Skript

Theoretische Informatik II

Dr. Dominik D. Freydenberger
freydenberger@em.uni-frankfurt.de

28. Juli 2014

14:39

Inhaltsverzeichnis

1 Einleitung

1.1 Über diese Vorlesung

Viele Problemstellungen der Informatik lassen sich als *formale Sprachen*, also Mengen von Wörtern definieren. Hauptthema dieser Vorlesung sind die zur Definition solcher Sprachen verwendeten Mechanismen. Der Schwerpunkt liegt dabei auf den Klasse der *regulären Sprachen* und der *kontextfreien* Sprachen.

Bei regulären Sprachen betrachten wir verschiedene Berechnungsmodelle, die für ihre Definition benutzt werden können (deterministische und nichtdeterministische endliche Automaten, reguläre Ausdrücke und reguläre Grammatiken). Die Ausdrucksstärke der Klasse der regulären Sprachen ist zwar vergleichsweise gering, dafür sind die zu ihrer Definition verwendeten Mechanismen beherrschbar. So lassen sich zum Beispiel deterministische endliche Automaten effizient minimieren und auswerten. Endliche Automaten, ihre Erweiterungen (z. B. probabilistische Automaten) und reguläre Ausdrücke werden daher in verschiedenen Gebieten eingesetzt, wie zum Beispiel Compilerbau (in der lexikalischen Analyse), Verifikation, Spracherkennung, Netzwerkprotokolle oder auch Algorithmen zur Suche in Wörtern.

Die kontextfreien Sprachen verfügen über eine größere Ausdrucksstärke, dafür sind die zur Definition verwendeten Mechanismen – kontextfreie Grammatiken und Kellerautomaten – weniger beherrschbar (zum Beispiel lassen sie sich nicht minimieren). Kontextfreie Grammatiken werden zur Definition von Programmiersprachen verwendet und sind dort von immenser Bedeutung. Außerdem gibt es noch eine Vielzahl weitere Anwendungen, wie zum Beispiel in der Computerlinguistik.

Dennoch ist die Ausdrucksstärke der kontextfreien Sprachen beschränkt: Nicht jede entscheidbare Sprache ist kontextfrei. Wir betrachten daher in einem weiteren Kapitel Sprachklassen, die nicht-kontextfreie Sprachen enthalten. Dabei befassen wir uns mit den höheren Ebenen der sogenannten Chomsky-Hierarchie, vor allem den *kontextsensitiven Sprachen* und den entsprechenden Grammatik- und Automatenmodellen. Da hier deren Ausdrucksstärke durch eine stark verringerte Handhabbarkeit erkauft werden muss, gehen wir auf diese Modelle nur oberflächlich ein. Zusätzlich dazu betrachten wir noch zwei Sprachklassen, die anhand von Wiederholungsoperatoren definiert werden. Die erste, die sogenannten *Patternsprachen*, ist vor allem wegen ihrer Negativresultate interessante. Die zweite, die anhand von *erweiterten regulären Ausdrücken* definiert wird, orientiert sich an regulären Ausdrücken wie sie inzwischen in der Praxis häufig verwendet werden.

Gegen Ende der Vorlesung wenden wir uns zwei Anwendungsfeldern der Sprachtheorie zu. Einerseits befassen wir uns kurz mit der Definition von Sprachen von XML-Dokumenten durch sogenannte *Document Type Definitions (DTDs)*, andererseits gehen wir auf das *Pattern-Match-Problem* ein.

1.2 Über dieses Skript

Wie Sie leicht erkennen können, ist dieses Skript noch in einem recht unfertigen Stadium. Zwar sind alle Inhalte der Vorlesung vorhanden (und gelegentlich auch zusätzliches Material, das nicht besprochen wurde); allerdings ist der Schriftsatz noch nicht optimiert, außerdem fehlen momentan viele erläuternde Erklärungen zur Motivation und zum größeren Kontext (im Grunde ist es zur Zeit also eher eine unfertige Vorlesungsmitschrift als ein Skript). Ich plane, diese fehlenden Zusatzinformationen nach und nach hinzufügen; bis dahin verweise ich Sie dafür einfach auf die Vorlesung, verschiedene Lehrbücher sowie auf das Skript zur Vorlesung *Theoretische Informatik* von Herrn Prof. Schnitger [14] sowie auf das Skript zur Vorlesung *Diskrete Modellierung* von Frau Prof. Schweikardt [16] (in letzterem finden sie auch Definition für der hier nicht definierten, aber verwendeten elementaren Begriffe, wie z. B. von Mengen).

Eigentlich sollte darüber hinaus keine weitere Literatur notwendig sein. Natürlich kann es aber nicht schaden, von Zeit zu Zeit einen Blick in das eine oder andere Buch zu werfen. Die eigentlich Auswahl ist dabei Geschmacksfrage; ich persönlich finde Hopcroft und Ullman [8] und Sipser [20] besonders hilfreich. Empfehlenswert sind aber außerdem auch Diekert et al. [4], Rozenberg und Salomaa [12], Schöning [15], Shallit [19], und Wegener [21, 22] (die Reihenfolge hat keine besondere Bedeutung).

Wie so oft bei längeren Texten haben sich außerdem sicherlich einige Fehler eingeschlichen. Für Hinweise auf Fehler bin ich besonders dankbar (natürlich besonders für inhaltliche Fehler, aber auch für Tippfehler und ähnliches).

Um Ihnen den Umgang mit diesem Skript zu erleichtern sind alle Verweise innerhalb des Skripts als Links gestaltet. Klicks auf Literaturangaben führen zum Literaturverzeichnis, die Einträge im Inhaltsverzeichnis (Seite 2) und im Index (am Ende des Dokuments) führen zu den entsprechenden Seiten. Außerdem sind Verweise auf Definitionen und Resultate (z. B. Definition 3.1 oder Lemma 3.10) ebenfalls klickbar.

Neben dem **Beweisendzeichen** □ verwenden wir außerdem auch das Symbol ♢ als **Beispielendzeichen** und das Symbol □ als Endzeichen für Beweisideen.

Hinweis 1.1 An verschiedenen Stellen des Skriptes befinden sich graue Hinweiskästen (wie dieser). In diesen werden wichtige Informationen nochmals betont, wie zum Beispiel Hinweise auf häufige Fehler oder eine Zusammenfassung verschiedener Ansätze zum Lösen häufiger Aufgabentypen.

Jeder dieser Hinweise ist auch im Index unter dem Stichwort „Hinweis“ aufgeführt.

1.3 Danksagungen

Dieses Skript und die dazugehörende Vorlesung sind natürlich nicht in einem Vakuum entstanden, ich wurde dabei von einer Vielzahl an Vorbildern beeinflusst. Besondere Erwähnung verdienen hier die Vorlesungen und/oder Skripte von Frau Prof. Schweikardt, Herrn Prof. Schnitger und Herrn Prof. Wotschke (alle drei an der Goethe-Universität in Frankfurt) sowie von Herrn Prof. Wiehagen an der TU Kaiserslautern.

Beim Erstellen der Vorlesung habe ich außerdem bemerkt, dass ich ungemein von der Breite und Tiefe der Vorlesungen im Rahmen der *5th International PhD School in Formal Languages and Applications* an der URV Tarragona profitiert habe. Direkte Einflüsse, die mir bewusst sind, stammen aus den Vorlesungen der Professoren Hendrik Jan Hoogeboom, Masami Ito, Manfred Kudlek, Victor Mitrana, Alexander Okhotin, Jean-Éric Pin, Kai Salomaa und Sheng Yu.

Besonderer Dank gebührt auch Herrn Joachim Bremer für eine Vielzahl hilfreicher Hinweise, sowie für seinen außergewöhnlichen Einsatz beim Erstellen der Übungsblätter, ohne den ich weit weniger Zeit für dieses Skript gehabt hätte.

Für Hinweise auf verschiedene kleinere Fehler danke ich[1]:

Florian Aul, Daniel Bauer, Vedad Cizmic, Sorin Constantinescu, Mario Holldack, Tim Ingelfinger, Fabian Knöller, Lukas Larisch, Marc Pohl, Elias Rieb, Nikolaj Schepsen, Nadine Seibel, Jesika Stasilojc, Rafael Tisch und Christopher Wolf.

Bisher wurden alle ernsteren inhaltlichen Fehler von mir selbst gefunden. Wenn Sie mich auf einen solchen ernsteren Fehler aufmerksam machen, den noch niemand zuvor bemerkt hat, werde ich Ihren Namen in eine gesonderte Liste mit besonderem Dank aufnehmen.

[1] Hier könnte Ihr Name stehen! Wenn Sie mich auf einen Tippfehler oder ähnliches aufmerksam machen, den noch niemand zuvor bemerkt hat, werde ich Ihren Namen in diese Liste aufnehmen.

2 Grundlagen und Notationen

Dieses Kapitel stellt einige der grundlegenden Definitionen und Konzepte vor. Undefinierte Begriffe sind im Skript zur Vorlesung *Diskrete Modellierung* [16] zu finden.

2.1 Mengen

Wir bezeichnen die Menge der **natürlichen Zahlen** als $\mathbb{N} := \{0, 1, 2, 3, \ldots\}$. Für jedes $k \in \mathbb{N}$ sei $\mathbb{N}_{>k} := \{i \mid i \in \mathbb{N}, i > k\}$ (also ist $\mathbb{N}_{>0}$ die Menge aller positiven natürlichen Zahlen). Die Rechenoperation $\text{mod} : \mathbb{N} \times \mathbb{N}_{>0} \to \mathbb{N}$ definieren wir, indem $m \bmod n$ als Resultat den Rest der ganzzahligen Teilung von m durch n erhält.

Wir verwenden die Symbole $\subseteq$ und $\subset$ um die **Teilmengenbeziehung** bzw. die **echte Teilmengenbeziehung** auszudrücken. Analog dazu verwenden wir $\supseteq$ und $\supset$ für die **Obermengenbeziehung** und die **echte Obermengenbeziehung**.

Wir bezeichnen die **leere Menge** mit $\emptyset$. Sind A und B Mengen, so ist die **Mengendifferenz** $A - B$ definiert als $A - B := \{a \in A \mid a \notin B\}$. Die **symmetische Differenz** zweier Mengen A und B bezeichnen wir mit $A \triangle B$, diese ist definiert als $A \triangle B := (A - B) \cup (B - A)$. Die **Potenzmenge** einer Menge A (also die Menge aller Teilmengen von A) bezeichnen wir mit $\mathcal{P}(A)$. Mit $\mathcal{P}_F(A)$ bezeichnen wir die Menge aller endlichen Teilmengen von A. Zwei Mengen A und B sind **unvergleichbar**, wenn weder $A \subseteq B$, noch $B \subseteq A$ gilt.

Eine Menge heißt **endlich**, wenn sie endlich viele Elemente enthält, und **unendlich**, wenn sie unendlich viele Wörter enthält. Eine Menge heißt **kofinit** (oder **koendlich**), wenn ihr Komplement endlich ist.

Wir bezeichnen die **Mächtigkeit** einer Menge A mit $|A|$ (falls A endlich ist, entspricht $|A|$ der Zahl der Elemente von A).

2.2 Wörter und Sprachen

Ein **Alphabet** Σ ist eine endliche, nicht-leere Menge von Buchstaben. Ist $|\Sigma| = 1$, so bezeichnen wir Σ als **unäres Alphabet**. Für jedes $n \in \mathbb{N}$ ist

$$\Sigma^n := \{a_1 a_2 \cdots a_n \mid a_i \in \Sigma\}$$

die Menge aller **Wörter** der Länge n über Σ, und ε bezeichnet das **leere Wort**[2]. Insbesondere gilt $\Sigma^0 = \{\varepsilon\}$. Außerdem bezeichnet

$$\Sigma^* := \bigcup_{n \in \mathbb{N}} \Sigma^n$$

[2] In der Literatur werden anstelle von ε auch ϵ, e, λ und 1 verwendet.

die Menge aller Wörter über Σ, und

$$\Sigma^+ := \bigcup_{n \in \mathbb{N}_{>0}} \Sigma^n$$

die Menge aller nicht-leeren Wörter über Σ. Eine **(formale) Sprache** (über Σ) ist eine Teilmenge von Σ^*. Das **Komplement** $\overline{L}$ einer Sprache $L \subseteq \Sigma^*$ ist definiert als $\overline{L} := \Sigma^* - L$. Ist $|\Sigma| = 1$, so bezeichnen wir L als **unäre Sprache**.

