCII-Wert* angelegt und es werden kontextabhängige (z.B. *nicht deklarierte Variablen*) und syntaktische Fehler (*falsche Schreibweise*) gesucht.

6.1 Beispiel

Betrachten wir das Programmfragment `IF i <= n THEN a(/i + 1/) := ∅ ELSE i := i + 1 ;`

Die Wörter zwischen den senkrechten Strichen werden als *Token* interpretiert:



Als Basis der lexikalischen Analyse wird meist ein deterministischer endlicher Automat verwendet, bei dem jedes akzeptierte Wort einem Token entspricht. Die Menge der Token kann sehr groß sein, ist aber eine reguläre Menge und kann somit auch durch einen rationalen Ausdruck beschrieben werden.

In dieser Phase des Übersetzens wird bereits eine Syntaxtabelle angelegt, welche die spätere semantische Analyse vereinfacht. Der Programmteil für die lexikalische Analyse wird als *Scanner* bezeichnet.

6.1.2. Syntaktische Analyse

Bei kontextfreier Syntaxdefinition einen Ableitungsbaum zu der von der lexikalischen Analyse erstellten Token-Kette erzeugen. Zu der Folge $|i| := |i| + |1|$ könnte z.B. der Ableitungsbaum aus Abbildung 6.1 gehören.

Es wird überprüft, ob der Programmtext gemäß der Programmiersprachengrammatik wohlgeformt ist. Dieser Programmteil des Compilers wird als *Parser* bezeichnet. Der hier erstellte Ableitungsbaum stellt die Grundlage für die folgende Phase der semantischen Analyse dar.

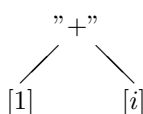
6.1.3. Semantische Analyse und Codegenerierung

In einer Synthese-Phase wird nun der sogenannte **abstrakte Syntaxbaum** aus dem Ableitungsbaum konstruiert, der ähnlich zu bilden ist, wie die Baumdarstellung eines arithmetischen Ausdrucks.

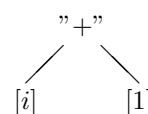
Der Ableitungsbaum wird nach Fehlern durchsucht und es werden in Vorbereitung auf die Phase der Codegenerierung Adressen im Ziel-Code festgelegt. Dies geschieht u.a. für Konstanten, Variablen und Zwischenergebnisse.

Anschließend gehört oftmals noch eine Phase der „Normalisierung“ dazu, bei der semantisch gleiche Sprachkonstrukte in eine einheitliche Form gebracht werden.

In dem in Abbildung 6.2 dargestellten abstrakten Syntaxbaum wird z.B. der linke Teilbaum:



durch den rechten Teilbaum ersetzt:



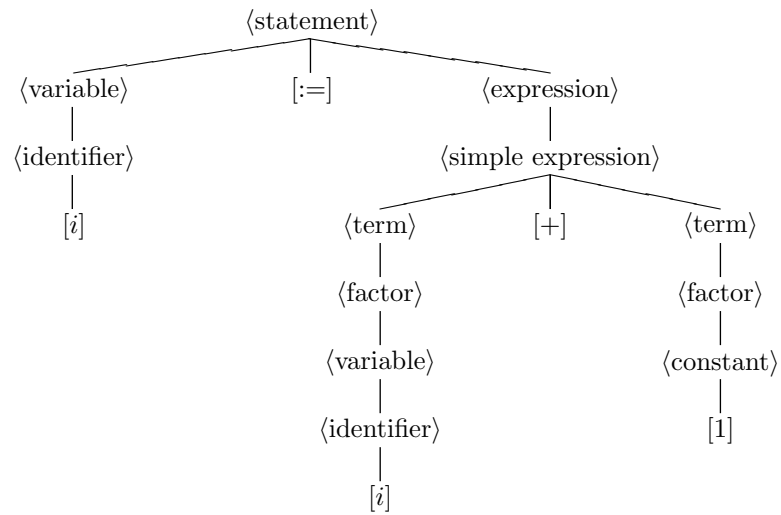
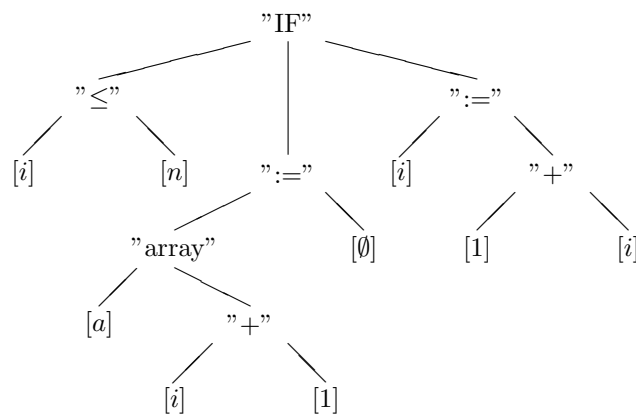
Abbildung 6.1.: Ableitungsbaum zu $|i| := |i| + |1|$ 

Abbildung 6.2.: Ein noch zu normalisierender Ableitungsbaum

6. Syntaxanalyse kontextfreier Sprachen

Die Trennung zwischen der syntaktischen und der semantischen Analyse ist hier genauso unscharf, wie die zwischen semantischer Analyse und Codegenerierung.

Die Erkennung gemeinsamer Teilausdrücke, wie im Beispiel des Ausdrucks „ $i + 1$ “ an zwei verschiedenen Stellen des abstrakten Syntaxbaums gehört dazu, sowie deren teilweise Ersetzung durch entsprechende Pointer zu einem Auftreten.

Einfachste Konstantenrechnungen könnten auch hier schon durchgeführt werden.

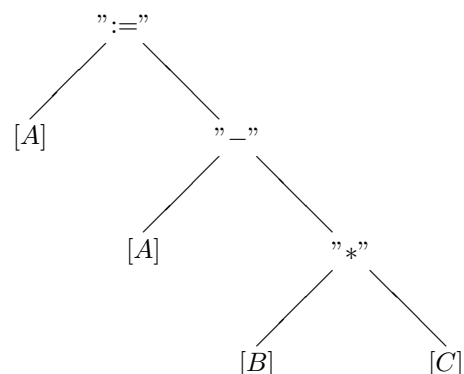
6.1.4. Codeoptimierung

Vielleicht nicht in allen Compilern verwendet, werden hier einige Maßnahmen zur Generierung von teiloptimiertem Code durchgeführt. Wichtig ist es, schon jetzt einzelne Operationen durch besonders gut implementierte, d.h. schnell ausführbare Operationen, zu ersetzen. Bei der Summe „+“ mit zwei Operanden könnte statt „+1“ zum Beispiel ein Befehl „increment“ benutzt werden.

Mehrdimensionale Felder können zum schnelleren Zugriff linearisiert werden. Attribute wie Feld-Länge, Deklarationstiefe, etc. können am abstrakten Syntaxbaum eingetragen werden.

Vereinfachungen von einfachen Laufschleifen durch Splitten, Zusammenfassen, Abwickeln oder Herausziehen von Anweisungen können vorgenommen werden.

Die eigentliche Codegenerierung hängt stark von der Maschine und der maschinennahen Assemblersprache ab. Aus dem optimierten abstrakten Syntaxbaum des Beispiels könnte ein einfacher Nulladresscode entstehen.



Dieser ist einer klammerfreien (sog. polnischen) Notation des arithmetischen Ausdrucks sehr ähnlich:

ID A, LOAD, ID A, LOAD, ID B, LOAD, ID C, LOAD, MUL, SUB, STORE

Wird ein 1-Adress- oder ein 1,5-Adress-Code verwendet, so werden noch Hilfsvariablen und weitere Optimierungen eingesetzt. Die Anweisung $A := A - B * C$ führt zu folgenden Assemblercodes:

1-Adress-Code	1,5-Adress-Code
LOAD B	LOAD R ₁ : B
MUL C	MUL R ₁ : C
STORE H ₁	LOAD R ₂ : A
LOAD A	SUB R ₂ : R ₁
SUB H ₁	STORE A : R ₂
STORE	

6.2. Deterministische Syntaxanalyse

Auch wenn in der heutigen Zeit vielleicht nicht jede(r) Informatiker[in] einen Compiler schreiben muss, werden doch oft genug Anpassungen schon existierender Software an vorhandene Compiler, oder umgekehrt Änderungen an existierenden Compilern nötig, um neue Sprachelemente verwenden zu können. Auch aus diesen Gründen ist die Kenntnis der prinzipiellen Möglichkeiten zur Entwicklung von Compilern angezeigt.

Für die Entwicklung von Compilern gibt es unterschiedliche Methoden, von denen einige inzwischen schon halbautomatisch mit Hilfe von Compiler-Generatoren vorgenommen werden.

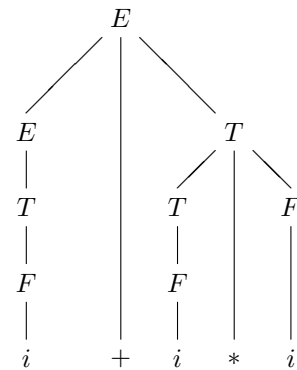
Wir wenden uns nun den wichtigsten deterministischen Syntaxanalyseverfahren zu, die bei zugrundeliegenden kontextfreien Grammatiken angewendet werden. Bevor die einzelnen Varianten der kontextfreien Grammatiken genau definiert werden, sollen an einzelnen Beispielen die grundsätzlichen Verfahrensweisen bei der Syntaxanalyse beleuchtet werden. Dadurch sollen die nachfolgenden formalen Definitionen leichter verständlich werden.

Insbesondere werden die Methoden des *bottom-up parsing* und des *top-down parsing* unterschieden. Wir behandeln sie in den Abschnitten 6.2.1 und 6.2.2

6.2.1. Bottom-up Parsing mit Keller

Sei G die bekannte CFG für einfache arithmetische Ausdrücke mit den Regeln:

$$\begin{aligned} E &\longrightarrow E + T \mid T, & \text{und dem Ableitungsbaum für} \\ T &\longrightarrow T * F \mid F, & \text{das aus dem Startsymbol } E \\ F &\longrightarrow (E) \mid i & \text{ableitbare Wort } i + i * i : \end{aligned}$$



Die Eingabe für den Algorithmus ist nun die Zeichenkette $i + i * i$, die unter Benutzung eines Kellers mit einem Symbol Vorausschau analysiert werden soll. Die Eingabe wird dabei von links nach rechts gelesen, und das Zeichen der Vorausschau wird in einem Puffer (endliche Kontrolle) gespeichert. Dazu wird versucht, die in einer Rechtsableitung zuletzt angewendete Produktion rückgängig zu machen, also die vorhergehende Satzform zu erzeugen. Der jeweilige Kellerinhalt, zusammen mit der noch zu lesenden Kette von Eingabesymbolen, bildet dabei immer die Satzform der Rechtsableitung. In Tabelle 6.1 sind die einzelnen Reduktionsschritte, ähnlich den Konfigurationen des PDA, untereinander aufgeführt. In der Situation *) darf der Kellerinhalt $E + T$ nicht reduziert werden, da das Symbol $*$ in der Vorausschau Priorität hat! Mit *shift* und *reduce* Schritten wird auf dem Keller gearbeitet, weswegen diese Methode auch den Namen Shift-Reduce-Verfahren bekommen hat. Das oben am Beispiel dargestellte Verfahren wird meistens jedoch als LR(1) bezeichnet: Das „L“ steht für *Eingabe von links nach rechts* und das

Tabelle 6.1.: Beispiel für LR(1)-Parsing

Kellerboden ↓	↓ Vorausschau	zu lesende Eingabe	
Kellerinhalt			
		$i + i * i$	<i>Start</i>
	i	$+ i * i$	<i>shift</i>
i	$+$	$i * i$	<i>shift</i>
F	$+$	$i * i$	<i>reduce</i>
T	$+$	$i * i$	<i>reduce</i>
E	$+$	$i * i$	<i>reduce</i>
$+ E$	i	$* i$	<i>shift</i>
$+ E i$	$*$	i	<i>shift</i>
$F + E$	$*$	i	<i>reduce</i>
$T + E$	$*$	i	<i>reduce *)</i>
$* T + E$	i		<i>shift</i>
$i * T + E$			<i>shift</i>
$F * T + E$			<i>reduce</i>
$T + E$			<i>reduce</i>
E			<i>reduce</i>
			<i>accept</i>

„R“ für *eine Rechtsableitung suchen*. Die endliche Kontrolle wird als Steuerung über eine Tafel verstanden, weshalb dies ein tafelgesteuertes Verfahren ist. Für einen Compiler fehlen noch die semantischen Aktionen und die für eine Übersetzung nötigen Ausgaben. In der theoretischen Informatik betrachtet man daher auch Kellerautomaten mit Ausgabe, die, ähnlich wie die endlichen Automaten mit Ausgabe, als akzeptierende Keller-Transduktoren verwendet werden.

6.2.2. Top-down Parsing mit Keller

Da die zuvor verwendete Grammatik Linksrekursionen¹ enthält, muss diese noch nach dem üblichen Verfahren² zum Entfernen von Linksrekursionen verändert werden. Wir verwenden demnach die äquivalente CFG mit folgenden Regeln:

$$\begin{aligned}
 E &\longrightarrow TE' \\
 E' &\longrightarrow +TE' \mid \lambda \\
 T &\longrightarrow FT' \\
 T' &\longrightarrow *FT' \mid \lambda \\
 F &\longrightarrow (E) \mid i
 \end{aligned}$$

Das top-down Verfahren versucht zu der vorgelegten Zeichenkette eine Linksableitung zu finden, wobei der gespiegelte Kellerinhalt zusammen mit der noch zu lesenden Eingabe die jeweils abgeleitete Satzform

¹Eine Linksrekursion liegt in einer Grammatik vor, wenn es in ihr Regeln der Form $A \longrightarrow Aw$ gibt.

²Das Verfahren wird bei der Transformation von beliebigen kontextfreien Grammatiken in Greibach-Normalform angewendet und ist im Beweis zu Theorem 4.25 ausgeführt.

ergibt.

Der Ableitungsbaum für $i + i * i$ ist in Abbildung 6.3 dargestellt.

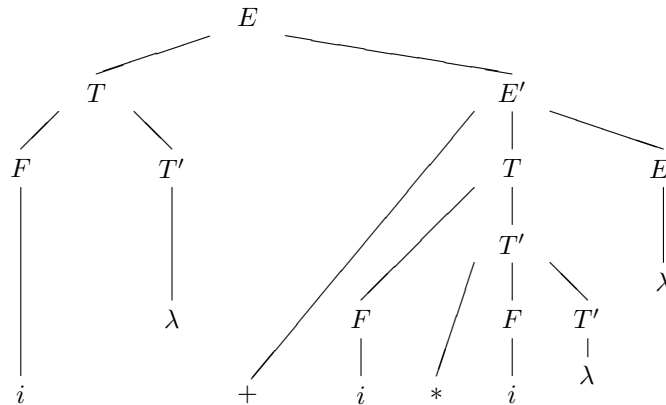


Abbildung 6.3.: Ableitungsbaum für $i + i * i$

Die Analyseschritte für dieses sogenannte LL(1)-Parsing sind wie beim LR(1)-Verfahren in Tabelle 6.2 knapp dargestellt:

Immer wenn ein Terminalsymbol auf dem *top* des Kellers erscheint, also ganz links von der Grammatik generiert wird, wird dieses mit dem Symbol der Eingabe, welches sich im Vorausschau-Puffer befindet, verglichen und beide gelöscht, falls sie identisch sind. Auf diese Weise werden auf dem Keller entweder *expand*- oder *reduce*-Schritte durchgeführt.

Die in diesem Verfahren benutzte CFG ist eine sogenannte LL(1)-Grammatik, die eine deterministische Syntaxanalyse mit einem Kellerautomaten bei einem Zeichen Vorausschau gestattet. Das Symbol in der Vorausschau bestimmt eindeutig, welche Produktion der Grammatik jeweils bei der Expansion zu verwenden ist.

Andererseits lässt eine LL(1) Grammatik auch sehr einfach das Verfahren des rekursiven Abstiegs zu. Letzteres werden wir hier am gleichen Beispiel durch Angabe der Analyseprozeduren vorstellen. Die Prozeduren werden direkt aus der Umsetzung der Produktionen gewonnen und sind selbsterklärend. Sie sind in Algorithmus 6.2 dargestellt. Die Eingabe für diese Prozeduren bestehen aus den Symbolen des zu analysierenden Textes, der hier z.B. in Form einer Liste mit den üblichen Operationen zur Verfügung gestellt wird.

6.3. LR(k)-Grammatiken

Wir behandeln nun etwas formaler die LR-Parsingmethode. LR-Parsing ist aus folgenden Gründen attraktiv:

- Jede deterministische kontextfreie Sprache kann mit diesem Bottom-up Verfahren analysiert und übersetzt werden

6. Syntaxanalyse kontextfreier Sprachen

Tabelle 6.2.: Beispiel für LL(1)-Parsing

Präfix der Satzform im Keller Kellerboden ↓ ↓ Vorausschau		zu lesende Eingabe	
Kellerinhalt			
E		$i + i * i$	<i>Start</i>
E	i	$+ i * i$	<i>positioning</i>
TE'	i	$+ i * i$	<i>expand</i>
$FT'E'$	i	$+ i * i$	<i>expand</i>
$iT'E'$	i	$+ i * i$	<i>expand</i>
$T'E'$		$+ i * i$	<i>reduce</i>
$T'E'$	$+$	$i * i$	<i>positioning</i>
E'	$+$	$i * i$	<i>expand</i> ($T' \rightarrow \lambda$)
$+TE'$	$+$	$i * i$	<i>expand</i>
TE'		$i * i$	<i>reduce</i>
TE'	i	$* i$	<i>positioning</i>
$FT'E'$	i	$* i$	<i>expand</i>
$iT'E'$	i	$* i$	<i>expand</i>
$T'E'$		$* i$	<i>reduce</i>
$T'E'$	$*$	i	<i>positioning</i>
$*FT'E'$	$*$	i	<i>expand</i>
$FT'E'$		i	<i>reduce</i>
$FT'E'$	i		<i>positioning</i>
$iT'E'$	i		<i>expand</i>
$T'E'$			<i>reduce</i>
E'			<i>reduce</i>
			<i>accept</i>

- Dieses Verfahren kann effizient implementiert werden.
- Syntaktische Fehler können so früh wie möglich erkannt werden.
- Viele existierende Compiler-Generatoren bieten sogar Hilfen an, wenn eine Grammatik keine LR-Grammatik ist, Mehrdeutigkeiten aufweist und daher geändert werden muss.
- Die LR-Parsingmethode ist das am besten ausgebaute Verfahren, und beinahe jede Programmiersprachensyntax besitzt eine LR(k)-Grammatik.

Es muss dazu betont werden, dass die im Compiler verwendete Syntaxbeschreibung einer Programmiersprache nicht die gleiche zu sein braucht, die die Benutzer zur Spezifikation der Sprache erhalten. Die im Compiler verwendete Syntaxbeschreibung wäre für die Benutzer auch mit zu vielen Details befrachtet.

Wir definieren zunächst formal den Begriff der LR(k)-Grammatik nach [Hopcroft&Ullman].

6.3 Definition (LR(k)-Grammatik)

Sei $G := (V_N, V_T, P, S)$ eine reduzierte kontextfreie Grammatik, bei der das Startsymbol S in keiner

6.2 Algorithmus

```

proc  expression is:
    begin term; restexpression end

```

```

proc  term is:
    begin factor; restterm end

```

```

proc  factor is:
    begin
        if head(input) = "i" then goto OK
        if head(input) = "(" then
            begin input := tail(input); expression;
                if head(input) = ")" then goto OK else FEHLER
            end
        else FEHLER
    OK: input := tail(input)
    end

```

```

proc  restexpression is:
    begin
        if head(input) = "+" then
            begin
                input := tail(input);
                term; restexpression;
            end
    end

```

```

proc  restterm is:
    begin
        if head(input) = "*" then
            begin
                input := tail(input);
                factor; restterm;
            end
    end

```

6. Syntaxanalyse kontextfreier Sprachen

rechten Seite einer Produktion vorkommt.³ Für alle $k \in \mathbb{N}$ und jedes $w \in V^*$ bezeichne $\text{first}_k(w)$ den Präfix von w der Länge k . D.h.:

$$\text{first}_k(w) := \begin{cases} w, & \text{falls } |w| \leq k \\ u, & \text{falls } |u| = k \text{ und } w = u \cdot v \end{cases}$$

Die CFG G heißt $\text{LR}(k)$, falls die Bedingungen 1., 2. und 3. die Aussage 4. implizieren:

1. $\exists \alpha, \beta, \gamma \in V^* \exists A \in V_N : S \xrightarrow[\text{re}]{*} \alpha A \gamma \xRightarrow{\text{re}} \alpha \beta \gamma$
2. $\exists \alpha', \gamma', \gamma'' \in V^* \exists B \in V_N : S \xrightarrow[\text{re}]{*} \alpha' B \gamma' \xRightarrow{\text{re}} \alpha \beta \gamma''$
3. $\text{first}_k(\gamma) = \text{first}_k(\gamma'')$
4. $A = B, \alpha = \alpha' \text{ und } \gamma' = \gamma''$

Diese Definition spiegelt genau das wieder, was im Beispiel bei dem bottom-up Verfahren zu erkennen war: Die Symbole in der Vorausschau von k Zeichen (im Beispiel war $k = 1$) bestimmen eindeutig, welche Regel für die Reduktion (den *reduce*-Schritt) zu wählen war. Um diese Situation einfach beschreiben zu können benutzen wir die Definition des Schlüssels einer Rechtsableitung aus dem wir die Produktion für die Reduktion erkennen können.

6.4 Definition (Schlüssel einer Ableitung)

Zu einer Rechtsableitung $S \xrightarrow[\text{re}]{*} \alpha A w \xRightarrow{\text{re}} \alpha \beta w$ heißt die Zeichenkette β mit der Angabe ihrer Position der **Schlüssel** (Griff, handle) dieser Ableitung. Wir schreiben diesen als $(A \rightarrow \beta, |\alpha\beta|)$.

Wie man der Definition leicht entnimmt, ist der Schlüssel bei einer $\text{LR}(k)$ -Grammatik immer eindeutig. Daher ist es nicht verwunderlich, dass jede $\text{LR}(k)$ -Grammatik ausschließlich eindeutige Ableitungen besitzt.

6.5 Beispiel

Betrachte die CFG mit den Produktionen $S \rightarrow aAc, A \rightarrow Abb \mid b$.

Für jedes $i \geq 1$ hat das Wort $aAb^{2i}c$ den Schlüssel $(A \rightarrow Abb, 4)$ und $ab^{2i+1}c$ hat den Griff $(A \rightarrow b, 2)$.

6.6 Theorem

Jede $\text{LR}(k)$ -Grammatik ist eindeutig.

Beweis: Betrachte zwei beliebige Rechtsableitungen $S \xrightarrow[\text{re}]{m} w$ und $S \xrightarrow[\text{re}]{n} w$, mit denen das Wort w mit m bzw. n Schritten generiert wird. Für den Beweis, dass beide Ableitungen die selben Regeln an den gleichen Stellen und in der selben Reihenfolge benutzen, verwenden wir Induktion über die Zahl $m \in \mathbb{N}$.

³In [Harrison] ist anstelle der strengen Bedingung „keine rechte Seite einer Produktion enthält das Startsymbol S “ lediglich gefordert, dass folgende Ableitung nicht möglich ist: „ $S \xrightarrow[\text{re}]{+} S$ “. Dadurch ergibt sich bei den $\text{LR}(0)$ -Grammatiken eine andere Sprachklasse. Wir weisen darauf mit einigen Beispielen hin und bezeichnen die $\text{LR}(k)$ -Grammatiken nach der Definition von Harrison hier als $\text{LR}'(k)$ -Grammatiken, werden aber kaum Beweise für die zitierten Resultate angeben.

Induktionsbasis: $m = 0$ bedeutet $w = S$ und, da G reduziert ist und niemals $S \xRightarrow{\pm} S$ vorkommen kann, muss folglich auch $n = 0$ gelten.

Induktionsschritt: Seien o.B.d.A. $1 \leq m \leq n$, $S \xRightarrow[m]{\text{re}} \alpha A \gamma \xRightarrow{\text{re}} \alpha \beta \gamma = w$ und $S \xRightarrow[n-1]{\text{re}} \alpha' B \gamma' \xRightarrow{\text{re}} \alpha \beta \gamma = w$. Aus der LR(k) Eigenschaft folgt nun $A = B$ und $\alpha = \alpha'$ und $\gamma = \gamma'$. Nach der Induktionsannahme sind schon die Ableitungen $S \xRightarrow[n-1]{\text{re}} \alpha A \gamma$ und $S \xRightarrow[n-1]{\text{re}} \alpha' B \gamma' = \alpha A \gamma$ gleich. Also müssen auch die ursprünglichen Ableitungen $S \xRightarrow[m]{\text{re}} w$ und $S \xRightarrow[n]{\text{re}} w$ identisch sein.

□

Nach der LR'(k) Definition von Harrison ist die Grammatik mit den Regeln $S \rightarrow Sa \mid a$ eine LR'(0)-Grammatik nicht jedoch nach der Definition in [Hopcroft&Ullman]. Für die hier gemäß [Hopcroft&Ullman] definierten LR(k)-Grammatiken gilt folgender Satz, den wir ohne Beweis zitieren:

6.7 Theorem

Folgende drei Aussagen sind äquivalent:

1. L ist präfixfreie, deterministische kontextfreie Sprache
2. $L = L(G)$ für eine LR(0)-Grammatik
3. $L = N(A)$ für einen DPDA

Für die LR'(0)-Grammatiken nach der Definition von Harrison erhalten wir eine echte Obermenge der präfixfreien, deterministischen kontextfreien Sprachen.

Für die LR'(0)-Sprachen gilt das Charakterisierungsergebnis:

6.8 Theorem

L ist LR'(0)-Sprache genau dann, wenn gilt:

$$\forall x, y \in V_T^* \forall z \in V_T^* : (x \in L) \wedge (xz \in L) \wedge (y \in L) \rightarrow (yz \in L)$$

Nach Harrison ist also die Dycksprache D_1 aller wohlgeformten Klammerausdrücke eine LR(0')-Sprache, während sie dies nach unserer Definition nicht ist, denn sie ist nicht präfixfrei. Gilt $k \neq 0$, so unterscheiden sich die Sprachen der LR(k)- und der LR'(k)-Grammatiken nicht mehr. Hier gilt dann:

6.9 Theorem

Folgende drei Aussagen sind äquivalent:

1. L ist deterministisch kontextfrei, d.h. $L = L(A)$ für einen DPDA A .
2. $L = L(G)$ für eine LR(1)-Grammatik G .
3. $L = L(G)$ für eine LR(k)-Grammatik G für ein $k \geq 1$.

6. Syntaxanalyse kontextfreier Sprachen

Trotz der Gleichheit der durch $LR(k)$ - und $LR(k+1)$ -Grammatiken beschriebenen Sprachen, bilden die Grammatiken selbst in folgender Weise eine Hierarchie:

6.10 Theorem

Zu jedem $k \geq 1$ gibt es eine kontextfreie Grammatik die $LR(k)$ aber nicht $LR(k-1)$ ist.

Es ist sogar so, dass im allgemeinen nicht einmal entschieden werden kann, ob eine vorgegebene Grammatik eine deterministische Sprache generiert. Es ist aber zu fester Zahl $k \in \mathbb{N}$ entscheidbar, ob diese Grammatik $LR(k)$ ist.

$LR(k)$ -Grammatiken wurden 1965 von D. Knuth definiert. Meistens werden jetzt gewisse Abwandlungen verwendet, deren Definitionen auf de Remer zurückgehen. Dies sind die sogenannten *simple* $LR(k)$ -Grammatiken, kurz $SLR(k)$ -Grammatiken und die $LALR(k)$ -Grammatiken mit einem zusätzlichen *lookahead*. Für $k = 1$ sind dies die heute am häufigsten verwendeten Grammatiken für Programmiersprachensyntax. Ein $LR(1)$ -Parser für PASCAL, z.B., hat mehrere tausend Zustände, während ein $LALR(1)$ -Parser für die gleiche Sprache mit wenigen hundert Zuständen auskommt.

Für die Entwicklung von Compilern werden heute meist folgende Verfahren benutzt:

parser-generator: Aus der kontextfreien Grammatik wird ein Analysator für die Sprache erzeugt, der im wesentlichen einem Kellerautomaten mit Ausgabe gleicht. Dies wird in der Regel interaktiv durchgeführt.

scanner-generator: Zu der Grammatik wird ein Analysator gewonnen, der die lexikalische Analyse vornimmt. Dieser gleicht im wesentlichen einem endlichen Automaten.

syntax-directed translation engine: Jeder kontextfreien Produktion wird eine semantische Aktion zugeordnet, die dann bei der Analyse mit einem Kellerverfahren ausgegeben wird. Als Beispiel seien hier nur zwei Produktionen angegeben:

$$\begin{aligned} E &\longrightarrow E + T & \text{zugeordnet wird: } E_{\text{val}} &:= E_{\text{val}} + T_{\text{val}} \\ F &\longrightarrow i & \text{zugeordnet wird: } F_{\text{val}} &:= i_{\text{lexval}} \end{aligned}$$

Nach dieser Idee werden die sogenannten syntaxgesteuerten Übersetzungen gestaltet, wo schon bei der Grammatikdefinition die semantische Auswertung parallel zu jeder Regel mit entsprechenden kontextfreien Produktionen die Auswertungsregeln formuliert werden. Auch diese sogenannten attributierten Grammatiken können hier nicht im Detail vorgestellt werden.

Für die deterministische top-down Analyse werden $LL(k)$ -Grammatiken verwendet. Am Beispiel der einfachen arithmetischen Ausdrücke wurde das angestrebte Vorgehen bei der syntaktischen Analyse schon erläutert. Im Prinzip geht es immer darum für ein zu analysierendes Wort $w := a_1 a_2 \dots a_i a_{i+1} a_n$ für das schon die Ableitung $S \implies a_1 a_2 \dots a_i A v$ gefunden wurde, aus $\text{first}_k(v)$, dem Präfix von v der Länge k eindeutig festlegen zu können, welche Produktion für die Ersetzung von A gewählt werden muss.

6.11 Beispiel

Betrachten wir die $LR(1)$ -Grammatik mit dem Startsymbol E und zwei sehr lange Wörter $w_1 := (i + i + \dots + i) + i$ und $w_2 := (i + i + \dots + i)$.

$$E \longrightarrow E + T \mid T,$$

$$\begin{aligned} T &\longrightarrow T * F \mid F, \\ F &\longrightarrow (E) \mid i \end{aligned}$$

Offensichtlich ist die erste anzuwendende Regel für die Ableitung von w_1 die Produktion $E \longrightarrow E + T$, aber zur Ableitung des Wortes w_2 wird die Produktion $E \longrightarrow T$ benötigt. Zu dieser Entscheidung ist aber die Kenntnis des rechten Endes des jeweiligen Wortes nötig, und das geht niemals mit einer Vorausschau beschränkter Länge! Auch das Fehlen von direkter Linksrekursionen ist alleine nicht ausreichend, denn diese kann ja mittelbar entstehen: $A \Longrightarrow Bw_1 \Longrightarrow Cw_2w_1 \Longrightarrow Aw_3w_2w_1$.