Hinweis 2.1 Ein häufiger Fehler ist die Annahme, dass eine unendliche Sprache unendlich lange Wörter enthalten muss. Diese Annahme enthält gleich zwei Fehler:

1. Im Kontext dieser Vorlesung besteht jedes Wort aus endlich vielen Buchstaben. Es ist ist zwar möglich, unendliche Wörter zu definieren, aber wir werden uns hier ausschließlich mit *endlich langen* Wörtern befassen.

2. Eine Sprache ist unendlich, wenn sie *unendlich viele* Wörter enthält. Diese Wörter können beliebig lang werden, aber trotzdem ist jedes dieser Wörter von endlicher Länge.

Für jedes Wort $w \in \Sigma^*$ bezeichnet $|w|$ die **Länge** von w (also die Zahl seiner Buchstaben). Es gilt $|\varepsilon| = 0$. Für jedes $w \in \Sigma^*$ und jedes $a \in \Sigma$ bezeichnet $|w|_a$ die **Anzahl der Vorkommen** von a in w.

Seien $u = u_1 \cdots u_m$ und $v = v_1 \cdots v_n$ $(m, n \in \mathbb{N})$ Wörter über Σ. Die **Konkatenation** $u \cdot v$ von u und v ist definiert als $u \cdot v = u_1 \cdots u_m v_1 \cdots v_n$. Als Konvention vereinbaren wir, dass der Konkatenationspunkt $\cdot$ auch weggelassen werden kann, also gilt $uv = u \cdot v$.

Sei $w = w_1 \cdots w_n$ $(n \in \mathbb{N})$ ein Wort über Σ. Wir definieren die Menge $\text{prefix}(w)$ aller **Präfixe** von w als

$$\begin{aligned} \text{prefix}(w) &:= \{w_1 \cdots w_i \mid 1 \leq i \leq n\} \cup \{\varepsilon\} \\ &= \{u \in \Sigma^* \mid \text{ es existiert ein } v \in \Sigma^* \text{ mit } uv = w\}. \end{aligned}$$

Ein Präfix $p \in \text{prefix}(w)$ ist ein **echtes Präfix** (von w), wenn $|p| < |w|$ ist (also $p \neq w$). Analog dazu definieren wir die Menge $\text{suffix}(w)$ der **Suffixe** von w als

$$\begin{aligned} \text{suffix}(w) &:= \{w_i \cdots w_n \mid 1 \leq i \leq n\} \cup \{\varepsilon\}, \\ &= \{v \in \Sigma^* \mid \text{ es existiert ein } u \in \Sigma^* \text{ mit } uv = w\}, \end{aligned}$$

und $s \in \text{suffix}(w)$ ist ein **echtes Suffix** (von w) wenn $|s| < |w|$. Es gilt also: Jedes Wort ist ein Präfix und Suffix von sich selbst (aber kein echtes), und ε ist echtes Präfix und echtes Suffix jedes Wortes außer ε.

Beispiel 2.2 Sei $\Sigma := \{\texttt{a}, \texttt{b}, \texttt{c}\}$ und $w := \texttt{abcab}$. Dann ist

$$\begin{aligned} \text{prefix}(w) &= \{\texttt{abcab}, \texttt{abca}, \texttt{abc}, \texttt{ab}, \texttt{a}, \varepsilon\}, \\ \text{suffix}(w) &= \{\texttt{abcab}, \texttt{bcab}, \texttt{cab}, \texttt{ab}, \texttt{b}, \varepsilon\}. \end{aligned}$$ ◊

Für jedes Wort $w \in \Sigma^*$ bezeichnet w^R das Wort, das entsteht, wenn w rückwärts gelesen wird. Ist also $w = w_1 w_2 \cdots w_n$ (mit $n \in \mathbb{N}$ und $w_i \in \Sigma$), so ist $w^R = w_n \cdots w_2 w_1$. Wir nennen diese Operation auch den **Reverseral-Operator** ^{R}.

Das **Shuffle-Produkt** zweier Wörter $x, y \in \Sigma^*$ ist die Wortmenge

$$\text{shuffle}(x,y) := \big\{x_1 y_1 x_2 y_2 \cdots x_n y_n \mid n \in \mathbb{N}, x = x_1 \cdots x_n, y = y_1 \cdots y_n \\ \text{mit } x_1, \ldots, x_n, y_1, \ldots, y_n \in \Sigma^*\big\}.$$

Weniger formal gesehen enthält $\text{shuffle}(x,y)$ alle Wörter, die durch Ineinanderschieben von x und y erzeugt werden, genauso wie beim Zusammenschieben von zwei Stapeln mit Spielkarten (daher auch der Name).

Beispiel 2.3 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. Es gilt:

$$\begin{aligned} \text{shuffle}(\mathtt{ab}, \mathtt{cd}) &= \{\mathtt{abcd}, \mathtt{acbd}, \mathtt{acdb}, \mathtt{cabd}, \mathtt{cadb}, \mathtt{cdab}\}, \\ \text{shuffle}(\mathtt{ab}, \mathtt{ba}) &= \{\mathtt{abab}, \mathtt{abba}, \mathtt{baab}, \mathtt{baba}\}, \\ \text{shuffle}(\mathtt{ab}, \mathtt{baa}) &= \{\mathtt{abbaa}, \mathtt{ababa}, \mathtt{abaab}, \mathtt{babaa}, \mathtt{baaba}, \mathtt{baaab}\}. \end{aligned}$$ ◊

Unter einer **n-stelligen Operation auf Sprachen** ($n \in \mathbb{N}$) verstehen wir eine Funktion, die n Sprachen auf eine Sprache abbildet. Beispiele für zweistellige Operationen sind die bekannten Mengenoperationen Vereinigung ($\cup$), Schnitt ($\cap$) und Mengendifferenz ($-$), eine einstellige Operation ist die Komplementbildung $\overline{L}$. Die vorgestellten Operationen auf Wörtern lassen sich leicht zu Operationen auf Sprachen erweitern.

Wir beginnen mit der Konkatenation: Sei $L \subseteq \Sigma^*$ eine Sprache. Dann ist

$$L_1 \cdot L_2 := \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}.$$

Wir erhalten also die Sprache aller Wörter, die sich aus Konkatenation eines Wortes aus L_1 mit einem Wort aus L_2 bilden lassen.

Beispiel 2.4 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$, sowie $L_1 := \{\mathtt{a}, \mathtt{ab}\}$ und $L_2 := \{\mathtt{a}, \mathtt{b}\}$. Dann ist

$$\begin{aligned} L_1 \cdot L_2 &= \{\mathtt{a}, \mathtt{ab}\} \cdot \{\mathtt{a}, \mathtt{b}\} \\ &= \{\mathtt{a}, \mathtt{ab}\} \cdot \{\mathtt{a}\} \cup \{\mathtt{a}, \mathtt{ab}\} \cdot \{\mathtt{b}\} \\ &= \{\mathtt{aa}, \mathtt{aba}\} \cup \{\mathtt{ab}, \mathtt{abb}\} \\ &= \{\mathtt{aa}, \mathtt{ab}, \mathtt{aba}, \mathtt{abb}\}. \end{aligned}$$

Sei $L_3 := \{\mathtt{a}\}$, $L_4 := \{\mathtt{a}\}^*$. Dann ist

$$\begin{aligned} L_3 \cdot L_4 &= \{\mathtt{a}\} \cdot \{\mathtt{a}\}^* \\ &= \{\mathtt{a}\} \cdot \{\mathtt{a}^i \mid i \in \mathbb{N}\} \\ &= \{\mathtt{a}^{1+i} \mid i \in \mathbb{N} \\ &= \{\mathtt{a}^j \mid j \in \mathbb{N}_{>0}\} \\ &= \{\mathtt{a}\}^+ \end{aligned}$$

Außerdem gilt

$$L \cdot \emptyset = \emptyset \cdot L = \emptyset,$$
$$L \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L = L$$

für alle Sprachen L (über jedem Alphabet). ◊

Nach dem gleichen Prinzip lässt sich das **Shuffle-Produkt** auf Sprachen erweitern. Seien $L_1, L_2 \subseteq \Sigma^*$ zwei beliebige Sprachen. Dann ist

$$\text{shuffle}(L_1, L_2) := \bigcup_{x \in L_1, y \in L_2} \text{shuffle}(x, y).$$

Ähnlich wird mit prefix und suffix verfahren: Sei $L \subseteq \Sigma^*$ eine Sprache. Dann ist

$$\text{prefix}(L) := \bigcup_{w \in L} \text{prefix}(w),$$
$$\text{suffix}(L) := \bigcup_{w \in L} \text{suffix}(w).$$

Die Menge der Präfixe (Suffixe) von L ist also genau die Menge aller Präfixe (Suffixe) von Wörtern aus L.

Beispiel 2.5 Seien $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und $L_1 := \{\mathtt{abc}, \mathtt{cc}\}$. Dann ist

$$\begin{aligned}\text{prefix}(L_1) &= \text{prefix}(\mathtt{abc}) \cup \text{prefix}(\mathtt{cc}) \\ &= \{\mathtt{abc}, \mathtt{ab}, \mathtt{a}, \varepsilon\} \cup \{\mathtt{cc}, \mathtt{c}, \varepsilon\} \\ &= \{\varepsilon, \mathtt{a}, \mathtt{c}, \mathtt{ab}, \mathtt{cc}, \mathtt{abc}\}.\end{aligned}$$

Sei nun $L_2 := \{\mathtt{a}^i\mathtt{b} \mid i \in \mathbb{N}_{>0}\}$. Dann ist

$$\begin{aligned}\text{prefix}(L_2) =& \{\mathtt{a}^i\mathtt{b} \mid i \in \mathbb{N}_{>0}\} && \text{(jedes Wort aus } L_2 \text{ ist in der Menge)} \\ &\cup \{\mathtt{a}^i \mid i \in \mathbb{N}_{>0}\} && \text{(Abschneiden von } \mathtt{b}\text{)} \\ &\cup \{\mathtt{a}^j \mid j \in \mathbb{N}\} && \text{(Abschneiden von } \mathtt{b} \text{ und bel. vielen } \mathtt{a}\text{)} \\ =& \{\mathtt{a}^i\mathtt{b} \mid i \in \mathbb{N}_{>0}\} \cup \{\mathtt{a}^j \mid j \in \mathbb{N}\}.\end{aligned}$$

Außerdem gilt $\text{prefix}(L) \supseteq L$ und $\text{suffix}(L) \supseteq L$ für alle Sprachen L über allen Alphabeten. ◊

Nach dem gleichen Prinzip können wir den **Reversal-Operator** ^{R} von Wörtern zu Sprachen erweitern: Sei $L \subseteq \Sigma^*$ eine beliebige Sprache. Dann ist

$$L^R := \{w^R \mid w \in L\}.$$

Um eine Sprache umzudrehen, drehen wir also einfach jedes ihrer Wörter um.

Beispiel 2.6 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{\mathtt{a}, \mathtt{ab}, \mathtt{abb}\}$. Dann ist $L^R = \{\mathtt{a}, \mathtt{ba}, \mathtt{bba}\}$. ◊

Analog zum Reversal-Operator lässt sich unsere Definition von Σ^n zu einer Operation auf Sprachen erweitern: Sei $L \subseteq \Sigma^*$ eine beliebige Sprache. Dann ist

$$L^0 := \{\varepsilon\} \qquad \text{und}$$
$$L^{n+1} := L \cdot L^n$$

für alle $n \in \mathbb{N}$. Die Sprache L^n enthält also genau die Wörter, die sich durch Konkatenation von n Wörtern aus L bilden lassen. Wir nennen diese Operation **n-fache Konkatenation**. Schließlich verallgemeinern wir auch die Definitionen von Σ^* und Σ^+ zu Operationen auf Sprachen:

$$L^* := \bigcup_{n \in \mathbb{N}} L^n, \qquad L^+ := \bigcup_{n \in \mathbb{N}_{>0}} L^n.$$

Die Operationen L^* und L^+ heißen **Kleene-Stern** und **Kleene-Plus**[3].

Beispiel 2.7 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und $L := \{\mathtt{a}, \mathtt{bc}\}$. Dann ist:

$$\begin{aligned}
&L^0 = \{\varepsilon\},\\
&L^1 = L \cdot L^0 = L \cdot \{\varepsilon\} = L = \{\mathtt{a}, \mathtt{bc}\},\\
&L^2 = L \cdot L^1 = L \cdot L = \{\mathtt{aa}, \mathtt{abc}, \mathtt{bca}, \mathtt{bcbc}\},\\
&L^3 = L \cdot L^2 = \{\mathtt{aaa}, \mathtt{abc}, \mathtt{abca}, \mathtt{abcbc}, \mathtt{bcaa}, \mathtt{bcabc}, \mathtt{bcbc}, \mathtt{bcbcbc}\},\\
&\quad\vdots
\end{aligned}$$

Die Sprache L^* enthält nun alle Wörter, die man durch Konkatenation beliebig vieler Vorkommen von $\mathtt{a}$ oder $\mathtt{bc}$ bilden kann.