6.4. LL(k)-Grammatiken

6.12 Definition (FIRST $_k(w)$)

Sei $G := G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik, $k \in \mathbb{N}$ und $w \in (V_N \cup V_T)^*$. Dann sei $FIRST_k(w)$ definiert durch:

$$\begin{aligned} FIRST_k(w) &:= \{ u \in V_T^* \mid w \xRightarrow{*}_{\text{li}} u \text{ und } |u| < k \text{ oder } \exists v \in (V_N \cup V_T)^* : \\ &\quad w \xRightarrow{*}_{\text{li}} uv \text{ mit } |u| = k \} \end{aligned}$$

Man beachte, dass $FIRST_k(w)$ und $\text{first}_k(w)$ aus Definition 6.3 nicht identisch sind. Man betrachte dazu die Grammatik mit den Produktionen $S \longrightarrow aAS \mid b$ und $A \longrightarrow bSA \mid a$. Es gilt $\text{first}_2(aAS) = aA$ aber $FIRST_2(aAS) = \{aa, ab\}$ und $FIRST_2(S) = \{b, aa, ab\}$ aber $\text{first}_2(S) = S$.

6.13 Definition (LL(k)-Grammatik)

Eine reduzierte, kontextfreie Grammatik $G := G = (V_N, V_T, P, S)$ heißt LL(k) für ein $k \in \mathbb{N}$, falls für je zwei Linksableitungen

$$S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{*}_{\text{li}} uv\alpha \xRightarrow{*}_{\text{li}} uw \text{ und } S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{*}_{\text{li}} uv'\alpha \xRightarrow{*}_{\text{li}} uw'$$

mit $u, w, w' \in V_T^*$ und $\alpha, v, v' \in (V_N \cup V_T)^*$, aus $FIRST_k(w) = FIRST_k(w')$ stets $v = v'$ folgt.

Da die Definition der LL(k)-Grammatik Eigenschaften über ableitbare Wörter voraussetzt, ist ein lokaler Test nicht ganz einfach. Der folgende Satz erlaubt einen etwas einfacheren Test:

6.14 Theorem

Eine reduzierte CFG $G := (V_N, V_T, P, S)$ ist genau LL(k), wenn für je zwei verschiedene Produktionen $A \longrightarrow w$ und $A \longrightarrow w'$ die folgende Bedingung gilt:

$$(*) \quad \forall u \in V_T^* \forall \alpha \in (V_N \cup V_T)^* : (S \xRightarrow{*}_{\text{li}} uA\alpha) \Rightarrow (FIRST_k(w\alpha) \cap FIRST_k(w'\alpha) = \emptyset)$$

Beweis: Sei $L := L(G)$ für eine LL(k)-Grammatik G , die die geforderten Voraussetzungen erfüllt, aber die Bedingung $(*)$ verletzt. Falls $\gamma \in FIRST_k(w\alpha) \cap FIRST_k(w'\alpha)$ ist, dann gilt nach Definition

6. Syntaxanalyse kontextfreier Sprachen

6.6.12 der Menge FIRST_k , dass folgende Ableitungen existieren: $w\alpha \xRightarrow{*}_{\text{li}} \gamma\beta$ und $w'\alpha \xRightarrow{*}_{\text{li}} \gamma\beta'$. Da G reduziert ist, existieren aber terminale Wörter $\delta, \delta' \in V_T^*$ und die beiden Ableitungen

$$S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{\text{li}} uw\alpha \xRightarrow{*}_{\text{li}} u\gamma\beta \xRightarrow{*}_{\text{li}} u\gamma\delta \text{ und } S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{\text{li}} uw'\alpha \xRightarrow{*}_{\text{li}} u\gamma\beta \xRightarrow{*}_{\text{li}} u\gamma\delta'$$

Für $|\gamma| \leq k$ folgt $\delta = \delta'$, mithin $\{\gamma\} = \text{FIRST}_k(\gamma\delta) \cap \text{FIRST}_k(\gamma\delta')$. Da G eine $\text{LL}(k)$ -Grammatik ist, folgt $A \rightarrow w = A \rightarrow w'$ im Widerspruch zu der Voraussetzung über G .

Umgekehrt sei G keine $\text{LL}(k)$ -Grammatik, dann existieren die beiden Linksableitungen

$$S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{\text{li}} uw\alpha \xRightarrow{*}_{\text{li}} u\beta \text{ und } S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{\text{li}} uw'\alpha \xRightarrow{*}_{\text{li}} u\beta'$$

mit $\text{FIRST}_k(\beta) = \text{FIRST}_k(\beta')$ aber $w \neq w'$. Damit folgt aber

$$\text{FIRST}_k(\beta) = \text{FIRST}_k(\beta') \subseteq \text{FIRST}_k(w\alpha) \cap \text{FIRST}_k(w'\alpha)$$

und dieser Durchschnitt ist nicht leer, die Bedingung (*) also auch nicht erfüllt. Damit sind beide Implikationen bewiesen. \square

Als Folgerung ergibt sich für die $\text{LL}(1)$ -Eigenschaft ein noch einfacherer Überprüfungstest:

6.15 Korollar

Eine reduzierte, kontextfreie Grammatik $G := (VN, VT, P, S)$ ohne λ -Produktionen ist genau dann $\text{LL}(1)$, wenn $\text{FIRST}_1(w) \cap \text{FIRST}_1(w') = \emptyset$ für je zwei verschiedene Produktionen $A \rightarrow w$ und $A \rightarrow w'$ ist.

Beweis: Da G reduziert ist, existieren die beiden Ableitungen

$$S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{\text{li}} uw\alpha \text{ und } S \xRightarrow{*}_{\text{li}} uA\alpha \xRightarrow{\text{li}} uw'\alpha$$

Die Bedingung aus Theorem 6.6.14 lautet nunmehr $\text{FIRST}_1(w\alpha) \cap \text{FIRST}_1(w'\alpha) = \emptyset$, und weil $|w| \neq 0 \neq |w'|$ ist und G keine λ -Produktionen besitzt, ist dies gerade äquivalent zu der Formulierung des Korollars. \square

6.16 Theorem

Jede reguläre Menge besitzt eine $\text{LL}(1)$ -Grammatik die sie erzeugt.

Beweis: Sei $A := (Z, X, K, \{z_0\}, Z_{\text{end}})$ ein vollständiger DFA, dann definieren wir eine CFG mit folgenden Produktionen: $\{[z_i] \rightarrow x[z_j] \mid (z_i, x, z_j) \in K\} \cup \{[z_i] \rightarrow \lambda \mid z_i \in Z_{\text{end}}\}$ und dem Symbol $[z_0]$ als Startsymbol. Da A ein DFA ist, folgt $\text{FIRST}_1(x[z_j]) \cap \text{FIRST}_1(y[z_k]) = \emptyset$ für je zwei verschiedene $[z_i]$ -Produktionen der Art

$$[z_i] \rightarrow x[z_j], [z_i] \rightarrow x[z_k] \text{ oder } [z_i] \rightarrow \lambda$$

\square

6.4.1. Ergebnisse zu $LL(k)$ -Grammatiken:

Die Liste der, im folgenden ohne Beweis zusammengestellten, Resultate zu den $LL(k)$ -Grammatiken finden die Leser(innen) in der Literatur. Wir werden nur eine Liste von Resultaten zur Verfügung stellen, damit die Leserinnen und Leser einen Überblick erhalten. Hier seien aus dem reichhaltigen Angebot nur [Aho&Ullman], [Aho,Sethi&Ullman], [Foster] und [Maurer] erwähnt.

- Keine $LL(k)$ -Grammatik besitzt Linksrekursionen.
- Jede $LL(k)$ -Grammatik ist eindeutig und zugleich eine $LR(k)$ -Grammatik. Folglich ist jede $LL(k)$ -Sprache eine deterministische, kontextfreie Sprache.
- Zu festem $k \in \mathbb{N}$ gibt es einen Algorithmus, der zu jeder beliebig vorgegebenen Grammatik G feststellt, ob dies eine $LL(k)$ -Grammatik ist.
- Es gibt keinen Algorithmus, der von jeder beliebig vorgelegten CFG feststellt, ob diese eine $LL(k)$ -Sprache für irgend ein $k \in \mathbb{N}$ generiert.
- Es gibt kein Verfahren, das für jede beliebige CFG feststellt, ob diese eine $LL(1)$ -Sprache erzeugt.
- Es ist unentscheidbar, d.h. es gibt kein Verfahren, das feststellt, ob eine beliebig gegebene $LR(k)$ -Grammatik zugleich für irgendein $n \in \mathbb{N}$ eine $LL(n)$ -Grammatik ist.
- Die $LL(k)$ -Sprachen bilden eine echte Hierarchie: Zu jedem $k \in \mathbb{N}$ gibt es eine $LL(k)$ -Grammatik G_k , so dass keine $LL(k-1)$ -Grammatik die gleiche Sprache erzeugt, d.h. $L(G_k) \neq L(G_{k-1})$ für jede $LL(k-1)$ -Grammatik G_{k-1} .
- Nicht jede deterministische, kontextfreie Sprache besitzt eine $LL(k)$ -Grammatik: Die Sprache $\{a^n b^m \mid 1 \leq n \leq m\}$ besitzt eine $LR(k)$ -Grammatik, aber für kein $k \in \mathbb{N}$ eine $LL(k)$ -Grammatik.
- Für beliebige $LL(k)$ -Grammatiken G und G' ist es entscheidbar, ob $L(G) = L(G')$ gilt.

Wir sehen also, dass die LL -Sprachen eine echte Teilmenge der LR -Sprachen sind und selbst eine echte, unendliche Hierarchie innerhalb der deterministischen kontextfreien Sprachen bilden.

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

Als Einleitung zu diesem Kapitel werden wir zunächst den Begriff des Algorithmus' etwas präziser fassen, und damit an die Darstellung der Entwicklung der Informatik aus der Einleitung anknüpfen. An diese möchten wir hier, wegen ihrer allgemeinen Wichtigkeit und der Bedeutung der historischen Einordnung, noch einmal ausdrücklich erinnern. Der in den meisten Büchern verwendete Begriff des Algorithmus' ist die folgende:

7.1 Definition

*Ein **Algorithmus** ist eine präzise, das heißt in einer festgelegten Sprache abgefaßte endliche Beschreibung eines allgemeinen Verfahrens unter Verwendung ausführbarer, elementarer Verarbeitungsschritte.*

Ähnliche Definitionen finden sich in den Büchern [BauerGoos] (S. 47-55), [Loeckx] (S. 10 ff.) sowie [Maurer] (S. 14).

Aus dieser Definition folgt, daß es terminierende und nicht-terminierende Algorithmen geben kann. Mit die wichtigste Eigenschaft aber ist, daß die einzelnen anwendbaren Schritte und Aktionen in jeder Situation zweifelsfrei und eindeutig bestimmt sind.

In [Turing] beschrieb Alan M. Turing die nach ihm benannten Maschinen, um einen formalen Zugang zu dem Begriff der „Berechenbarkeit“ zu erhalten.

Da wir Algorithmen nicht gut auf der Grundlage von natürlicher Sprache definieren und trotzdem Korrektheit oder Terminationsfragen formal exakt untersuchen können, benutzen wir dazu die **Turing-Maschine**. Diese Maschinen enthalten alle wesentlichen Bestandteile eines Algorithmus' und werden hier als Präzisierung des Algorithmusbegriffs verwendet werden. Alternative Ansätze und Definitionen findet man bei Gödel (1934), Rosser (1935), Kleene (1936), Church oder Post im selben Jahr und auch bei Markov (1951). Die Turing'sche These besagt, daß jede intuitiv berechenbare Funktion auch von einer Turing-Maschine berechnet werden kann. Die Church'sche These behauptet das gleiche für den λ -Kalkül. Bislang sind alle formalen Definitionen für Berechenbarkeit als gleich mächtig bewiesen worden. Es ist also ausreichend, wenn man die wesentlichen Einsichten mit Hilfe des Modells der Turing-Maschine gewinnt.

Wollen wir zum Beispiel algorithmisch testen, ob ein bestimmtes Wort $w \in \{L, R\}^*$ ein wohlgeformtes Klammergebilde aus der Menge D_1 ist, so brauchen wir dazu ein Verfahren, das uns in jeder Situation *eindeutig* bestimmt, was wir zu tun haben. Das, was wir dann tun sollen, muß natürlich auch *effektiv durchführbar* sein und die gesamte Anleitung dazu soll *in einem endlichen Text niedergelegt* sein. Dies sind die wesentlichen Eigenschaften, die wir von einem Algorithmus forderten.

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

Als Beispiel geben wir zwei Algorithmen an, die testen, ob eine beliebige Zeichenkette $w \in \{L, R\}^*$ ein Dyck-Wort aus D_1 ist.

7.2 Algorithmus

A Schritt 0: Eingabe sei $w \in \{L, R\}^*$.

Schritt 1: Markiere das am weitesten links stehende unmarkierte Symbol L in w .

Schritt 2: Markiere das nächstgelegene unmarkierte Symbol R rechts davon.

Schritt 3: Wiederhole Schritte 1 und 2 in dieser Reihenfolge solange, bis:

- alle Symbole von w markiert sind (Erfolg und $w \in D_1$)
- oder Schritte 1 und 2 waren *nichtbeide* ausführbar (Mißerfolg und $w \notin D_1$).

7.3 Algorithmus

B Schritt 0: Eingabe sei $w \in \{L, R\}^*$ und $k \in \mathbb{N}$.

Schritt 1: Setze $k := 0$ und beginne w von links nach rechts Symbol für Symbol zu lesen. Dabei bilde $k := k + 1$ für jedes auftretende L , und $k := k - 1$ für jedes auftretende R . Wenn k dabei niemals negativ werden mußte und zum Schluß den Wert 0 besitzt, so ist $w \in D_1$.

Um nun die recht allgemein gehaltenen Formulierungen der obigen Verfahren auf einem - wenn auch nur formalen - Modell ausführen zu können, wird die Turing-Maschine in Anlehnung an die Arbeitsweise eines einfachen Rechners ähnlich definiert, wie die bisher benutzten abstrakten Maschinen und Automaten.

7.1. Turing-Maschinen

Eine Turing-Maschine besitzt eine endliche Kontrolle wie ein endlicher Automat, ein in Felder unterteiltes, einseitig unendliches Band mit seinem Ende links und einen Lese-/Schreibkopf (LSK), der in beide Richtungen über das Band von Feld zu Feld geschoben werden kann. Auf jedem Feld des Bandes kann die (hier zunächst deterministische) Turing-Maschine (DTM) mit dem LSK feststellen welches Zeichen auf dem Feld steht. Steht dort kein Symbol, so wird dies mit einem speziellen Zeichen (hier dem „Schweinegatter“ $\#$, engl.: *sharp*) kenntlich gemacht. Dadurch sind alle Felder des Arbeitsbandes rechts von dem letzten von $\#$ verschiedenen Symbol ausschließlich mit dem Symbol $\#$ beschriftet.

7.4 Definition

Eine **deterministische Turing-Maschine** (DTM) wird angegeben durch ein Tupel $A := (Z, X, Y, \delta, q_0, Z_{end})$ wobei:

Z endliche Menge von Zuständen

Y endliches Bandalphabet

X endliches Eingabealphabet, mit $X \subsetneq Y$, und $Y \cap Z = \emptyset$

$\# \in (Y \setminus X)$ ist das spezielle Symbol für das unbeschriftete Feld auf dem Arbeitsband

$q_0 \in Z$ Startzustand

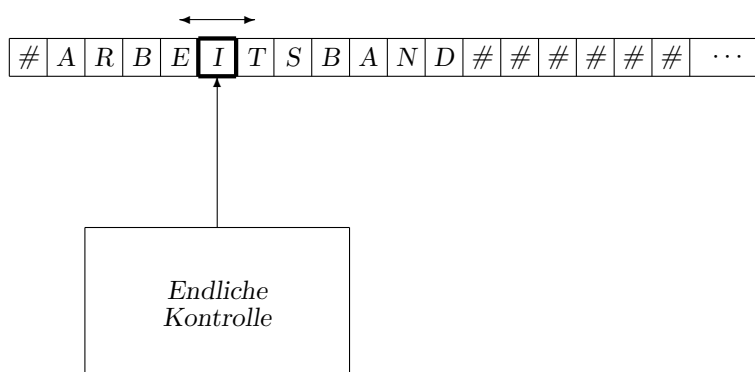
$Z_{end} \subseteq Z$ Menge der Endzustände

$\delta : (Z \times Y) \longrightarrow (Y \times \{L, R\} \times Z)$ ist die partielle Übergangs-Funktion.

Diese gibt durch $\delta(p, y) = (z, b, q)$ an, was geschehen muß, wenn der LSK der DTM A im Zustand $p \in Z$ ein Zeichen $y \in Y$ liest. Mit (z, b, q) wird festgelegt, daß die DTM A das Zeichen z anstelle von y auf das Feld schreiben, dann die Kopfbewegung $b \in L, R$ ausführen und zuletzt in den Zustand q übergehen soll.

Bemerkung:

Es kann gezeigt werden, daß für jede Turing-Maschine M eine äquivalente Turing-Maschine M' mit nur zwei Bandsymbolen existiert. Die Eingabe- und Bandsymbole von M werden in M' binär codiert. Wenn in diesem Fall das Blank-Symbol $\#$ nicht in dem Eingabealphabet X , so müßten alle Eingaben unär codiert werden. Das wiederum würde Probleme bei den Komplexitätsbetrachtungen ergeben. Bei größeren Alphabeten X jedoch empfiehlt sich die Einschränkung $\# \in (Y \setminus X)$, weil dann das Ende der Eingabe immer (leicht) erkannt werden kann.



Zur bequemen Notation von Turing-Maschinen gibt es unterschiedliche Möglichkeiten:

Die **Turing-Tafel** ist die lineare Aufschreibung der Elemente der Menge $K \subseteq Z \times Y \times Y \times \{L, R\} \times Z$, definiert durch $(p, y, z, b, q) \in K$ gdw. $\delta(p, y) = (z, b, q)$. Wegen der Disjunktheit der Alphabete wird beim Aufschreiben auf Kommata und Klammern verzichtet:

Für DTM $A := (\{q_0, q_1, q_2, q_3, q_4\}, \{a, b\}, \{a, b, \hat{a}, \hat{b}\}, \delta, q_0, \{q_4\})$

Tabelle der Übergangsfunktion:

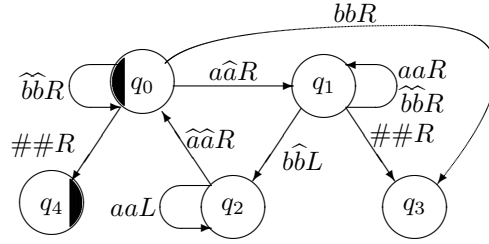
δ	$\#$	a	\hat{a}	b	\hat{b}
q_0	$\#Rq_4$	$\hat{a}Rq_1$	—	bRq_3	$\hat{b}Rq_0$
q_1	$\#Rq_3$	aRq_1	—	$\hat{b}Lq_2$	$\hat{b}Rq_1$
q_2	—	aLq_2	$\hat{a}Rq_0$	—	$\hat{b}Lq_2$
q_3	—	—	—	—	—
q_4	—	—	—	—	—

Turing-Tafel für δ als Relation $\delta \subseteq Z \times Y \times Y \times \{L, R\} \times Z$:

$q_0\hat{b}\hat{b}Rq_0$	$q_1a\hat{a}Rq_1$	$q_2a\hat{a}Lq_2$
$q_0b\hat{b}Rq_3$	$q_1\hat{b}\hat{b}Rq_1$	$q_2\hat{b}\hat{b}Lq_2$
$q_0\#\hat{b}Rq_4$	$q_1\#\hat{b}Rq_3$	$q_2\hat{a}\hat{a}Rq_0$
$q_0a\hat{a}Rq_1$	$q_1b\hat{b}Lq_2$	—

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

Oder als **Zustands-Diagramm**:



Der (System-)Zustand der Turing-Maschine zusammen mit der **wesentlichen** Inschrift des Arbeitsbandes wird in der sog. **Konfiguration** beschrieben. Wesentlich ist diejenige Inschrift des Arbeitsbandes, die vom am weitesten links stehenden bis zum am weitesten rechts stehenden Bandsymbol $y \in Y \setminus \{\#\}$ reicht. Weiterhin wichtig ist die Stellung des LSK **auf**, **vor** oder **hinter** dieser Inschrift. Daher müssen alle $\#$, die zwischen LSK (bzw. Bandanfang, bei Benutzung des einseitig unendlichen Bandes) und der wesentlichen Inschrift stehen, in der Konfiguration erfaßt werden.

7.5 Definition

Ein Wort $w \in Y^* \cdot Z \cdot Y^*$ heißt **Konfiguration** der TM $A := (Z, X, Y, \delta, q_0, Z_{end})$. Dabei bedeutet $w = upv$ mit $u, v \in Y^*$ und $p \in Z$, daß A sich im Zustand p befindet, die Bandinschrift uv ist und das erste Zeichen von v sich unter dem LSK befindet. Für den Fall, daß $v = \lambda$ ist, soll $\#$ unter dem LSK stehen. Falls $v \neq \lambda$ dann ist $v \in Y^*(Y \setminus \{\#\})$, d.h. das am weitesten rechts stehende Symbol von v ist nicht das Symbol $\#$ und rechts von v stehen ausschließlich die Symbole $\#$. Die **Menge aller Konfigurationen** der Turing-Maschine M sei $KONF_M$.

Zusammenfassend sei hier noch einmal dargestellt, wie eine DTM arbeitet:

Entsprechend der Übergangsfunktion δ verändert sie durch Überdrucken des Zeichens auf dem Arbeitsband dessen Inschrift und bewegt ihren Lese-/Schreibkopf um jeweils ein Feld nach rechts oder links. Ist $\delta(p, y) = (z, b, q)$, so bedeutet dies, daß die TM, wenn sie sich im Zustand p befindet und auf dem Arbeitsband das Zeichen $y \in Y$ liest (wobei $\#$ für das leere Feld steht), dieses mit dem Symbol z überschreibt, danach den LSK nach rechts bzw. links aufs Nachbarfeld setzt je nachdem, ob $b = R$ oder $b = L$ ist, und zuletzt in den Zustand q übergeht. Natürlich kann die TM den LSK nicht über den linken Rand des Bandes hinaus bewegen. Verlangt die Überföhrungsfunktion dieses, so blockiert die TM, so als wäre der LSK vom Arbeitsband heruntergefallen.

Im Einzelnen:

7.6 Definition

Für eine DTM A ist die Schrittrelation $\vdash_A \subseteq KONF_A \times KONF_A$ erklärt durch:

$w \vdash_A w'$ gilt genau dann, wenn eine der folgenden vier Bedingungen 1., 2., 3. oder 4. erfüllt ist, wobei im folgenden stets $u, v, w \in Y^*$, $x, y, z \in Y$ und $p, q \in Z$ sein soll:

1. $w = uypxv$ **und**

$$w' = \begin{cases} uqyzv, & \text{falls } (v \neq \lambda \text{ oder } z \neq \#) \text{ und } \delta(p, x) = (z, L, q) \\ uqy, & \text{falls } v = \lambda, y \neq \# \text{ und } \delta(p, x) = (\#, L, q) \\ uq, & \text{falls } v = \lambda, y = \# \text{ und } \delta(p, x) = (\#, L, q) \\ uyzqv, & \text{falls } \delta(p, x) = (z, R, q) \end{cases}$$

2. $w = uyp$ **und**

$$w' = \begin{cases} uqyz, & \text{falls } z \neq \# \text{ und } \delta(p, \#) = (z, L, q) \\ uqy, & \text{falls } y \neq \# \text{ und } \delta(p, \#) = (\#, L, q) \\ uq, & \text{falls } y = \# \text{ und } \delta(p, \#) = (\#, L, q) \\ yzq, & \text{falls } \delta(p, \#) = (z, R, q) \end{cases}$$

Und weil Bewegungen des LSK nach links am linken Rand nicht möglich sind:

3. $w = pxv$ **und**

$$w' = zqv, \text{ falls } \delta(p, x) = (z, R, q)$$

4. $w = p$ **und**

$$w' = zq, \text{ falls } \delta(p, \#) = (z, R, q)$$

Mit \vdash_A^* wird die reflexive, transitive Hülle von \vdash_A bezeichnet. Wenn keine Verwirrung entstehen kann, lassen wir den Index A weg und schreiben anstelle von \vdash_A nur \vdash .

Formal ist diese Definition nur deshalb so umständlich, weil das Wort v , das den rechten Teil der signifikanten Bandinschrift beschreibt, am Ende die Zeichen $\#$ nicht enthalten soll und auch der Teil links vom Kopf kein Zeichen enthält, wenn der LSK am linken Ende steht.

7.7 Definition

Jede Folge von Konfigurationen $k_1 \vdash k_2 \vdash \dots \vdash k_i \vdash k_{i+1} \vdash \dots$ heißt **Rechnung** der TM wobei eine endliche Rechnung $k_1 \vdash k_2 \vdash \dots \vdash k_t$ **Erfolgsrechnung** heißt, wenn $k_1 \in Y^* \cdot \{q_0\} \cdot Y^*$ und $k_t \in Y^* \cdot Z_{\text{end}} \cdot Y^*$ gilt.

Akzeptierte Sprache einer Turing-Maschine Turing-Maschinen akzeptieren ähnlich wie endliche Automaten formale Sprachen und können andererseits auch zum Berechnen von Funktionen verwendet werden. Letztlich sind dies beides nur unterschiedliche Sichtweisen eines gleichartigen Sachverhalts.

7.8 Definition

Für die DTM $A = (Z, X, Y, \delta, q_0, Z_{\text{end}})$ bezeichnet $L(A)$ (andernorts auch $|A|$) die von A akzeptierte Sprache:

$$L(A) := \{w \in X^* \mid \exists u, v \in Y^* \exists q \in Z_{\text{end}} : q_0 w \vdash^* uqv\}$$

Achtung: eine Turing-Maschine akzeptiert immer die ganze endliche Bandinschrift, auch wenn sie diese gar nicht „angeguckt“ hat.

Aufgabe 7.1:

Konstruieren Sie eine DTM, die eine der folgende Sprachen akzeptiert:

1. $\{a^n \mid \exists k \in \mathbb{N} : n = 2^k\}$
2. $\{ww^{\text{rev}} \mid w \in \{a, b\}^*\}$
3. $\{w\$w \mid w \in \{a, b\}^*\}$
4. $\{a^n b^n c^n \mid n \in \mathbb{N}\}$

Achtung: eine Turing-Maschine akzeptiert immer die ganze endliche Bandinschrift, auch wenn sie diese gar nicht „angeguckt“ hat.

7.2. Berechenbarkeit

Präzisierungen des Begriffes *Berechenbarkeit* gab es wie schon angedeutet viele verschiedene: λ -Definierbarkeit, berechenbare arithmetische Funktionen, μ -rekursive Funktionen, Markov-Algorithmen, Post'sche Systeme und eben die Turing-Berechenbarkeit.

7.9 Definition

Sei X ein Alphabet. Eine (Wort-)Funktion $f : X^* \xrightarrow{p} X^*$ heißt (**Turing-**)**berechenbar** oder **partiell rekursiv** genau dann, wenn es eine DTM $A = (Z, X, Y, \delta, q_0, Z_{end})$ gibt mit $q_0 w \vdash_A^* q_e v$, für einen Endzustand $q_e \in Z_{end}$ genau dann, wenn $f(w) = v$ ist.

Funktionen $f : \mathbb{N} \xrightarrow{p} \mathbb{N}$ sind natürlich auch berechenbar, wenn man eine geeignete Kodierung vornimmt. Dabei können die Zahlen unär oder binär kodiert werden, und zwar $i \in \mathbb{N}$ bei unärer Kodierung durch die Zeichenkette 0^{i+1} , wobei statt der 0 auch jedes andere Symbol verwendet werden kann. Dadurch ist es möglich, auch die natürliche Zahl *Null* zu kodieren, was durch $0^0 = \lambda$ nicht möglich wäre. Bei binärer Kodierung wird einfach die Binärdarstellung der Zahl $i + 1$ (ohne führende Nullen) als Code für die Zahl i benutzt und ein stets gleicher Block von Nullen als Trennsymbol.

Aufgabe 7.1:

Schreibe eine DTM, die ein Wort $w \in \{0, 1\}^*$ als Binärzahl interpretiert und dazu die durch w dargestellte Zahl unär erzeugt. Die TM soll mit der Anfangskonfiguration qw starten und mit der Konfiguration pa^m anhalten. Hierbei bezeichnet m die mit der Binärzahl w dargestellte Zahl.

Aufgabe 7.2:

Zeige, daß folgende Definition von TM-berechenbaren Funktionen äquivalent zu Definition 7.9 ist und eine korrekte Alternative dazu wäre:

Eine Funktion $f : X^* \xrightarrow{p} X^*$ heiße TM-berechenbar genau dann, wenn eine DTM B existiert, für die $L(B) = \{w\$v \mid f(w) = v\} \subseteq X^* \cdot \{\$ \} \cdot X^*$ ist.

7.10 Definition

Eine (i.a. partielle) Funktion $f : \mathbb{N}^r \xrightarrow{p} \mathbb{N}^s$ heißt (**TM-**)**berechenbar** oder **partiell rekursiv** genau dann, wenn eine DTM $A = (Z, X, Y, \delta, q_0, Z_{end})$ existiert mit $q_0 0^{m_1+1} 1 \dots 10^{m_r+1} \vdash_A^* p 0^{n_1+1} 10^{n_2+1} 1 \dots 10^{n_s+1}$, $p \in Z_{end}$ und $f(m_1, m_2, \dots, m_r) = (n_1, n_2, \dots, n_s)$, falls der Funktionswert definiert ist.

Einfache totale berechenbare Funktionen von $\mathbb{N} \times \mathbb{N}$ nach \mathbb{N} sind:

$$f(m, n) := m + n, \quad f(m, n) := m \cdot n, \quad f(m, n) := n^m$$

Daß oft genug partielle Funktionen vorkommen, kennen wir alle von dem Problem bei der Division durch Null: $f : \mathbb{Z} \times \mathbb{Z} \xrightarrow{p} \mathbb{Z}$ mit $f(p, q) := \frac{p}{q}$ ist aber nicht nur deswegen partiell, sondern auch, weil nicht jeder Bruch ganzer Zahlen wieder eine ganze Zahl ist.

7.3. Varianten der Turing-Maschine

Wenn wir uns das von A. Turing zuerst definierte Modell der nach ihm benannten universellen Maschine ansehen würden, dann stellten wir fest, daß dort ein nach beiden Seiten unbegrenztes Arbeitsband verwendet wird.

Letztlich ist dieses Modell genauso mächtig wie das mit nur einseitig unendlichem Arbeitsband. Ein Unterschied besteht in der vielleicht einfacheren, weil symmetrischeren, Notation für die Folgekonfiguration mit Hilfe einer Relation \vdash auf den Konfigurationen.

Aufgabe 7.1:

Definiere bei einer TM mit beidseitig unendlichem Band den Begriff der Konfiguration und die Relation \vdash für die Nachfolgekonfiguration k_j von k_i , d.h. das, was $k_i \vdash k_j$ bedeuten soll.

7.11 Definition

Zwei Turingmaschinen A und B heißen genau dann **äquivalent**, wenn sie die gleiche Sprache akzeptieren, d.h. $L(A) = L(B)$ gilt.

7.12 Theorem

Zu jeder DTM A mit einseitig unendlichem Arbeitsband gibt es eine äquivalente DTM B mit beidseitig unendlichem Arbeitsband und umgekehrt.

Beweis-Idee: A wird von B simuliert, indem sie sich links neben das Eingabewort eine Markierung schreibt und dann genauso wie A arbeitet. Falls dieses Feld einmal erreicht werden sollte, so muß B die Simulation beenden ohne zu akzeptieren.

B wird von A simuliert, indem A zwei Spuren auf dem Arbeitsband einrichtet, die das Band von B links und rechts des LSK darstellen. War $B = (Z, X, Y, \delta, q_0, Z_{end})$, so erhält $A = (Z', X, Y', \delta', q'_0, Z_{end})$ als Bandalphabet $Y' := Y \times Y \cup \{[y, *] \mid y \in Y\}$. Die endliche Kontrolle von A merkt sich, auf welchem Teil des Bandes B bei der Simulation gerade arbeitet und wechselt dies nur, wenn sie ein Zeichen $[y, *]$ liest. Im übrigen führt A jeden Schritt von B auf der Spur aus, die demjenigen Teil von B 's Band zugeordnet ist, auf dem B arbeitet. Die jeweils andere Spur ignoriert A , indem sie diese nicht verändert. Die Zustandsmenge von A ist dazu $Z' := \{[q, O], [q, U] \mid q \in Z\} \cup \{q'_0\}$, wobei $[q, U]$ bedeutet, daß B im Zustand q ist und der LSK links von dem mit 0 bezeichneten Feld des Arbeitsbandes ist. Dieses Feld ist jenes, auf dem sich der LSK in der Anfangskonfiguration von B befand.

Band von B :

...			-5	-4	-3	-2	-1	0	1	2	3	4	5		...
beidseitig unendliches Band															

Band von A :

0	1	2	3	4	5										...
*	-1	-2	-3	-4	-5										...
einseitig unendliches Band															

Ist die Anfangskonfiguration von B beschrieben durch $q_0 x_1 x_2 x_3 x_4 \dots x_n$, so sei die von A dann

$$q'_0[x_1, \#][x_2, \#][x_3, \#] \dots [x_n, \#].$$

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

Die unbeschriebenen Felder des Bandes von A sind dann die Symbole $[\#, \#]$, die mit dem Zeichen $\#$ identifiziert werden.

Wenn wir mit δ_A bzw. δ_B die Überföhrungsfunktionen von A und B bezeichnen, so definieren wir δ_A wie folgt:

$$\delta_A(q'_0, [a, \#]) := ([x, *], R, [q, O]), \text{ falls } \delta_B(q_0, a) = (x, R, q), \quad \forall a, * \in Y$$

B bewegt sich im ersten Schritt nach rechts, und

$$\delta_A(q'_0, [a, \#]) := ([x, *], R, [q, U]), \text{ falls } \delta_B(q_0, a) = (x, L, q), \quad \forall a, * \in Y,$$

B bewegt sich im ersten Schritt nach links.

Für jedes $[y_1, y_2] \in Y'$ mit $y_2 \neq *$ sei

$$\delta_A([p, O], [y_1, y_2]) := ([z, y_2], t, [q, O])$$

falls $\delta_B(p, y_1) = (z, t, q)$, mit $t \in \{R, L\}$.

B wird auf der oberen Spur simuliert, da der LSK von B rechts von seiner Anfangsposition steht und

$$\delta_A([p, U], [y_1, y_2]) := ([y_1, z], L, [q, U]), \text{ falls } \delta_B(p, y_2) = (z, R, q) \text{ bzw.}$$

$$\delta_A([p, U], [y_1, y_2]) := ([y_1, z], R, [q, U]), \text{ falls } \delta_B(p, y_2) = (z, L, q)$$

B wird auf der unteren Spur simuliert und der LSK von A bewegt sich entgegengesetzt zu dem von B .

Der Spurwechsel geschieht wie folgt:

$$\delta_A([p, O], [x, *]) := \delta_A([p, U], [x, *]) := ([z, *], R, [q, O]), \text{ falls } \delta_B(p, x) = (z, R, q)$$

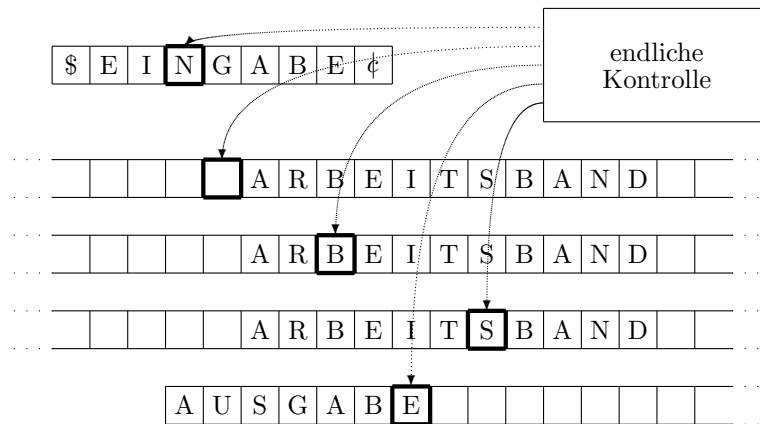
$$\delta_A([p, O], [x, *]) := \delta_A([p, U], [x, *]) := ([z, *], R, [q, U]), \text{ falls } \delta_B(p, x) = (z, L, q)$$

und A geht immer nach rechts.

Es sollte leicht nachzuvollziehen sein, daß jedem Übergang $k_1 \xrightarrow{B} k_2$ der TM B ein entsprechender Übergang $k'_1 \xrightarrow{A} k'_2$ entspricht. \square

So wie eben ein Turingband mit zwei Spuren verwendet wurde, kann man ebenso leicht, nur unter Vergrößerung des Alphabets der Bandzeichen, mehrere Spuren verwenden, um so die einzelnen Informationen darauf deutlicher werden zu lassen. Markierungen von einzelnen Zeichen der Bandinschrift können dann, ohne diese auf der ihr zugeordneten Spur zu verändern, einfach in die darunterliegende Spur geschrieben werden. Das bei einer TM mit k Spuren verwendete Alphabet Y' ist dann letztlich das cartesische Produkt $Y_k := \underbrace{Y \times \dots \times Y}_{k\text{-mal}}$ des Alphabets Y einer TM mit nur einer einzigen Spur.

Diese Überlegungen föhren zum Modell der sog. mehrbändigen off-line Turing-Maschine.



7.13 Definition

Eine ***k*-Band off-line Turing-Maschine** (TM) besitzt *k* beidseitig unbeschränkte Arbeitsbänder mit jeweils ihrem eigenen LSK sowie ein Eingabeband, auf dem die TM ausschließlich lesen kann, aber den Lese-Kopf dabei in beide Richtungen bewegen darf. Weiterhin besitzt diese TM ein Ausgabeband, auf dem sie nur schreiben kann und den Schreibkopf ausschließlich von links nach rechts weiterbewegt.

Die Konfiguration einer *k*-Band off-line TM wird entsprechend definiert, wobei das Gesamtalphabet ein entsprechend großes cartesisches Produkt von Alphabeten ist, die den Bändern einzeln zugeordnet sind.

Genauere Angaben sind hier nicht nötig und das folgende Resultat läßt sich mit dem nötigen formalen Aufwand auch exakt beweisen. Plausibel wird es sofort mit Blick auf die schon einmal verwendete Technik der Einteilung des Arbeitsbandes in Spuren.

7.14 Theorem

Zu jeder deterministischen *k*-Band off-line Turing-Maschine *A* mit $k \geq 1$ gibt es eine äquivalente DTM *B* mit nur einem Band.

Der Beweis sei den Leser(inne)n als Übung überlassen.

Die Notwendigkeit von off-line Turing-Maschinen wird klar, bei der Beschränkung des zur Berechnung zur Verfügung stehenden Platzes auf dem Arbeitsband auf weniger als die Länge der Eingabe selbst.

7.4. Nichtdeterministische Turing-Maschinen

Ähnlich wie bei den endlichen Automaten werden auch bei Turing-Maschinen Modelle definiert, die in einer Konfiguration nicht nur höchstens eine, sondern eine feste Zahl von verschiedenen möglichen Nachfolgekongfigurationen besitzt. Diese Maschinen werden nichtdeterministische Turing-Maschinen (NTM) genannt und ganz entsprechend definiert:

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

7.15 Definition

$M = (Z, X, Y, K, q_0, Z_{end})$ heißt **nichtdeterministische Turingmaschine (NTM)** genau dann, wenn zu jedem Paar $(p, y) \in Z \times Y$ eine endliche Zahl von möglichen Übergängen möglich ist, und damit die Übergangs-Funktion als $\delta : Z \times Y \rightarrow 2^{Y \times \{L, R\} \times Z}$ geschrieben werden müßte.

Wir verwenden aber hier stattdessen $K \subseteq Z \times Y \times Y \times \{L, R\} \times Z$ als Übergangsrelation. Bei einer nicht-deterministischen TM gibt es zu jedem Zustand q und jedem Symbol x unter dem LSK im allgemeinen mehrere verschiedene Möglichkeiten für die Folgekonfiguration. Alle anderen Definitionen entsprechen denen für die DTM.

Der folgende Beweis benötigt den Begriff der **Baum-Adresse**, den wir deswegen hier einführen.

7.16 Definition

Sei X ein endliches Alphabet, dessen Elemente mit $<$ total geordnet sind. Die sogenannte **lexikalische** oder natürliche Ordnung \ll auf X^* ist wie folgt definiert:

Für beliebige Wörter $u, v \in X^*$ gilt $u \ll v$ genau dann, wenn entweder $|u| < |v|$ oder $|u| = |v|$ und $u \leq_{lex} v$.

Hierbei ist \leq_{lex} die **lexikographische Erweiterung** der Ordnung $<$ von X auf X^* : $u \leq_{lex} v$ gdw. $u = v$ oder $u, v \in X$ und $u < v$ oder $u = wau_2$ und $v = wbv_2$ mit $a < b$ für passende Wörter $w, u_2, v_2 \in X^*$.

Als **Baum-Adressen** bezeichnet man meist die unter \ll geordnete Menge $\{1, 2, \dots, k\}^*$, wobei hier die Zahlen $1, 2, \dots, k$ und die daraus gebildeten Wörter nicht als Dezimalzahlen aufgefaßt werden, sondern als eine unter $<$ linear geordnete Menge von eindeutigen Bezeichnungen der Knoten von k -ären Bäumen in der oben beschriebenen Punktnotation. Das leere Wort λ wird dann meist für die Bezeichnung der Wurzel benutzt.

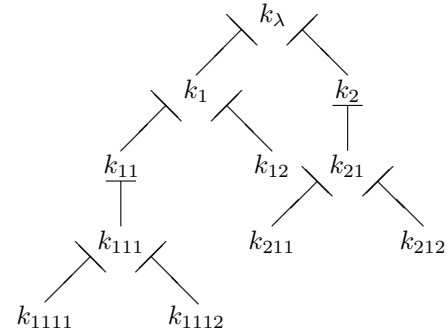
Auch bei Turing-Maschinen sind deterministische und nichtdeterministische Modelle äquivalent.

7.17 Theorem

Jede von einer nichtdeterministischen Turing-Maschine erkannte Sprache (bzw. berechnete Funktion) kann auch von einer deterministischen Turing-Maschine erkannt (bzw. berechnet) werden, kurz: $NTM \cong DTM$.

Beweis: Da jede DTM auch eine NTM ist, bleibt nur noch zu zeigen, wie die beliebige NTM $A := (Z, X, Y, K, q_0, Z_{end})$ durch eine DTM $B := (Z', X, Y', \delta, q'_0, Z'_{end})$ simuliert werden kann. Zu jedem Paar $(p, y) \in Z \times Y$ existiert eine begrenzte Anzahl von Kanten in K , die wir auch als Menge $\delta(p, y)$ geschrieben haben. Sei $k := \max \{|\delta(p, y)| \mid p \in Z \wedge y \in Y\}$ die maximal mögliche Zahl von verschiedenen Nachfolgekonfigurationen zu einer fest vorgegebenen.

So wie die Knoten der i -ten Schicht eines k -nären Baumes mit Tupeln $w \in \mathbb{N}^i$, den sogenannten Baum-Adressen (*tree-domain*), eindeutig bezeichnet werden, so werden dies auch die jeweils möglichen Nachfolge-Konfigurationen. Statt eines Tupels $(n_1, n_2, \dots, n_i) \in \mathbb{N}^i$, schreiben wir $n_1.n_2 \dots n_i$, wie bei der Numerierung hierarchischer Inhaltsverzeichnisse. Mit k_λ werde die Anfangskonfiguration von A bezeichnet.



$k_{u,i}$ ist für $1 \leq i \leq r$ diejenige Konfiguration, welche die NTM A aus k_u erreicht, wenn A sich im Zustand p befindet, das Symbol y liest und dann – bei einer vorausgesetzten linearen Ordnung der Elemente in jeder der Mengen $\delta(p, y) \subseteq Y \times \{L, R\} \times Z$ – das i -te Element der Menge $\delta(p, y)$ benutzt. Statt der Adresse $\lambda.i$ schreiben wir bei dieser Adressierungsmethode kurz i . Die DTM B hat drei Arbeitsbänder, wovon das erste die Eingabe von A enthält und auf dem zweiten Band Wörter $w \in \mathbb{N}^i$ in der sogenannten lexikalischen Ordnung auf \mathbb{N}^k generiert werden, d.h. nach aufsteigender Zahl der Komponenten und bei gleicher Komponentenzahl lexikographisch. Diese Ordnung ist total, und entspricht der bekannten Breitensuche, so daß jede Bezeichnung einer im Prinzip denkbaren (Nachfolge-) Konfiguration von k_0 in A erzeugt werden kann. Nach jeder auf dem zweiten Band erzeugten Numerierung $w \in \mathbb{N}^k$ (in Punktschreibweise), simuliert B die Rechnung von A auf dem dritten Band, indem sie die durch w vorgegebenen Überführungen vornimmt. Erreicht B dabei eine Endkonfiguration von A , so akzeptiert B . Da jede endliche Rechnung von A auf diese Weise irgendwann einmal vorkommt, wird von B natürlich genau die Sprache $L(A)$ akzeptiert. \square

7.5. Entscheidbarkeit und Aufzählbarkeit

Meist möchte man Turing-Maschinen nicht nur zum Akzeptieren von Wortmengen oder zum Berechnen von Funktionen verwenden, sondern auch um komplizierte Probleme algorithmisch zu lösen oder Ja-Nein-Fragen so zu entscheiden.

7.18 Definition

Eine Menge $M \subseteq X^*$ heißt (relativ zu X^*) **entscheidbar** genau dann, wenn die charakteristische Funktion $\chi_M : X^* \rightarrow \{0, 1\}$ berechenbar ist. Die Klasse aller entscheidbaren Mengen wird mit \mathcal{REC} (recursive sets) bezeichnet.

Aufgabe 7.1:

Zeige, daß eine Menge $M \subseteq X^*$ genau dann entscheidbar ist, wenn eine Turing-Maschine A existiert, die $M = L(A)$ akzeptiert und auf *jeder* Eingabe $w \in X^*$ anhält. Beachte dabei, daß unter „anhalt“ das reguläre Halten wegen fehlenden Eintrags in der Turing-Tafel, wie auch das Blockieren wegen nicht durchführbaren Zustandsüberganges, zu verstehen ist!

Neben den entscheidbaren und den aufzählbaren Mengen gibt es solche, die nicht einmal aufzählbar sind. Solche Mengen werden wir bald kennenlernen. Die von Turing-Maschinen akzeptierten Sprachen nennt man auch aufzählbare Mengen und verwendet häufig eine Definition, die die Ähnlichkeit zu der von den abzählbaren Mengen (siehe Definition 22.30) besonders deutlich macht.

7.19 Definition

Eine Menge $M \subseteq X^*$ heißt (**rekursiv**) **aufzählbar** genau dann, wenn $M = \emptyset$ ist, oder eine totale berechenbare Funktion $g : \mathbb{N} \rightarrow X^*$ existiert, für die $g(\mathbb{N}) = M$ ist. Die Familie aller aufzählbaren Mengen wird mit \mathcal{RE} (recursively enumerable sets) bezeichnet.

Damit der Begriff *aufzählbar* in diesem Zusammenhang deutlicher wird, sehen wir uns das folgende Ergebnis an.

7.20 Theorem

Eine Menge M ist genau dann aufzählbar, wenn eine k -Band off-line TM existiert, die jedes Wort der Menge M genau einmal auf ihr Ausgabeband schreibt. Dies ist genau dann der Fall, wenn $M = L(A)$ für eine DTM A ist.

Beweis-Idee: $M = L(A)$ für DTM $A \Rightarrow \exists k$ -Band off-line TM B und M ist aufzählbar:

B erzeugt Paare $(i, j) \in \mathbb{N} \times \mathbb{N}$ nacheinander in natürlicher Ordnung \gg auf \mathbb{N}^2 . Im Beispiel: $(0, 0) \gg (1, 0) \gg (0, 1) \gg (2, 0) \gg (1, 1) \gg (0, 2) \gg (3, 0) \gg (2, 1) \gg \dots$ Diese Ordnung ist erklärt durch: $(x, y) \gg (r, s)$ genau dann, wenn **entweder** $(x + y) < (r + s)$ **oder** $[(x + y) = (r + s) \text{ und } y < s]$. Diese Ordnung entspricht der lexikalischen Ordnung $<^{\text{lg-lex}}$ auf $\{a, b\}^*$ mit $a < b$. Es gilt $(i, j) \gg (r, s)$ gdw. $a^i b^j <^{\text{lg-lex}} a^r b^s$. Die Reihenfolge kann einfach bestimmt werden: Zuerst werden aufsteigend Summen der zwei Komponenten erzeugt, und dann bei gleicher Komponentensumme in den Vektoren werden diese nach absteigender erster Komponente erzeugt. Dann wird für jedes erzeugte Paar (i, j) , das i -te Wort $w_i \in X^*$ bezüglich der lexikalischen Ordnung auf X^* generiert, und sodann die DTM A genau j Schritte auf dem Wort w_i simuliert. Akzeptiert A in genau j Schritten, so schreibt B das Wort $w_i \#$ auf das Ausgabeband. Da jede endliche Rechnung von A auf jedem $w \in X^*$ vorkommt und bei Akzeptierung die deterministische TM A auch nur genau eine Rechnung macht, wird jedes von A akzeptierte Wort genau einmal auf das Ausgabeband geschrieben, sauberlich durch $\#$ getrennt. Auf die gleiche Weise kann eine TM konstruiert werden, die bei einer Eingabe $i \in \mathbb{N}$ nur das i -te erzeugte Wort auf ihr einziges Band schreibt.

k -Band off-line TM $B \Rightarrow \exists \text{ TM } A : M = L(A)$:

A wird als TM mit $k + 2$ Spuren definiert, deren Spuren den Bändern von B zugeordnet sind und die B simuliert. Dabei vergleicht sie nacheinander jedes (auf der dem Ausgabeband von B zugeordneten Spur) erzeugte Wort w mit der ihr vorgelegten (auf die erste Spur geschriebenen) Eingabe. Ähnlich geht A vor, wenn B bei Eingabe von $i \in \mathbb{N}$ diese Ausgabe erzeugt und so $g(\mathbb{N}) = L(A)$ berechnet. Stimmen beide überein, so akzeptiert A . \square

Was wir nun sehen und beweisen wollen, ist folgende Beziehung zwischen den bislang kennengelernten Klassen von Mengen:

Klasse der **entscheidbaren Mengen** \subsetneq Klasse der **aufzählbaren Mengen** \subsetneq Klasse der **abzählbaren Mengen** \neq Klasse der **überabzählbaren Mengen**.

Daß es überabzählbare, d.h. nicht mehr abzählbare Mengen gibt, sahen wir schon im ersten Kapitel in Theorem 2.2.35.

Bevor wir sehen, daß es Mengen gibt, die nicht aufzählbar sind, beachten wir einen Zusammenhang zwischen diesen und den entscheidbaren Mengen.

7.21 Theorem

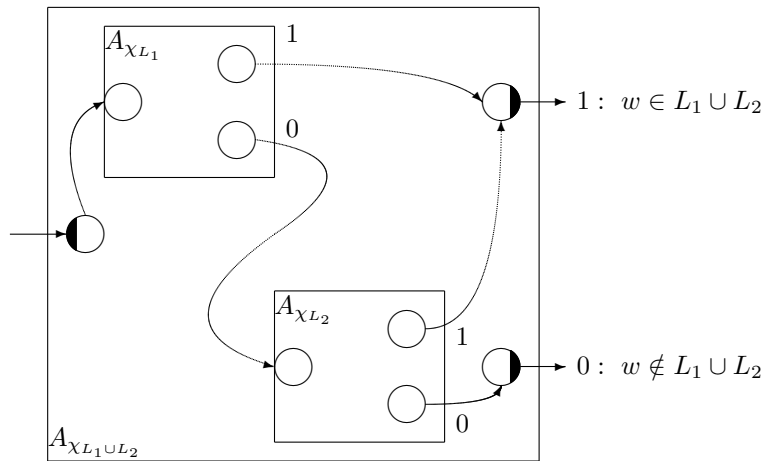
Eine Menge $M \subseteq X^*$ ist entscheidbar genau dann, wenn M und $X^* \setminus M$ beide aufzählbar sind.

Beweis: Da jede entscheidbare Menge aufzählbar ist, braucht nur noch die Umkehrung bewiesen werden: Seien A_M und $A_{\overline{M}}$ Turing-Maschinen, die M bzw. \overline{M} akzeptieren, aber nicht notwendiger Weise stets halten. Daraus konstruieren wir eine TM B_M , die M entscheidet. B_M verwendet A_M und $A_{\overline{M}}$ leicht modifiziert, indem die Folgekonfigurationen immer abwechselnd erzeugt werden und jede TM den nächsten Übergang nur machen kann, wenn die andere nicht angehalten hatte. Beide Maschinen bekommen die gleiche Eingabe w und werden solange simuliert, bis die erste angehalten hat und das Ergebnis $w \in M$? entschieden wurde. \square

7.22 Theorem

Die Klasse der entscheidbaren Mengen bildet eine Boolesche Mengen-Algebra.

Beweis: Da eine Menge M entscheidbar ist, wenn ihre charakteristische Funktion χ_M berechenbar ist, brauchen wir die TM, die die Funktion χ_M berechnet, für das Komplement $\overline{M} := X^* \setminus M$ von M nur leicht zu ändern: die Ausgaben der Symbole 1 und 0 werden nur vertauscht. So erhalten wir eine TM zur Berechnung von $\chi_{\overline{M}}$ und auch \overline{M} ist entscheidbare Menge. Wir sehen weiter an dem unten stehenden Bild, wie eine TM entworfen werden kann, die die charakteristische Funktion der Vereinigung zweier entscheidbarer Mengen berechnet. In der Skizze sind die Turing-Maschinen angedeutet, die die charakteristischen Funktionen der Mengen L_1 und L_2 berechnen. Der Zustand links ist jeweils der frühere Startzustand und die beiden Zustände rechts in jedem Bild einer dieser Turing-Maschinen waren die jeweiligen Endzustände. Diese werden entsprechend der eingezeichneten Kanten miteinander gekoppelt und die entstehende TM berechnet dann die charakteristische Funktion der Menge $L_1 \cup L_2$. Die DTM zur Entscheidung der Menge $M_1 \cap M_2 = X^* \setminus ((X^* \setminus M_1) \cup (X^* \setminus M_2))$ erhält man unter Anwendung dieser Konstruktionen ebenso einfach. \square



Das Cantor'sche Diagonalverfahren aus dem Beweis von Theorem 2.35 kann auch verwendet werden, um auf der Basis von zwei bekannten aufzählbaren Mengen eine neue zu definieren, die dann nicht mehr aufzählbar sein kann. Dies ist der kürzeste Beweis für die Existenz einer solchen Menge.

7.23 Definition

Eine totale, injektive, berechenbare Funktion $i : M \rightarrow \mathbb{N}$ von einer Menge $M \subseteq X^*$ in die Menge \mathbb{N} , heißt **Gödelisierung** von M , falls $\text{range}(i)$ eine entscheidbare Menge ist, und auch i^{-1} berechenbar ist.

In den meisten Fällen werden wir die Gödelisierungsabbildung selbst nicht benötigen, und werden für eine Turing-Maschine A anstelle von $g(A)$ dann $\langle A \rangle$ notieren, wobei A dann als Turingtafel angesehen wird.

In den hier interessierenden Fällen, wird $\mathbb{N} = \text{range}(i)$ in der Regel durch $G^* = \text{range}(i)$, für ein endliches Alphabet G , ersetzt werden. Die lexikalische Ordnung auf G^* ist total, und definiert eine weitere Gödelisierung $g_{lex} : G^* \rightarrow \mathbb{N}$, die dann der Abbildung i nachgeschaltet werden kann, damit letztlich $g_{lex} \circ i$ die verlangte Gödelisierung $\langle M \rangle$ von M in \mathbb{N} darstellt.

Oft werden auch Kodierungen g verwendet, die zunächst den einzelnen Buchstaben x_i eines Alphabetes $X := \{x_1, x_2, \dots, x_n\}$ die i -te Primzahl der Reihe $2, 3, 5, 7, 11, \dots$ zuordnet. Einem Wort $w \in X^*$, mit $w = x_{i_1}x_{i_2} \dots x_{i_n}$, wird dann folgende, eindeutig bestimmte, natürliche Zahl zugeordnet:

$$g(w) := \prod_{j=1}^n p_j^{i_j}$$

Aus $n \in \mathbb{N}$ erhält man $g^{-1}(n)$ durch fortgesetztes Dividieren.

Mit $X := \{x_1, x_2, x_3\}$ erhalten wir so $g(x_3x_2x_1x_2) = 17640$.

7.24 Theorem

Sei G ein Alphabet zur Kodierung von Turing-Maschinen bzw. Wörtern, sowie w_i das i -te Wort und A_i die i -te DTM in der lexikalischen Aufzählung der Wörter $\langle A_i \rangle \in G^*$. Dann ist die Menge $L_d := \{w_i \mid w_i \notin L(A_i)\}$ nicht aufzählbar.

Beweis: Daß G^* in der lexikalischen Ordnung aufzählbar ist, ist offensichtlich. Die Teilmenge von G^* aller derjenigen Wörter, die Kodierung einer Turing-Maschine sind, ist aufzählbar, weil eine TM existiert, die für jedes Wort $w \in G^*$ entscheiden kann, ob dieses die zulässige Kodierung einer TM ist. Betrachtet man die unendliche Matrix mit den Spalten w_1, w_2, w_3, \dots , den Zeilen A_1, A_2, A_3, \dots und den Einträgen: 1, falls $w_i \in L(A_i)$ bzw. 0, falls $w_i \notin L(A_i)$, so entspricht L_d gerade der Diagonalen, von der nur die Einträge 0 Berücksichtigung fanden. Es bleibt nur noch zu zeigen, daß L_d selbst nicht aufzählbar sein kann.

Angenommen, es sei L_d doch aufzählbar, d.h. es gibt eine TM A_j in der Aufzählung aller (Kodierungen von) Turing-Maschinen die L_d akzeptiert: $L_d = L(A_j)$. Nun gilt für das Wort w_j entweder $w_j \in L_d$ oder $w_j \notin L_d$. In beiden Fällen ergibt sich ein Widerspruch wie folgt: $w_j \in L_d \Rightarrow w_j \notin L(A_j)$ (Def. von L_d) $\Rightarrow w_j \notin L_d$ (Annahme). Andererseits gilt genauso folgende Schlußkette: $w_j \notin L_d \Rightarrow w_j \notin L(A_j)$ (Annahme) $\Rightarrow w_j \in L_d$ (Def. von L_d). In beiden Fällen ergibt sich also ein Widerspruch und die Annahme L_d sei aufzählbar ist nicht mehr zu halten. \square

Da wir jedoch zur Zeit kein Verfahren haben, um L_d anders zu beschreiben, wollen wir eine ähnliche Menge auch noch auf einem direkteren Wege erhalten. Zunächst jedoch stellen wir fest, daß eine Menge, die nicht aufzählbar ist, natürlich auch nicht entscheidbar sein kann. Mithin sind (wegen Theorem 7.21) $G^* \setminus L_d$ und auch L_d keine entscheidbaren Mengen. $G^* \setminus L_d = \{w_i \mid w_i \in L(A_i)\}$ ist jedoch

eine aufzählbare Menge, wie die Leser(innen) bitte durch eine(n) eigene(n) Beweisidee (Beweis) zeigen wollen. Vergleichen Sie dazu bitte Aufgabe 3.

Aufgabe 7.2:

Definiere eine Gödelisierung über dem Alphabet $G := \{0, 1\}$ mit der *jede* Turing-Maschine $A = (Z, X, Y, K, q_0, Z_{end})$ als eine Zeichenkette $\langle A \rangle$ über G^* dargestellt werden kann. Nehmen Sie dazu auch Literatur aus der Bibliothek zu Hilfe, falls das nötig wird.

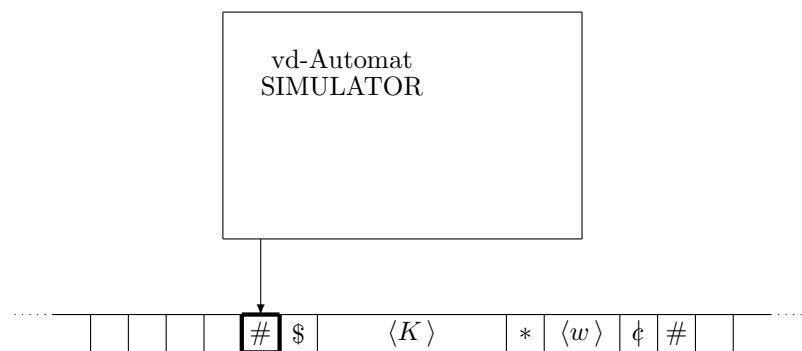
Nicht jedes Wort $w \in G^*$ ist freilich die Kodierung einer Turing-Maschine und daher definiere eine *möglichst einfache* TM, die überprüft, ob eine Zeichenkette über dem Alphabet $G := \{0, 1\}$ gemäß der angegebenen Gödelisierung die Kodierung einer Turingmaschine ist. Ist letztere Aufgabe auch mit einem Keller- oder gar endlichen Automaten zu lösen?

Aufgabe 7.3:

Zeige, daß die Menge $G^* \setminus L_d = \{w_i \mid w_i \in L(A_i)\}$ aufzählbar ist. Dies kann z.B. geschehen, durch Skizzieren einer möglichen TM (NTM oder DTM), die diese Sprache akzeptiert, oder einer k -Band off-line TM, die diese Menge aufzählt. Jede andere Methode ist natürlich auch willkommen.

7.6. Die universelle Turing-Maschine (UTM)

Jeder endliche Automat ist natürlich eine spezielle Turing-Maschine, die das Arbeitsband nur von links nach rechts lesen und niemals darauf schreiben kann. Die Zustandsübergänge in der endlichen Kontrolle der TM entsprechen denen im zu simulierenden Automaten. Der Nachteil dabei ist jedoch, daß für jeden endlichen Automaten eine eigene Turing-Maschine zu entwerfen ist. Dies paßt natürlich wenig zu dem Modell eines allgemeinen Algorithmusbegriffes. Eine Methode *jeden beliebigen* endlichen Automaten mit einer einzigen Turing-Maschine simulieren zu können wäre viel sinnvoller und ist tatsächlich leicht möglich.



Die obenstehend skizzierte TM kann als Simulator eines beliebigen, vollständigen DFA $A = (Z, X, K, \{q_0\}, Z_{end})$ benutzt werden. Jeder Zustand $q_i \in Z := \{q_0, q_1, \dots, q_n\}$ wird als Zeichenkette z^{i+1} kodiert und jedes Symbol $x_i \in X$ als x^i . Um die Gödelisierung $\langle K \rangle$ der Kantenmenge K zu erhalten, fehlen noch Symbole, um die Endzustände von anderen zu unterscheiden, Trennsymbole und Markierungszeichen. Wird die Kante $(q_i, x_j, q_k) \in K$ durch $-z^{i+1}x^jz^{k+1}$ kodiert, wobei das Zeichen $-$ (bzw. $+$) vor dem ersten z , die Tripel aus K voneinander trennt, und $+$ (im Unterschied zu $-$) den

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

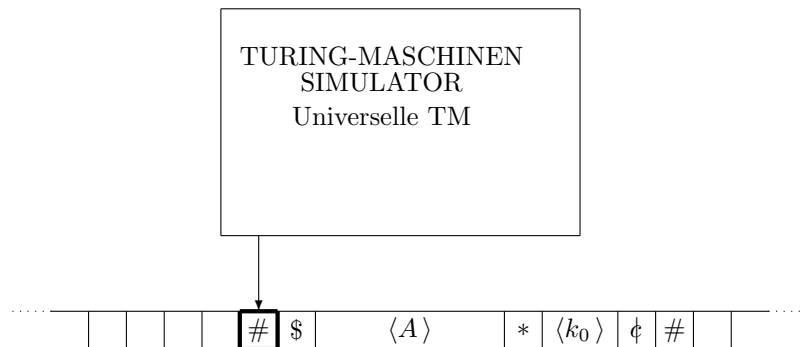
Zustand als Endzustand markiert, so kann K – wie auch das Eingabewort w des endlichen Automaten – über dem Alphabet $\{-, +, x, z\}$ kodiert werden.