Betrachten wir nun die Sprache $L_2 := \{\mathtt{ab}\}$. Dann ist

$$\begin{aligned}
&(L_2)^0 = \{\varepsilon\},\\
&(L_2)^1 = L_2 \cdot (L_2)^0 = \{\varepsilon\} \cdot L_2 = \{\mathtt{ab}\},\\
&(L_2)^2 = L_2 \cdot (L_2)^1 = L_2 \cdot L_2 = \{\mathtt{abab}\},\\
&(L_2)^3 = L_2 \cdot (L_2)^2 = \{\mathtt{ababab}\},\\
&\quad\vdots\\
&(L_2)^n = \{(\mathtt{ab})^n\}.
\end{aligned}$$

Wir stellen also fest: $(L_2)^* = \{(\mathtt{ab})^n \mid n \in \mathbb{N}\}$ und $(L_2)^+ = \{(\mathtt{ab})^n \mid n \in \mathbb{N}_{>0}\}$. ◊

Eine weitere zweistellige Operation auf Sprachen ist der sogenannte **Rechts-Quotient**, auch **Quotient** genannt. Seien $L_1, L_2 \subseteq \Sigma^*$ zwei beliebige Sprachen. Dann ist der Rechts-Quotient von L_1 mit L_2 definiert als

$$L_1/L_2 := \{x \in \Sigma^* \mid xy \in L_1, y \in L_2\}.$$

[3] Benannt nach Stephen Cole Kleene

Beispiel 2.8 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$, und sei

$$L_1 := \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\},$$
$$L_2 := \{\mathtt{b}^i \mid i \in \mathbb{N}\}.$$

Dann enthält L_1/L_2 alle Wörter, die entstehen, indem man von einem Wort aus L_1 von rechts ein Wort aus L_2 abtrennt. Es gilt also: $L_1/L_2 = \{\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N}, i \geq j\}$. ◊

2.2.1 Wortproblem

In der Informatik werden formale Sprachen zur Modellierung verwendet. Ein Problem, das dabei eine wichtige Rolle spielt, ist das sogenannte **Wortproblem**. Das Wortproblem einer Sprache $L \subseteq \Sigma^*$ ist das folgende Entscheidungsproblem:

WORTPROBLEM VON L

Eingabe: Ein Wort $w \in \Sigma^*$.
Frage: Ist $w \in L$?

Ein einfaches Beispiel für eine Anwendung eines Wortproblems ist ein Mailfilter, der eine Liste von unerwünschten Emailadressen hat, die ausgefiltert werden sollen. Da Emailadressen Wörter sind, ist die Menge der zu filternden Emailadressen eine Sprache L_{filter}. Bei jeder eingehenden Mail muss der Mailfilter nun entscheiden, ob die Absenderadresse der Mail zur Sprache L_{filter} gehört – dazu muss das Wortproblem von L_{filter} gelöst werden.

Wir werden im Verlauf der Vorlesungen eine Vielzahl von Möglichkeiten kennenlernen, Sprachen zur Modellierung zu benutzen. Dabei steht jeweils für die Anwendung der erstellten Modelle das Wortproblem im Hintergrund.

2.3 Klassen von Sprachen

Eine Menge von Sprachen wird auch als **Klasse** bezeichnet. Ein Beispiel für eine Klasse von Sprachen ist die Klasse FIN aller endlichen Sprachen:

$$\mathrm{FIN}_\Sigma := \{L \subset \Sigma^* \mid L \text{ ist endlich}\},$$
$$\mathrm{FIN} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{FIN}_\Sigma\,.$$

Eigentlich könnte man nun beliebige Mengen von Sprachen als Klassen bezeichnen. Aber im Lauf der Vorlesung werden wir den Begriff „Klasse“ vor allem für solche Mengen verwenden, deren Sprachen bestimmte Gemeinsamkeiten haben (so wie zum Beispiel alle Sprachen in FIN endlich sind).

Durch diese Gemeinsamkeit in der Definition lassen sich auch andere gemeinsame Eigenschaften ableiten. Zum Beispiel ist das Resultat der Vereinigung zweier endlicher Sprachen ebenfalls immer endlich. Wir sagen: „FIN ist unter Vereinigung abgeschlossen“.

Indem man die **Abschlusseigenschaften** einer Sprachklasse unter verschiedenen Operationen untersucht kann man sich einen formalen Werkzeugkasten erstellen, der einem die Arbeit mit der Klasse deutlich erleichtert. Für endliche Sprachen sind diese Beobachtungen noch nicht besonders tiefgründig; später werden wir die Abschlusseigenschaften von spannenderen Sprachklassen untersuchen.

Zuerst formalisieren wir den Begriff der Abgeschlossenheit: Sei Op eine n-stellige ($n \in \mathbb{N}_{>0}$) Operation auf Sprachen. Eine Klasse $\mathcal{C}$ von Sprachen heißt **abgeschlossen** unter Op, wenn für alle Sprachen $L_1, \ldots, L_n \in \mathcal{C}$ auch stets $\mathrm{Op}(L_1, \ldots, L_n) \in \mathcal{C}$ gilt. Anschaulich heißt das: Unabhängig davon, welche Sprachen aus der Klasse $\mathcal{C}$ mit der Operation Op „bearbeiten“, es kommt immer eine Sprache aus der Klasse $\mathcal{C}$ heraus.

Für die Klasse der endlichen Sprachen können wir leicht Folgendes feststellen:

Lemma 2.9 *Die Klasse* FIN *ist*

1. *abgeschlossen unter* $\cup$ *(Vereinigung),*
2. *abgeschlossen unter* prefix *und* suffix*,*
3. *abgeschlossen unter* $\cap$ *(Schnitt),*
4. *abgeschlossen unter* $-$ *(Mengendifferenz),*
5. *abgeschlossen unter* ^{R} *(Reversal-Operator),*
6. *abgeschlossen unter* $\cdot$ *(Konkatenation),*
7. *abgeschlossen unter* shuffle *(Shuffle-Produkt),*
8. *für jedes* $n \in \mathbb{N}$ *abgeschlossen unter* L^n *(n-fache Konkatenation),*
9. *nicht abgeschlossen unter* $\overline{L}$ *(Komplementierung*[4]*),*
10. *nicht abgeschlossen unter* $^+$ *und* * *(Kleene + und Kleene ∗).*

Beweis: *Zu 1:* Seien $L_1, L_2 \in \mathrm{FIN}$. Dann ist L_1 eine endliche Sprache über einem Alphabet Σ_1, und L_2 eine endliche Sprache über einem Alphabet Σ_2. Somit ist $L := L_1 \cup L_2$ eine endliche Sprache über dem Alphabet $\Sigma := \Sigma_1 \cup \Sigma_2$. Es gilt $L \in \mathrm{FIN}_\Sigma$, und somit $L \in \mathrm{FIN}$.

Zu 2: Sei $L \in \mathrm{FIN}$. Dann existiert ein Alphabet Σ mit $L \in \mathrm{FIN}_\Sigma$, und L ist endlich. Allgemein gilt, dass jedes Wort w (einer beliebigen Sprache) nicht mehr als $|w| + 1$ Präfixe (oder Suffixe) haben kann. Das heißt insbesondere, dass jedes Wort $w \in L$ nur endliche viele Präfixe (und Suffixe) haben kann, also ist $\mathrm{prefix}(L)$ (und $\mathrm{suffix}(L)$) eine Vereinigung endlich vieler endlicher Mengen und muss damit selbst endlich sein. Also ist $\mathrm{prefix}(L) \in \mathrm{FIN}_\Sigma$ und somit $\mathrm{prefix}(L) \in \mathrm{FIN}$ (und analog $\mathrm{suffix}(L) \in \mathrm{FIN}$).

Zu 3, 4, 5, 6, 7 und 8: Übung.

Zu 9: Für unser Gegenbeispiel können wir hier eine beliebige Sprache aus einem beliebigen FIN_Σ wählen, da Komplementierung aus einer endlichen Sprache stets eine unendliche Sprache macht. Der Vollständigkeit halber formal: Sei Σ ein beliebiges Alphabet

[4] Für die Komplementbildung einer Sprache $L \in \mathrm{FIN}$ können wir jedes Σ wählen, für das $L \subseteq \Sigma^*$ gilt.

und sei L eine beliebige Sprache mit $L \in \text{FIN}_\Sigma$. Also ist L endlich, und da Σ^* unendlich ist, muss $\overline{L} = \Sigma^* - L$ ebenfalls unendlich sein. Somit gilt $\overline{L} \notin \text{FIN}$.

Zu 10: Auch hier ist es eigentlich egal, welche endliche Sprache wir für unser Gegenbeispiel nehmen, solange diese mindestens ein Wort (außer ε) enthält: Durch Anwendung von $^+$ oder * wird daraus immer eine unendliche Sprache. Formaler: Sei $\Sigma = \{\mathtt{a}\}$ und $L = \{\mathtt{a}\}$. Es gilt: $L \in \text{FIN}_\Sigma$, also $L \in \text{FIN}$. Aber $L^+ = \{a\}^+$ ist eine unendliche Sprache; somit gilt weder FIN_Σ, noch kann ein anderes Alphabet Σ' mit $L \in \text{FIN}_{\Sigma'}$ existieren. Also gilt $L^+ \notin \text{FIN}$. (Der Fall für * verläuft analog.) □

2.4 Aufgaben

Aufgabe 2.1 Sei Σ ein Alphabet.

- Beweisen Sie die folgende Behauptung: Seien $L_1, L_2 \subseteq \Sigma^*$. Ist $\varepsilon \in L_1$, dann gilt $L_2 \subseteq (L_1 \cdot L_2)$. Ist $\varepsilon \in L_2$, dann gilt $L_1 \subseteq (L_1 \cdot L_2)$.
- Geben Sie Sprachen $L_1, L_2 \subseteq \Sigma^*$ an, so dass
  1. $\varepsilon \in L_1$, $(L_1 \cdot L_2) \neq L_2$,
  2. $\varepsilon \in L_1$, $(L_1 \cdot L_2) = L_2$.

Aufgabe 2.2 Geben Sie ein Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$ an mit $\text{prefix}(L) = L$.

Aufgabe 2.3 Geben Sie ein Alphabet Σ und eine Sprache $L \subseteq \Sigma^*$ an, für die sowohl $\text{prefix}(L) \neq L$ als auch $\text{suffix}(L) \neq L$, aber gleichzeitig auch $\text{prefix}(L) = \text{suffix}(L)$.

Aufgabe 2.4 Zeigen Sie: Für jedes Alphabet Σ und jede Sprache $L \subseteq \Sigma^*$ gilt:

$$\varepsilon \in L \text{ genau dann wenn } L^+ = L^*.$$

Aufgabe 2.5 Beweisen Sie die Teile 3, 4, 5, 6, 7 und 8 von Lemma 2.9.

2.5 Bibliographische Anmerkungen

Dieser Abschnitt ist momentan nur ein Platzhalter. In Kürze werden hier einige Kommentare zu den verwendeten Quellen und weiterführendem Lesematerial zu finden sein.

3 Reguläre Sprachen und endliche Automaten

Dieses Kapitel beginnt mit einer genaueren Betrachtung der deterministischen endlichen Automaten und der durch sie definierten regulären Sprachen. Neben der Frage, wie diese minimiert werden können, lernen wir verschiedene Techniken kennen, um mit Automaten andere Automaten zu simulieren.

Danach befassen wir uns mit anderen Darstellungsformen regulärer Sprachen: Wir betrachten nichtdeterministische endliche Automaten, reguläre Ausdrücke und reguläre Grammatiken und beweisen, dass alle diese Modelle die gleiche Ausdrucksstärke haben.

3.1 Deterministische Endliche Automaten

Eines der wichtigsten Modelle in dieser Vorlesung ist der deterministische endliche Automat, kurz DFA[5]:

Definition 3.1 Ein **deterministischer endlicher Automat (DFA)** A über einem Alphabet Σ wird definiert durch:

1. eine nicht-leere, endliche Menge Q von **Zuständen**,
2. eine partielle Funktion $\delta : Q \times \Sigma \to Q$ (die **Übergangsfunktion**),
3. einen Zustand $q_0 \in Q$ (der **Startzustand**),
4. eine Menge $F \subseteq Q$ von **akzeptierenden Zuständen**.

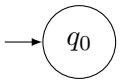
Wir schreiben dies als $A := (\Sigma, Q, \delta, q_0, F)$. Ist δ eine totale Funktion, so nennen wir A **vollständig**.

DFAs lassen sich auch als gerichtete Graphen darstellen. Dabei verwenden wir die folgende Konvention für die Darstellung eines DFA $A = (\Sigma, Q, \delta, q_0, F)$:

- Jeder Zustand $q \in Q$ wird durch einen mit q beschrifteten Knoten dargestellt:

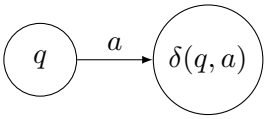
[5] Diese Abkürzung beruht auf der englischen Bezeichnung *deterministic finite automaton*. In der deutschsprachigen Literatur wird oft auch die Abkürzung DEA verwendet.

- Der Startzustand q_0 wird durch einen auf ihn zeigenden Pfeil markiert:

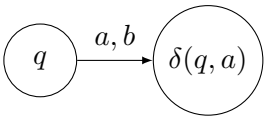


- Jeder akzeptierende Zustand $q \in F$ erhält eine doppelte Umrandung:

- Für jeden Zustand $q \in Q$ und jeden Buchstaben $a \in \Sigma$ gilt: Ist $\delta(q, a)$ definiert, so gibt es in der graphischen Darstellung von A einen Pfeil von q nach $\delta(q, a)$, der mit a beschriftet ist.



  Wenn zu einem Zustand q mehrere unterschiedliche Buchstaben $a, b \in \Sigma$ den gleichen Folgezustand haben (also $\delta(q, a) = \delta(q, b)$ mit $a \neq b$), können wir diese an einer Kante zusammenfassen:



Gelegentlich schreiben wir die Übergangsfunktion δ auch als eine Übergangstabelle. Ein Beispiel dafür finden Sie in Beispiel 3.4.

Die Arbeitsweise eines DFA lässt sich einfach informell beschreiben: Ein DFA bearbeitet ein Wort w, indem er in seinem Startzustand beginnt und w buchstabenweise von links nach rechts abarbeitet. Dazu berechnet er mittels der Übergangsfunktion aus dem aktuellen Zustand und dem aktuellen Buchstaben des Wortes den Folgezustand. Wird auf diese Art ein akzeptierender Zustand erreicht, so wird das Wort akzeptiert. Formal ausgedrückt:

Definition 3.2 Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein DFA. Die Übergangsfunktion δ wird zu einer partiellen Funktion $\delta : Q \times \Sigma^* \to Q$ **erweitert**, und zwar durch die folgende rekursive Definition für alle $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$:

$$\begin{aligned} \delta(q, \varepsilon) &:= q, \\ \delta(q, wa) &:= \delta\left(\delta\left(q, w\right), a\right). \end{aligned}$$

Der DFA A **akzeptiert** ein Wort $w \in \Sigma^*$, wenn $\delta(q_0, w)$ definiert ist und in F liegt. Die von A **akzeptierte Sprache** $\mathcal{L}(A)$ ist definiert als die Menge aller von A

akzeptierten Wörter, also

$$\mathcal{L}(A) := \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

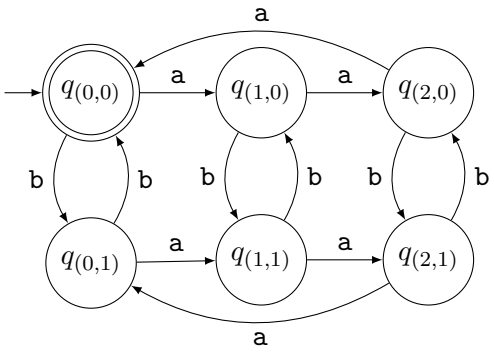
Hinweis 3.3 Ein häufiges Missverständnis bezieht sich auf die Bedeutung des Wortes „endlich“ im Begriff „endlicher Automat“. Endliche Automaten heißen *nicht* deswegen endliche Automaten, weil sie nur endliche Sprachen akzeptieren, sondern weil sie nur *endlich viele Zustände* haben.

Außerdem ist zu beachten, dass jedes Wort in der Sprache eines Automaten endlich ist, da im Rahmen dieser Vorlesung ausschließlich endliche Wörter betrachtet werden (siehe Hinweis 2.1). Daher kann ein Automat beim Lesen eines Wortes zwar beliebig lange Schleifen durchlaufen, trotzdem sind diese Schleifen aber immer nur endlich lang.

Ein Vorteil von DFAs ist, dass das Wortproblem der durch sie definierten Sprachen leicht zu lösen ist. Um zu entscheiden, ob für einen DFA A und ein gegebenes Wort $w \in \Sigma^*$ gilt, dass $w \in \mathcal{L}(A)$, muss man nur w buchstabenweise abarbeiten.

Auch der Beweis, dass ein DFA eine bestimmte Sprache akzeptiert, ist oft zwar ein wenig aufwändig, aber nicht unbedingt schwer. Wir betrachten dazu das folgende Beispiel:

Beispiel 3.4 Gegeben sei der folgende DFA A:



Der Automat ist vollständig, wenn man ihn als DFA über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ auffasst. (Man könnte ihn aber auch als DFA über einem beliebigen Alphabet $\Sigma \supset \{\mathtt{a}, \mathtt{b}\}$ interpretieren, in diesem Fall wäre er nicht vollständig.) An diesem Beispiel sieht man übrigens auch, dass der Startzustand nicht unbedingt q_0 heißen muss. Die Übergangstabelle für A (über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$) lautet wie folgt:

	a	b
$q_{(0,0)}$	$q_{(1,0)}$	$q_{(0,1)}$
$q_{(1,0)}$	$q_{(2,0)}$	$q_{(1,1)}$
$q_{(2,0)}$	$q_{(0,0)}$	$q_{(2,1)}$
$q_{(0,1)}$	$q_{(1,1)}$	$q_{(0,0)}$
$q_{(1,1)}$	$q_{(2,1)}$	$q_{(1,0)}$
$q_{(2,1)}$	$q_{(0,1)}$	$q_{(2,0)}$

Der DFA A akzeptiert die Menge aller Wörter $w \in \{\mathtt{a}, \mathtt{b}\}^*$, die eine gerade Anzahl von $\mathtt{b}$ und eine durch 3 teilbare Anzahl von $\mathtt{a}$ enthalten, also

$$\mathcal{L}(A) = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid |w|_{\mathtt{a}} \bmod 3 = 0 \text{ und } |w|_{\mathtt{b}} \bmod 2 = 0\}.$$

Um dies zu beweisen, zeigen wir die folgende Aussage: Für alle $w \in \Sigma^*$ mit $i := |w|_{\mathtt{a}} \bmod 3$ und $j := |w|_{\mathtt{b}} \bmod 2$ ist $\delta(q_{(0,0)}, w) = q_{(i,j)}$. Wir zeigen dies durch Induktion über den Aufbau von w.

Induktionsanfang: $w = \varepsilon$.
Behauptung: $\delta(q_{(0,0)}, w) = q_{(i,j)}$, für $i := |w|_{\mathtt{a}} \bmod 3$ und $j := |w|_{\mathtt{b}} \bmod 2$.
Beweis: Da $w = \varepsilon$ ist $i = j = 0$. Somit gilt $\delta(q_{(0,0)}, w) = q_{(0,0)} = q_{(i,j)}$.

Induktionsschritt: Seien $w \in \Sigma^*$, $c \in \Sigma$ beliebig.
Induktionsannahme: $\delta(q_{(0,0)}, w) = q_{(i,j)}$, für $i := |w|_{\mathtt{a}} \bmod 3$ und $j := |w|_{\mathtt{b}} \bmod 2$.
Behauptung: $\delta(q_{(0,0)}, wc) = q_{(i',j')}$, für $i' := |wc|_{\mathtt{a}} \bmod 3$ und $j' := |wc|_{\mathtt{b}} \bmod 2$.
Beweis: Es gilt:

$$\begin{aligned} \delta(q_{(0,0)}, wc) &= \delta(\delta(q_{(0,0)}, w), c) && \text{(nach Definition von } \delta) \\ &= \delta(q_{(i,j)}, c) && \text{(nach Induktionsannahme)} \end{aligned}$$

Wir unterscheiden nun zwei Fälle (abhängig vom Wert von c):
1. Fall: Angenommen, $c = \mathtt{a}$. Dann ist $i' = (i + 1) \bmod 3$ und $j' = j$. Außerdem folgt aus der Definition von δ, dass der Automat beim Lesen eines $\mathtt{a}$ vom Zustand $q_{(m,n)}$ (mit $m \in \{0, 1, 2\}$, $n \in \{0, 1\}$) in den Zustand $q_{\left((m+1) \bmod 3, n\right)}$ wechselt. Es gilt also $\delta(q_{(i,j)}, c) = q_{(i',j)} = q_{(i',j')}$. Somit ist die Behauptung in diesem Fall korrekt.
2. Fall: Angenommen, $c = \mathtt{b}$. Dann ist $j' = (j + 1) \bmod 2$, $i' = i$ und außerdem $\delta(q_{(i,j)}, c) = q_{(i,j')} = q_{(i',j')}$. Die Behauptung stimmt also auch in diesem Fall, und der Induktionsbeweis ist abgeschlossen.

Wir wissen nun: Für alle $w \in \Sigma^*$ mit $i := |w|_{\mathtt{a}} \bmod 3$ und $j := |w|_{\mathtt{b}} \bmod 2$ ist $\delta(q_{(0,0)}, w) = q_{(i,j)}$. Da $q_{(0,0)}$ der einzige akzeptierende Zustand ist, akzeptiert A genau die Wörter w, für die $(|w|_{\mathtt{a}} \bmod 3) = (|w|_{\mathtt{b}} \bmod 2) = 0$ gilt. ◊

Zur Erleichterung der Arbeit mit DFAs führen wir außerdem folgende Begriffe ein:

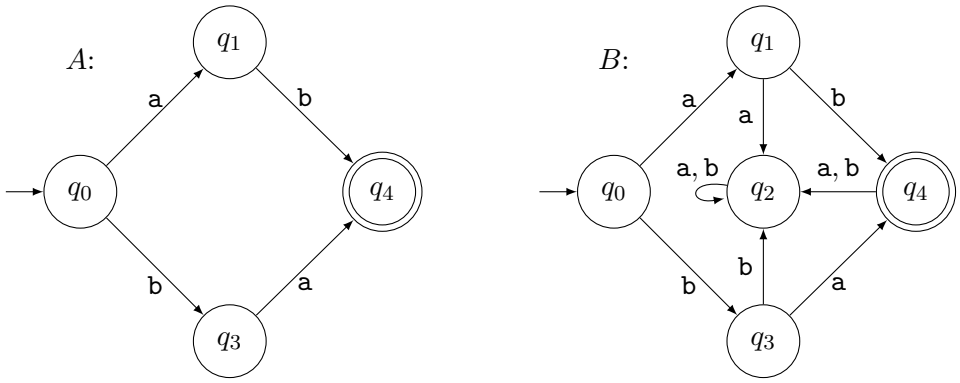
Definition 3.5 Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein DFA. Seien $p, q \in Q$ Zustände. Der Zustand q ist **erreichbar von** p, wenn ein Wort $w \in \Sigma^*$ existiert, so dass $\delta(p, w) = q$.
Ein Zustand $q \in Q$ heißt

- **erreichbar**, wenn q von q_0 erreichbar ist,
- **Sackgasse**, wenn für alle $w \in \Sigma^*$ gilt: $\delta(q, w) \notin F$.

Der DFA A ist **reduziert**, wenn alle Zustände aus Q erreichbar sind.