Zusätzliche Symbole können dann noch markieren, welches der bei der Simulation von A gerade erreichte Zustand ist. Es sollte klar sein, daß jeder solchermaßen kodierte DFA von der TM simuliert werden kann, indem diese, durch Markieren des in A nach Einlesen des Symbols x_j erreichten Zustandes in der kodierten Kantenmenge K , dort den abgelesenen Folgezustand aufsucht und dann diesen markiert. Bei stets erfolgreicher Markierung des jeweils gelesenen Buchstabens der (kodierten) Eingabe w von A kann die TM dann bei Erreichen des Endes von $\langle w \rangle$ feststellen, ob $w \in A$ gilt.

Nach demselben Schema entwirft man eine TM, die jede beliebige andere Turing-Maschine A simuliert. Diese werden der sogenannten *universellen Turing-Maschine* (UTM) in ähnlicher Weise in kodierter (gödelisierter) Form als Zeichenketten $\langle A \rangle$ vorgesetzt, wie dies eben mit den endlichen Automaten geschehen war.

7.25 Definition

Für eine beliebige DTM A mit Anfangskonfiguration k_0 sei $\langle A \rangle$ (bzw. $\langle k_0 \rangle$) die Kodierung von A (bzw. von k_0) über dem Alphabet $G := \{0, 1\}$. Die DTM $U := (Z, X, Y, \delta, q_0, Z_{end})$ heißt **Universelle Turing-Maschine** (UTM), wenn sie mit der Anfangskonfiguration $q_0 \langle A \rangle \langle k_0 \rangle$ beginnend folgendes leistet: Für jeden Übergang $k_i \mapsto k_j$, den die DTM A nach dem Start in k_0 macht, gibt es in U entsprechende Übergänge von $\langle A \rangle \langle k_i \rangle$ in mehreren Schritten nach $\langle A \rangle \langle k_j \rangle$, wobei hier nur die relevante Bandinschrift von U gemeint ist, ohne die Stellungen ihres LSK und der Zustände.



Es ist bei der Simulation der TM A durch die UTM nur wichtig, daß die Konfigurationen von A in der gleichen Reihenfolge durchlaufen werden. Dazwischen liegen natürlich viele andere Konfigurationen der UTM, die diese zu der Simulation braucht. Dazu muß diese in der Turing-Tafel von A nachsehen, in welchem Zustand A sich gerade befindet und in welchen Nachfolgezustand A mit welcher Aktion zu gehen hat. Entsprechend wird dann die (kodierte) Konfiguration $\langle k_0 \rangle$ geändert.

7.7. Unentscheidbarkeit des Halteproblems

Das Halteproblem wird als die folgende formale Sprache H definiert.

7.26 Definition

Sei $H := \{ \langle A \rangle \langle w \rangle \mid \text{die TM } A \text{ hält bei Eingabe von } w \in \{0, 1\}^* \text{ an} \} \subseteq \{0, 1\}^*$.

Aufgabe 7.1:

Zeigen Sie, daß H eine aufzählbare Menge ist.

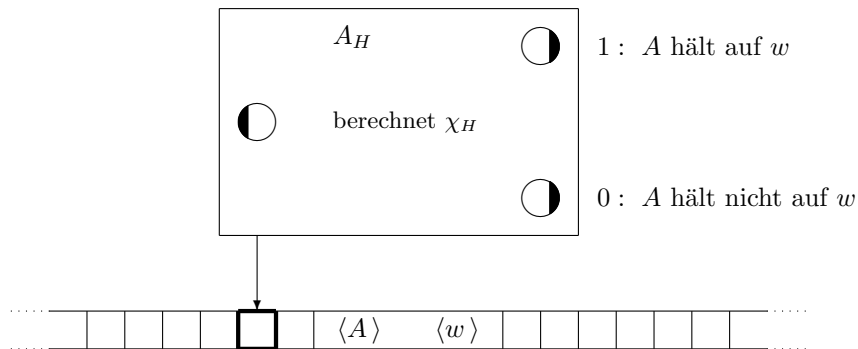
Aufgabe 7.2:

Zeigen Sie daß zu jeder Turing-Maschine A eine äquivalente TM B konstruiert werden kann, die auf einem Wort w genau dann hält, wenn $w \in L(A)$ ist.

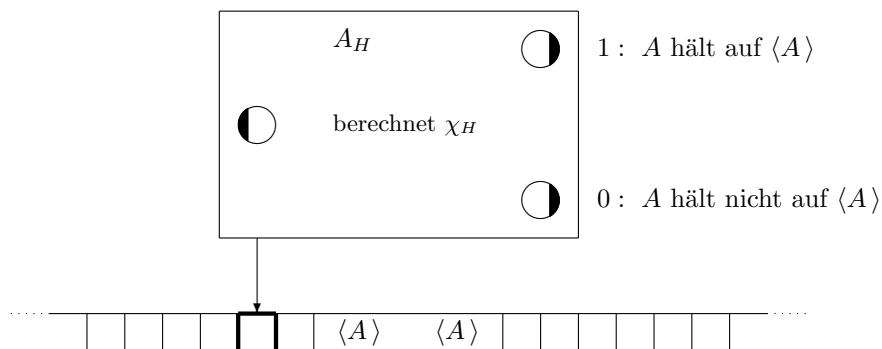
7.27 Theorem

(Halteproblem für Turing-Maschinen) Die Menge H aus Definition 7.26 ist nicht entscheidbar.

Beweis (indirekt): Angenommen, H wäre eine entscheidbare Menge, d.h. es gibt eine TM A_H die die charakteristische Funktion $\chi_H : X^* \rightarrow \{0, 1\}$ berechnet. A_H bekommt also Wörter der Form $\langle A \rangle \langle w \rangle \in \{0, 1\}^*$ als Eingabe und druckt eine 1, wenn A auf w hält, und eine 0 andernfalls.

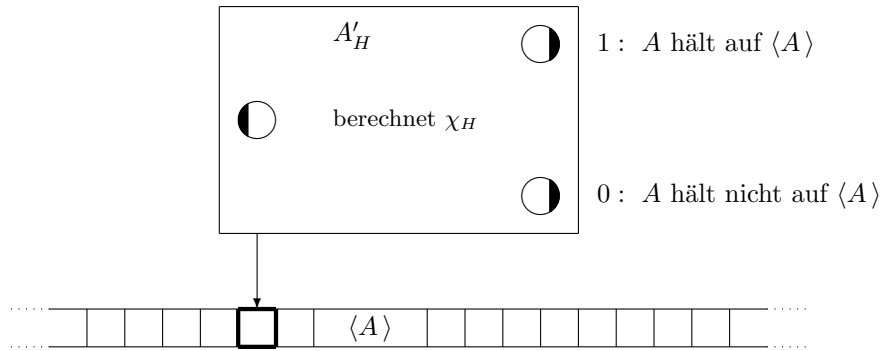


Statt des Wortes $\langle w \rangle$ kann natürlich auch die Kodierung $\langle A \rangle$ von A auf das Band von A_H geschrieben werden. In diesem Falle druckt die TM A_H genau dann eine 1, wenn A auf ihrer eigenen Kodierung anhält.

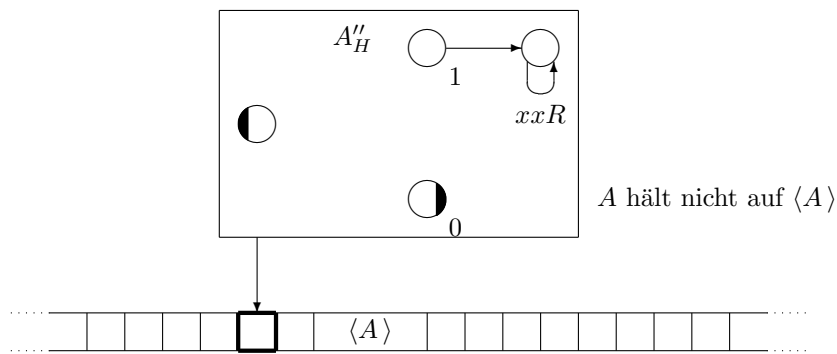


Wenn nun A_H so in eine TM A'_H geändert wird, daß A'_H die Eingabe einmal kopiert und danach wie A_H arbeitet, so ändert sich an dieser Eigenschaft fast nichts.

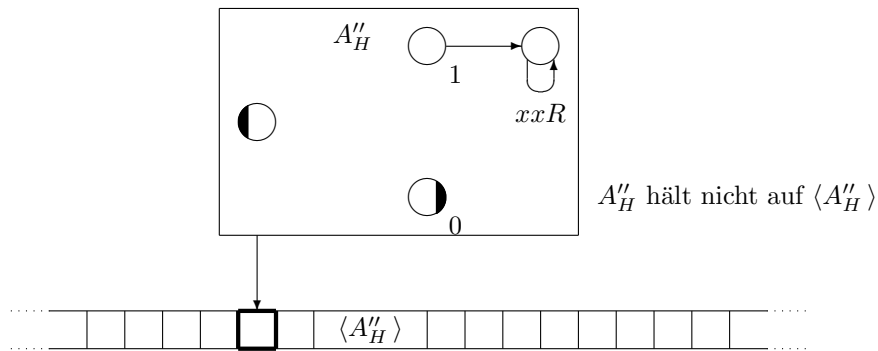
7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit



Nun ändern wir A'_H in A''_H so um, daß diese nun statt eine 1 zu drucken niemals anhält. A''_H hält also genau dann an und druckt eine 0, wenn A nicht auf $\langle A \rangle$ anhält.



Nun war die TM A eine TM, die nur auf ihren akzeptierten Eingaben hält, aber ansonsten ganz beliebig. A''_H war auch so eine TM und daher können wir auch deren Kodierung auf das Band schreiben.

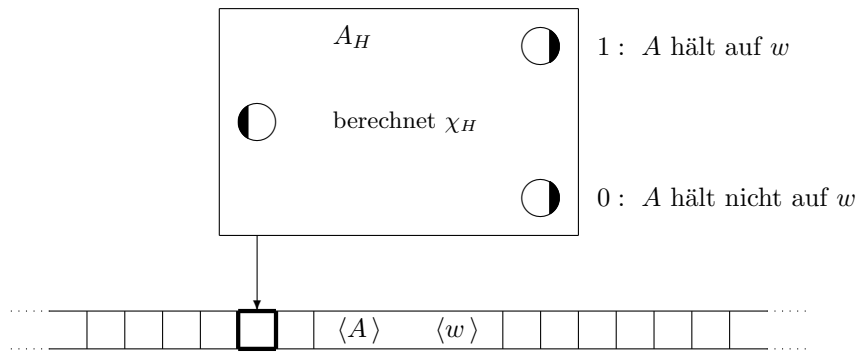


Dies ergibt aber nun einen Widerspruch, denn A''_H hält nun auf $\langle A''_H \rangle$ (und druckt 0) genau dann, wenn A''_H nicht auf $\langle A''_H \rangle$ hält. \square

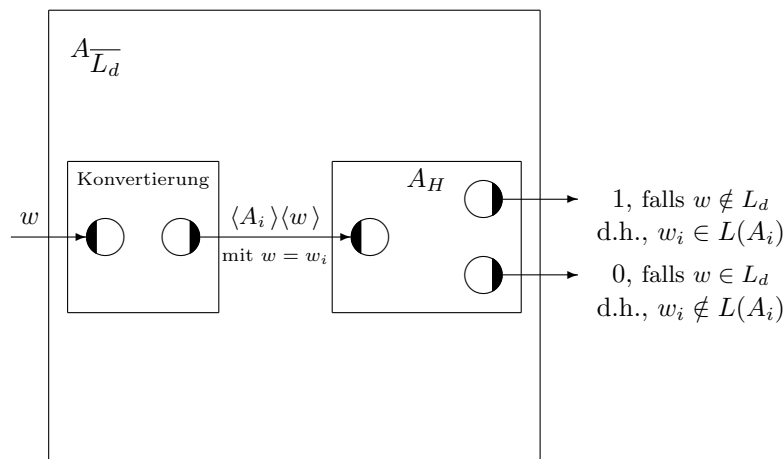
Als Konsequenz ergibt sich, daß H eine unentscheidbare Menge ist, und anders ausgedrückt, die charakteristische Funktion χ_H eine nicht berechenbare Funktion ist. Da, wie hoffentlich in Aufgabe 1 gezeigt wurde, H aufzählbar ist, folgt aus Theorem 7.21 die Nichtaufzählbarkeit des Komplements von H . Mit $\{0, 1\}^* \setminus H$ ist nun neben L_d also eine weitere nicht aufzählbare Menge bekannt.

Ohne diesen direkten Beweis für die Nichtentscheidbarkeit der Menge H hätten wir auch noch die Möglichkeit, dieses mit einer sogenannten Reduktion von H auf das Komplement von L_d zu zeigen. Mit L_d ist das Komplement $\overline{L_d}$ dann ja ebenfalls eine unentscheidbare Menge.

Angenommen, wir wüßten noch nicht, daß H unentscheidbar ist, und nähmen deshalb an, es gäbe einen Algorithmus für H , also eine TM A_H für die Berechnung der charakteristischen Funktion χ_H .



Unter Verwendung der TM A_H konstruieren wir eine TM, die die charakteristische Funktion von $\overline{L_d}$ berechnen könnte.



Diese neue TM $A_{\overline{L_d}}$ konstruiert aus der Eingabe von w zunächst den Index $i \in \mathbb{N}$ von w in der lexikalischen Ordnung, einfach durch Aufzählen aller Wörter in aufsteigender Reihenfolge. In gleicher Weise bestimmt sie die Kodierung der i -ten TM in der Aufzählung aller Gödelisierungen von Turing-Maschinen. Das Wort $\langle A_i \rangle \langle w_i \rangle$ wird dann der TM A_H als Eingabe vorgesetzt, und A_H bestimmt dann, ob A_i auf w_i hält. Tut sie dies, so kann anschließend leicht bestimmt werden, ob $w_i \in L(A_i)$ ist oder nicht. Der Widerspruch ist überzeugend, denn $\overline{L_d}$ ist ja als nicht entscheidbare Menge bekannt. Somit ist hiermit ein anderer Beweis für die Nichtentscheidbarkeit der Menge H gegeben. Die bloße Existenz solcher Mengen kann man natürlich ebenfalls, und eventuell etwas kürzer, mit dem klassischen Diagonalverfahren beweisen. Doch davon wollen wir hier absehen, denn zwei unterschiedliche Beweismethoden für ein wichtiges Ergebnis sollten ausreichen.

Da universelle Programmiersprachen die Möglichkeit geben, Interpreter (also UTM) zu konstruieren, ist also die Termination dieses universalen Interpreters auf beliebigen Eingaben nicht entscheidbar. Es

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

ist sogar schlimmer, denn sogar bei den von Haus aus stets terminierenden Programmen, ist es nicht entscheidbar, ob sie etwas Sinnvolles berechnen.

7.28 Definition

Mit \mathcal{M}_T sei die Menge aller (Kodierungen von) Turing-Maschinen M bezeichnet, die totale Funktionen $f_M : X^* \rightarrow \{0, 1\}$ berechnen.

7.29 Theorem

Es gibt keinen Algorithmus, der für eine beliebige Funktion $f_M \in \mathcal{M}_T$ entscheidet, ob $f_M(w) = 1$ für mindestens ein $w \in X^*$ gilt.

Beweis (wie so häufig indirekt): Angenommen, $\mathcal{P} := \{M \in \mathcal{M}_T \mid \exists w \in X^* : f_M(w) = 1\}$ sei eine entscheidbare Menge und $A_{\mathcal{P}}$ die TM, die die charakteristische Funktion $\chi_{\mathcal{P}}$ von \mathcal{P} berechnet. Es gälte also:

$$\chi_{\mathcal{P}}(\langle M \rangle) = \begin{cases} 1, & \text{falls } M \in \mathcal{P} \\ 0, & \text{sonst} \end{cases}$$

Wir wählen folgende Teilmenge von \mathcal{P} um einen Widerspruch herzuleiten: Die Klasse aller Turing-Maschinen $M_{B,w}$, die bei Eingabe von $n \in \mathbb{N}$ die genau n Schritte der TM B bei Eingabe von w simulieren. $M_{B,w}$ soll halten und eine 1 drucken, wenn die TM B in genau n Schritten auf w hält und eine 0, wenn B dies nicht tut. Nun gilt folgendes:

$\chi_{\mathcal{P}}(\langle M_{B,w} \rangle) = 1$ gdw. $M_{B,w} \in \mathcal{P}$ gdw. $M_{B,w}$ druckt eine 1 bei Eingabe von n gdw. B hält auf w nach n Schritten an gdw. $w \in L(B)$.

Damit würde $A_{\mathcal{P}}$ aber das Halteproblem entscheiden, weil für jede TM B und jede Eingabe w die TM $M_{B,w}$ dazu konstruiert werden kann. \square

Damit ist nun z.B. folgende Frage unentscheidbar:

Eingabe: eine beliebige stets haltende Turing-Maschine

Frage: wird bei allen Eingaben stets die Ausgabe 0 berechnet?

Man gewinnt als Korollar auch folgendes Ergebnis:

7.30 Korollar

Es ist unentscheidbar, ob eine beliebige, durch eine TM definierte, entscheidbare Menge M leer ist.

Beweis: Es gilt $M = \emptyset$ gdw. es gibt kein w mit $\chi_M = 1$ gdw. $A_M \notin \mathcal{P}$ \square

Teilmenge der Familie der aufzählbaren Sprachen, also bestimmte Mengen von aufzählbaren Sprachen, werden durch Eigenschaften bzw. Prädikate definiert, die diese Sprachen erfüllen sollen um zu dieser Teilmenge zu gehören.

Wir identifizieren im folgenden also jede solche Menge mit der Eigenschaft die sie definiert. Kurz: eine Eigenschaft ist eine Teilmenge der aufzählbaren Sprachen.

So ist die Familie \mathcal{REG} der regulären Mengen definiert als Teilmenge der Familie \mathcal{RE} der aufzählbaren Sprachen durch:

$$\mathcal{REG} := \{L \in \mathcal{RE} \mid \exists \text{ NFA } A \text{ mit } L = L(A)\}$$

7.31 Definition

Jede (Teil-)Menge \mathcal{S} von aufzählbaren Sprachen nennen wir eine **Eigenschaft** der aufzählbaren Sprachen. Eine Eigenschaft \mathcal{S} heißt **trivial** genau dann, wenn \mathcal{S} entweder die leere Menge ist oder alle aufzählbaren Mengen enthält.

Zum Beispiel ist $L = \emptyset \wedge \bar{L} = \emptyset$ eine triviale Eigenschaft, denn bei keiner Menge kann diese selbst und auch ihr Komplement leer sein. Die Menge aller Sprachen mit dieser Eigenschaft ist also leer.

Nichttriviale Eigenschaften zeichnen sich dadurch aus, daß es sowohl Mengen gibt, die sie erfüllen, wie auch andere, die diese Eigenschaft nicht erfüllen.

Das folgende Theorem ist in seiner Tragweite nicht zu unterschätzen:

7.32 Theorem

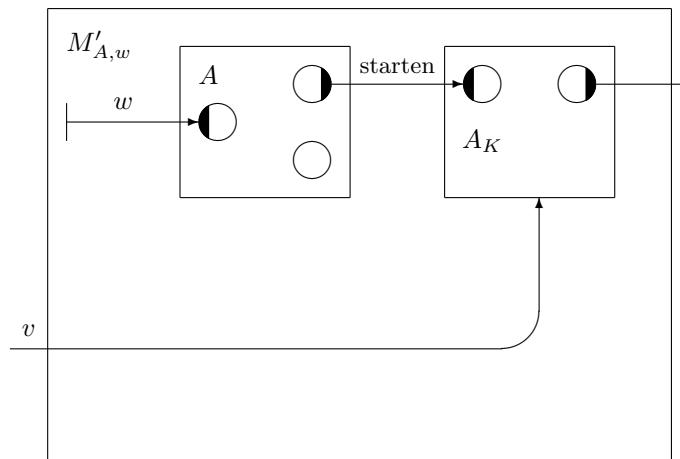
(Theorem von Rice) Jede nichttriviale Eigenschaft \mathcal{S} der aufzählbaren Sprachen ist unentscheidbar.

Beweis: O.B.d.A. sei $\emptyset \notin \mathcal{S}$, sonst wählen wir das Komplement von \mathcal{S} . Da \mathcal{S} nichttrivial sein sollte, existiert eine Menge $K \in \mathcal{S}$ mit $K = L(A_K)$ für eine TM A_K .

Wenn wir annehmen, daß \mathcal{S} eine entscheidbare Menge sei, dann gibt es eine auf jeder Eingabe haltende TM $A_{\mathcal{S}}$ mit

$$L(A_{\mathcal{S}}) = \{\langle A \rangle \mid L(A) \in \mathcal{S}\}$$

Sei $M'_{A,w}$ eine TM mit $L(M'_{A,w}) \in \mathcal{S}$ gdw. $w \in L(A)$ für eine vorgegebene TM A , die auf jeder Eingabe w genau dann hält, wenn $w \in L(A)$ ist (vergl. Übungsaufgabe 2). Die TM $M'_{A,w}$ kann aus $\langle A \rangle$ und $\langle w \rangle$ zusammen mit A_K leicht konstruiert werden und sieht wie folgt aus:



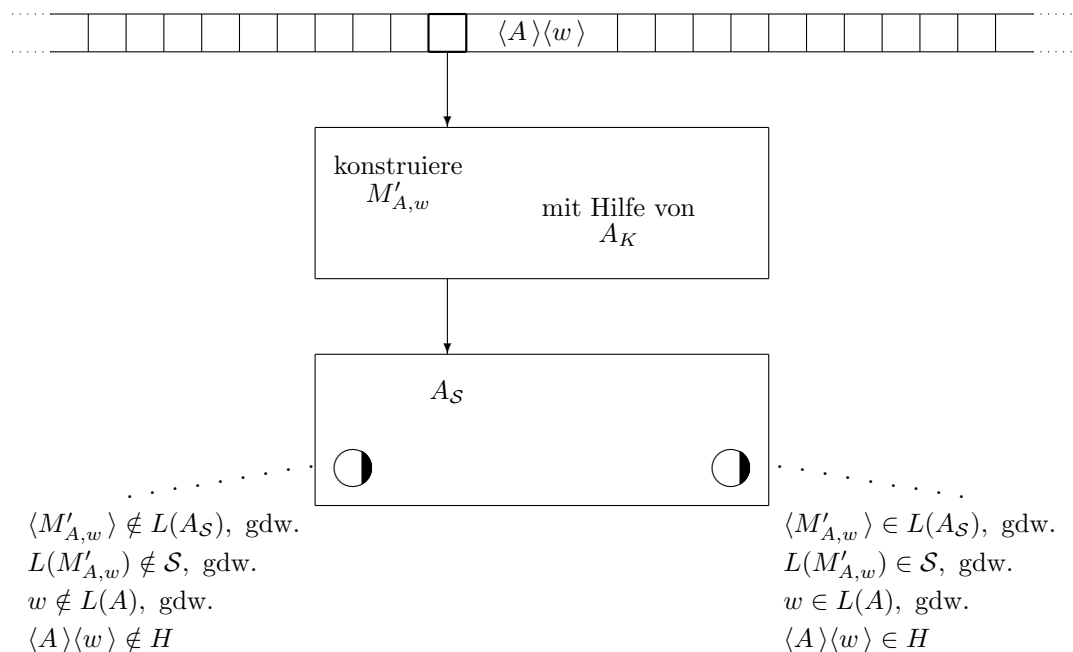
Hier gilt nun: $w \in L(A)$ impliziert $L(M'_{A,w}) = K$ und $w \notin L(A)$ impliziert $L(M'_{A,w}) = \emptyset$. Da $\emptyset \notin \mathcal{S}$ galt und $K \in \mathcal{S}$ war, bedeutet dies folgendes:

$$L(M'_{A,w}) \in \mathcal{S} \iff w \in L(A) \iff \langle A \rangle \langle w \rangle \in H$$

7. Turingmaschinen: Entscheidbarkeit und Berechenbarkeit

Dies wird nun benutzt, um die TM A_H zu konstruieren, von der wir ja wissen, daß es sie nicht geben kann.

Dazu benutzen wir die Maschine A_K , mit deren Hilfe wir bei Eingabe von $\langle A \rangle \langle w \rangle$ die TM $M'_{A,w}$ konstruieren werden, sowie die hypothetisch existierende TM A_S , die die charakteristische Funktion der Menge \mathcal{S} berechnet.



□

Mit dem Satz von Rice ergeben sich als Korollar nun viele unentscheidbare Probleme über aufzählbare Sprachen, die durch beliebige Turing-Maschinen definiert werden. Hier eine Auswahl:

Eingabe: Eine Turing-Maschine A

Frage: Ist $L := L(A)$ eine endliche Menge?

Dies wird im folgenden kürzer notiert als:

Ist L endliche Menge?

Ebenso sind die anderen Entscheidungsprobleme des Korollars zu sehen:

7.33 Korollar

Folgende Probleme sind unentscheidbar:

Ist L endliche Menge?

Ist L leere Menge?

Ist L unendliche Menge?

Ist L reguläre Menge?

Ist L kontextfreie Sprache?

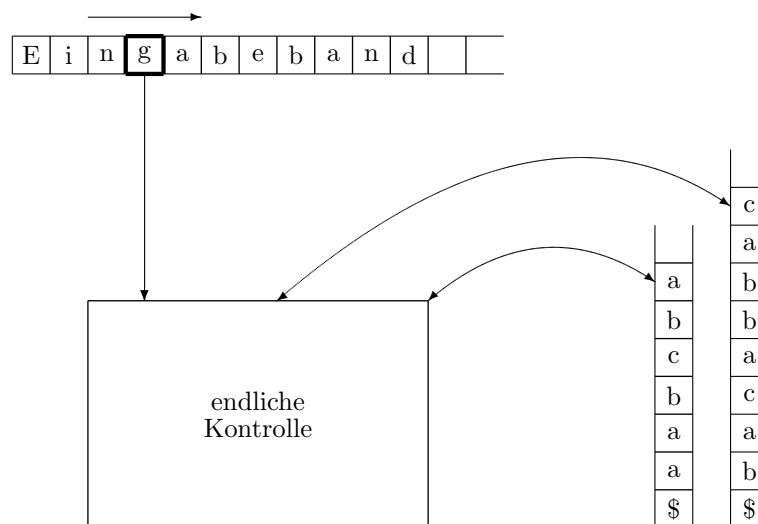
Ist L entscheidbare Sprache?

Ist L eine Menge mit genau einem Element?

Den Beweis für die einzelnen Aussagen mögen die Leser(innen) als Übung versuchen.

7.8. k -Keller-Automaten

Turing-Maschinen haben immer die Fähigkeit, auf ihren Arbeitsbändern an jeder erreichbaren Stelle zu lesen und zu schreiben. Zur Akzeptierung von kontextfreien Sprachen verwendet man aber meist Kellerautomaten, die einen Keller als Arbeitsband benutzen und deren Eingabeband nur in einer Richtung durchlaufen und dabei auch nur gelesen werden darf (*one-way*). Wir definieren hier eine Turing-Maschine mit mehreren Kellern, anstelle des sonst üblichen Kellerautomaten aus Kapitel 5.



Maschine mit zwei unabhängigen Kellern

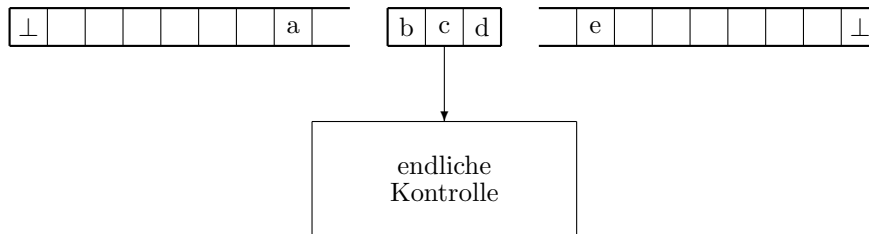
7.34 Definition

Ein **k -Keller-Automat** ist eine k -Band off-line TM mit einem **one-way Eingabeband** und k **Arbeitsbändern**, die alle am linken Ende einseitig beschränkt sind und nur in folgender Weise benutzt werden können: Der LSK liest ausschließlich das von $\#$ verschiedene Symbol am rechten Ende. Eine Veränderung des Kellers besteht aus mehreren Schritten der TM: Das oberste Symbol wird gelesen und nach dem Zustandsübergang gelöscht, nicht verändert oder es wird ein weiteres Symbol auf den Keller geschrieben. Nach jeder Kellerveränderung bewegt sich der LSK wieder über das letzte von $\#$ verschiedene Symbol. Um das linke Ende des Arbeitsbandes zu erkennen, ist jeweils zu Beginn ein spezielles **Kellerbodenzeichen**, z.B. \perp , vor das erste Symbol gesetzt worden. Es sollte klar sein, in welcher Weise eine TM diese Operationen ausführen muß.

7.35 Theorem

Jede aufzählbare Menge wird auch von einem 2-Keller-Automaten akzeptiert.

Beweis: Wir legen die beiden Keller waagerecht mit den oberen Enden gegen einander zeigend. Dazwischen ist ein Pufferfeld für drei Symbole. Auf diese Weise wollen wir das beidseitig unbeschränkte Turing-Band mit den zwei Kellern simulieren. Die drei Symbole in dem Pufferfeld sind die Zeichen links, unter und rechts neben dem LSK der TM und dieses wird in der endlichen Kontrolle der 2-Keller-Maschine gehalten.



Bewegt sich der LSK nach dem Ersetzen des gelesenen Symbols nach rechts, so wird das linke Symbol aus dem Puffer auf den linken Keller geschrieben und das oberste Symbol des rechten Kellers in den Puffer ans rechte Ende gesetzt, nachdem vorher im Puffer die Symbole alle um eine Position nach links geschoben wurden. Für die Bewegung des LSK nach links gilt entsprechendes (vertausche in der Formulierung *links* mit *rechts*). Wenn das Kellerbodenzeichen im Puffer auftaucht, so befindet sich der LSK auf einem leeren Feld mit Inschrift # und entsprechend wird dann der Keller behandelt. Die Details mögen die Leser selbst ausführen. \square

Nach Definition ist umgekehrt ein k -Keller-Automat eine spezielle Turing-Maschine, kann also auch nicht mehr leisten als diese.

Bedeutet es nun *geringere* Leistungsfähigkeit für einen 2-Keller-Automaten, wenn dieser seine Keller nur mit einem einzigen Symbol beschriften kann? Keller-Automaten, die ihren Keller *nur so* benutzen können, nennt man Zähler-Automaten.

7.36 Definition

Ein **k -Zähler-Automat** ist ein k -Keller-Automat, bei dem das Bandalphabet für die Keller (zusätzlich zu dem benutzten **Kellerboden-Zeichen** \perp) nur aus einem einzigen Symbol, z.B. $*$, besteht.