Eine Sackgasse ist also ein Zustand, von dem aus kein akzeptierender Zustand erreicht werden kann. Im Gegensatz zu unerreichbaren Zuständen sind Sackgassen allerdings nicht nutzlos, da wir durch sie einen unvollständigen Automaten vollständig machen können. Wir betrachten dazu zuerst das folgende Beispiel:

Beispiel 3.6 Über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ seien die DFAs A und B wie folgt definiert:



Es gilt: $\mathcal{L}(A) = \mathcal{L}(B) = \{\mathtt{ab}, \mathtt{ba}\}$. Allerdings ist B vollständig, während A dies nicht ist. Die beiden DFAs sind fast identisch. Der einzige Unterschied ist, dass B zusätzlich die Sackgasse q_2 besitzt. Immer wenn in A ein Übergang nicht definiert ist, geht B stattdessen in q_2 über. So ein Zustand wird daher auch als **Fehlerzustand** oder **Falle** (engl. *trap*) bezeichnet. ◊

Durch Einbauen eines Fehlerzustands kann jeder DFA in einen vollständigen DFA umgewandelt werden:

Lemma 3.7 *Zu jedem unvollständigen DFA A existiert ein vollständiger DFA A' mit $\mathcal{L}(A) = \mathcal{L}(A')$.*

Beweis: Sei $A = (\Sigma, Q, \delta, q_0, F)$. Wir definieren nun $A' := (\Sigma, Q', \delta', q_0, F)$. Hierbei ist $Q' := Q \cup \{q_{trap}\}$, wobei q_{trap} ein neuer Zustand ist mit $q_{trap} \notin Q$, und δ' ist definiert als

$$\delta'(q, a) := \begin{cases} \delta(q, a) & \text{falls } q \neq q_{trap} \text{ und } \delta(q, a) \text{ definiert,} \\ q_{trap} & \text{falls } q = q_{trap} \text{ oder } \delta(q, a) \text{ undefiniert.} \end{cases}$$

Es ist leicht zu sehen, dass A' vollständig ist. Da q_{trap} eine Sackgasse ist, ändert sich nichts an der Erreichbarkeit der akzeptierenden Zustände, daher gilt $\mathcal{L}(A) = \mathcal{L}(A')$. □

Eigentlich genügt es, genau einen Fehlerzustand zu haben. Bei Automaten mit vielen Zuständen kann dadurch allerdings die graphische Darstellung schnell überaus unübersichtlich werden.

3.1.1 (Nicht-)Reguläre Sprachen

Wir verwenden endliche Automaten um eine Sprachklasse zu definieren, mit der wir uns in dieser Vorlesung ausgiebig beschäftigen werden:

Definition 3.8 Sei Σ ein Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt **regulär** wenn ein DFA A über Σ existiert, so dass $L = \mathcal{L}(A)$. Wir bezeichnen die **Klasse aller regulären Sprachen** über dem Alphabet Σ mit REG_Σ, und definieren die Klasse aller regulären Sprachen $\mathrm{REG} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{REG}_\Sigma$.

Reguläre Sprachen sind also genau die Sprachen, die von DFAs akzeptiert werden. Als erste einfache Beobachtung stellen wir fest, dass jede endliche Sprache auch regulär ist:

Satz 3.9 $\mathrm{FIN} \subset \mathrm{REG}$

Beweis: Der Beweis $\mathrm{FIN} \subseteq \mathrm{REG}$ ist eine Übungsaufgabe (Aufgabe 3.1).

Um $\mathrm{FIN} \neq \mathrm{REG}$ zu zeigen, genügt das folgende einfache Gegenbeispiel: Sei $\Sigma := \{\mathtt{a}\}$ und $L = \{\mathtt{a}\}^*$. Dann ist L unendlich, und somit $L \notin \mathrm{FIN}$. Allerdings wird L von dem DFA $A := (\Sigma, \{q_0\}, \delta, q_0, \{q_0\})$ mit $\delta(q_0, \mathtt{a}) = q_0$ akzeptiert. □

Hierbei stellt sich natürlich sofort die Frage, ob alle Sprachen regulär sind. Wie aus den vorherigen Vorlesungen bekannt sein sollte, lautet die Antwort natürlich „nein“. Ein sehr hilfreiches Werkzeug um dies zu beweisen, ist das **Pumping-Lemma** (auch bekannt als *uvw-Lemma*):

Lemma 3.10 (Pumping-Lemma) *Sei Σ ein Alphabet. Für jede reguläre Sprache $L \subseteq \Sigma^*$ existiert eine **Pumpkonstante** $n_L \in \mathbb{N}_{>0}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n_L$ folgende Bedingung erfüllt ist: Es existieren Wörter $u, v, w \in \Sigma^*$ mit*

1. *$uvw = z$,*
2. *$|v| \geq 1$,*
3. *$|uv| \leq n_L$,*
4. *und für alle $i \in \mathbb{N}$ ist $uv^i w \in L$.*

Beweisidee: Zu jeder regulären Sprache L existiert ein DFA A, der L akzeptiert. Sobald Wörter von L länger sind, als die Zahl der Zustände von A, muss A dazu Schleifen verwenden. Diese Schleifen können wir zum Pumpen gebrauchen. □

Beweis: Sei $L \subseteq \Sigma^*$ eine reguläre Sprache. Dann existiert ein DFA $A := (\Sigma, Q, \delta, q_0, F)$ mit $\mathcal{L}(A) = L$. Wir wählen als n_L die Anzahl der Zustände von A, also $n_L := |Q|$.

Sei nun $z \in L$ ein Wort mit $|z| \geq n_L$. Da $\mathcal{L}(A) = L$ wird z von A akzeptiert. Wir bezeichnen die Folge der Zustände, die A dabei durchläuft, mit $q_0, \ldots, q_{|z|}$ (dabei muss $q_{|z|} \in F$ gelten).

Da $|z| + 1 > |z| \geq n_L$ ist, und wir $n_L = |Q|$ gewählt haben, muss (mindestens) einer dieser Zustände (mindestens) zweimal durchlaufen werden. Also existieren ein $m \geq 0$ und ein $k \geq 1$ mit $q_m = q_{m+k}$ und $m + k \leq n_L$.

Sei nun $z = z_1 \cdots z_{|z|}$ (mit $z_i \in \Sigma$ für $1 \leq i \leq |z|$). Wir wählen nun u, v, w wie folgt:

- $u := z_1 \cdots z_m$ (also besteht u aus den ersten m Buchstaben von z; insbesondere gilt $u = \varepsilon$ falls $m = 0$),
- $v := z_{m+1} \cdots z_{m+k}$ (v besteht aus den nächsten k Buchstaben von z), sowie
- $w := z_{m+k+1} \cdots z_{|z|}$ (w besteht aus dem Rest, falls $m + k + 1 > |z|$ gilt $w = \varepsilon$).

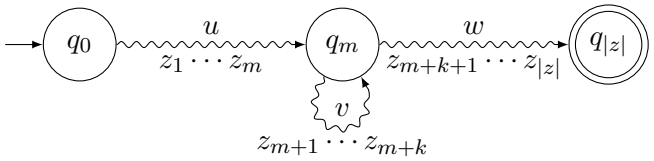
Wir stellen fest

1. $uvw = z$ gilt wegen unserer Wahl von u, v und w,
2. $|v| \geq 1$ gilt wegen $|v| = k$ und $k \geq 1$, und
3. $|uv| \leq n_L$ gilt wegen $|uv| = m + k$ und $m + k \leq n_L$.

Außerdem gilt Folgendes:

$$\delta(q_0, u) = q_m, \qquad \delta(q_m, v) = q_{m+k} = q_m, \qquad \delta(q_{m+k}, w) = q_{|z|}.$$

Daher muss unser DFA A die folgenden Zustände enthalten, die außerdem wie angegeben erreichbar sind:



(Dabei kann $q_0 = q_m$ oder $q_m = q_{|z|}$ gelten, aber das ist kein Problem.) Die mit v beschriftete Schleife von q_m zu q_m kann also ausgelassen oder beliebig oft wiederholt werden, es gilt also $\delta(q_0, uv^i w) = q_{|z|}$ für jedes $i \in \mathbb{N}$, und somit auch $uv^i w \in \mathcal{L}(A)$. □

Das Pumping-Lemma gilt natürlich auch für endliche Sprachen: Da in dem Fall keine Wörter existieren, die länger sind als n_L, ist die Pump-Behauptung trivialerweise erfüllt.

Das Pumping-Lemma kann verwendet werden um zu zeigen, dass eine Sprache *nicht* regulär ist. Das entsprechende Beweisschema betrachten wir anhand der beiden folgenden Beispiele:

Beispiel 3.11 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{xx \mid x \in \Sigma^*\}$. Angenommen, L ist regulär. Sei n_L eine Pumpkonstate für L.

Wir wählen das Wort $z := \mathtt{a}^{n_L}\mathtt{b}\mathtt{a}^{n_L}\mathtt{b}$. Es gilt $z \in L$, also existiert eine Zerlegung $uvw = z$, die das Pumping-Lemma erfüllt. Da $|uv| \leq n_L$ gilt, können u und v kein $\mathtt{b}$ enthalten. Also gilt:

- es existiert ein $m \geq 0$ mit $u = \mathtt{a}^m$,
- es existiert ein $k \geq 1$ mit $v = \mathtt{a}^k$,
- $m + k \leq n_L$,
- $w = \mathtt{a}^{n_L-(m+k)}\mathtt{ba}^{n_L}\mathtt{b}$.

Gemäß Pumping-Lemma ist $uv^iw \in L$ für alle $i \geq 0$, insbesondere auch für $i = 0$. Es gilt:

$$\begin{aligned} uv^0w &= uw \\ &= \mathtt{a}^m\ \mathtt{a}^{n_L-(m+k)}\mathtt{ba}^{n_L}\mathtt{b} \\ &= \mathtt{a}^{m+n_L-(m+k)}\mathtt{ba}^{n_L}\mathtt{b} \\ &= \mathtt{a}^{n_L-k}\mathtt{ba}^{n_L}\mathtt{b}. \end{aligned}$$

Da $k \geq 1$ ist $n_L - k \neq n_L$. Also lässt sich uv^0w nicht in $uv^0w = xx$ zerlegen, somit ist $uv^0w \notin L$. Widerspruch.

Diese Sprache L wird in der Literatur auch als „Copy-Sprache“ oder „copy language“ über Σ bezeichnet und ist eine der bekanntesten nicht-regulären Sprachen. ◊

Beispiel 3.12 Sei $L := \{a^{n^2} \mid n \in \mathbb{N}\}$. Angenommen, L ist regulär. Sei n_L eine Pumpkonstate für L.

Wir wählen das Wort $z := \mathtt{a}^{(n_L)^2}$. Es gilt $z \in L$, also existiert eine Zerlegung $uvw = z$, die das Pumping-Lemma erfüllt. Also gilt:

- es existiert ein $m \geq 0$ mit $u = \mathtt{a}^m$,
- es existiert ein $k \geq 1$ mit $v = \mathtt{a}^k$,
- $m + k \leq n_L$,
- $w = \mathtt{a}^{(n_L)^2-(m+k)}$.

Gemäß Pumping-Lemma ist $uv^iw \in L$ für alle $i \geq 0$, insbesondere auch für $i = 2$. Es gilt:

$$\begin{aligned} uv^2w &= \mathtt{a}^m\ \mathtt{a}^{2k}\ \mathtt{a}^{(n_L)^2-(m+k)} \\ &= \mathtt{a}^{(n_L)^2+k}. \end{aligned}$$

Die kleinste Quadratzahl, die größer ist als $(n_L)^2$ ist $(n_L+1)^2 = (n_L)^2 + 2n_L + 1$. Wegen $k \leq (m + k) \leq n_L$ ist $k < 2n_L + 1$, und somit

$$(n_L)^2 < (n_L)^2 + k < (n_L + 1)^2.$$

Also kann $(n_L)^2 + k$ keine Quadratzahl sein, und somit ist $uv^2w \notin L$. Widerspruch. ◊

Andererseits kann das Pumping-Lemma nicht verwendet werden, um zu zeigen, dass eine Sprache regulär ist. Es gibt nämlich Sprachen, die nicht regulär sind, aber die Pumping-Eigenschaft trotzdem erfüllen:

Beispiel 3.13 Sei $L := \{b^i a^{j^2} \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N}\} \cup \{a^k \mid k \in \mathbb{N}\}$. Diese Sprache erfüllt das Pumping-Lemma (siehe dazu Aufgabe 3.5). ◊

Um zu zeigen, dass diese Sprache nicht regulär ist, brauchen wir weitere Werkzeuge. Dafür gibt es verschiedene Möglichkeiten: Man kann eine verallgemeinerte Form des Pumping-Lemmas benutzen[6], man verwendet geeignete Abschlusseigenschaften (siehe dazu Abschnitt 3.1.2, Beispiel 3.24) oder man argumentiert über die Äquivalenzklassen der Nerode-Relation (siehe dazu Abschnitt 3.1.3).

3.1.2 Komplement- und Produktautomat

Endliche Automaten werden häufig verwendet, um Berechnungen und andere Prozesse zu simulieren. Dafür gibt es einige verbreitete Simulationstechniken, von denen wir in diesem Abschnitt zwei kennenlernen werden. Wir benutzen sie hier vor allem um DFAs zu konstruieren, die andere DFAs mit einer „ähnlichen“ Sprache simulieren. Ein Beispiel dafür ist die Konstruktion, die im Beweis des folgenden Resultats verwendet wird:

Lemma 3.14 *Die Klasse* REG *ist abgeschlossen unter Komplementbildung.*

Beweis: Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine reguläre Sprache. Dann existiert ein DFA $A := (\Sigma, Q, \delta, q_0, F)$ mit $\mathcal{L}(A) = L$. Wegen Lemma 3.7 können wir ohne Beeinträchtigung der Allgemeinheit annehmen, dass A vollständig ist.

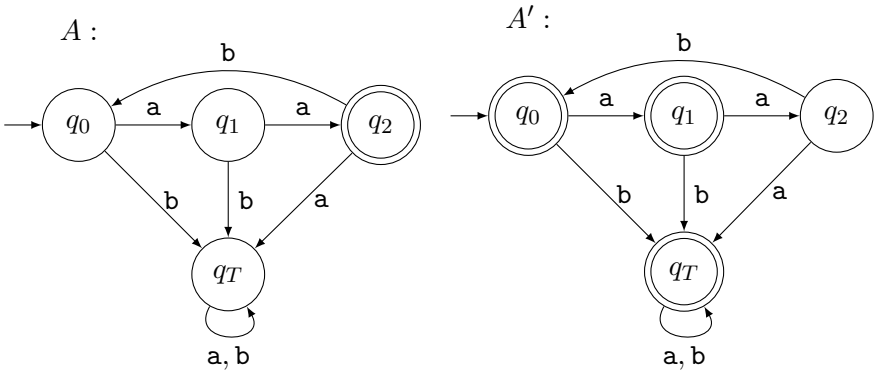
Wir definieren nun einen DFA $A' := (\Sigma, Q, \delta, q_0, F')$ durch $F' := Q - F$ (wir machen also aus jedem akzeptierenden Zustand einen nicht-akzeptierenden, und umgekehrt). Da A' vollständig ist, ist $\delta(q_0, w)$ für alle $w \in \Sigma^*$ definiert. Außerdem gilt: $\delta(q_0, w) \in F$ genau dann wenn $\delta(q_0, w) \notin F'$. Somit ist $w \in \mathcal{L}(A)$ genau dann wenn $w \notin \mathcal{L}(A')$.

Also ist $\mathcal{L}(A') = \Sigma^* - \mathcal{L}(A)$, und somit $\overline{\mathcal{L}(A)} \in$ REG. □

Intuitiv kann man den Beweis von Lemma 3.14 so verstehen, dass der DFA A' den DFA A simuliert, aber genau das Gegenteil akzeptiert. Wir nennen diesen Automaten daher auch den **Komplementautomaten**.

Beispiel 3.15 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $L := \{\mathtt{aa}\} \cdot \{\mathtt{baa}\}^*$. Im folgenden Bild sehen Sie links einen DFA A mit $\mathcal{L}(A) = L$, und rechts den Komplementautomaten A' zu A, es gilt also $\mathcal{L}(A') = \overline{\mathcal{L}(A)}$. Beachten Sie bitte, dass A ein vollständiger Automat ist.

[6] In dieser Vorlesung werden wir uns nicht mit solchen Verallgemeinerungen befassen. Falls Sie neugierig sind: Eine kleine Sammlung verallgemeinerter Versionen des Pumping-Lemmas finden Sie zum Beispiel in Yu [23], Abschnitt 4.1.



◊

Dank Lemma 3.14 können wir unserem „Werkzeugkasten“ eine weitere nützliche Beobachtung hinzufügen:

Korollar 3.16 *Jede kofinite Sprache ist regulär.*

Beweis: Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine kofinite Sprache. Nach Definition muss das Komplement $\overline{L}$ eine endliche Sprache sein (und somit nach Satz 3.9) regulär. Da REG gemäß Lemma 3.14 unter Komplement abgeschlossen ist, ist auch L regulär. □

Durch eine leichte Abwandlung der Konstruktion des Komplementautomaten lässt sich auch Folgendes zeigen:

Korollar 3.17 *Die Klasse* REG *ist abgeschlossen unter der Operation* prefix.

Beweis: Übung 3.7. □

Eine etwas kompliziertere und gleichzeitig mächtigere Simulations-Konstruktion finden wir im Beweis für das folgende Resultat:

Lemma 3.18 *Die Klasse* REG *ist abgeschlossen unter Schnitt.*

Beweisidee: Zu zeigen ist: Zu jedem Paar von regulären Sprachen L_1 und L_2 existiert ein DFA A mit $\mathcal{L}(A) = L_1 \cap L_2$. Wir konstruieren dazu einen DFA A, der gleichzeitig sowohl einen DFA A_1 für L_1 und einen DFA A_2 für L_2 simuliert und akzeptiert, wenn sowohl A_1 als auch A_2 akzeptiert. □

Beweis: Sei Σ ein Alphabet und seien $L_1, L_2 \in \Sigma^*$ reguläre Sprachen. Dann existieren DFAs

$$A_1 := (\Sigma, Q_1, \delta_1, q_{(0,1)}, F_1),$$
$$A_2 := (\Sigma, Q_2, \delta_2, q_{(0,2)}, F_2)$$

mit $\mathcal{L}(A_i) = L_i$ $(i \in \{1, 2\})$. Gemäß Lemma 3.7 nehmen wir an, dass A_1 und A_2 vollständig sind (aus dem gleichen Grund können wir auch fordern, dass beide DFAs über dem gleichen Alphabet definiert sind). Wir definieren nun den DFA $A := (\Sigma, Q, \delta, q_0, F)$ wie folgt:

- $Q := Q_1 \times Q_2$,
  (*Idee:* In einem Zustand von A sind gleichzeitig ein Zustand von A_1 und einer von A_2 gespeichert.)

- $q_0 := (q_{(0,1)}, q_{(0,2)})$,
  (*Idee:* A beginnt die Simulation damit, dass A_1 und A_2 in ihrem jeweiligen Startzustand sind.)

- $F := F_1 \times F_2$,
  (*Idee:* A akzeptiert, wenn A_1 *und* A_2 akzeptieren.)

- $\delta\big((q_1, q_2), a\big) := \big(\delta_1(q_1, a), \delta_2(q_2, a)\big)$ für alle $q_1 \in Q_1$, $q_2 \in Q_2$ und $a \in \Sigma$.
  (*Idee:* A simuliert das Verhalten von A_1 und A_2).

Da A_1 und A_2 vollständig sind, ist auch A vollständig. Außerdem gilt nach Definition von δ

$$\delta(q_0, w) = (\delta_1(q_{(0,1)}, w), \delta_2(q_{(0,2)}, w))$$

für alle $w \in \Sigma^*$. Daher gilt

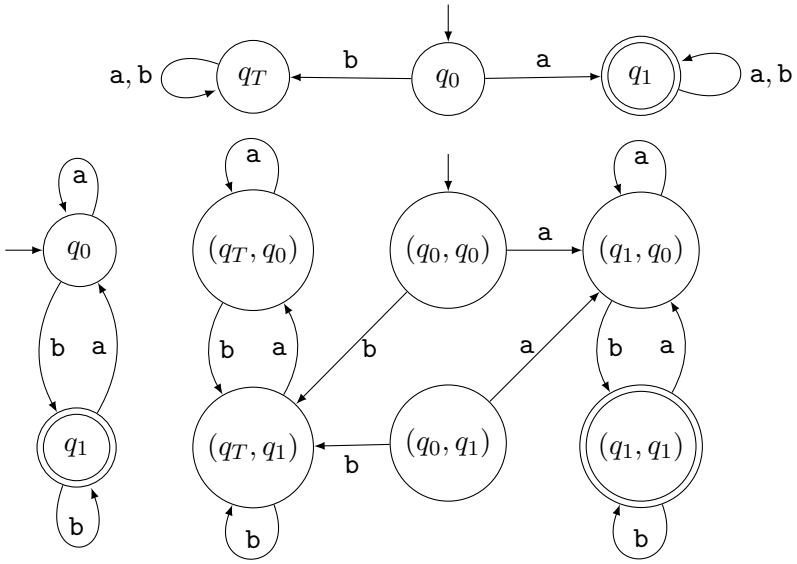
$$\begin{aligned}
& \delta(q_0, w) \in F \\
\Leftrightarrow \quad & \delta(q_0, w) \in F_1 \times F_2 \\
\Leftrightarrow \quad & (\delta_1(q_{(0,1)}, w), \delta_2(q_{(0,2)}, w)) \in F_1 \times F_2 \\
\Leftrightarrow \quad & \delta_1(q_{(0,1)}, w) \in F_1 \text{ und } \delta_2(q_{(0,2)}, w) \in F_2 \\
\Leftrightarrow \quad & w \in L_1 \text{ und } w \in L_2 \\
\Leftrightarrow \quad & w \in L_1 \cap L_2.
\end{aligned}$$

Also ist $\mathcal{L}(A) = L_1 \cap L_2$. Wir stellen fest: $L_1 \cap L_2$ eine reguläre Sprache. Somit ist REG abgeschlossen unter Schnitt. □

Der im Beweis von Lemma 3.18 konstruktierte Automat heißt auch **Produktautomat** (der Name kommt vom in der Definition von Q verwendeten Kreuzprodukt). Wir betrachten nun ein Beispiel:

Beispiel 3.19 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und seien $L_1 := \{\mathtt{a}\} \cdot \Sigma^*$ und $L_2 := \Sigma^* \cdot \{\mathtt{b}\}$ (L_1 ist also die Sprache aller Wörter, die mit dem Buchstaben `a` beginnen, und L_2 die aller Wörter, die auf den Buchstaben `b` enden).

Die folgende Darstellung zeigt einen DFA für L_1 (oben), einen DFA für L_2 (links) und den Produktautomaten für $L_1 \cap L_2$:



Dieses Beispiel zeigt auch, dass die Produktautomatenkonstruktion in Bezug auf die Zahl der Zustände nicht optimal ist: Der Zustand (q_0, q_1) ist nicht erreichbar (und daher überflüssig), und die beiden Sackgassen (q_T, q_0) und (q_T, q_1) können zusammengefasst werden. ◊

Anhand des Produktautomaten können wir zwei weitere Abschlusseigenschaften beweisen:

Korollar 3.20 *Die Klasse* REG *ist abgeschlossen unter Vereinigung und Differenz.*

Beweis: Wir können dieses Resultat auf zwei verschiedene Arten beweisen: Durch eine Modifikation des Produktautomaten aus dem Beweis von Lemma 3.18, oder durch Anwendung von Abschlusseigenschaften.

Variante 1 (modifizierter Produktautomat): Seien $L_1, L_2 \subseteq \Sigma^*$ reguläre Sprachen. Wir konstruieren den Produktautomaten A wie im Beweis zu Lemma 3.18. Der einzige Unterschied ist die Definition von F (die Korrektheit der jeweiligen Konstruktion ist leicht zu zeigen):

- Für $L_1 \cup L_2$ definieren wir $F := (Q_1 \times F_2) \cup (F_1 \times Q_2)$.
- Für $L_1 - L_2$ definieren wir $F := F_1 \times (Q_2 - F_2)$.

Variante 2 (Abschlusseigenschaften): Seien $L_1, L_2 \subseteq \Sigma^*$ reguläre Sprachen. Es gilt:

- $L_1 \cup L_2 = \overline{\overline{L_1} \cap \overline{L_2}}$, und
- $L_1 - L_2 = L_1 \cap \overline{L_2}$.

Da reguläre Sprachen unter Schnitt (Lemma 3.18) und Komplementbildung (Lemma 3.14) abgeschlossen sind, sind auch $L_1 \cup L_2$ und $L_1 - L_2$ reguläre Sprachen. □

Man kann Abschlusseigenschaften verwenden, um zu zeigen, dass Sprachen regulär sind:

Beispiel 3.21 Wir betrachten die folgenden Sprachen:

1. $L_1 := \{\mathtt{ba}\}^* - \{w \in \{\mathtt{a},\mathtt{b}\}^* \mid |w| = 123456789\}$,
2. $L_2 := \{\mathtt{ba}\}^* \cup \{\mathtt{c}\}^{327007}$,

Ohne Abschlusseigenschaften wäre der Nachweis der Regularität dieser Sprachen vergleichsweise mühsam. Wir könnten zwar jeweils einen DFA definieren und (mit einigem langweiligen Schreibaufwand oder „Händewedeln“) sicherstellen, dass er die jeweilige Sprache erzeugt. Aber dank einiger Abschlusseigenschaften ist unser Leben leichter.