Üblicherweise identifiziert man den Kellerinhalt $*^n$ mit der Zahl $n \in \mathbb{N}$ und addiert bzw. subtrahiert 1 von dem Inhalt des i -ten Zählers k_i . Jeder k -Zähler-Automat kann also als modifizierter endlicher Automat gesehen werden, der bei den Zustandsübergängen die Einweg-Eingabe liest, die Zähler verändert oder diese auf Null abfragt. In der Maschinen-Tafel eines 3-Zähler-Automaten A bedeutet z.B. der Ausdruck $(p, x, +1, -1, 0, q)$, daß A im Zustand p und bei dem Eingabesymbol x folgendes tun kann: den ersten Zähler k_1 um eins erhöhen ($k_1 := k_1 + 1$), den zweiten Zähler k_2 um eins erniedrigen, sofern dies noch möglich ist (**if** $k_2 \geq 0$ **then** $k_2 := k_2 - 1$) und prüfen ob der Zähler k_3 leer ist ($k_3 = 0?$).

7.37 Theorem

Eine Menge M ist genau dann aufzählbar, wenn sie von einem Automaten der folgenden Art erkannt werden kann:

1. Einer deterministischen Turing-Maschine (DTM).
2. Einer nichtdeterministischen Turing-Maschine (NTM).
3. Einem 2-Keller-Automaten.
4. Einem 2-Zähler-Automaten.

Beweis:

1., 2. und 3. wurden schon gezeigt in den Sätzen 7.17, 7.20, und 7.35. Was noch fehlt, ist der Nachweis, daß jeder 2-Zähler-Automat genauso viel kann wie ein 2-Keller-Automat. Wir werden dazu einen Keller zunächst mit zwei Zählern und danach den so entstehenden 4-Zähler-Automaten erneut mit zwei Zählern simulieren.

Für ein Kelleralphabet $Y = \{y_1, y_2, \dots, y_{k-1}\}$ mit $k-1$ Symbolen wird der Kellerinhalt $y_{i_1} y_{i_2} y_{i_3} \dots y_{i_m}$ durch die Zahl $i_1 \cdot k^{m-1} + i_2 \cdot k^{m-2} + \dots + i_{m-1} \cdot k + i_m$ in eindeutiger Weise k -när kodiert. Die Keller-Operationen werden wie folgt auf einem Zähler z durchgeführt, wobei der zweite Zähler für die nötigen Berechnungen zu Hilfe genommen wird:

push(y_i) entspricht der Änderung $z := z \cdot k + i$

pop entspricht der Änderung $z := \lfloor \frac{z}{k} \rfloor$

top= y_i entspricht dem Test $i = z \bmod k$

Auf diese Weise erhält man einen 4-Zähler-Automaten, der den 2-Keller-Automaten simuliert. Diese vier Zähler kodieren wir nun anders, indem wir ihren vier Inhalten $i, j, k, l \in \mathbb{N}$, eineindeutig die Zahl $2^i 3^j 5^k 7^l$ zuordnen. Wird einer der ursprünglichen vier Zähler verändert, so muß jetzt diese Zahl „nur“ mit einer der vier zugeordneten Primzahlen 2, 3, 5 oder 7 multipliziert bzw. dividiert werden. Geht das Ergebnis der Division ohne Rest auf, so ist der entsprechende Zähler um 1 verringert. Bleibt bei der Division ein Rest, so war der simulierte Zähler gerade auf 0 gesunken. Auch dies kann mit Hilfe des zweiten zur Verfügung stehenden Zählers erledigt werden. Details schenken wir uns hier und verweisen für die Umkehrung wieder auf die Definition. \square

Es stellt sich nun auch die Frage, ob die aufzählbaren Mengen auch durch eine bestimmte Art von Grammatiken erzeugt oder generiert werden können, genauso, wie wir dies von den kontextfreien Sprachen kennen, die sich einerseits mit kontextfreien Grammatiken generieren lassen, andererseits aber ebenso von 1-Keller-Automaten akzeptiert werden können. Dies werden wir im nächsten Kapitel untersuchen.

8. Chomsky Typ-0 und Chomsky Typ-1 Grammatiken

Wenn wir unendliche Mengen definieren und behandeln wollen, brauchen wir stets endliche Beschreibungen von ihnen. Die übliche Darstellung von Mengen mit Hilfe von Prädikaten, denen die Elemente der Menge genügen sollen, ist eine häufig benutzte Weise. Für Wortmengen kennen wir, z.B. von den regulären Mengen, auch sogenannte endliche Ausdrücke, die rationalen Ausdrücke. Grammatiken, wie sie N. Chomsky 1956 bis 1959 einführte, sehen wir eigentlich in jeder Syntaxdefinition einer Programmiersprache. Ableitungs-Kalküle, wie sie auch in der Logik verwendet werden, stammen letztlich von Emil Post 1936 und als Ersetzungssysteme für Zeichenketten schon 1914 von Axel Thue. Diesen werden wir uns in einem späteren Kapitel ausführlicher widmen.

8.1. Typ-0 Grammatiken, Semi-Thue Systeme

Wir unterscheiden Ableitungs-Kalküle, meist formale Systeme genannt, von den oft generative Grammatiken genannten Ersetzungs-Kalkülen. Letztere stammen von den sogenannten *semi-Thue Systemen* ab und bildeten letztlich die Grundlage für die sogenannten Chomsky-Grammatiken. Daher beginnen wir mit der Definition dieser Systeme, die den eigentlichen Beginn der Theorie der formalen Sprachen bedeuteten:

8.1 Definition

Ein **semi-Thue System** (STS) über dem Alphabet X ist eine (endliche oder unendliche) Teilmenge $S \subseteq X^* \times X^*$, notiert meist als S anstelle des korrekten Tupels (X, S) . Die Elemente $(u, v) \in S$ nennt man Regeln und schreibt diese meistens als $u \rightarrow v$. Die zu S gehörende **einschrittige Ableitungsrelation** $\xrightarrow{(S)}$ $\subseteq X^* \times X^*$, ist wie folgt erklärt:

$$w_1 \xrightarrow{(S)} w_2 \text{ wenn } w_1 = \alpha u \beta \text{ und } w_2 = \alpha v \beta \text{ für } \alpha, \beta \in X^* \text{ und } u \rightarrow v \in S$$

Die reflexive, transitive Hülle von $\xrightarrow{(S)}$ wird wie üblich mit $\xrightarrow{*}_{(S)}$ bezeichnet, und ist die von S definierte **Ableitungsrelation**. Weitere Relationen auf der Basis von $\xrightarrow{(S)}$ sind für $n \in \mathbb{N}$ wie folgt erklärt:

$$\xrightarrow[n]{(S)} := \xrightarrow[n-1]{(S)} \circ \xrightarrow{(S)}$$

wobei $\xrightarrow[0]{(S)} := Id_S$. Dies beschreibt die Ableitungen in genau n Schritten. Den Index (S) lassen wir weg, wenn dadurch keine Verwirrung entstehen kann.

8. Chomsky Typ-0 und Chomsky Typ-1 Grammatiken

8.2 Definition

Eine **Typ-0** oder **Phrasenstruktur-Grammatik** wird durch ein Tupel $G := (V_N, V_T, P, S)$ spezifiziert. Hierbei gilt:

V_N ist ein endliches Alphabet von **Nonterminalen** (oder auch: **syntaktische Kategorie**, **metalinguistische Variable** oder **Hilfszeichen**).

V_T ist endliches Alphabet von **Terminalsymbolen** mit $V_N \cap V_T = \emptyset$.

$V := V_N \cup V_T$ ist das **Gesamtalphabet** von G .

$S \in V_N$ ist das **Startsymbol**.

$P \subseteq V^*V_NV^* \times V^*$ ist endliche Menge von **Produktionen** (oder **Regeln**), also ein spezielles STS über V .

Eine Regel $(u, v) \in P$ wird auch hier meist als $u \longrightarrow v$ geschrieben.

Die von G generierte oder erzeugte Sprache ist $L(G) := \{w \in V_T^* \mid S \xrightarrow[\overline{(P)}]{*} w\}$, und mit \mathcal{L}_0 wird die Familie aller von Typ-0 Grammatiken erzeugbaren Sprachen bezeichnet. D.h., $\mathcal{L}_0 := \{L \mid L = L(G) \text{ für eine Typ-0 Grammatik } G\}$.

Interessieren uns alle Satzformen die G generiert, so betrachten wir die **Satzformsprache** $S(G) := \{w \in V^* \mid S \xrightarrow[\overline{(P)}]{*} w\}$.

Das durch die Regelmengemenge P definierte STS im Index (P) wird meist durch (G) ersetzt, wenn dies überhaupt nötig sein sollte. Eine einzelne Grammatik wird oft nur durch Angabe ihrer Regeln notiert, wobei dann die Nonterminale stets die Großbuchstaben sind, während die Terminalsymbole mit kleinen Buchstaben abgekürzt werden.

Beispiel:

G sei gegeben durch die Regeln:

$$S \longrightarrow abc, \quad H_r b \longrightarrow bH_r, \quad bH_l \longrightarrow H_l b,$$

$$S \longrightarrow aH_r bc, \quad H_r c \longrightarrow H_l bcc, \quad aH_l \longrightarrow aaH_r, \quad aH_l \longrightarrow aa$$

Dann gilt $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

8.3 Theorem

Jede von einer Turing-Maschine akzeptierte Sprache kann auch von einer Typ-0 Grammatik generiert werden und umgekehrt, kurz: $\mathcal{L}_0 = \mathcal{RE}$.

Beweis: $\mathcal{L}_0 \subseteq \mathcal{RE}$:

Zu der Typ-0 Grammatik G konstruiert man eine NTM, bei der das Eingabewort w auf eine Spur des Arbeitsbandes kopiert wird und auf einer zweiten Spur zu Beginn nur das Startsymbol S von G steht. Die NTM führt die einzelnen Ableitungsschritte auf der zweiten Spur aus, indem die linke Seite einer Regel von G , in der dort stehenden Satzform, durch die zugehörige rechte Seite ersetzt wird. Die NTM schafft sich dabei den nötigen Platz, falls die rechte Seite der Regel länger als die linke ist. War die rechte Seite kürzer, so werden die entstehenden freien Felder durch Zusammenschieben beider entstandenen Teile

vernichtet. Die NTM prüft nach jeder Ersetzung nach, ob das Ergebnis mit der Eingabe übereinstimmt und akzeptiert bei Gleichheit. Es ist klar, dass diese NTM genau $L(G)$ akzeptiert, denn wenn das Wort abgeleitet werden kann, so existiert diese Ableitung auch in einer der durchgeführten Rechnungen der NTM.

$\mathcal{RE} \subseteq \mathcal{L}_0$:

Wir nehmen an, dass $L \in \mathcal{RE}$ von einer DTM $A := (Z, X, Y, \delta, q_0, Z_{end})$ akzeptiert wird, die ein beidseitig unendliches Band besitzt. Die Grammatik G zu dieser DTM wird zunächst aus dem Startsymbol S jede beliebige Anfangskonfiguration $q_0w \in ZX^*$ erzeugen und rechts folgend das Wort w noch einmal als Kopie. Dies ist mit leichter Änderung der Grammatik für die Sprache $\{ww \mid w \in X^*\}$ möglich. Aus S wird also die Menge $\{\$Q_0w\phi w \mid w \in X^*\}$ abgeleitet. Diese und alle folgenden Satzformen werden in dem Teil, der die TM Konfigurationen beschreiben wird, entsprechend der Übergangsfunktion δ verändert. Dabei wird zu jedem Zustand $q_i \in Z$ das Nonterminal Q_i zusammen mit den linken und rechten Nachbarsymbolen ersetzt. Die Zeichen $\$, \phi \notin Y$ fungieren dabei als Randmarkierungen. Jedes abgeleitete Wort, das einer Konfiguration entspricht, die einen Endzustand aus Z enthält, wird durch Löschen der Hilfszeichen für die Ränder und den Zustand in ein Terminalwort überführt. Die Regeln der Grammatik für die Änderungen der Konfigurationen sind dann folgende, wobei wir diejenigen zur Generierung der Menge $\{\$Qw\phi w \mid w \in X\}$, weggelassen haben. (vergleiche Übungsaufgabe 1)

$$\begin{aligned} yQ_ix &\longrightarrow Q_jyz, \forall y \in Y, \text{ falls } \delta(q_i, x) = (q_j, z, L) \\ \$Q_ix &\longrightarrow \$Q_j\#z, \text{ falls } \delta(q_i, x) = (q_j, z, L) \\ Q_ix &\longrightarrow zQ_j, \text{ falls } \delta(q_i, x) = (q_j, z, R) \text{ und } x \neq \phi \\ Q_i\phi &\longrightarrow zQ_j\phi, \text{ falls } \delta(q_i, \#) = (q_j, z, R) \end{aligned}$$

Zum Erzeugen des akzeptierten Wortes benötigen wir noch Regeln zum Löschen des gesamten Teils zwischen dem $\$$ und dem ϕ Symbol:

$$\begin{aligned} Q_i &\longrightarrow F, \text{ falls } q_i \in Z_{end} \\ Fy &\longrightarrow F, \forall y \in Y \\ yF &\longrightarrow F, \forall y \in Y \\ \$F\phi &\longrightarrow \lambda \end{aligned}$$

Was übrig bleibt, ist das akzeptierte Wort $w \in V_T^*$. Das Nonterminalalphabet V_N ist erklärt durch $V_N := (Y \setminus X) \cup \{Q_i \mid q_i \in Z\} \cup \{S, \$, \phi, F\}$ disjunkt vereinigt mit den Hilfszeichen, die zur Erzeugung von $\{\$Q_0w\phi w \mid w \in X\}$ benötigt werden. Als Terminalalphabet wird natürlich $V_T := X$ gewählt. Es gilt dann $L(G) = L(A)$ für die DTM A . \square

Diese Konstruktion würde prinzipiell auch bei einer NTM funktionieren, nur dann müßte statt der Übergangsfunktion δ dann entsprechend die Kantenmenge K in der Formulierung verwendet werden.

Aufgabe 8.1:

1. Schreiben Sie eine Typ-0 Grammatik G für die Sprache:

$$L(G) = \{w \mid \exists v \in \{a, b\}^* : w = \$v\phi v\}$$

2. Ist es einfacher, die Menge $\{w \mid \exists v \in \{a\}^* : w = \$v\phi v\}$ zu erzeugen? Von welchem Typ ist diese Sprache?

8.2. Typ-1 Grammatiken, kontextsensitive Grammatiken

Die Sprachklassen, die wir bisher kennengelernt hatten, waren auf unterschiedlichste Weisen definiert worden. Alle diese Verfahren haben ihre Vor- und Nachteile: endliche Ausdrücke sind knapp und leicht verständlich, Grammatiken spezifizieren mehr die Syntax, während Automaten oder Maschinen die Implementationen von Algorithmen in den Vordergrund stellen. Es ist wichtig zu wissen, mit welchen Automatenmodellen Sprachen akzeptiert werden können, die durch gewisse Typen von Grammatiken generiert werden. Ein erstes Beispiel dafür waren die hinlänglich bekannten regulären Mengen mit ihren *rationalen Ausdrücken*, den *endlichen Automaten* (NFA oder DFA, was immer am besten geeignet ist) oder den *einseitig linearen kontextfreien (Typ-3) Grammatiken*. Ein anderes Beispiel war die Menge $\mathcal{L}_0 = \mathcal{RE}$, die nach Theorem 8.3 wahlweise durch Typ-0 Grammatiken oder Turing-Maschinen definiert werden kann.

Damit wir Möglichkeiten bekommen, bisher unbekannte und neue Klassen von formalen Sprachen einzustufen und diese mit schon bekannten zu vergleichen, benutzen wir beweisbare Eigenschaften über diese Familien von Sprachen. Der Zweig der Theoretischen Informatik, der sich damit hauptsächlich befaßt, ist die sogenannte AFL-Theorie (*abstract formal languages*).

8.4 Definition

Eine Menge \mathcal{L} heißt **Sprachfamilie** genau dann, wenn folgendes gilt:

1. Für jede Sprache $L \in \mathcal{L}$ gibt es ein endliches Alphabet X_L mit $L \subseteq X_L^*$.
2. Es existiert eine Sprache $L \in \mathcal{L}$ mit $L \neq \emptyset$.

Alle bisher definierten Klassen von Sprachen waren in diesem Sinne *Sprachfamilien*. Operationen, die man auf Sprachen anwendet, wie z.B. den mengentheoretischen Durchschnitt kann man zwar auch auf die Sprachfamilien anwenden, da diese ja selbst Mengen sind, jedoch ist dies im allgemeinen nicht das Ziel. Offensichtlich ergeben sich dabei wenig neue Gesichtspunkte: $\mathcal{CF} \cap \mathcal{REG} = \mathcal{REG}$ ist bekannt, denn jede reguläre Menge ist auch eine kontextfreie Sprache.

Was wir eigentlich betrachten wollen, ist hier eher der Durchschnitt der Sprachen aus den beiden Familien. Man bezeichnet in der AFL Theorie dann auch korrekterweise diese Operationen auf den Sprachen als *Operatoren* auf den Sprachfamilien, um so den Unterschied deutlich zu machen. Wir wollen das hier nicht weiter ausführen und betrachten hier nur sogenannte Abschlusseigenschaften.

8.5 Definition

Eine Sprachfamilie \mathcal{L} ist **abgeschlossen unter einer Operation** $o : 2^X \times 2^X \rightarrow 2^X$ genau dann, wenn $o(L_1, L_2) \in \mathcal{L}$, für alle $L_1, L_2 \in \mathcal{L}$ gilt.

Aufgabe 8.1:

Stellen Sie eine Liste der bisher betrachteten Sprachfamilien zusammen und geben Sie für jede einzelne die bisher bekannten Abschlusseigenschaften an. Welche kommen dafür bisher in Frage? Greifen Sie erst dann zu Literatur, wenn die Matrix anscheinend zu klein ausfällt und eigene Beweise nicht einfallen wollen.

8.6 Definition

Eine Typ-0 Grammatik $G = (V_N, V_T, P, S)$ wird genau dann **Typ-1** oder **kontextsensitive Grammatik** genannt, falls $u \rightarrow v \in P$ nur dann gilt, wenn entweder $u = \alpha A \beta$, $v = \alpha w \beta$ mit $A \in V_N$, $w \in V^+$ und $\alpha, \beta \in V^*$ oder $u = S$, $v = \lambda$ und S kommt in keiner Regel auf der rechten Seite vor.

Die Familie der **kontextsensitiven Sprachen** wird mit \mathcal{CS} oder \mathcal{L}_1 abgekürzt, und es ist $\mathcal{L}_1 := \{L \mid L = L(G) \text{ für eine Typ-1 Grammatik } G\}$.

Bei Chomsky heißen die einseitig linearen Grammatiken **Typ-3 Grammatiken**, während die kontextfreien Grammatiken als **Typ-2 Grammatiken** bezeichnet wurden. Entsprechend wurden dann die dazugehörigen Sprachfamilien mit \mathcal{L}_3 und \mathcal{L}_2 abgekürzt.

In kontextsensitiven Grammatiken wird ein Nonterminal nur im Kontext der Worte α und β ersetzt. Ist dieser nicht vorhanden, so liegt nur noch eine *kontextfreie Grammatik* vor, die bis auf die Sonderregel $S \rightarrow \lambda$ auch noch λ -frei ist. In der ursprünglichen Definition von N. Chomsky kam die praktische Regelung für das Startsymbol mit der Regel $S \rightarrow \lambda \in R$ noch nicht vor.

Betrachten wir die bisher definierten Sprachfamilien, so werden allein aus den Definitionen schon einige Glieder der nachfolgenden Inklusionskette sichtbar. Es gilt $\mathcal{REG} = \mathcal{L}_3 \subseteq \mathcal{CF} = \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{REC} \subseteq \mathcal{RE} = \mathcal{L}_0$, und wir werden zeigen, daß alle Inklusionen dieser sogenannten Chomsky-Hierarchie echt sind. Die Ungleichheit der Familien \mathcal{REG} und \mathcal{CF} wurde schon früher bewiesen und die echte Inklusion $\mathcal{L}_2 \subsetneq \mathcal{L}_1$ folgt aus der Tatsache, daß die nicht kontextfreie Sprache $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ eine kontextsensitive Grammatik besitzt. Die Familie der entscheidbaren Mengen \mathcal{REC} wurde von Chomsky noch nicht dieser Hierarchie zugerechnet, aber wir werden sehen, daß jede kontextsensitive Sprache entscheidbar ist. Daß es aufzählbare Mengen gibt, die nicht entscheidbar sind, hatten wir in Kapitel 7 ebenfalls bewiesen, so daß lediglich noch die folgenden Ergebnisse zu beweisen sind: $\mathcal{L}_1 \subseteq \mathcal{REC}$, $\mathcal{L}_1 \neq \mathcal{REC}$ und $\{a^n b^n c^n \mid n \in \mathbb{N}\} \in \mathcal{L}_1$.

Bevor wir dies tun, werden wir eine praktischere Form von Grammatiken definieren, mit denen ebenfalls genau die kontextsensitiven Sprachen generiert werden können.

Eine Regel der Form $AB \rightarrow BA$ ist nicht kontextsensitiv aber in der Wortlänge nicht verkürzend. Keine kontextsensitive Regeln (bis auf $S \rightarrow \lambda$) ist verkürzend und man kann zeigen, daß dies eine charakterisierende Eigenschaft für die Typ-1 Sprachen ist.

8.7 Definition

Eine Typ-0 Grammatik $G = (V_N, V_T, P, S)$ heißt **monoton**, falls $\forall u \rightarrow v \in P : (|u| \leq |v|)$ oder ($u = S$, $v = \lambda$ und S kommt in keiner Regel rechts vor).

$\mathcal{MON} := \{L \mid L = L(G) \text{ für eine monotone Grammatik}\}$ bezeichnet die Familie der von diesen Grammatiken erzeugbaren Sprachen.

8.8 Theorem

Es gilt: $\mathcal{CS} = \mathcal{L}_1 = \mathcal{MON}$

Beweis: $\mathcal{L}_1 \subseteq \mathcal{MON}$ ist offensichtlich, da alle kontextsensitiven Regeln monoton sind.

Um $\mathcal{MON} \subseteq \mathcal{L}_1$ zu zeigen, konstruieren wir zu beliebig vorgegebener monotoner Grammatik $G := (V_N, V_T, P, S)$ die äquivalente Typ-1 Grammatik $G' := (V'_N, V'_T, P', S')$ mit $L(G) = L(G')$. Wir definieren dazu V'_N mit Hilfe neuer Symbole $\binom{A}{k}$ so:

8. Chomsky Typ-0 und Chomsky Typ-1 Grammatiken

$$V'_N := \left\{ \binom{A}{k} \mid A \in V \text{ und } 1 \leq k \leq |P| \right\} \cup V$$

Die neue Regelmenge P' wird für jede Produktion aus P einen ganzen Satz von kontextsensitiven Regeln enthalten, den wir für jede einzelne wie folgt beschreiben: Sei die k -te Regel von P die Regel $A_1 A_2 \dots A_n \rightarrow B_1 B_2 \dots B_m$, so werden folgende neue Produktionen definiert:

$$\begin{aligned} A_1 A_2 \dots A_n &\rightarrow \binom{A_1}{k} A_2 \dots A_n \\ \binom{A_1}{k} A_2 \dots A_n &\rightarrow \binom{A_1}{k} \binom{A_2}{k} \dots A_n \\ &\vdots \\ \binom{A_1}{k} \binom{A_2}{k} \dots \binom{A_{n-1}}{k} A_n &\rightarrow \binom{A_1}{k} \dots \binom{A_n}{k} \binom{B_{n+1}}{k} \dots \binom{B_m}{k} \\ \binom{A_1}{k} \dots \binom{A_n}{k} \binom{B_{n+1}}{k} \dots \binom{B_m}{k} &\rightarrow B_1 \binom{A_2}{k} \dots \binom{B_m}{k} \\ &\vdots \\ B_1 B_2 \dots \binom{B_m}{k} &\rightarrow B_1 B_2 \dots B_m \end{aligned}$$

Da die Nonterminalzeichen von V'_N für je zwei Regeln von P alle disjunkt sind, kann eine einmal begonnene Ableitung von solch einem zusammengehörenden Regelsatz aus P' nur erfolgreich terminieren, wenn er ganz zu Ende ausgeführt wird. Einen noch formaleren Beweis mögen ungläubige Leser(innen) selbst ausführen. \square

8.9 Theorem

Jede kontextsensitive Sprache ist entscheidbar, d.h. $\mathcal{L}_1 \subseteq \mathcal{REC}$.

Beweis: Für eine Sprache $L \in \mathcal{L}_1$ sei $L = L(G)$ für eine monotone Grammatik G , was nach Theorem 8.8 möglich ist. Um nun zu entscheiden ob ein beliebiges Wort w in L vorkommt, betrachten wir alle möglichen Ableitungen $S \xrightarrow[\overline{(G)}]{*} v$ mit $|v| \leq |w|$. Da es davon stets nur endlich viele ohne Wiederholungen gibt, und weil auch jede darin vorkommende Satzform nicht länger als das Wort w sein kann, kann dieses mit einer TM leicht geprüft werden. \square

Die Frage, ob nun vielleicht umgekehrt jede entscheidbare Menge auch kontextsensitiv ist, kann wieder mit einem Diagonalbeweis negativ entschieden werden.

8.10 Theorem

Es gilt: $\mathcal{L}_1 \not\subseteq \mathcal{REC}$

Beweis: Wir konstruieren die entscheidbare Menge $L_e \notin \mathcal{L}_1$: G_1, G_2, \dots bzw. $\langle G_1 \rangle, \langle G_2 \rangle, \dots$ sei eine Aufzählung aller monotonen Grammatiken. Dies ist möglich, weil für jedes Wort über dem Kodierungs-Alphabet $\{0, 1\}$ entschieden werden kann, ob es die Gödelisierung einer monotonen Grammatik ist.

Weiter sei $f : \{0, 1\}^* \rightarrow \mathbb{N}$ eine berechenbare Bijektion, die jedem Wort w die Zahl $f(w) = i$ zuordnet, wenn w das i -te Wort in der lexikalischen Ordnung auf $\{0, 1\}^*$ ist. Nun definieren wir:

$$L_e := \{w \mid w \notin L(G_{f(w)})\}$$

Wir beobachten, daß L_e entscheidbar ist: Für ein beliebiges Wort $v \in \{0,1\}^*$ berechnet man $f(v)$, konstruiert dann die monotone Grammatik $G_{f(v)}$ und testet, ob $v \notin L(G_{f(v)})$ gilt. $L_e \notin \mathcal{L}_1$ folgt nun durch Widerspruchsbeweis: Falls $L_e \in \mathcal{L}_1$ wäre, so gäbe es eine monotone Grammatik G_n mit $L(G_n) = L_e$. Für das n -te Wort u der lexikalischen Aufzählung von $\{0,1\}^*$ mit $f(u) = n$ gilt nun *entweder* $u \in L_e$ *oder* $u \notin L_e$. In beiden Fällen entsteht ein Widerspruch wie folgt:

$$u \in L_e \xrightarrow{\text{Def. } L_e} u \notin L(G_n) \xrightarrow{\text{Annahme}} u \notin L_e$$

Andererseits ergibt sich auch:

$$u \notin L_e \xrightarrow{\text{Def. } L_e} u \in L(G_n) \xrightarrow{\text{Annahme}} u \in L_e$$

Mithin der Widerspruch zur Annahme $L_e \in \mathcal{L}_1$. □

8.3. Linear beschränkte Automaten

Nachdem wir sahen, daß die Typ-0 Sprachen von allgemeinen Turing-Maschinen akzeptiert werden, stellt sich die Frage, wie die Maschinen aussehen könnten, die zu den kontextsensitiven Sprachen gehören. Dies wird eine NTM sein, die einer Beschränkung in der Benutzung ihres einzigen Arbeitsbandes unterworfen ist, dem linear beschränkten Automaten, (LBA).

8.11 Definition

Ein **linear beschränkter Automat (LBA)** ist eine NTM, bei der auf dem Arbeitsband für eine Konstante $c \in \mathbb{R}^+$ höchstens $c \cdot |w|$ Felder bis zur Akzeptierung besucht werden. Arbeitet die Turing-Maschine bei gleicher Beschränkung ihres Arbeitsbandes deterministisch, so wird der linear beschränkte Automat mit **DLBA** abgekürzt.

Die Familie der von LBA's bzw. DLBA's akzeptierten Sprachen wird mit \mathcal{LBA} bzw. \mathcal{DLBA} bezeichnet.

8.12 Theorem

Es gilt: $\mathcal{LBA} = \mathcal{L}_1$

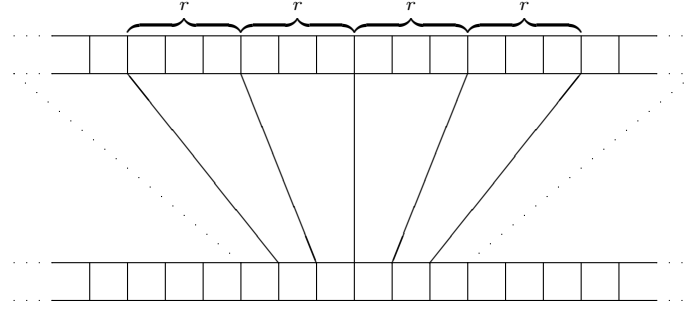
Beweis: Da $\mathcal{L}_1 = \mathcal{MON}$ ist, zeigen wir zunächst $\mathcal{MON} \subseteq \mathcal{LBA}$:

Die Konstruktion im Beweis von Theorem 8.3 für eine NTM, die eine Typ-0 Sprache akzeptiert, wird auch hier durchgeführt. Weil die vorgegebene Grammatik monoton war, muß diese NTM lediglich jedesmal prüfen, ob die Länge der Satzform auf der zweiten Spur länger wird als das Eingabewort. In diesem Fall darf die NTM nicht akzeptieren. Daher ist diese leicht modifizierte NTM tatsächlich ein LBA. Für jedes Wort v das auf der zweiten Spur erzeugt wird und dessen Länge gleich der von w ist, prüft dieser LBA nach, ob $v = w$ gilt. Tritt dieser Fall ein, so akzeptiert der LBA das Wort w . Also akzeptiert der aus der monotonen Grammatik konstruierte LBA die gesamte Sprache L .

Für den Beweis von $\mathcal{LBA} \subseteq \mathcal{MON} = \mathcal{L}_1$ konstruieren wir zu einem beliebigen LBA A , der mit Platzbeschränkung $c \cdot |w|$ arbeitet, eine monotone Grammatik. Falls nun die Konstante c größer als 1 ist, können wir die Inschrift des Arbeitsbandes des LBA nicht als Satzformen beim Ableiten verwenden. Die Konstruktion im Beweis von Theorem 8.3 ist leider auch nicht zu verwenden, weil diese Grammatik

8. Chomsky Typ-0 und Chomsky Typ-1 Grammatiken

nicht monoton ist. Wir zeigen also zunächst, daß jeder LBA statt mit $c \cdot |w|$ stets auch nur mit genau $|w|$ Arbeitsfeldern auskommen kann. Sei also der LBA A wie oben so, daß er mit $c \cdot |w|$ benutzten Arbeitsfeldern akzeptiert. Ist $0 < c \leq 1$, so ist nichts zu beweisen. Für $c > 1$ konstruieren wir den LBA B , der mit $|w|$ Platz auskommt und auch $L = L(A)$ akzeptiert. B hat als Bandalphabet die Menge Y^r , wobei Y das entsprechende Alphabet von A war und r eine noch festzulegende natürliche Zahl sein wird. Jedes Element aus Y^r stellt einen *Block* von r , auf dem Arbeitsband von A nebeneinander liegenden, Zeichen dar. Bewegt sich der LSK von A in solch einem Block, so kann dies der LBA B ohne Kopfbewegung nur innerhalb seiner endlichen Kontrolle tun.



B braucht auf diese Weise nur $\frac{c}{r} \cdot |w|$ Felder seines Arbeitsbandes zu betreten. Daß das Bandalphabet von B so viel größer werden mußte, ist dabei unerheblich. Wählt man $r := \min(n \in \mathbb{N} \mid n \geq c)$, so sind das höchstens $|w|$ Felder.