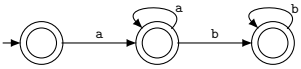
Zu L_1: Ein DFA für $\{\mathtt{ba}\}^*$ ist schnell angegeben, wir können daher davon ausgehen, dass diese Sprache regulär ist. Außerdem gilt $\{w \in \Sigma^* \mid |w| = 123456789\} = \Sigma^{123456789}$. Diese Sprache ist endlich und somit auch regulär (gemäß Satz 3.9). Also ist L_1 die Differenz zweier regulärer Sprachen und somit ebenfalls regulär (da REG abgeschlossen ist unter Differenz, siehe Korollar 3.20.

Zu L_2: Die Sprache $\{\mathtt{ba}\}^*$ ist regulär (siehe Absatz zu L_1). Die Sprache $\{\mathtt{c}\}^{327007}$ besteht aus einem einzigen Wort, sie ist also endlich und somit regulär (gemäß Satz 3.9). Also ist L_2 die Vereinigung zweier regulärer Sprachen und somit nach Korollar 3.20 ebenfalls eine reguläre Sprache. ◊

Abschlusseigenschaften erleichtern aber auch viele Beweise von Nichtregularität. Dazu beginnt man mit der zu untersuchenden Sprache und überführt sie mittels der Abschlusseigenschaften in eine Sprache, von der man bereits weiß, dass sie nicht regulär ist:

Beispiel 3.22 Sei $L := \{w \in \{\mathtt{a},\mathtt{b}\}^* \mid |w|_{\mathtt{a}} \neq |w|_{\mathtt{b}}\}$. (Also ist L die Menge aller Wörter über $\{\mathtt{a},\mathtt{b}\}$, die die gleiche Anzahl von `a` und `b` enthält.) Wir werden zeigen, dass L nicht regulär ist. Dazu nehmen wir erst an, L sein eine reguläre Sprache. Da REG abgeschlossen ist unter Komplementbildung, ist auch

$$\overline{L} = \{w \in \{\mathtt{a},\mathtt{b}\}^* \mid |w|_{\mathtt{a}} = |w|_{\mathtt{b}}\}$$

eine reguläre Sprache. Außerdem wissen wir, dass $\{\mathtt{a}\}^*\{\mathtt{b}\}^*$ eine reguläre Sprache ist. (Einen entsprechenden DFA können wir leicht konstruieren: 

). Da REG unter Schnitt abgeschlossen ist, ist auch die folgende Sprache regulär:

$$\overline{L} \cap \{\mathtt{a}\}^*\{\mathtt{b}\}^* = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}.$$

Dass diese Sprache nicht regulär ist, wissen wir bereits aus der diskreten Modellierung, oder wir zeigen dies einfach mit dem Pumping-Lemma. Widerspruch, also kann L nicht regulär sein. ◊

Beispiel 3.23 Sei $\Sigma := \{\mathtt{a},\mathtt{b}\}$ und $L := \Sigma^* - \mathrm{COPY}_\Sigma$, wobei

$$\mathrm{COPY}_\Sigma := \{xx \mid x \in \Sigma^*\}.$$

Angenommen, L ist regulär. Da die Klasse der reguläre Sprachen unter Komplementbildung abgeschlossen ist, ist dann auch $\overline{L}$ regulär. Aber $\overline{L} = \text{COPY}_\Sigma$, und wir wissen bereits, dass diese Sprache nicht regulär ist (aus Beispiel 3.11). Widerspruch. Also ist L nicht regulär. ◊

Beispiel 3.24 In Beispiel 3.13 haben wir die Sprache

$$L := \left\{ b^i a^{j^2} \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N} \right\} \cup \left\{ a^k \mid k \in \mathbb{N} \right\}$$

kennengelernt und festgestellt, dass diese Sprache die Eigenschaften des Pumping-Lemmas erfüllt. Allerdings haben wir auch behauptet, dass L nicht regulär ist. Eine Möglichkeit, dies zu beweisen, ist die Verwendung von Abschlusseigenschaften, zusammen mit dem Pumping-Lemma.

Angenommen, L ist regulär. Wir definieren

$$\begin{aligned} L' &:= L \cap (\{\texttt{b}\} \cdot \{\texttt{a}\}^*) \\ &= \left\{ ba^{j^2} \mid j \in \mathbb{N} \right\}. \end{aligned}$$

Da $(\{\texttt{b}\} \cdot \{\texttt{a}\}^*)$ regulär ist[7] und reguläre Sprachen abgeschlossen sind unter Schnitt, ist auch L' regulär. Im Gegensatz zu L können wir nun das Pumping-Lemma anwenden um zu zeigen, dass L' nicht regulär ist[8]. Widerspruch, also kann L nicht regulär sein. ◊

Allerdings ist im Umgang mit Abschlusseigenschaften Vorsicht geboten.

Hinweis 3.25 Wir können Abschlusseigenschaften auf die folgenden Arten verwenden:

- Um zu zeigen, dass eine Sprache L regulär ist, beginnen wir mit einer Auswahl von Sprachen, von denen wir wissen, dass sie regulär sind (oder das leicht zeigen können), und bauen uns aus diesen Sprachen unsere Zielsprache L mittels Operationen, unter denen REG abgeschlossen ist.
- Um zu zeigen, dass eine Sprache L *nicht* regulär ist, nehmen wir zuerst an, dass L regulär ist. Dann bearbeiten wir L solange mit Operationen, unter denen REG abgeschlossen ist, bis eine Sprache herauskommt, von der wir wissen, dass sie *nicht* regulär ist. Widerspruch, also kann L nicht regulär sein.

Ein häufiger Fehler ist die beiden Ansätze zu durchmischen, indem man mit einer Sprache L beginnt und sie mit Operationen bearbeitet (unter denen REG abgeschlossen ist), bis man eine reguläre Sprache erhält, um dann abschließend festzustellen, dass L aufgrund der Abschlusseigenschaften von REG eine reguläre Sprache sein muss. Das ist falsch, wie die folgenden Beispiele illustrieren:

[7] Einfache Übung: Warum ist $(\{\texttt{b}\} \cdot \{\texttt{a}\}^*)$ regulär?

[8] Dies sei Ihnen als Übung überlassen. In Beispiel 3.108 werden wir eine elegantere Methode kennenlernen.

Beispiel 3.26 Sei Σ ein beliebiges Alphabet und $L \subseteq \Sigma^*$ eine nicht-reguläre Sprache. Dann ist $L \cup \Sigma^* = \Sigma^*$. Wir wissen: $\Sigma^* \in$ REG, und REG ist abgeschlossen unter Vereinigung. Aber auch wenn das Resultat der Vereinigung von L und Σ^* eine reguläre Sprache ist, können wir daraus nicht schließen, dass L regulär ist.

Ein weiteres Beispiel: Sei Σ ein beliebiges Alphabet, und $L \subseteq \Sigma^*$ eine nicht-reguläre Sprache. Dann ist auch $\overline{L} \notin$ REG (falls $\overline{L} \in$ REG gelten würde, würde $L \in$ REG folgen, da REG abgeschlossen unter Komplement). Aber es gilt nach Definition: $L \cup \overline{L} = \Sigma^*$. $\Diamond$

Wir werden im weiteren Verlauf der Vorlesung noch andere Abschlusseigenschaften kennenlernen.

3.1.3 Die Nerode-Relation und der Äquivalenzklassenautomat

In diesem Abschnitt beschäftigen wir uns mit einem der wichtigsten Resultate zu regulären Sprachen. Neben einem weiteren Werkzeug zum Nachweis von Regularität und Nicht-Regularität werden wir dieses Material im Anschluss verwenden, um DFAs zu minimieren. Bevor wir mit der Definition und Betrachtung der Nerode-Relation[9] beginnen, führen wir zuerst den Begriff der Ableitung einer Sprache ein:

Definition 3.27 Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Für jedes Wort $x \in \Sigma^*$ definieren wir die **Ableitung von L nach** x, geschrieben $D_x L$, als

$$D_x L := \{z \mid xz \in L\}.$$

Die Ableitung von L nach x ist also die Menge aller Wörter, die wir an x anhängen können, um wieder ein Wort aus L zu erhalten. Bitte beachten Sie: x kann ein beliebiges Wort sein und muss nicht aus L gewählt werden. Außerdem kann $D_x L = \emptyset$ gelten. Weiter unten werden wir ein paar Beispiel zu Ableitungen betrachten.

Definition 3.28 Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Die **Nerode-Relation $\equiv_L$ (von L)** ist eine Relation $\equiv_L \subseteq \Sigma^* \times \Sigma^*$, die wie folgt definiert ist:

Für alle $x, y \in \Sigma^*$ gilt $x \equiv_L y$ genau dann, wenn $D_x L = D_y L$.

Mit anderen Worten: Es gilt $x \equiv_L y$ wenn $(xz \in L \Leftrightarrow yz \in L)$ für alle $z \in \Sigma^*$.

Wie sich leicht überprüfen lässt, ist $\equiv_L$ für jede Sprache L eine Äquivalenzrelation:

Proposition 3.29 *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine Sprache. Dann ist $\equiv_L$ eine Äquivalenzrelation.*

Beweis: Zu zeigen ist: Die Relation $\equiv_L$ ist reflexiv, transitiv, und symmetrisch.

1. Reflexivität: $x \equiv_L x$ gilt für alle $x \in \Sigma^*$ nach Definition, da $D_x L = D_x L$.

[9] Benannt nach Anil Nerode.

2. Transitivität: Seien $x_1, x_2, x_3 \in \Sigma^*$ mit $x_1 \equiv_L x_2$ und $x_2 \equiv_L x_3$. Dann gelten $D_{x_1}L = D_{x_2}L$ und $D_{x_2}L = D_{x_3}L$, und somit auch $D_{x_1}L = D_{x_3}L$. Wir stellen fest: $x_1 \equiv_L x_3$, also ist $\equiv_L$ transitiv.

3. Symmetrie: Aus $x \equiv_L y$ folgt $y \equiv_L x$ nach Definition, denn $D_xL = D_yL$ gdw. $D_yL = D_xL$. □

Da die Nerode-Relation $\equiv_L$ eine Äquivalenzrelation auf Σ^* ist, können wir sie zur Definition von Äquivalenzklassen verwenden:

Definition 3.30 Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Für jedes Wort $x \in \Sigma^*$ ist $[x]_{\equiv_L}$, die **Äquivalenzklasse von x (in Bezug auf $\equiv_L$)**, definiert als

$$[x]_{\equiv_L} := \{y \in \Sigma^* \mid x \equiv_L y\}.$$

Ist klar, auf welche Sprache L wir uns beziehen, können wir auch $[x]$ anstelle von $[x]_{\equiv_L}$ schreiben.

Der **Index von** $\equiv_L$, geschrieben index(L), ist die Anzahl der unterschiedlichen Äquivalenzklassen von $\equiv_L$.

Bitte beachten Sie: Genauso wie die Ableitung einer Sprache L ist auch ihre Nerode-Relation $\equiv_L$ auf *allen* Wörtern aus Σ^* definiert, nicht nur auf den Wörtern aus L. Wie wir in einem der folgenden Beispiele sehen werden, kann index(L) Werte zwischen 1 und ∞ annehmen, es gilt also index$(L) \in \mathbb{N}_{>0} \cup \{\infty\}$.

Beim Berechnen der Äquivalenzklassen von $\equiv_L$ kann uns das folgende Lemma helfen:

Lemma 3.31 *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Für alle $a \in \Sigma$ und alle $x, y \in \Sigma^*$ mit $x \equiv_L y$ gilt: $[xa]_{\equiv_L} = [ya]_{\equiv_L}$.*

Beweis: Angenommen, es existieren $x, y \in \Sigma^*$ mit $x \equiv_L y$ und ein $a \in \Sigma$ mit $[xa]_{\equiv_L} = [ya]_{\equiv_L}$. Dann ist $D_{xa}L \neq D_{ya}L$, also existiert ein $z \in \Sigma^*$ mit $z \in (D_{xa}L - D_{ya}L)$ oder $z \in (D_{ya}L - D_{xa}L)$. Ohne Beeinträchtigung der Allgemeinheit nehmen wir an, dass $z \in (D_{xa}L - D_{ya}L)$, also $z \in D_{xa}L$ und $z \notin D_{ya}L$. Also ist $xaz \in L$ und $yaz \notin L$. Daraus folgen unmittelbar $az \in D_xL$ und $az \notin D_yL$, also $x \not\equiv_L y$. Widerspruch. □

Wir wenden uns nun den versprochenen Beispielen zu:

Beispiel 3.32 Wir beginnen mit einem ganz einfachen Beispiel: Sei Σ ein beliebiges Alphabet, und sei $L_1 := \Sigma^*$. Dann gilt $D_xL_1 = \Sigma^*$ für alle $x \in \Sigma^*$, und somit auch $x \equiv_{L_1} y$ für alle $x, y \in \Sigma^*$. Da alle Wörter zueinander äquivalent sind, existiert nur eine einzige Äquivalenzklasse; es gilt also $[\varepsilon] = \Sigma^*$ und index$(L_1) = 1$.