Zu dem nichtdeterministischen LBA $B = (Z, X, Y, K, q_0, T)$ konstruiert man nun eine Grammatik $G := (V_N, V_T, P, S)$, bei der fast jedes Nonterminalsymbol vier Spuren besitzt und die wesentlichen Satzformen alle aus $|w|$ Zeichen bestehen werden. Die Anfangs-Konfiguration $q_0 w$ für ein beliebiges Wort $w \in X^*$ wird von S aus mit folgenden Regeln erzeugt:

$$S \longrightarrow A \begin{bmatrix} x \\ x \\ \text{\$} \\ \# \end{bmatrix}, A \longrightarrow A \begin{bmatrix} x \\ x \\ \# \\ \# \end{bmatrix} \text{ und } A \longrightarrow \begin{bmatrix} x \\ x \\ \text{\$} \\ q_0 \end{bmatrix}, \text{ für alle } x \in X$$

Dabei markieren $\text{\$}$ und $\text{\$}$ den linken bzw. den rechten Rand der Eingabe $w \in X^*$, die in den ersten beiden Spuren der Nonterminale steht. Das Symbol für den Zustand des LBA's steht immer in der letzten Spur desjenigen Zeichens x , das der LSK von B gerade besucht. Die anderen Regeln beschreiben dann die Übergänge der Konfigurationen anhand der Maschinen-Tafel K . Für alle Symbole $x_1, x_2, x_3 \in$

$X, y_1, y_2 \in Y$ werden für jedes Tupel $(p, y, z, L, q) \in K$ – bzw. jedes $(p, y, z, R, q) \in K$ – folgende Regeln in G aufgenommen:

$$\begin{array}{ccc} \begin{bmatrix} x_1 \\ y_1 \\ \# \\ \# \end{bmatrix} \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_1 \\ y_1 \\ \# \\ q \end{bmatrix} \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix}, \quad \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ \# \end{bmatrix} \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} \longrightarrow \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ q \end{bmatrix} \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix} \quad \text{sowie} \\ \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ \# \end{bmatrix} \begin{bmatrix} x_2 \\ y \\ \phi \\ p \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_1 \\ y_1 \\ \$ \\ q \end{bmatrix} \begin{bmatrix} x_2 \\ z \\ \phi \\ \# \end{bmatrix} \end{array}$$

Für die Bewegungen nach rechts lauten die Regeln ganz entsprechend:

$$\begin{array}{ccc} \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} \begin{bmatrix} x_3 \\ y_2 \\ \# \\ \# \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix} \begin{bmatrix} x_3 \\ y_2 \\ \# \\ q \end{bmatrix}, \quad \begin{bmatrix} x_2 \\ y \\ \# \\ p \end{bmatrix} \begin{bmatrix} x_3 \\ y_2 \\ \phi \\ \# \end{bmatrix} \longrightarrow \begin{bmatrix} x_2 \\ z \\ \# \\ \# \end{bmatrix} \begin{bmatrix} x_3 \\ y_2 \\ \phi \\ q \end{bmatrix} \quad \text{sowie} \\ \begin{bmatrix} x_2 \\ y \\ \$ \\ p \end{bmatrix} \begin{bmatrix} x_3 \\ y_2 \\ \phi \\ \# \end{bmatrix} & \longrightarrow & \begin{bmatrix} x_2 \\ z \\ \$ \\ \# \end{bmatrix} \begin{bmatrix} x_3 \\ y_2 \\ \phi \\ q \end{bmatrix} \end{array}$$

Zum Schluß müssen alle Nonterminale genau dann in das entsprechende Terminalsymbol überführt werden, wenn ein Endzustand bei der Berechnung des LBA erreicht wurde.

Die Regeln dafür sind für alle $x, z \in X$, für $y \in Y$ und $\& \in \{\#, \$, \phi\}$ wie folgt erklärt:

$$\begin{bmatrix} x \\ y \\ \& \\ q \end{bmatrix} \longrightarrow x \text{ falls } q \in T \text{ und } \begin{bmatrix} x \\ y \\ \& \\ \# \end{bmatrix} z \longrightarrow xz \text{ bzw. } \begin{bmatrix} x \\ y \\ \& \\ \# \end{bmatrix} z \longrightarrow zx$$

Mit diesen Regeln kann also ein terminales Wort genau dann erzeugt werden, wenn es eine Erfolgsrechnung für w im LBA mit $|w|$ Speicherplatz gibt. \square

Mit Hilfe von Charakterisierungen von Sprachklassen durch Automaten, lassen sich in vielen Fällen einige Abschlusseigenschaften der Sprachfamilien leichter beweisen als mit Grammatiken. Dies ist zum Beispiel beim Beweis des folgenden Theorems zu sehen:

8.13 Theorem

Die Familie der kontextsensitiven Sprachen $\mathcal{L}_1 = \mathcal{LBA}$ ist gegenüber Durchschnittsbildung abgeschlossen.

Beweis: Seien A und B zwei LBA's für $L_A := L(A)$ und $L_B := L(B)$. Man kann ohne größere Schwierigkeiten einen LBA C konstruieren, der A auf einer Spur des Arbeitsbandes und B auf einer zweiten Spur simuliert. C akzeptiert das Wort w genau dann, wenn w von A und B akzeptiert wird. Der Platzbedarf von C übersteigt den von A und von B natürlich nicht. \square

8. Chomsky Typ-0 und Chomsky Typ-1 Grammatiken

Die Beweise für die Abgeschlossenheit von \mathcal{L}_1 gegenüber *Vereinigung* oder *nicht-löschendem Homomorphismus*, sind natürlich einfacher mit Grammatiken zu zeigen. Daß auch das Komplement einer kontextsensitiven Sprache wieder eine kontextsensitive Sprache ist, wurde 1987 von N. Immerman (Yale Univ.) und – unabhängig davon – von R. Szelépcsenyi (ČSSR) bewiesen und ist daher in vielen Lehrbüchern noch nicht enthalten. Im Beweis werden platzbeschränkte Turing-Maschinen verwendet und das Resultat ist zudem noch allgemeiner formuliert (siehe dazu [Wegener]).

Aufgabe 8.1:

Untersuchen Sie folgende Abschlußeigenschaften für die Sprachfamilie \mathcal{L}_1 , und beweisen Sie die davon gültigen, mit der am einfachsten erscheinenden Methode:

1. Vereinigung - also: $A, B \in \mathcal{L}_1 \implies A \cup B \in \mathcal{L}_1$
2. Komplexprodukt - also: $A, B \in \mathcal{L}_1 \implies A \cdot B \in \mathcal{L}_1$
3. nicht-löschendem Homomorphismus - also: $A \subseteq X^*$, $A \in \mathcal{L}_1$ und $h : X^* \rightarrow Y^*$ mit $|w| \leq |h(w)|$ für jedes $w \in X^*$ $\implies h(L) \in \mathcal{L}_1$
4. beliebigem Homomorphismus - also: $h(L) \in \mathcal{L}_1$ für jedes $L \in \mathcal{L}_1$ und jeden Homomorphismus h
5. Kleene'sche Hülle, d.h. *-Abschluß - also: $L^* \in \mathcal{L}_1$ falls $L \in \mathcal{L}_1$
6. Quotientenbildung mit regulären Mengen - also: $L \in \mathcal{L}_1$ und $R \in \mathcal{REG} \implies L/R := \{u \mid \exists v \in R : uv \in L\} \in \mathcal{L}_1$

Denken Sie beim Nachschauen und dem Vergleich mit existierender Literatur daran, daß dort in vielen Fällen das leere Wort λ (wie in der Originaldefinition von N. Chomsky) nie in einer kontextsensitiven Sprache vorkommt.

8.4. Die Chomsky-Hierarchie

Die folgende Tabelle gibt eine Übersicht über alle Sprachfamilien an, die wir kennengelernt haben. Die Sprachen vom Typ-3 bis Typ-0 bilden dabei die Elemente der **Die Chomsky-Hierarchie**. Zu jeder Familie geben wir ein akzeptierendes Automaten- und ein generierendes Grammatikmodell an (sofern vorhanden). Außerdem ist angegeben, ob die jeweilige Sprachfamilie unter dem Operator (Vereinigung \cup , Schnitt \cap sowie Komplement $\bar{\cdot}$) abgeschlossen ist.

8.4. Die Chomsky-Hierarchie

Sprachfamilie	Automaten	Grammatik	Beispiel	\cup	\cap	$\bar{\cdot}$
endl. Mengen	–	–	$\{a, ab, abb\}$	+	+	–
$\mathcal{R}eg$	DFA=NFA	Typ-3 = rechts-lineare G.	$\{a\}^* \{b\}^*$	+	+	+
$det\mathcal{C}f$	DPDA	$LR(k), k \geq 1$	$\{a^n b^n \mid n \in \mathbb{N}\}$ $\{wcw^{rev} \mid w \in \{a, b\}^*\}$	–	–	+
$\mathcal{L}_2 = \mathcal{C}f$	PDA	Typ-2 = kontextfreie G.	$\{ww^{rev} \mid w \in \{a, b\}^*\}$	+	–	–
$\mathcal{L}_1 = \mathcal{C}s$	NLBA	Typ-1 = monotone G.	$\{a^n b^n c^n \mid n \in \mathbb{N}\}$	+	+	+
$\mathcal{R}ec$		–	L_e	+	+	+
$\mathcal{L}_0 = \mathcal{R}e$	DTM = NTM	Typ-0	$H, (G^* \setminus L_d)$	+	+	–
abzählbare Mengen	–	–	L_d, \mathbb{N}			

9. Strukturelle Komplexitätstheorie

Die Komplexitätstheorie ist ein Zweig der Theoretischen Informatik in dem quantitative Aspekte der Berechenbarkeit untersucht werden. Das Ziel ist dabei, eine Einstufung und Einordnung für den Mindestaufwand an Rechenzeit oder Speicherplatz zu erhalten. Dabei sind die unentscheidbaren Probleme in der Komplexitätstheorie aus naheliegenden Gründen uninteressant. Als Modell von realen Maschinen wird dabei zunächst wieder die bekannte Turing-Maschine verwendet. Probleme, die untersucht werden, sind entweder *Optimierungsprobleme* oder reine *Ja-Nein-Entscheidungsprobleme*. Beispiele für die erste Klasse sind Fragen wie „Finde in einem beliebig vorgegebenen kantengewichteten Graphen den kürzesten Weg vom Knoten A zum Knoten B “. Entscheidungsprobleme erlauben grundsätzlich nur zwei Antworten: *Ja* oder *Nein*. Ein Beispiel dafür wäre z.B. die Frage, ob eine gegebene kontextfreie Grammatik ein bestimmtes Wort generieren kann.

Ein Problem ist immer eine allgemeine Fragestellung, die für jede spezielle vorgegebene Problem-Instanz, die sich aus der allgemeinen Fragestellung durch Ersetzen der frei wählbaren Parameter ergibt, beantwortet werden soll. Damit ein spezielles Problem von einer Turing-Maschine bearbeitet werden kann, muß dieses in kodierter Form vorgelegt werden. Dies kann auf die unterschiedlichste Weise geschehen, wir wollen aber hier die gleiche Form der Kodierung benutzen, wie wir sie bei Turing-Maschinen verwendet haben, nämlich durch Darstellung mit ausreichend großem Alphabet, welches eine binäre Kodierung gestattet. Die Länge dieser Darstellung wird dann als die **Größe des speziellen Problems** bezeichnet.

Als Beispiel sei hier das Hamilton-Kreis Problem mit seiner Notation angeführt:

9.1 Definition

Hamilton-Kreis (HC) (Hamilton-Circuit)

Eingabe: Ein ungerichteter Graph $G = (V, E)$

Frage: Gibt es einen geschlossenen Kreis in G , bei dem jeder Knoten genau einmal durchlaufen wird, d.h. existiert eine Folge von Knoten v_1, v_2, \dots, v_n mit $n = |V|$, $V = \{v_1, v_2, \dots, v_n\}$ und $\{v_i, v_{i+1}\} \in E$ und $\{v_n, v_1\} \in E$ für alle $i \in \mathbb{N}$ mit $1 \leq i < n$?

Als Kodierung für einen Graphen $G = (V, E)$ könnte man die Liste der Kanten zusammen mit der der Knoten verwenden, was höchstens eine Länge von $|V| + 3 \cdot |E| - 1 + \lceil \log_k |V| \rceil \cdot (|V| + 2 \cdot |E|)$ ergibt, wenn die Knoten als Zahlen zur Basis k mit Trennsymbolen in der Knotenliste und der Kantenliste als Zeichenkette notiert werden.

G sei der folgende, sehr einfache Graph:

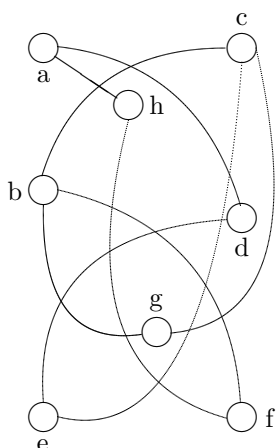


9. Strukturelle Komplexitätstheorie

Für diese Kodierung von G ist $\langle G \rangle = 1, 2, 3, 4\{1, 2\}\{2, 3\}\{3, 4\}$ mit der Länge: $\lceil \log_{10} 4 \rceil \cdot 10 + 12 = 22$.

Eine Darstellung mit den Zeilen der Adjenzmatrix ist stets $|V|^2 + |V| - 1$ Zeichen lang: 0100#1010#0101#0010 hat die Länge 19. Da in ungerichteten Graphen $|E| \leq |V|^2$ gilt, unterscheiden sich diese Größen im schlechtesten Fall nur unwesentlich – d.h. polynomiell – voneinander.

In der Notation für ungerichtete Graphen verwenden wir für die Kantenbezeichnung bei gerichteten Graphen statt der Tupel (v_1, v_2) , stets Teilmengen von V mit höchstens zwei Elementen. $\{a, b\}$ ist also eine Kante zwischen den Knoten a und b . Eine Schleife am Knoten c wird dann mit $\{c\}$ bezeichnet. Ansonsten entspricht die Definition der eines Graphen zu einer Relation gemäß Kapitel 2.



Zu Ja-Nein-Problemen, wie z.B. diesem, gehören immer jene Sprachen, die die Kodierungen von solchen Problem-Instanzen enthalten, für die die Antwort *Ja* gegeben wird. In obigem Beispiel also $L_{HC} := \{\langle G \rangle \mid G \text{ ist Graph mit einem Hamilton-Kreis}\}$. Das spezielle Problem für den gegebenen Graphen G' ist dann die Frage: Gilt $\langle G' \rangle \in L_{HC}$? Üblicherweise werden in solchen Fällen die Klammern \langle und \rangle weggelassen, da diese standardmäßig ergänzt werden können. Wir sehen hier leicht, daß $G' \in L_{HC}$ ist.

Aber ob eine Turing-Maschine dies nicht nur in diesem speziellen Fall, sondern für alle möglichen Problem-Instanzen mit vertretbarem Aufwand lösen kann, ist noch zu untersuchen. Woran wir interessiert sind, ist zunächst ein Algorithmus, der dieses Problem löst, der also die Menge L_{HC} akzeptiert und dabei stets hält, d.h. die Menge L_{HC} entscheidet.

Im Folgenden werden wir die Abkürzungen für die Probleme immer auch als Abkürzungen für die ihnen zugeordneten Sprachen verwenden. Statt L_{HC} schreiben wir also, wie überall üblich, nur HC und benutzen die Abkürzung des Problems immer auch als Bezeichnung für die ihm zugeordnete Sprache. Ist ein Problem entscheidbar, dann soll die dazu verwendete TM möglichst wenig Aufwand treiben, also wenig Schritte machen bzw. mit wenig Feldern auf dem Arbeitsband auskommen. Auch wollen wir dieses Problem mit anderen vergleichen, die in den unterschiedlichsten Anwendungen vorkommen. Die Mengen die alle Sprachen enthalten, die mit dem gleichen Aufwand an Rechenzeit oder Speicherplatz analysiert werden können, bilden natürlich wieder Sprachfamilien, die wir zu diesem Zweck mit denen vergleichen wollen, die uns aus anderer Blickrichtung schon vertraut sind.

9.1. Zeit- und Platzkomplexität

Um die Größe des Aufwands von Berechnungen auf Turing-Maschinen zu definieren, verwenden wir die k -Band off-line TM aus Definition 7.13.

Jede NTM hat für das Akzeptieren eines Wortes i.A. mehrere verschiedene Erfolgsrechnungen – d.h. Folgen von Konfigurationen – zur Verfügung. In der Literatur gibt es auch andere, die bei den meisten

betrachteten Funktionen die gleichen Komplexitätsklassen liefern, aber nicht so allgemein sind. Wir beginnen mit einzelnen Erfolgsrechnungen, dann betrachten wir alle Erfolgsrechnungen für ein festes Wort und sodann die Erfolgsrechnungen für alle Worte einer Länge.

9.2 Definition

In einer fest vorgegebenen Erfolgsrechnung **benötigt** eine NTM soviel **Zeit**, wie darin einzelne Konfigurationen durchlaufen werden.

Die NTM **benötigt** in dieser Erfolgsrechnung soviel **Platz**, wie maximal Felder auf einem der Arbeitsbänder besucht werden.

Für die Platzbeschränkung stelle man sich alle Arbeitsbänder der NTM als Spuren eines einzigen Bandes vor.

9.3 Definition

Eine NTM A akzeptiert $w \in L(A)$ mit der **Zeitbeschränkung** $t \in \mathbb{R}$ genau dann, wenn die kürzeste Erfolgsrechnung für w nur $\lceil t \rceil$ Zeit benötigt bzw. Schritte hat.

Eine NTM A akzeptiert $w \in L(A)$ mit der **Platzbeschränkung** $s \in \mathbb{R}$, wenn die Erfolgsrechnung mit dem geringsten Platzverbrauch für w nur $\lceil s \rceil$ Platz benötigt.

9.4 Definition

Eine NTM A ist $t(n)$ -**zeitbeschränkt** genau dann, wenn $t(n) \geq \max\{m \mid \exists w \in L(A) : n = |w| \text{ und } A \text{ akzeptiert } w \text{ mit der Zeitbeschränkung } m\}$ gilt.

Eine NTM A ist $s(n)$ -**platzbeschränkt** genau dann, wenn $s(n) \geq \max\{m \mid \exists w \in L(A) : n = |w| \text{ und } A \text{ akzeptiert } w \text{ mit der Platzbeschränkung } m\}$ gilt.

Wenn eine TM mit der Platzbeschränkung $s(n)$ arbeitet, dann sollte man davon ausgehen, daß $s(n) \geq 1$ für jedes $n \in \mathbb{N}$ ist, denn jede TM benutzt wenigstens ein Feld zum Ansehen auf dem Arbeitsband, selbst wenn nur $\#$ darauf steht. Wir sagen also, die TM arbeitet mit Platzbeschränkung $f(n)$, wenn sie in Wirklichkeit $\max\{1, f(n)\}$ -platzbeschränkt ist.

Es ist ebenfalls nützlich anzunehmen, daß eine TM ihre Eingabe stets vollständig liest und ein weiteres Feld vorrückt, um deren Ende festzustellen. Also sagen wir, daß eine TM mit Zeitbeschränkung $t(n)$ arbeitet, wenn sie tatsächlich $\max\{n + 1, t(n)\}$ -zeitbeschränkt ist.

Beispiel:

Die Sprache $\{w\$w^{rev} \mid w \in \{a, b\}^*\}$ ist kontextfrei, und kann daher mit dem Verfahren von Cocke/Younger/Kasami (Theorem 4.34) in Polynomzeit analysiert werden. Der Speicherplatz, der dabei benötigt wird, ist entsprechend groß. Diese spezielle Sprache kann aber nichtdeterministisch leicht mit der Platzkomplexität $s(n) = n$ akzeptiert werden, wobei hier n die Länge des jeweiligen Eingabewortes bezeichnet. Wenden wir das Kellerprinzip an, so genügt uns ein Speicherbedarf von $\frac{n+1}{2}$. Es ist aber auch möglich, durch geschickte Codierung nur noch mit $\log(n)$ Platzbedarf auszukommen. Der Präfix w in $w\$w^{rev}$ muß als Binärzahl so kodiert werden, daß jeder einzelne Buchstabe leicht ermittelt werden kann. Die Leser(innen) sind aufgerufen, sich an einem Verfahrensvorschlag zu versuchen.

Auf der Basis von Beschränkungsfunktionen kann man Sprachklassen definieren, wie zum Beispiel $\text{DTIME}(f(n)) := \{L \mid L = L(A) \text{ für eine } f(n)\text{-zeitbeschränkte Turing-Maschine } A\}$. Der Nachteil

9. Strukturelle Komplexitätstheorie

einer solchen Definition *an dieser Stelle* ist offensichtlich: Wir müßten diese Klasse von der unterscheiden, bei der statt $f(n)$ die Funktion $g(n) := c \cdot f(n)$ mit $c \neq 1$ verwendet würde. Daher untersuchen wir zunächst den Einfluß von konstanten Faktoren und Summanden auf Beschränkungsfunktionen. Das erste Resultat verallgemeinert den zweiten Teil des Beweises von Theorem 8.12:

9.5 Theorem

(Bandkompression) Zu jeder $s(n)$ -platzbeschränkten TM und jeder reellen Zahl $c \in \mathbb{R}$ mit $c > 0$ gibt es eine äquivalente TM die $c \cdot s(n)$ -platzbeschränkt ist.

Beweis: Die Konstruktion des $|w|$ -platzbeschränkten LBA B aus einem beliebigen LBA A aus Theorem 8.12 kann für jede TM durchgeführt werden. Wenn A $c \cdot s(n)$ -platzbeschränkt ist, so ist B dann nur noch $s(n)$ -platzbeschränkt. \square

Damit sind dann für die Platzbeschränkung z.B. die Funktionen $f(n) := 3n^2 - 70n + 5$ und $g(n) := n^2$ nicht von einander zu unterscheiden, denn $f(n) \leq 4n^2$ und nach Theorem 9.5 kann die mit $f(n)$ -Platzbeschränkung akzeptierte Sprache auch von einer $g(n)$ -platzbeschränkten TM akzeptiert werden.

Ein fast identisches Ergebnis erhalten wir für Zeitbeschränkungen:

9.6 Theorem

(lineare Beschleunigung, speed-up) Zu jeder $t(n)$ -zeitbeschränkten TM mit $\inf_{n \rightarrow \infty} (\frac{t(n)}{n}) = \infty$ und jedem $c \in \mathbb{R}$ mit $c > 0$ gibt es eine äquivalente $c \cdot t(n)$ -zeitbeschränkte TM.

Beweis: (Genauere Angaben in der Literatur, z.B. [HopcroftUllman])

Wieder werden jeweils r benachbarte Felder des Arbeitsbandes der zu simulierenden TM A zu einem neuen Symbol kodiert. Dies geschieht auch mit der Eingabe, die beim Lesen von dem read-only Eingabeband der Simulations-TM B zusammen mit dieser Blockbildung auf das Arbeitsband kopiert wird. Kopfbewegungen der zu simulierenden TM in diesem Bereich von r benachbarten Symbolen kostet die neue TM B keine einzige Kopfbewegung.

Ein ständiger Wechsel zwischen benachbarten Blöcken jedoch kostet genauso viel Zeit wie in der simulierten TM A . Um das zu sparen, werden mit dem aktuell wichtigen Block (das ist der, in dessen r Symbolen sich die TM A gerade befindet) auch gleich noch seine beiden Nachbarn in der endlichen Kontrolle gespeichert.

Dies geschieht mit vier Bewegungen des LSK von B (links-rechts-rechts-links) immer dann, wenn der nun aktuelle Block ein benachbarter wird, d.h. die Simulations-TM B ihren LSK um ein Feld weiter bewegen muß, will sie immer den aktuellen Block als Symbol kodiert unter dem LSK behalten.

Bewegt A ihren LSK r Schritte in eine Richtung, so wird im schlechtesten Fall der aktuelle Bereich 2-mal geändert, was B 8 Schritte kostet. $t(n)$ Schritte von A werden von B in höchstens $\lceil \frac{8t(n)}{r} \rceil$ Schritten simuliert.

Dazu kommen n Schritte, um die Eingabe zu lesen und in Symbolen von Blöcken aufeinanderfolgender Zeichen zu schreiben, sowie weitere $\lceil \frac{n}{r} \rceil$ Schritte, um den LSK von B auf den Anfang der Folge von kodierten Blöcken zu bewegen. Da stets $\lceil x \rceil < x + 1$ ist, sind insgesamt höchstens $n + 1 + \frac{n}{r} + 1 + 8 \cdot (\frac{t(n)}{r})$ Schritte von B nötig. Da $\inf_{n \rightarrow \infty} (\frac{t(n)}{n}) = \infty$ ist, gibt es für jede Konstante d eine Zahl n_d , so daß für alle

$n \geq n_d$ dann $(\frac{t(n)}{n}) \geq d$ ist. Bis auf endlich viele $n < \max\{n_d, 2\}$ ist dann also die Schrittzahl von B nicht größer als $t(n) (\frac{2}{d} + \frac{1}{dr} + \frac{8}{r})$, falls $n + 2 \leq 2n$, d.h. $n \geq 2$ ist.

Wählen wir nun $r := \lceil \frac{16}{c} \rceil$ und $d := \lceil \frac{4}{c} \rceil + \frac{1}{8}$ dann gilt $r \cdot c \geq 16$ und $d \geq \frac{32+c}{8c}$.

Setzt man diese Werte in obige Formel für die maximale Laufzeit von B ein, so bleibt diese unterhalb von $c \cdot t(n)$ für fast alle n . Um die Worte zu bearbeiten, die kürzer als $\max\{2, n_d\}$ sind braucht B nur die endliche Kontrolle und $n + 1$ Bewegungen um die Eingabe zu lesen. Damit ist B wie gewünscht eine $c \cdot t(n)$ -zeitbeschränkte TM. \square

9.2. Komplexitätsklassen

Nachdem nun klar wurde, daß konstante Faktoren oder additive Glieder bei den Beschränkungsfunktionen keine entscheidende Rolle spielen, können wir uns mit Aussagen über deren asymptotisches Verhalten begnügen und Sprachfamilien auf der Basis der Akzeptierungskomplexität besser definieren. Zur Klassifizierung von Funktionen verwendet man die sogenannten Landau-Schreibweisen, von denen wir hier nur eine wirklich benötigen:

9.7 Definition

Eine Abbildung $f : \mathbb{N} \rightarrow \mathbb{R}$ **wächst mit der Ordnung** $g(n)$ bzw. g , notiert durch $f \in O(g)$, falls $g : \mathbb{N} \rightarrow \mathbb{R}$ eine Abbildung ist und eine Konstante $c \in \mathbb{R}$ existiert, so daß $|f(n)| \leq c \cdot |g(n)|$ für alle, bis auf endlich viele $n \in \mathbb{N}$ gilt.

Es wird ebenfalls notiert:

$$\begin{aligned} f(n) &\in o(g(n)), \text{ falls } \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \\ f(n) &\in \Omega(g(n)), \text{ falls } g(n) \in O(f(n)) \\ f(n) &\in \omega(g(n)), \text{ falls } g(n) \in o(f(n)) \end{aligned}$$

Als Beispiel seien f und g definiert durch: $f(n) := 23n^4 - 7n^3 + 1993$ und $g(n) := 7n^3 - n$. Dann ist $g \in O(f)$ aber nicht $f \in O(g)$. Auch gilt $f \in O(n^4)$ oder $f \in O(442207n^4 - 1948)$. Andere Schreibweisen für $f \in O(g)$ sind auch $f = O(g)$ oder $f \simeq O(g)$.

In der Regel betrachten wir Funktionen, die nicht wie $f(n) := n \sin(n)$ periodisch schwanken, sondern solche, die monoton wachsen. In allen Fällen aber ist die Aussage wichtig, daß f nicht schneller wächst als g .

9.8 Definition

Für Funktionen $s, t : \mathbb{N} \rightarrow \mathbb{R}$ für die stets $t(n) \geq n + 1$ und $s(n) \geq 1$ ist, bezeichne:

$$\begin{aligned} \mathbf{DTIME}(t(n)) &:= \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte DTM } A\} \\ \mathbf{NTIME}(t(n)) &:= \{L \mid L = L(A) \text{ für eine } t(n)\text{-zeitbeschränkte NTM } A\} \\ \mathbf{DSpace}(s(n)) &:= \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte DTM } A\} \\ \mathbf{NSpace}(s(n)) &:= \{L \mid L = L(A) \text{ für eine } s(n)\text{-platzbeschränkte NTM } A\} \\ \mathbf{P} &:= \{L \mid \text{Es gibt ein Polynom } p : \mathbb{N} \rightarrow \mathbb{R} \text{ und eine} \end{aligned}$$

9. Strukturelle Komplexitätstheorie

$$\begin{aligned} \mathbf{NP} &:= \{L \mid \text{Es gibt ein Polynom } p : \mathbb{N} \rightarrow \mathbb{R} \text{ und eine} \\ &\quad p(n)\text{-zeitbeschränkte DTM } A \text{ mit } L = L(A)\} \end{aligned}$$

Damit ist wegen des speed-up Satzes:

$$P = \bigcup_{i \geq 1} DTIME(n^i) \quad \text{und} \quad NP = \bigcup_{i \geq 1} NTIME(n^i)$$

Ähnlich definiert man polynomiale Platz-Klassen:

$$\begin{aligned} PSPACE &:= \bigcup_{i \geq 1} DSPACE(n^i) \\ NPSPACE &:= \bigcup_{i \geq 1} NSPACE(n^i) \end{aligned}$$

9.9 Korollar

Die sich direkt aus den Definitionen ergebenden trivialen Beziehungen zwischen diesen Komplexitätsklassen sind offensichtlich die folgenden:

$$\begin{aligned} DSPACE(f) &\subseteq NSPACE(f) \\ DTIME(f) &\subseteq NTIME(f) \\ P &\subseteq NP \\ PSPACE &\subseteq NPSPACE \end{aligned}$$

Bekannt ist nur, daß alle nicht-deterministisch in Polynomzeit lösbaren Probleme $L \in NP$, stets mit exponentiellem Zeitaufwand deterministisch gelöst werden können. Bessere Aussagen sind nicht bekannt und die Beantwortung der Frage ob die Sprachklassen P und NP verschieden sind, ist – nicht nur in der Informatik – durchaus bedeutsam.

Viele der *praktisch wichtigen* Fragestellungen haben Lösungen mit Algorithmen, die nur dann in Polynomzeit arbeiten, wenn sie nicht-deterministisch sind. Für eine Implementierung sind aber stets deterministische Verfahren nötig.

In der Klasse NP sind natürlich auch solche Sprachen enthalten, die in sehr kurzer Zeit, z.B. in Linearzeit, erkannt werden können, also in $NTIME(n)$ liegen, wie auch andere, für die in ihrer bisher besten deterministischen Implementation bestenfalls exponentielle Zeit verbraucht wird.

Wir sehen uns Beispiele von ähnlich klingenden aber dennoch unterschiedlich lösbaren Problemen an:

P		NP
<p>Gegeben:</p> <p>Ungerichteter, kantenbewerteter Graph $G = (V, E)$, eine Gewichtsfunktion $g : E \rightarrow \mathbb{N}$, zwei Knoten $a, b \in V$ und eine Schranke $k \in \mathbb{N}$</p>		
<p>Frage:</p> <p>Gibt es einen einfachen Pfad p von a nach b mit $g(p) \leq k$?</p> <p>„Kürzester Weg zwischen zwei Knoten“</p>	<p>Frage:</p> <p>Gibt es einen einfachen Pfad p von a nach b mit $g(p) \geq k$?</p> <p>„Längster Weg zwischen zwei Knoten“</p>	

P		NP
Gegeben: Matrix $A \in \mathbb{Z}^{m \times n}$ und ein Vektor $b \in \mathbb{Z}^m$		
Frage: Gibt es $x \in \mathbb{Z}^n$ mit $Ax = b$	Frage: Gibt es $x \in \mathbb{N}^n$ mit $Ax = b$	

P		NP
Gegeben: Ungerichteter, kreisfreier Graph $G = (V, E)$		
Frage: Gibt es einen geschlossenen Kreis, in dem jede Kante genau einmal auftritt? „Euler-Kreis“	Frage: Gibt es einen geschlossenen Kreis, in dem jeder Knoten genau einmal auftritt? „Hamilton-Kreis“	

Wir kennen für alle der oben angegebenen Beispiele Turing-Maschinen, die zum Erkennen der dem jeweiligen Problem zugeordneten Sprachen nichtdeterministisch in Polynomzeit arbeiten, aber für obige Probleme aus der Familie NP kennt man bisher nichts Besseres. Alle deterministischen Verfahren arbeiten nicht polynomiell.

9.10 Definition

Sei $\mathbf{LW} := \{ \langle G \rangle \mid G \text{ besitzt einen Pfad } p \text{ von } a \text{ nach } b \text{ mit } g(p) \geq k \}$ die dem Problem „Längster Weg zwischen zwei Knoten“ zugeordnete Sprache.

9. Strukturelle Komplexitätstheorie

Und $\mathbf{KW} := \{ \langle G \rangle \mid G \text{ besitzt einen Pfad } p \text{ von } a \text{ nach } b \text{ mit } g(p) \leq k \}$ sei die Sprache, die zu dem Problem „Kürzester Weg zwischen zwei Knoten“ gehört.

Wie sieht die NTM zur Erkennung der „Längster Weg zwischen zwei Knoten“-Sprache LW mit polynomieller Zeitbeschränkung nun eigentlich aus?

Zu jedem kodierten Graphen der Eingabe schreibt die NTM eine beliebige Kantenfolge p von a nach b auf ihr Band, indem sie die Eingabe oft genug nach passenden Paaren $\{v_1, v_2\}$ und $\{v_3, v_4\}$ mit $v_2 = v_3$ durchsucht. Da jeder Knoten bei einem einfachen Pfad nur einmal durchlaufen werden darf, kann die NTM die ausgewählten Paare markieren, und muß so höchstens $|E|$ -mal die Eingabe durchlaufen. Dieser Zeitaufwand $t(n)$ ist dann von der Ordnung $O(n^2)$. Die Prüfung, ob $g(p) \geq k$ ist, kann durch Addieren der in der Liste für g notierten Gewichte in etwa der gleichen Zeit geschehen. Das selbe Verfahren können wir anwenden um zu zeigen, daß das Problem KW des kürzesten Weges ebenfalls in polynomieller Zeit gelöst werden kann.

Damit gilt nun für beide Probleme $\text{LW} \in \text{NP}$ und $\text{KW} \in \text{NP}$.

Die Frage ist nun, ob es nicht in beiden Fällen möglich ist, eine deterministische Turing-Maschine zu finden, die die gleiche Sprache erkennt und dabei in Polynomzeit arbeitet.

Für KW gelingt dies tatsächlich:

9.11 Theorem

Das Problem „kürzester einfacher Weg von a nach b “ ist deterministisch in Polynomzeit zu lösen, d.h. $\text{KW} \in P$.

Beweis-Skizze: Statt die DTM in allen Details anzugeben, skizzieren wir das Verfahren hier nur und geben eine Begründung dafür, daß nur polynomielle Zeit verbraucht wird:

Zuerst konstruiert die DTM aus der kodierten Eingabe $\langle G \rangle$ zu $G = (V, E)$, mit a und b als ausgezeichnete Knoten sowie der Angabe von $g(e)$ für jede Kante $e \in E$, eine Matrix $W \in \mathbb{N}^{n \times m}$, für $n := |E|$ und $m := |V|$. In der Matrix W gibt das Element $W_{i,j}$ in der i -ten Zeile und j -ten Spalte die Länge des kürzesten, einfachen Weges von $a := v_1$ zum Knoten v_j mit bis zu i Kanten an. Der zu erreichende Ziel-Knoten b wird durch passende Numerierung der Knoten zu v_r mit $r := |V|$. Nun wird $W_{i,j}$ zeilenweise durch Erhöhen des Index i von den Anfangswerten $W_{1,j} := \min(g(e) \mid e = \{v_1, v_j\} \in E \text{ oder } \omega, \text{ falls } e \notin E)$ bestimmt.

Wir definieren dazu: $\forall n \in \mathbb{N} : \omega > n$. Dann wird $W_{i+1,j}$ induktiv bestimmt durch:

$$W_{i+1,j} := \min_{1 \leq k \leq |V|} \{W_{1,j}, W_{i,k} + g(\{v_k, v_j\})\}$$

Da die Zeile $W_{i,j}$ für $1 \leq j \leq |V|$ bekannt ist und nur beschränkt viele, jedoch höchstens $|E|$ Kanten aus jedem Knoten heraus führen, berechnet sich der Aufwand zu $O(|V|^2 \cdot |E|)$ Schritten. Zu berücksichtigen ist $\min\{n, \omega\} := n$ sowie $n + \omega = \omega$ sowie der Aufwand für die Vergleiche und Addition.

Für das Problem des kürzesten Weges von einem Knoten zu einem anderen in einem Graphen (gerichtet oder nicht) gibt es viele andere Algorithmen. Man erinnere sich an das Verfahren von Kleene, das benutzt wird, um aus einem endlichen Automaten den äquivalenten Rationalen Ausdruck zu erzeugen. Auch dies Verfahren läßt sich so abwandeln, daß anstelle der Ausdrücke nur die minimalen Wege notiert werden.

Wir wollen hier nicht näher darauf eingehen und empfehlen die in dieser Hinsicht sehr ausführliche Literatur (z.B. [Mehlhorn]). \square

Aufgabe 9.1:

Schreiben Sie einen Algorithmus, der das „Kürzester Weg von a nach b “-Problem löst, und dabei nicht nur die bloße Existenz eines solchen herausfindet, sondern auch noch diesen Weg durch Angabe der darin vorkommenden Kanten spezifiziert. Benutze eine Notation, die leicht in reale Programme übertragen werden kann und bestimme die Zeitbeschränkung dieses speziellen Algorithmus.

Könnte man diesen Algorithmus auf einfache Weise dahingehend verallgemeinern, daß zwischen zwei Knoten im Graphen mehrere Kanten mit unterschiedlichen Gewichten vorkommen dürfen?

Wenn wir nach einem Algorithmus für das „Längster Weg von a nach b “-Problem suchen, so werden wir in der Regel nicht so erfolgreich sein.

Natürlich stellt sich sofort die Frage, ob das nun an unserem *Können* liegt oder vielleicht *in dem Problem selbst* begründet ist?

9.3. Problemreduktionen und vollständige Probleme

Ähnlich wie bei der Reduktion eines bisher unbekannten Entscheidungs-Problems A auf ein anderes – nennen wir es B , dessen Entscheidbarkeit uns bekannt ist, so gibt es auch Reduktionen der Sprachen einer Komplexitätsklasse auf die einer anderen.

In beiden Fällen geschieht das durch eine Abbildung $f : X^* \rightarrow Y^*$ für die $f(w) \in L_B$ für ein beliebiges Wort $w \in X^*$ genau dann gilt, wenn $w \in L_A$ ist. Wird nur nach der Entscheidbarkeit eines Problems gefragt, so reicht es, wenn f berechenbar ist.

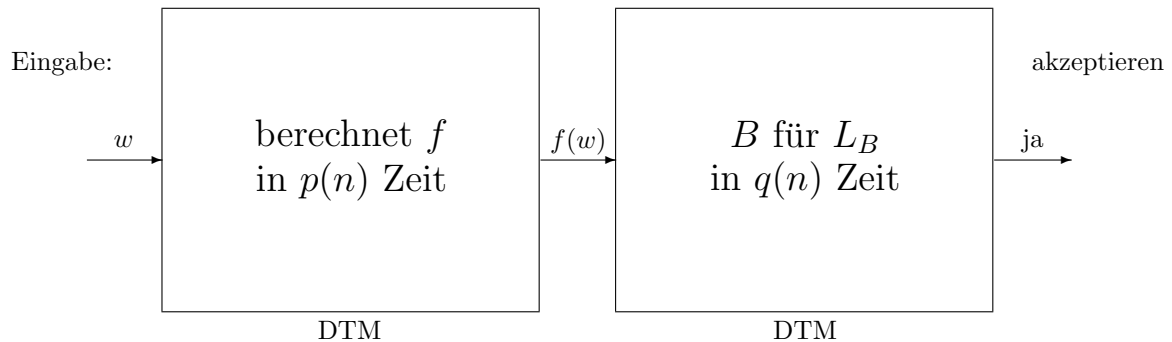
Wollen wir erreichen, daß die Berechnungskomplexität des Problems bei solch einer Reduktion erhalten bleibt, so müssen wir sicher stellen, daß diese Funktion f mit wenig Speicherplatz oder kurzer Zeit berechnet werden kann.

Dies führte zuerst zum Begriff der Reduktion, die in polynomieller Zeit durchgeführt werden kann.

9.12 Definition

Ein Problem A ist **polynomiell reduzierbar** auf das Problem B , notiert als $A \leq_{pol} B$ genau dann, wenn es eine Polynomzeit beschränkte deterministische Turing-Maschine gibt, die für die zu den Problemen gehörenden Sprachen $L_A \subseteq X^*$ und $L_B \subseteq Y^*$, eine Funktion $f : X^* \rightarrow Y^*$ berechnet mit $w \in L_A$ gdw. $f(w) \in L_B$.

Gilt $A \leq_{pol} B$, so kann aus jedem Verfahren für das Problem B eines für das Problem A gewonnen werden, welches dann deterministisch in Polynomzeit durchführbar ist, wenn dies für das Problem B galt.



Die Zeit, die nun für die Akzeptierung der Sprache L_A benötigt wird, berechnet sich zu $p(n) + q(f(w))$, wenn $n := |w|$ für $w \in L_A$ ist, und dies ist wieder ein Polynom, wenn q ein solches war.

Es gibt Sprachen, die für die Klasse NP typisch sind, weil entweder *alle* oder *keine* davon auch deterministisch in Polynomzeit zu erkennen sind. Diese Sprachen heißen *NP-vollständig* und sind alle paarweise aufeinander in Polynomzeit reduzierbar.

9.13 Definition

Eine Sprache L heißt **NP-vollständig** genau dann, wenn sie selbst in NP liegt und jede beliebige Sprache M aus NP sich auf L polynomiell reduzieren läßt, d.h. wenn gilt:

$$L \in NP \text{ und } \forall M \in NP : M \leq_{pol} L$$

Aus der Definition und dem oben Gesagten über die gesamte Dauer der Rechnung aus Reduktion gefolgt von Erkennung mit der Turing-Maschine für eine NP-vollständige Sprache, folgt leicht das folgende Korollar:

9.14 Korollar

Sei L NP-vollständig, dann gilt $L \in P$ gdw. $P = NP$.

Die NP-vollständigen Sprachen, bzw. die dazu gehörenden Probleme, werden zu recht als schwierig angesehen, kennt man bisher doch stets nur deterministische Verfahren, die in Exponentialzeit arbeiten. Das Problem „ $P = NP?$ “ ist seit seiner Formulierung durch S. Cook (1971) ungelöst. Die meisten Wissenschaftler vermuten die Ungleichheit dieser Klassen und vielleicht kann es dafür sogar mit unseren Mitteln niemals einen formalen Beweis geben.

Selbst wenn wir nicht sagen können, daß NP-vollständige Probleme beweisbar schwierig sind, so ist die Evidenz dafür deutlich genug. Mit den zur Zeit zur Verfügung stehenden Mitteln ist bislang keine Verbesserung möglich oder in Sichtweite.

Das erste Problem, das als NP-vollständig erkannt wurde, ist das *Erfüllbarkeitsproblem* (abgekürzt SAT für *satisfiability*).

9.15 Definition

$SAT := \{w \in X^* \mid w \text{ ist ein erfüllbarer Boolescher Ausdruck}\}$ ist bzw. beschreibt das **Erfüllbarkeitsproblem**, das wie folgt definiert ist:

Erfüllbarkeitsproblem:

Eingabe: Eine Menge V von Variablen und eine Boolesche Formel $B \in X^*$ darüber mit den üblichen Operatoren \vee, \wedge, \neg sowie Klammern.

Frage: Gibt es eine Belegung der Variablen mit $TRUE$ und $FALSE$ derart, daß der Wahrheitswert von B $TRUE$ wird?

9.16 Theorem

SAT ist NP-vollständig.

Beweis-Idee: (Den exakten Beweis entnehme man [GareyJohnson] oder [Mehlhorn])

Es ist einfach zu sehen, daß $SAT \in NP$ ist. Dazu wird für eine nichtdeterministisch beliebig ausgewählte Variablenbelegung geprüft, ob der Wahrheitswert einer Booleschen Formel $B \in SAT$ durch Auswerten den Wert $TRUE$ bekommt. Dies ist in Polynomzeit möglich.

Um zu zeigen, daß *jedes* Problem aus NP deterministisch in Polynomzeit auf SAT reduziert werden kann, werden zu jeder polynomzeitbeschränkten Turing-Maschine Boolesche Formeln aufgestellt, deren Konjunktion genau dann wahr ist, wenn die TM eine Erfolgsrechnung besitzt. Für jedes Polynom p wird es eine Reduktions-DTM geben, die dies leistet.

Sei $L \in NP$ beliebig und $A_L := (Z, X, Y, K, q_0, Z_{end})$ eine $p(n)$ -zeitbeschränkte NTM mit $L := L(A_L)$, wobei p ein Polynom ist. A_L hat in jeder ihrer Erfolgsrechnungen wegen der Zeitbeschränkung bei Eingabe von w mit $|w| = n$ höchstens $p(n)$ verschiedene Konfigurationen k_i , wobei die Länge $|k_i|$ einer jeden höchstens $p(n) + 1$ ist. In jedem der maximal $p(n)$ Schritte kann sich der LSK von A vom ersten bis zum $(p(n) + 1)$ -ten Feld bewegen und sich jeweils nur an genau einer von $p(n)$ verschiedenen Positionen befinden. Wir können o.B.d.A. annehmen, daß sich bei vorzeitigem Akzeptieren die Konfiguration in allen weiteren Schritten nicht mehr verändert und die aktuelle Bandinschrift stets alle $p(n)$ Felder beschriftet hat, notfalls mit aufgefüllten $\#$.

Nun können wir Bool'sche Variablen definieren, mit deren Hilfe man eine Formel F_L konstruiert, so daß „ $w \in L$ “ gdw. „Es gibt eine Erfolgsrechnung für w mit $p(n)$ Schritten“ gdw. „Es gibt eine F_L erfüllende Variablenbelegung“.

Die Variablen mit Ihrer Bedeutung sind:

$FELD(i, j, t) \triangleq$ In Konfiguration k_t steht das Zeichen x_j in Feld i .

$ZUSTAND(r, t) \triangleq$ In der Konfiguration k_t befindet sich die TM A im Zustand z_r .

$KOPF(i, t) \triangleq$ In der Konfiguration k_t steht der LSK auf dem Feld i .

Die Anzahl dieser Variablen berechnet sich zu einer Größe der Ordnung $O(p(n))$. Wir definieren die Formel $\oplus(a_1, a_2, \dots, a_r) := (a_1 \vee a_2 \vee \dots \vee a_r) \bigwedge_{i \neq j} (\neg a_i \vee \neg a_j)$ mit der Bedeutung: $\oplus(a_1, a_2, \dots, a_r) = 1$ gdw. genau ein $a_i = 1$. Die Länge dieser Formel ist von der Ordnung $O(r^2) = O(p^2(n))$.

Die Formel F_L hat die Form $F_L := A \wedge B \wedge C \wedge D \wedge E \wedge F \wedge G$ mit $A := A_1 \wedge A_2 \wedge \dots \wedge A_{p(n)}$.

Die einzelnen Teilformeln bedeuten:

$A_t \triangleq$ In der Konfiguration k_t steht der LSK von A_L auf genau einem Feld.

$B \triangleq$ In jeder Konfiguration k_t enthält jedes Feld genau ein Zeichen.

9. Strukturelle Komplexitätstheorie

$C \triangleq$ In jeder Konfiguration k_t befindet sich A_L in genau einem Zustand.

$D \triangleq$ Bei jedem Übergang wird genau das Feld verändert, auf das der LSK zeigt.

$E \triangleq$ Jeder Übergang im Zustand z_k mit x_j unter dem LSK entspricht der Turing-Tafel.

$F \triangleq$ Die erste Konfiguration ist $k_0 = z_1 w \# \dots \#$.

$G \triangleq$ Der Zustand in der letzten Konfiguration $k_{p(n)}$ ist Endzustand aus Z_{end} .

Als Beispiel für die Teilformeln und deren Größe geben wir hier nur die ersten an, die anderen werden auf die selbe Art gebildet und sind alle in $O(p^3(n))$ Zeit konstruierbar. Die Details entnehme man bitte der Literatur.

$$\forall t \leq p(n) \text{ sei } A_t := \oplus(\text{KOPF}(1, t), \text{KOPF}(2, t), \dots, \text{KOPF}(p(n), t))$$

Auch B und C sind zusammengesetzte Formeln:

$$\begin{aligned} B &:= \bigwedge_{1 \leq i, t \leq p(n)} B(i, t) \text{ mit} \\ B(i, t) &:= \oplus(\text{FELD}(i, 1, t), \dots, \text{FELD}(i, m, t)), \quad m := |Y| \\ C &:= \bigwedge_{1 \leq t \leq p(n)} C_t \text{ mit} \\ C_t &:= \oplus(\text{ZUSTAND}(1, T), \dots, \text{ZUSTAND}(s, t)), \quad s := |Z| \end{aligned}$$

An diesen Formeln kann man beispielhaft sehen, wie jedes einzelne Detail der Erfolgsrechnungen ausgedrückt werden kann. \square

Aufgabe 9.1:

1. Skizzieren Sie eine deterministische Turingmaschine, die für eine beliebige Boolesche Formel $B \in \{\vee, \wedge, \neg, (,), 0, 1\}^*$ mit einer vorgegebenen Variablenbelegung mit 0 für FALSE und 1 für TRUE die feststellt, ob B zu 1 ausgewertet wird oder nicht und dieses deterministisch in Polynomzeit erledigt. Z.B. wäre ein Formel der Art $(\neg(\neg 0 \wedge 1) \vee (\neg(1 \wedge (1 \vee \neg 1)) \wedge 1))$ zu 0 auszuwerten.
2. Mit Hilfe der in 1. konstruierten DTM als Modul entwickle daraus eine NTM für SAT.
3. Bisher hatte die Formel B in 1. keine Variablen. Nun sollen einzelne Variablensymbole x_i benutzt werden und in der Kodierung von B die Form $x\alpha$ haben, wobei $\alpha \in \{0, 1\}^*$ die Binärdarstellung der Nummer i der Variablen x_i ist.

Wieso ist die Länge (d.h. die Anzahl der verwendeten Symbole) der kodierten Form $\langle B \rangle$ der Formel B nun polynomiell in der Ordnung der Länge der unkodierten Formel? Wie lang ist $\langle B \rangle$ maximal, wenn B selbst n Zeichen hatte?

Wenn wir einem neuen Problem begegnen dessen Komplexität wir nicht kennen, so hilft – oft noch vor der Suche nach einem effizienten Verfahren – die Reduktion auf ein schon bekanntes, so daß dann dessen einfache Methode mitverwendet werden kann. Oder man findet eine Reduktion von einem schwerem, z.B. NP-vollständigen Problem, auf das eigene. Im letzteren Fall kann man nicht mehr mit einem in allen Fällen optimalen Algorithmus rechnen.

Dies kann zum Beispiel mit der Variante KNF des Problems SAT durchgeführt werden, bei der die Booleschen Formeln alle auf konjunktive Normalform eingeschränkt sind.

9.17 Definition

Das Erfüllbarkeitsproblem Boolescher Formeln in konjunktiver Normalform wird durch die Sprache $KNF \not\subseteq SAT$ gegeben, wobei:

$$KNF := \{w \in X^* \mid w \text{ ist eine erfüllbare Boolesche Formel in konjunktiver Normalform}\}$$

9.18 Theorem

KNF ist NP-vollständig.

Beweis: Beweisskizze. KNF ist in NP, denn wir können in Polynomzeit feststellen, ob eine Formel in KNF vorliegt und SAT schon in NP ist.

Für die NP-Vollständigkeit bleibt zu zeigen, dass eine Reduktion von SAT auf KNF möglich ist.

(1. Versuch). Sei F eine Formel. Wir konstruieren zu F eine äquivalente KNF H :

1. Forme F so zu G um, das in G die Negationen nur noch vor den Atomen vorkommen:

$$\neg(X \wedge Y) \equiv \neg X \vee \neg Y \quad \neg(X \vee Y) \equiv \neg X \wedge \neg Y \quad \neg\neg X \equiv X$$

2. Forme G in eine KNF H um:

$$X \vee (Y \wedge Z) \equiv (X \vee Y) \wedge (X \vee Z) \quad (X \wedge Y) \vee Z \equiv (X \vee Z) \wedge (Y \vee Z)$$

Leider klappt das so nicht, denn die Umformung von G nach H geschieht i.a. *nicht* in Polynomzeit. Bsp.: $(A_1 \wedge B_1) \vee \dots \vee (A_n \wedge B_n) \equiv (A_1 \vee A_2 \vee \dots \vee A_n) \wedge \dots \wedge (B_1 \vee B_2 \vee \dots \vee B_n)$. Also: 2^n Klauseln. Diese können in Polynomzeit nicht einmal hingeschrieben werden.

(2. Versuch). Sei F eine Formel. Konstruiere zu F eine KNF H mit der Eigenschaft:

$$F \text{ erfüllbar} \quad \text{gdw.} \quad H \text{ erfüllbar}$$

Es wird also nicht mehr die Äquivalenz der KNF angestrebt, sondern lediglich die schwächere Erfüllbarkeitsäquivalenz.

1. Forme F – wie zuvor – zu äquivalenter Formel G um, indem die Negationen nach innen getrieben werden.
2. Forme F -wie zuvor- zu äquivalenter Formel G um. Konstruiere die KNF $H = H(G)$ rekursiv aus G :

- $G = A$: Setze $H(G) = A$
- $G = (G_1 \wedge G_2)$: Setze $H(G) = H(G_1) \wedge H(G_2)$.
- $G = (G_1 \vee G_2)$: (Dieser Fall ist der trickreiche.) Sei $H(G_1) = (K_1 \wedge \dots \wedge K_m)$ und $H(G_2) = (L_1 \wedge \dots \wedge L_n)$. Wähle ein frisches Atom A und setze: $H(G) = (A \vee K_1) \wedge \dots \wedge (A \vee K_m) \wedge (\neg A \vee L_1) \wedge \dots \wedge (\neg A \vee L_n)$.

Es gilt: $H(G)$ ist in Polynomzeit konstruierbar und ist genau dann erfüllbar, wenn F dies ist. (Letzteres ist dem Leser als Übung empfohlen.) \square

Wir können die syntaktische Struktur der Formeln noch weiter einschränken.

9.19 Definition

Die Sprache $3\text{-SAT} \subsetneq \text{KNF} \subsetneq \text{SAT}$ ist gegeben durch **3-SAT** := $\{w \in X^* \mid w \text{ ist erfüllbarer Boolescher Ausdruck in konjunktive Normalform mit genau 3 Literalen in jeder Klausel}\}$.

9.20 Theorem

3-SAT ist NP-vollständig.

BeweisSkizze: 3-SAT ist in NP, das SAT schon in NP ist und wir in Polynomzeit leicht feststellen können, ob eine Formel eine 3-KNF ist. Unser Beweisziel ist daher: Reduktion von KNF auf 3-SAT .

Sei F eine Formel in KNF. Konstruiere zu F eine äquivalente Formel in 3-KNF .

Sei $F = (K_1 \wedge \dots \wedge K_m)$ die KNF. Fallunterscheidung in Bezug auf die Anzahl der Literale in K_i :

- $K_i = A$: Wähle zwei neue Atome B und C .
Setze $L_i := (A \vee B \vee C) \wedge (A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (A \vee \neg B \vee \neg C)$.
- $K_i = (A \vee B)$: Wähle ein neues Atom C .
Setze $L_i := (A \vee B \vee C) \wedge (A \vee B \vee \neg C)$.
- $K_i = (A \vee B \vee C)$: Setze $L_i := K_i$.
- $K_i = (A_1 \vee \dots \vee A_n)$ mit $n > 3$:
Setze $L_i := (A_1 \vee A_2 \vee B_1) \wedge (A_3 \vee \neg B_1 \vee B_2) \wedge (A_4 \vee \neg B_2 \vee B_3) \wedge \dots \wedge (A_{n-1} \vee A_n \vee B_{n-3})$ für neue Atome B_1, \dots, B_{n-3} .

Dann ist $G = (L_1 \wedge \dots \wedge L_m)$ äquivalent zu F (Übung) und in 3-SAT . □

Das Problem HC wurde ganz zu Anfang von Kapitel 9 definiert. Die Frage nach der Existenz eines Hamilton-Kreises ist (im Unterschied zum Euler-Kreis) sicher dann schwerer ist, wenn $P \neq NP$ ist, denn HC ist NP-vollständig.

9.21 Theorem

Das Hamilton-Kreis Problem HC ist NP-vollständig.

Beweis(: Skizze) Dass HC in NP liegt, ist klar: Wir raten einen Kreis und testen, ob alle Knoten genau einmal enthalten sind.

Wir zeigen NP-Vollständigkeit per Reduktion von 3-SAT auf HC.

Wir konstruieren dazu zu jeder Formel F einen Graphen $G(F)$:

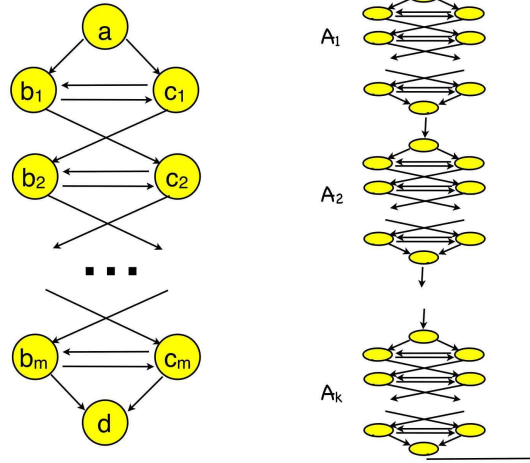
$$F \text{ erfüllbar} \quad gdw. \quad G(F) \text{ besitzt einen HC}$$

Sei $F = (A_1 \vee B_1 \vee C_1) \wedge \dots \wedge (A_n \vee B_n \vee C_n)$.

Wir erzeugen uns mehrere „Bausteine“:

- Zu jedem vorkommenden Literal A_i erzeugen wir einen Teilgraph.
- Zu jeder Klausel $K_i = (A_i \vee B_i \vee C_i)$ erzeugen wir einen Teilgraph.

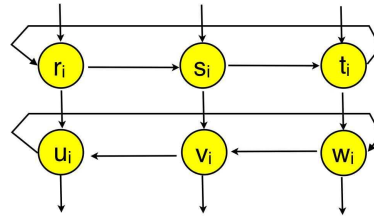
1. Baustein: Sei m die Anzahl des Auftretens eines festen Literals A in F . Zu A konstruieren wir den folgenden Baustein (Abb. links):



Von a nach d existieren nur zwei Pfade, die alle Knoten besuchen. Die Wahl der Kante in Knoten a legt den Pfad bis zum Ende fest. Idee: Der eine Pfad kodiert die Belegung „wahr“, der andere „falsch“ für A .

Wir erstellen für jedes Literal A_i der Formel F einen solchen Teilgraph und verknüpfen diese zu einem Kreis (dargestellt in der Abbildung, rechts). Betrachten wir einen Hamilton-Kreis dieses Graphen: Dieser muss oben ein- und unten austreten. Für jeden A_i -Teilgraph haben wir 2 Möglichkeiten. Bei k Atomen haben wir damit 2^k mögliche Pfade. Jeder Hamilton-Kreis kodiert somit genau eine Belegung der Formel.

2. Baustein: Für jede Klausel $K_i = (A_i \vee B_i \vee C_i)$ erzeugen wir den Teilgraph:



Wenn der Teilgraph in einem Kreis vorkommt, dann wird er oben betreten und unten verlassen, d.h. es gibt genau drei Wege: r_i/u_i , s_i/v_i und t_i/w_i .

Um $G(F)$ zu erhalten, verbinden wir jetzt die beiden Bausteine:

- Wenn das Literal A_i unnegiert auftritt, dann ziehen wir eine Kante von dem ersten „freien“ b -Knoten aus dem A_i -Graph zu r_i und von u_i zu dem c -Knoten, der diagonal unter dem b -Knoten liegt.
- Umgekehrt: Wenn das Atom A_i negiert auftritt, dann ziehen wir eine Kante von dem ersten „freien“ c -Knoten zu r_i und von u_i zu dem b -Knoten, der diagonal unter dem c -Knoten liegt.

Analog für die Literale B_i und C_i .

Durch diese Konstruktion verknüpfen wir die Belegung mit der Formel. Es gilt nun: Der Graph $G(F)$ hat einen HC gdw. F erfüllbar ist. (Beweis als Übung)

9. Strukturelle Komplexitätstheorie

Wir halten noch fest, dass die Konstruktion in Polynomzeit möglich ist. Damit ist der Reduktionsbeweis abgeschlossen. \square

Wir hatten vorher gesehen, daß das Problem LW des „längsten Weges zwischen zwei Knoten“ zwar in Polynomzeit aber dies bisher nur nicht-deterministisch lösbar ist. Wie schon angedeutet, ist dieses Problem tatsächlich NP-vollständig, was eine leichte Reduktion von einem anderen NP-vollständigen Problem auf LW zeigen wird. Das Problem, das wir dazu benutzen ist die Frage nach der Existenz eines Hamilton-Kreises.

9.22 Theorem

Das Problem LW („Längster Weg zwischen zwei Knoten“) ist NP-vollständig.

Beweis: Wir benutzen eine Reduktion für $HC \leq_{pol} LW$, denn $HC \in NP$ ist mit Theorem 9.21 bekannt. Sei G ein Graph, dessen Kantenbewertungen alle gleich sind und den Wert 1 haben, dann ist jeder Hamilton-Kreis ein einfacher Pfad vom Knoten v_0 zum gleichen Knoten v_0 zurück, dessen Länge mit $|V|$ maximal in G ist. Umgekehrt ist jeder einfache Pfad der Mindestlänge $|V|$ immer auch ein Hamilton-Kreis. Also löst die Frage nach der Existenz solch eines einfachen Pfades immer auch das Problem HC. \square

NP-vollständige Probleme gibt es in allen Bereichen, einige weitere seien hier zusätzlich angegeben:

9.23 Theorem

Folgende Probleme sind NP-vollständig:

Partition (partition)

Eingabe: Eine endliche Menge A und eine Funktion $g : A \rightarrow \mathbb{N}$

Frage: Gibt es eine Teilmenge $B \subsetneq A$ derart, daß

$$\sum_{a \in B} g(a) = \sum_{a \in A \setminus B} g(a)$$

Clique (clique)

Eingabe: Ungerichteter Graph $G := (V, E)$ und $k \in \mathbb{N}$ mit $k \leq |V|$

Frage: Enthält G eine k -Clique, d.h. eine Teilmenge V_K von V derart, daß $V_K \times V_K \setminus Id_V \subseteq E$, mit k Knoten?

Handlungsreisender (traveling salesman)

Eingabe: Ein ungerichteter Graph $G := (V, E)$, eine Gewichtsfunktion $g : E \rightarrow \mathbb{N}$ und eine Beschränkung $k \in \mathbb{N}$

Frage: Gibt es einen Hamilton-Kreis p in G mit $g(p) \leq k$?

Wenn wir ein Problem in die Komplexitätsklasse P einordnen können, so meistens deshalb, weil wir einen konkreten Algorithmus angeben können, der es deterministisch „relativ schnell“ löst. Solche Einordnungen sind uns leider nicht immer effektiv möglich. Auf jeden Fall aber wissen wir, daß dies Problem dann nicht so schwierig zu lösen ist wie viele der wichtigen Fragestellungen.

9.3. Problemreduktionen und vollständige Probleme

Was wir bisher nicht untersucht haben ist der Zusammenhang zwischen Zeit- und Platz-Komplexitätsklassen sowie zu den bekannten Sprachklassen der Chomsky-Hierarchie.

Oft werden die Zeit-Komplexitätsklassen als besonders bedeutungsvoll angesehen, aber auch nach dem Platzbedarf können viele Probleme eingestuft und verglichen werden. Für die Klasse PSPACE aller Probleme, die mit polynomielltem Platzbedarf gelöst werden können, z.B., existieren sogar vollständige Sprachen. Möchte man allerdings Reduktionen für diesen Vollständigkeitsbegriff verwenden, so muß sichergestellt werden, daß diese auch mit der erforderlichen Platzbeschränkung arbeiten. Eine Reduktion mit polynomieller Zeitbeschränkung ist dazu in der Regel nicht mehr zu verwenden, da diese viel zu viel Platz benutzen könnte.

Das gilt zwar nicht für die Klasse PSPACE, jedoch bei Platzklassen mit geringerer Platzbeschränkung würde dies leicht auftreten. Man definierte daher schon frühzeitig Reduktionen, die mit wenig (genauer: logarithmischem) Platzbedarf auskommen müssen.

Für diesen Teil der Komplexitätstheorie, und die offen stehenden Fragen, verweisen wir auf die Literatur und die weiterführenden Vorlesungen im Hauptstudium.

A. Historischer Ursprung

Die Bezeichnung „Algorithmus“ für ein nach festen Regeln ablaufendes Verfahren, geht auf den persischen Mathematiker Ibn Musa Al Chwarizmi (al-Khowarizmi) zurück, der etwa um 820 n. Chr. das Buch *Kitab al jabr w'almuqabala* („Regeln der Wiedereinsetzung und Reduktion“) über die Behandlung algebraischer Gleichungen schrieb. Um 1300 suchte der katalanische Mystiker, Dichter und Missionar Ramón Llull (sprich: „Jui“, bekannt als Raimundus Lullus, 1232–1316) nach einer allgemeinen Methode zum Auffinden aller „Wahrheiten“ auf kombinatorischer Grundlage. Mathematisch von geringer Bedeutung, beeinflusste sein Werk, die *ars magna*, viele spätere Mathematiker. Unter anderen den italienischen Philosophen, Arzt und Mathematiker Geronimo Cardano (1501–1576), René Descartes (Renatus Cartesius, 1596–1650) und Gottfried Wilhelm Leibnitz (1646–1716).

Leibnitz kann man, wegen seiner Überlegungen, Ideen und Beiträge, als den eigentlichen Vater der (Theoretischen) Informatik bezeichnen, obwohl seine Schriften keinen unmittelbaren Einfluss auf die Entwicklung der mathematischen Logik hatte. In seiner Schrift: *Explication de l'Arithme'tique Binaire* beschreibt er das Rechnen im dualen Zahlensystem und hatte wohl als erster die Idee zu einer Rechenmaschine auf der Basis des Dualsystems. Beeinflusst von Lullus, drückt Leibnitz in dem Werk *De scientia universalis seu calculo philosophico* seine Ideen zur symbolischen Darstellung von Begriffsinhalten aus. Er vermutete, dass eines Tages jedes wissenschaftliche Streitgespräch von Philosophen durch „Ausrechnen“ entschieden werden könne.

Weniger philosophisch/mathematisch orientierte Manipulationen mit Symbolen traten schon früh bei dem Gebrauch von Geheimschriften auf. Die einfachen Transpositionen von Julius Cäsar sind, vom Standpunkt der heutigen Kryptologie, kaum ernstzunehmen. Das erste erhaltene Werk zur Kryptologie stammt von Leon Battista Alberti (ital. Baumeister und Kunsthistoriker, 1404–1472).

Von einer anderen Seite kommend suchten die Logiker nach möglichst universell anwendbaren Algorithmen. Die auf Aristoteles (384–322 v. Chr.) zurückgehende Syllogistik umfasst nur einen kleinen Teil der für Mathematiker relevanten Schlüsse. Leibnitz beschrieb (wieder als erster) 1672–1676 die wesentlichen Ansätze für einen logischen Kalkül. Mit einer präzisen Universalsprache, der *characteristica universalis*, wollte er Aussagen beschreiben und mit dem Kalkül, dem *calculus ratiocinator*, weiter behandeln. Noch heute folgt man in wesentlichen Teilen dieser Methode zur Definition von Kalkülen, wie in diesem Skript im Detail ausgeführt und an einigen Beispielen erläutert wurde.

Der Schweizer Mathematiker Leonard Euler (1707–1783) wie auch der Engländer John Venn gaben Diagramme zur Erklärung und Durchführung logischer Operationen. Zu den Wegbereitern der modernen Logik zählt vor allem der Engländer George Boole (1815–1864) und sein Kollege Augustus de Morgan (1806–1871). Boole's algebraische Sichtweise der (Aussagen-)Logik wurde später, besonders durch G. Peano (1858–1932), mehr und mehr durch eine, der natürlichen Sprache angepasste, Symbolik ersetzt. Mit seiner *Begriffsschrift* bemühte sich Gottlob Frege (1848–1925) um eine exakte Fassung der logischen Regeln. Alfred North Whitehead (1861–1949) und Bertrand Russell (1872–1970) formulierten

A. Historischer Ursprung

mit dem monumentalen Werk: *Principia Mathematica* eine logische Grundlage der Mathematik. Auf der Suche nach Beweisen für den Prädikatenkalkül erster Stufe führten die Arbeiten von Skolem, etwa 1920, 1930 von Herbrand, und 1933 von Gerhard Gentzen (1909–1945) sowie 1956 von J.A. Robinson (1930 –) zu Formalen Systemen mit den Begriffen von Folgerbarkeit und Ableitbarkeit. Einen krönenden Abschluss fand die vorangegangene Entwicklung 1930 mit dem Gödel’schen Vollständigkeitssatz für die Prädikatenlogik erster Stufe und 1931 mit dem Beweis dafür, dass jede widerspruchsfreie Axiomatisierung der Zahlentheorie nicht entscheidbare Aussagen enthält. Das heißt, dass es eine nicht leere Menge von wahren Aussagen gibt, für die es jedoch kein Verfahren gibt, einen Beweis dieser Tatsache für alle Aussagen dieser Menge zu finden.

Die Suche nach einer Klärung des Begriffs „Algorithmus“ ist aufs engste verbunden mit den Versuchen einer formalen Definition der als intuitive berechenbar angesehenen Funktionen. Auf David Hilbert (1862–1943) und P. Bernays geht der Begriff der primitiven Rekursion zurück, während Stephen C. Kleene die Bezeichnung der primitiv rekursiven Funktionen prägte, die 1931 von Kurt Gödel (1906–1978) noch als rekursive Funktionen bezeichnet wurden. W. Ackermann beschrieb 1928 eine offensichtlich intuitiv berechenbare Funktion, die nicht primitiv rekursiv ist und zeigte so, dass dieser Begriff nicht alle intuitiv berechenbaren Funktionen umfasst. 1936 schlug Alonzo Church (1903–1994) vor, den Begriff der λ -definierbaren Funktionen mit dem der intuitiv berechenbaren Funktionen zu identifizieren. Dies wird heute als Church’sche These bezeichnet. Da diese Funktionen nicht überall definiert sein müssen, werden sie heute meist als partiell rekursiv bezeichnet.

Präzisierungen des Algorithmusbegriffs wurden mit unterschiedlichen formalen Methoden etwa gleichzeitig entwickelt: J. Herbrand 1932, Kurt Gödel 1934 und Stephen C. Kleene 1936 benutzten den Gleichungskalkül, Alonzo Church 1936 den Lambda-Kalkül, Alan M. Turing (1912–1954) formulierte 1937 die jetzt nach ihm benannten Maschinen und Emil L. Post (1887–1954) definierte 1936 formale Wortersetzungssysteme. Axel Thue (1863–1922) definierte 1906 die sogenannten Thue-Systeme und 1914 die heutigen semiThue-Systeme. Alle diese Präzisierungen haben sich als äquivalent erwiesen und beschreiben das, was die Erfinder als intuitiv berechenbar ansahen. Die Thesen, alles intuitiv Berechenbare sei mit dem jeweiligen Kalkül erfassbar, werden jetzt als Turing’sche beziehungsweise Church’sche These bezeichnet. Tatsächlich stellten sich alle weiteren formulierten Definitionen des Algorithmus-Begriffs immer wieder als äquivalent zu den vorgenannten heraus. Damit fand diese Entwicklung einen bis heute noch gültigen Abschluss.

Literaturverzeichnis

- [BauerGoos] F.L. Bauer & G. Goos:
„Informatik, Eine einführende Übersicht, Teil I“,
Springer-Verlag, Heidelberg (1982)
- [Loeckx] J. Loeckx:
„Algorithmentheorie“,
Springer-Verlag Heidelberg (1976)
- [Maurer] H. Maurer:
„Theoretische Grundlagen der Programmiersprachen“,
B.I. Wissenschaftsverlag, Stuttgart (1969)
- [Turing] A.M. Turing:
„On computable numbers with an application to the Entscheidungsproblem“,
Proc. London Math. Soc. 42 (1936) 230-265,
43 (1937) 544-546
- [Wegener] I. Wegener:
„Theoretische Informatik“,
B.G. Teubener, Stuttgart (1993)
- [HopcroftUllman] J.E. Hopcroft & J.D. Ullman:
„Introduction to Automata Theory, Languages, and Computation“,
Addison-Wesley, Reading Mass. (1979)
- [Mehlhorn] K. Mehlhorn:
„Graph Algorithms and NP-Completeness“,
EATCS-Monographs 2, Springer-Verlag (1984)
- [GareyJohnson] M.R. Garey & D.S. Johnson:
„Computers and Intractability“,
Freeman (1979)
- [Aho&Ullman] A.V. Aho & J.D. Ullman:
„The Theory of Parsing, Translation and Compiling“,
Prentice Hall, vol I (1972), vol II (1973)
- [Aho,Hopcroft&Ullman] A.V. Aho, J.E. Hopcroft & J.D. Ullman:
„The Design and Analysis of Computer Algorithms“,
Addison Wesley, Reading Mass. (1974)

Literaturverzeichnis

- [Aho,Sethi&Ullman] A.V. Aho, R. Sethi & J.D. Ullman:
“Compilers – Principles, Techniques, and Tools“,
Addison Wesley, Reading Mass. (1986)
- [Aigner] M. Aigner:
“Diskrete Mathematik“,
Friedr. Vieweg & Sohn, Verlagsgesellschaft mbH., Braunschweig, Wies-
baden (1993)
- [Baase] S. Baase:
“Computer Algorithms
– Introduction to Design and Analysis“,
Addison Wesley, Reading Mass. (1988)
- [Biggs] N.L. Biggs:
“Discrete Mathematics“
Oxford University Press Inc. New York (1998)
- [Brauer] W. Brauer:
“Automatentheorie“,
Teubner, Stuttgart (1984)
- [Bronstein et al.] I.N. Bronstein & K.A. Semedjajew & G. Musiol & H. Mühlig:
“Taschenbuch der Mathematik“
Verlag Harri Deutsch, Frankfurt a.M. (1995)
- [Duden Informatik] “Duden Informatik“
Dudenverlag, Mannheim (1993)
- [Floyd&Beigel] R. Floyd & R. Beigel:
“Die Sprache der Maschinen“,
Internat. Thomson Publ., Bonn, Albany (1996)
- [Graham et al.] R.L. Graham, D.E. Knuth, & O. Patashnik:
“Concrete Mathematics“,
Addison-Wesley, Reading, Ma. (1989)
- [Gries&Schneider] D. Gries & F.B. Schneider:
“A Logical Approach to Discrete Math“.
Springer-Verlag, Heidelberg (1993)
- [Gruska] J. Gruska:
“foundations of computing“.
Int. Thomson Computer Press, London, Boston, Bonn (1997)
- [Foster] J.M. Foster:
“Automatische Syntax-Analyse“,
Carl Hanser Verlag, München (1971)
- [Harrison] M.A. Harrison:
“Introduction to Formal Language Theory“,
Addison-Wesley, Reading Mass. (1978)

- [Hopcroft,Motwani&Ullman] J.E. Hopcroft, R. Motwani & J.D. Ullman:
“Introduction to Automata Theory, Languages, and Computation“,
Addison-Wesley, 2nd edition (2001)
- [Hopcroft&Ullman] J.E. Hopcroft & J.D. Ullman:
“Introduction to Automata Theory, Languages, and Computation“,
Addison-Wesley, Reading Mass. (1979)
- [Kinber&Smith] E. Kinber & C. Smith:
“Theory of Computing – A Gentle Introduction“,
Prentice Hall, Inc. (2001)
- [Loeckx,Mehlhorn&Wilhelm] J. Loeckx, K. Mehlhorn & R. Wilhelm:
“Grundlagen der Programmiersprachen“,
Teubner, Stuttgart (1986)
- [Maurer] H. Maurer:
“Theoretische Grundlagen der Programmiersprachen“,
B.I. Wissenschaftsverlag, Stuttgart (1969)
- [Möller] H. Möller:
“Algorithmische Lineare Algebra – Eine Einführung für Mathematiker
und Informatiker“,
Friedr. Vieweg & Sohn, Verlagsgesellschaft mbH., Braunschweig, Wies-
baden (1997)
- [Sander,Stucky&Herschel] P. Sander, W. Stucky & R. Herschel:
“Automaten, Sprachen, Berechenbarkeit“,
Teubner, Stuttgart (1992)
- [Schöning] U. Schöning:
“Theoretische Informatik kurz gefasst“,
B.I. Wissenschaftsverlag, Mannheim (1992)
- [Turing] A.M. Turing:
“On computable numbers with an application to the Entscheidungspro-
blem“,
Proc. London Math. Soc. 42 (1936) 230-265,
43 (1937) 544-546
- [Wegener] I. Wegener:
“Theoretische Informatik“,
B.G. Teubner, Stuttgart (1993)
- [Zobel] R. Zobel:
“Diskrete Strukturen – eine angewandte Algebra für Informatiker“,
Reihe Informatik Band 49,
B.I. Wissenschaftsverlag, Mannheim (1987)

Index

- A-Ableitungsbaum, 106
- Abbildung, *siehe* Funktion, totale
- Ableitungsbaum, 106
- Ableitungsrelation: $\xRightarrow[G]{*}$, 97
- Abschluss
 - reflexiver und transitiver, 20
 - transitiver, 20
- abstrakter Syntaxbaum, 136
- Adjazenzmatrix, 58
- Äquivalenz
 - Bestimmung äquivalenter Zustände, 83–86
 - von Grammatiken, 98
 - von Zuständen, 79
- Äquivalenzautomat, 82
- Äquivalenzklasse: $[a]_R$, 76
- Äquivalenzproblem
 - für DFA's, 94
 - für DFAs, 94
- Äquivalenzrelation, 76
 - Äquivalenzklasse: $[a]_R$, 76
 - Index, 77
 - Partition, 76
 - rechtsinvariant, 77
- AFL-Theorie, 87
- Algebra, *siehe* Struktur algebraische
- Algorithmus
 - äquivalente Zustände, 83
 - Chomsky Normalform, 100
 - Cocke-Younger-Kasami, 113
 - dezimal nach b-adisch, 48
 - dezimal nach b-när, 47
 - Entfernen der λ -Kanten in CFG, 101
 - Greibach-Normalform, 108
 - initialer Zusammenhang, 67
 - $\lambda \in L(G)$, 103
 - Leerheitsproblem: NFA, 93
 - Tripelkonstruktion, 126
- Alphabet, 34
 - ASCII, 35
- Assoziativgesetz, 32
- Automorphismus, 33
- b -adische Zahlendarstellung, 47
- b -näre Zahlendarstellung, 46
- Backus/Naur-Notation, 95
- Baum, 104
- Bijektion, *siehe* Funktion, bijektive
- binäre Zahlendarstellung, 46
- Blatt, 104
- boolesches Matrizenprodukt: \otimes , 58
- bottom-up parsing*, 139
- cartesisches Produkt, 15
- ceiling*, 13
- Chomsky-Normalform einer CFG, 100
- Codegenerierung, 136
- codomain*, *siehe* Wertebereich
- Compiler, 135
- Definitionsbereich, 15
- Diagonalisierung, 29
- Differenzbildung
 - regulärer Mengen, 88
- disjunkte Vereinigung, *siehe* Menge, disjunkte
 - Vereinigung: $M_1 \uplus M_2$
- domain*, *siehe* Definitionsbereich
- Durchschnitt
 - Menge, *siehe* Menge, Durchschnitt: $M_1 \cap M_2$

Index

- regulärer Mengen, 88
- dyadische Zahlendarstellung, 47
- Dyck-Sprache, 37, 106, 128
- egrep, 69
- eindeutige CFG, 107
- Eingabealphabet: Σ , 51
- endlicher Automat
 - λ -freier NFA, 57
 - Äquivalenzautomat, 82
 - akzeptierte Sprache
 - DFA, 52
 - NFA, 57
 - buchstabierender NFA, 57, 65
 - deterministischer: DFA, 51
 - Durchschnitt, 88
 - Erfolgsrechnung im NFA, 57
 - initial zusammenhängender DFA:
 - izDFA, 53
 - initiale Zusammenhangskomponente, 67
 - λ -freier NFA, 56, 63
 - minimaler DFA, 79
 - nichtdeterministischer: NFA, 56
 - Potenzautomat, 65
 - Produktautomat, 88
 - Rechnung, 57
 - reduzierter DFA, 79
 - Vereinigungs NFA, 70
 - vollständiger DFA: vDFA, 53
 - Bestimmung äquivalenter Zustände, 83–86
- Endomorphismus, 33
- Endzustände: Z_{end} , 51
- Epimorphismus, 33
- Erfolgsrechnung
 - im NFA, 57
- erreichbares Nonterminal einer CFG, 99
- Erzeugendensystem, 33
 - freies, 33
- Familie
 - kontextfreie Sprachen: \mathcal{Cf} , 98
- Familie der regulären Mengen: \mathcal{Reg} , 54
- Fibonacci-Zahlen
 - geschlossene Form, 10
 - rekursive Definition, 10
- FIRST $_k(w)$, 147
- first $_k(w)$, 142
- floor, 13
- Folge, 10
- formale Sprache, 36
 - COPY, 131
 - DUP, 79, 91
 - Dyck-Sprache: D_1 , 37, 128
 - PAL, 131
- Funktion
 - bijektive, 26
 - charakteristische: χ_M , 12
 - Gödelisierungs~, 44
 - injektive, 26
 - partielle, 24
 - Stelligkeit, 24
 - surjektive, 26
 - totale, 24
- ganze Zahlen: \mathbb{Z} , 13
- gerichteter Graph, *siehe* Graph, gerichteter
- Gleichheitszeichen
 - relational: $=$, 9
 - zur Definition: $:=$, 9
- Grammatik, 96
 - kontextfrei: CFG, 97
 - eindeutig, 107
 - einseitig linear, 103
 - linear, 103
 - linkslinear, 103
 - LL(k), 148
 - LR(0), 144, 145
 - LR(0'), 145
 - LR(k), 142
 - mehrdeutig, 107
 - Nonterminalalphabet: V_N , 97
 - Produktion, 97
 - Produktionsmenge: P , 97
 - rechtslinear, 103
 - reduziert, 99
 - Startsymbol: S , 97
 - Terminalalphabet: V_T , 97

- reguläre, *siehe* Grammatik, kontextfrei:
 - CFG, einseitig linear
- Typ-3, *siehe* Grammatik, kontextfrei: CFG,
 - einseitig linear
- Graph
 - gerichteter, 58
 - kantenbewerteter, 59
 - zu einer Relation, 58
- Greibach-Normalform, 108
- Gödelisierung, 44–49
- Halbgruppe, 32
- Halbordnung, *siehe* Striktordnung
- handle*, *siehe* Schlüssel
- Hasse-Diagramm, 18
- Homomorphismus, 33, 89
 - inverser, 89
- Identitätsrelation, 18, 20
- iff*, *siehe* Junktoren, logischer: Biimplikation
- Index einer Äquivalenzrelation, 77
- Infimum, *glb*, 18
- initiale Zusammenhangskomponente, 67
- Injektion, *siehe* Funktion, injektive
- Interpreter, 135
- inverser Homomorphismus, 89
- Isomorphismus, 33
- Junktor, logischer, 11
 - Allquantor: \forall , 11
 - Biimplikation: \leftrightarrow , 11
 - Disjunktion: \vee , 11
 - Existenzquantor: \exists , 11
 - Implikation: \rightarrow , 11
 - Konjunktion: \wedge , 11
 - Negation: \neg , 11
- kantenbewerteter Graph, *siehe* Graph, Kanten
 - bewerteter
- Kellerautomat
 - Äquivalenz, 123
 - akzeptierte Sprache
 - mit Endzustand: $L(A)$, 122
 - mit leerem Keller: $N(A)$, 122
 - buchstabierend, 123
 - deterministisch, 123
 - fast-buchstabierend, 123
 - Konfiguration, 121
 - nichtdeterministisch: PDA, 119
 - Überführungsrelation, 122
- Kettenregel, 102
- Kleene, Satz von \sim : $Reg = Rat$, 72
- Kleene-Hülle, *siehe* Sternbildung, reguläre Men-
 - ge
- Komplementbildung
 - regulärer Mengen, 88
- komplexe Zahlen: \mathbb{C} , 13
- Komplexprodukt, 32
- Komponentenfunktion, 44
- Komposition
 - von Relationen, 16
- Konkatenation, 36
- kontextfreie Grammatik, *siehe* Grammatik,
 - kontextfrei
- kontextfreie Produktion, *siehe* Grammatik, kon-
 - textfrei, Produktion
- kontextfreie Regel, *siehe* Grammatik, kontext-
 - frei, Produktion
- Korrespondenz, *siehe* Relation, binäre
- Kugelautomat, 3
- λ -freie CFG, 97
- λ -freier NFA, *siehe* endlicher Automat, λ -freier
- λ -Produktion
 - einer CFG, 97
- λ -Kante, 56
- Leerheitsproblem
 - für NFA, 93
- Leistung eines Zustandes: $L(z)$, 79
- Lexikalische Analyse, 135
- lexikalische Ordnung: $\preceq^{\text{lg-lex}}$, 39
- lexikographische Ordnung: \preceq^{lex} , 38
- Linksableitung einer CFG, 107
- Linksrekursion, 140
- $LL(k)$ -Grammatiken, 148
- $LR'(0)$ -Grammatik, 145
- $LR(0)$ -Grammatik, 144, 145
- $LR(k)$ -Grammatik, 142, 145
- LR-Parsing, 141

Index

Matrizenprodukt

boolesches: \otimes , 58

Maximum, 17

Mealy-Automat, 62

mehrdeutige CFG, 107

Menge, 10

abzählbare, 27

der ganzen Zahlen: \mathbb{Z} , 13

der komplexen Zahlen: \mathbb{C} , 13

der natürlichen Zahlen: \mathbb{N} , 13

der rationalen Zahlen: \mathbb{Q} , 13

der reellen Zahlen: \mathbb{R} , 13

disjunkte Vereinigung: $M_1 \uplus M_2$, 13

Durchschnitt: $M_1 \cap M_2$, 13

Kardinalität: $|M|$, 12

Komplement: \overline{M} , 13

leere: \emptyset , 11

Mengendifferenz: $C \setminus M$, 13

partiell geordnet, **poset**, 17

Potenzmenge: 2^M , 12

unendliche: $|M| = \infty$, 12

Vereinigung: $M_1 \cup M_2$, 13

minimaler DFA, 79

Minimum, 17

Mittel

arithmetisches, *mean*, 21

Mittelwert

arithmetischer, 9

geometrischer, 9

monadische Zahlendarstellung, 47

Monoid, 32

freies, 33–34, 36

Monomorphismus, 33

Moore-Automat, 61

Morse-Code, 105

Nachbereich, 15

Nachfolger, 104

natürliche Zahlen: \mathbb{N} , 13

Nerode-Äquivalenz, 77

neutrales Element, 32

Nonterminal, *siehe* Nonterminalalphabet

erreichbares, 99

produktives, 99

Nonterminalalphabet: V_N , 96

Operation, 31

disjunkte Vereinigung: $M_1 \uplus M_2$, 13

Durchschnitt: $M_1 \cap M_2$, 13

inverser Homomorphismus, 89, 90

Komplement: \overline{M} , 13

Mengendifferenz: $C \setminus M$, 13

Spiegelwort, 90

Substitution, 89

Vereinigung: $M_1 \cup M_2$, 13

Operator, 87

\wedge , 87, 115

\vee , 87, 115

inverser Homomorphismus, 89, 90

Komplement, 115, 132

Mengendifferenz, 115

Spiegelwort, 90

Stern: $()^*$, 87, 115

Substitution, 89

Ordnung

lexikalische Erweiterung: $<^{\text{lg-lex}}$, 40

lexikalische: $\preceq^{\text{lg-lex}}$, 39

lexikographische Erweiterung: $<^{\text{lex}}$, 40

lexikographische: \preceq^{lex} , 38

lineare, 18

totale, *siehe* Ordnung, lineare

wohl-fundierte, 18

Ordnungsrelation, 17

Paarfunktion: *pair*, 41

Palindrom: $w = w^{\text{rev}}$, *siehe* Spiegelwort

partielle Ordnung, 17

Partition, 13, 76

Potenzautomat, 65

Präordnung, 17

preorder, 17

Produkt

cartesisches, 15

regulärer Mengen, 71

Produktautomat, 88

Produktion einer CFG

A -Produktion, 107

λ -Produktion, 97

- linksrekursiv, 107
- Produktionsmenge einer CFG, *siehe* Grammatik, kontextfrei, Produktionsmenge
- produktives Nonterminal einer CFG, 99
- Prädikat
 - Iverson'sches: $[P]$, 11
- Pumping Lemma
 - für Cf , *siehe* $uvwx$ -Theorem
- Pumping-Lemma
 - für Reg , *siehe* uvw -Theorem
- Quantoren
 - existiert: \exists , 11
 - für alle: \forall , 11
- Quasiordnung, 17
- Quellprogramm, 135
- range*, *siehe* Wertebereich
- rationale Zahlen: \mathbb{Q} , 13
- rationaler Ausdruck, 68
- Rechtsquotient, 91
- reduzierte CFG, 99, *siehe* Grammatik, kontextfrei, reduziert
- reduzierter DFA, 79
- reelle Zahlen: \mathbb{R} , 13
- reguläre Menge, 54
 - Differenzbildung, 88
 - Durchschnitt, 88
 - Komplementbildung, 88
 - Produkt, 71
 - Rechtsquotient, 91
 - Sternbildung, 71
 - Vereinigung, 70
- regulärer Ausdruck, *siehe* rationaler Ausdruck
- Relation
 - n -stellige, 15
 - antisymmetrische, 16
 - asymmetrische, 16
 - binäre, 15
 - dyadische, 15
 - feiner, 16
 - gröber, 16
 - Identitäts-, 20
 - inverse, 16
 - irreflexive, 16
 - Komposition von, 16
 - lineare, 16
 - reflexive, 16
 - symmetrische, 16
 - transitive, 16
- Relationalstruktur, *siehe* Struktur, relationale
- Relationensystem, *siehe* Struktur, relationale
- Satzform einer Grammatik, 98
- Schlüssel, 144
- Schlüsselwortsuche, 74
- Schranke
 - obere, 17
 - untere, 17
- Schubfachprinzip
 - einfachste Variante, 21
 - leicht verallgemeinertes, 21
 - verallgemeinertes, 22
- Semantische Analyse, 136
- source code*, *siehe* Quellprogramm
- Spiegelwort: w^{rev} , 37
- Sprache
 - eindeutige Cf , 107
 - formale, 36
 - mehrdeutige Cf , 107
 - präfixfrei, 129
 - kontextfrei, 129
 - von CFG erzeugte, 98
 - von DFA akzeptierte, 52
 - von dPDA akzeptierte, 123
 - von NFA akzeptierte, 57
 - von PDA akzeptierte, 122
- Sprachfamilie, 54
 - $\mathcal{A}kz(\Sigma)$, 54
 - Cf , 98, 128
 - $\det Cf$, 123, 129
 - Rat , 69
 - Reg , 54
- Startsymbol einer CFG, *siehe* Grammatik, kontextfrei, Startsymbol
- Startzustand: z_0 , 51
- Stelligkeit
 - einer Funktion, 24

Index

- Sternbildung
 - regulärer Mengen, 71
- Striktordnung, *spo*, 18
- Struktur
 - algebraische, 31
 - relationale, 31
- Substitution
 - endliche, 89
 - reguläre, 89
- Supremum, *lub*, 18
- Surjektion, *siehe* Funktion, surjektive
- Symbol
 - Häufigkeit des Auftretens: $|w|_x$, 36
 - Kellerbodenzeichen: \perp , 119
- Syntaktische Analyse, 136
- syntaktische Rechtskongruenz, 77
- Syntaxdiagramm, 96
- target code*, *siehe* Zielcode
- Terminal, *siehe* Terminalalphabet
- Terminalalphabet: V_T , 96
- top-down parsing*, 140
- Transitionsmonoid, 75
- transitive Hülle, *siehe* Abschluss, transitiver
- transponierter Vektor: \vec{x}^\top , 15
- Tripelkonstruktion, 126
- Tupel, 15
- Tupelfunktion, 44
- Typ-2-Grammatik, *siehe* Grammatik, kontextfrei
- Typ-3-Grammatik, *siehe* Grammatik, kontextfrei: CFG, einseitig linear
- Überführungsabbildung
 - einfache: δ , 52
 - erweiterte: $\hat{\delta}$, 52
- Universalitätsproblem
 - für DFA's, 94
 - für DFAs, 94
- Unterhalbgruppe, 33
 - von M erzeugt: M^+ , 33
- Untermonoid, 33
 - von M erzeugt: M^* , 33
- uvw*-Theorem, 86
- uvwx*-Theorem, 110
- Vektoren
 - lexikalische Ordnung: $\leq^{\text{lg-lex}}$, 40
 - lexikographische Ordnung: \leq^{lex} , 40
 - transponierte, *siehe* transponierter Vektor: \vec{x}^\top
- Vereinigung, *siehe* Menge, Vereinigung: $M_1 \cup M_2$
 - regulärer Mengen, 70
- Verknüpfung, 31
- Vorbereich, 15
- Vorgänger, 104
- Wertebereich, 16
- Wohlordnung, 18
- Wort, 36
 - gespiegeltes: w^{rev} , 37
 - leeres: λ , 36
 - Länge: $|w|$, 36
 - Präfix
 - beliebiges: \sqsubseteq , 36
 - echtes: \sqsubset , 36
 - Teilwort, 36
 - z und z' unterscheidendes, 83
- Wortproblem
 - allgemeines
 - für \mathcal{Cf} , 113
 - für \mathcal{Rat} , 92
 - spezielles
 - für \mathcal{Cf} , 112, 113
 - für \mathcal{Reg} , 92
- Wurzel, 104
- Zahlendarstellung
 - b -adische, 47
 - b -näre, 46
 - binäre, 46
 - dyadische, 47
 - monadische, 47
- Zeichenkette, *siehe* Wort
- Zustandsgraph, 52
 - eines DFA, 52
 - eines NFA, 59
 - eines PDA, 121