Für die folgenden Beispiele verwenden wir $\Sigma := \{\mathtt{a}, \mathtt{b}\}$.

Sei nun $L_2 := \{\mathtt{aa}\}$. Wir bestimmen die Äquivalenzklassen von $\equiv_{L_2}$ anhand der Ableitungen und beginnen mit $[\varepsilon]$. Es gilt $D_\varepsilon L_2 = \{\mathtt{aa}\}$; da $y \cdot \mathtt{aa} \in L_2$ nur für $y = \varepsilon$ gilt, enthält $[\varepsilon]$ keine anderen Wörter. Ähnliches gilt für $[\mathtt{a}]$ und $[\mathtt{aa}]$, da $D_\mathtt{a}L_2 = \{\mathtt{a}\}$ und $D_\mathtt{aa}L_2 = \{\varepsilon\}$ enthalten diese Klassen ebenfalls keine weiteren Wörter als $\mathtt{a}$ bzw. $\mathtt{aa}$. Die

letzte Klasse ist $[\mathtt{b}]$, mit $D_{\mathtt{b}}L_2 = \emptyset$ (da kein Wort aus L_2 mit $\mathtt{b}$ beginnt). Diese enthält alle restlichen Wörter, also $[\mathtt{b}] = \Sigma^* - \{\varepsilon, \mathtt{a}, \mathtt{aa}\}$. Somit hat $\equiv_{L_2}$ die vier Äquivalenzklassen $[\varepsilon]$, $[\mathtt{a}]$, $[\mathtt{aa}]$ und $[\mathtt{b}]$, und es gilt $\text{index}(L_2) = 4$.

Sei $L_3 := \{\mathtt{a} \cdot \mathtt{b}\}^*$. Dann hat $\equiv_{L_3}$ die folgenden Äquivalenzklassen mit entsprechenden Ableitungen:

$$\begin{aligned} [\varepsilon] &= \{\mathtt{ab}\}^* = \{\varepsilon, \mathtt{ab}, \mathtt{abab}, \ldots\}, & D_\varepsilon L_3 &= \{\mathtt{ab}\}^*, \\ [\mathtt{a}] &= \mathtt{a} \cdot \{\mathtt{ba}\}^* = \{\mathtt{a}, \mathtt{aba}, \mathtt{ababa}, \ldots\}, & D_{\mathtt{a}} L_3 &= \mathtt{b} \cdot \{\mathtt{ab}\}^*, \\ [\mathtt{b}] &= \Sigma^* - ([\varepsilon] \cup [\mathtt{a}]), & D_{\mathtt{b}} L_3 &= \emptyset. \end{aligned}$$

Es gilt also $\text{index}(L_3) = 3$.

Abschließend betrachten wir ein letztes Beispiel: Sei $L_4 := \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Diese Sprache ist keine reguläre Sprache (das kann man zum Beispiel mit dem Pumping-Lemma beweisen). Die Äquivalenzklassen von L_4 lauten wie folgt:

$$\{[\varepsilon], [\mathtt{b}]\} \cup \{[\mathtt{a}], [\mathtt{aa}], [\mathtt{aaa}], \ldots\} \cup \{[\mathtt{ab}], [\mathtt{aab}], [\mathtt{aaab}], \ldots\}$$

Dies zu beweisen ist allerdings ein wenig aufwändig und sei Ihnen als Übung überlassen. Wir betrachten stattdessen zur Veranschaulichung die folgende Übersicht aller Klassen und der dazu gehörenden Ableitungen:

$$\begin{aligned} [\varepsilon] &= \{\varepsilon\}, & D_\varepsilon L_4 &= L_4, \\ [\mathtt{b}] &= \Sigma^* - \big(\text{prefix}(L_4)\big), & D_{\mathtt{b}} L_4 &= \emptyset. \end{aligned}$$

Außerdem ist für jedes $n \in \mathbb{N}_{>0}$

$$\begin{aligned} [\mathtt{a}^n] &= \{\mathtt{a}^n\}, & D_{\mathtt{a}^n} L_4 &= \{\mathtt{a}^i\mathtt{b}^{n+i} \mid i \in \mathbb{N}\}, \\ [\mathtt{a}^n\mathtt{b}] &= \{\mathtt{a}^i\mathtt{b}^{i+1-n} \mid i \in \mathbb{N}, i \geq n\}, & D_{\mathtt{a}^n\mathtt{b}} L_4 &= \{\mathtt{b}^{n-1}\}. \end{aligned}$$

Anschaulicher erklärt: Die Klasse $[\varepsilon]$ enthält nur das leere Wort u, da dies das einzige Wort ist, das vor Wörter $v \in L_4$ geschrieben werden kann und trotzdem zu einem Wort $uv \in L_4$ führt (daher ist $D_\varepsilon L_4 = L_4$). Die Klasse $[\mathtt{b}]$ enthält alle Wörter, die kein Wort aus L_4 sind und auch nicht durch Anhängen eines geeigneten Wortes zu einem Wort aus L_4 gemacht werden können. Daher ist $D_{\mathtt{b}} L_4 = \emptyset$. Die Klasse $[\mathtt{a}^1\mathtt{b}]$ (also $[\mathtt{ab}]$) enthält genau die Wörter aus $L_4 - \{\varepsilon\}$. An diese Wörter kann nur ε angehängt werden, daher ist $D_{\mathtt{a}^1\mathtt{b}} L_4 = \{\mathtt{b}^{1-1}\} = \{\mathtt{b}^0\} = \{\varepsilon\}$. Für $n > 1$ enthält $[\mathtt{a}^n\mathtt{b}]$ genau die Wörter, an die $n-1$ viele $\mathtt{b}$ angehängt werden müssen, um ein Wort aus L_4 zu erhalten. Die Klassen $[\mathtt{a}^n]$ ($n \in \mathbb{N}_{>0}$) enthalten die Wörter, die sowohl mit einer geeigneten Zahl von $\mathtt{b}$, als auch mit geeigneten Wörtern der Form $\mathtt{a}^i\mathtt{b}^{n+i}$ zu einem Wort aus L_4 vervollständigt werden können.

Wir stellen abschließend fest, dass $\text{index}(L_4) = \infty$. ◊

Die Nerode-Relation und Ableitungen sind zwar für beliebige Sprachen definiert; allerdings können wir bei regulären Sprachen noch zusätzliche Aussagen treffen. Insbesondere können wir die Ableitung einer regulären Sprache L nach einem bestimmten Wort direkt aus einem DFA ermitteln, der L akzeptiert:

Lemma 3.33 *Sei Σ ein Alphabet, sei $A := (\Sigma, Q, \delta, q_0, F)$ ein vollständiger DFA und sei $L := \mathcal{L}(A)$. Für jedes $x \in \Sigma^*$ sei der DFA A_x definiert durch $A_x := (\Sigma, Q, \delta, q_x, F)$, wobei $q_x := \delta(q_0, x)$. Dann gilt für alle $x \in \Sigma^*$: $D_x L = \mathcal{L}(A_x)$.*

Beweis: Die beiden DFAs A und A_x sind fast identisch; der einzige Unterschied ist, dass A_x als Startzustand $\delta(q_0, x)$ verwendet. Man kann dies so verstehen: Der DFA A_x simuliert (noch vor Abarbeiten seiner Eingabe) den DFA A beim Einlesen von x. Danach arbeitet A_x seine Eingabe so ab, wie A es tun würde. Es gilt also für alle $z \in \Sigma^*$, dass $\delta(q_x, z) = \delta(q_0, xz)$. Es gilt also

$$\mathcal{L}(A_x) = \{z \mid xz \in \mathcal{L}(A)\} = D_x \mathcal{L}(A).$$

Da $\mathcal{L}(A) = L$ ist somit $\mathcal{L}(A_x) = D_x L$. □

Daraus resultiert unmittelbar die folgende Beobachtung:

Korollar 3.34 *Sei Σ ein Alphabet, sei $A := (\Sigma, Q, \delta, q_0, F)$ ein vollständiger DFA und sei $L := \mathcal{L}(A)$. Dann gilt für alle $x, y \in \Sigma^*$: Ist $\delta(q_0, x) = \delta(q_0, y)$, so ist $x \equiv_L y$.*

Beweis: Aus $\delta(q_0, x) = \delta(q_0, y)$ folgt $\mathcal{L}(A_x) = \mathcal{L}(A_y)$. Gemäß Lemma 3.33 folgt hieraus $D_x L = D_y L$, und somit $x \equiv_L y$. □

Vielleicht ist Ihnen bereits aufgefallen, dass die Nerode-Relationen der drei regulären Sprachen (L_1, L_2, L_3) in Beispiel 3.32 einen endlichen Index haben, während die Nerode-Relation $\equiv_{L_4}$ der nicht-regulären Sprache L_4 einen Index von ∞ hat. Das ist kein Zufall, die Regularität einer Sprache und der Index ihrer Nerode-Relation sind nämlich eng miteinander verbunden. Der folgende Satz[10] formalisiert diesen Zusammenhang:

Satz 3.35 *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Dann gilt:*

L ist genau dann regulär, wenn $\mathrm{index}(L)$ *endlich ist.*

Beweis: $\Rightarrow$: Angenommen, L ist regulär. Dann existiert ein vollständiger DFA $A := (\Sigma, Q, \delta, q_0, F)$ mit $\mathcal{L}(A) = L$. Nach Korollar 3.34 folgt für alle $x, y \in \Sigma^*$ aus $\delta(q_0, x) = \delta(q_0, y)$ stets $x \equiv_L y$. Also kann $\equiv_L$ nicht mehr Äquivalenzklassen haben als A Zustände hat. Daher gilt $\mathrm{index}(L) \leq |Q|$, somit ist $\mathrm{index}(L)$ endlich.

$\Leftarrow$: Um diese Richtung zu beweisen konstruieren wir einen DFA $A_{\equiv_L}$, der auch als der **Äquivalenzklassenautomat** oder der **Nerode-Automat** zu L bekannt ist. Sei also $\mathrm{index}(L)$ endlich. Wir verwenden nun die Nerode-Relation $\equiv_L$ von L und ihre Äquivalenzklassen, um einen DFA $A_{\equiv_L} := (\Sigma, Q, \delta, q_0, F)$ zu definieren, und zwar wie folgt:

- $Q := \{[x] \mid x \in \Sigma^*\}$, wir verwenden für jede Äquivalenzklasse von $\equiv_L$ einen Zustand. Da wir voraussetzten, dass $\mathrm{index}(L)$ endlich ist, ist garantiert, dass Q ebenfalls endlich ist.
- $q_0 := [\varepsilon]$, der Startzustand entspricht der Klasse des leeren Wortes.

[10] Dieser Satz ist eine leichte Abwandlung eines Resultats, das als *Satz von Myhill und Nerode* bekannt ist.

- $F := \{[x] \mid x \in L\}$, wir akzeptieren in allen Zuständen, deren Klasse einem Wort aus L entspricht.
- $\delta([x], a) = [xa]$ für alle $x \in \Sigma^*$, $a \in \Sigma$.

Zuerst müssen wir zeigen, dass $A_{\equiv_L}$ wohldefiniert ist. Potentiell problematisch sind die Definitionen von F und δ. Hier müssen wir jeweils sicherstellen, dass es egal ist, welches x wir als Repräsentanten der Klasse $[x]$ wählen. Glücklicherweise gilt folgendes für alle $x, y \in \Sigma^*$ mit $x \equiv_L y$:

$$x \in L \text{ genau dann, wenn } y \in L.$$

Dies zu beweisen ist Ihnen überlassen (Aufgabe 3.11). Aufgrund dieser Beobachtung wissen wir, dass für jede Äquivalenzklasse $[x]$ eindeutig definiert ist, ob der entsprechende Zustand akzeptierend ist oder nicht. Aufgrund von Lemma 3.31 wissen wir außerdem, dass für jede Äquivalenzklasse $[x]$ und jedes $a \in \Sigma$ ein eindeutiger Folgezustand $[xa]$ existiert (denn für alle $y \in [x]$ ist $[ya] = [xa]$).

Es bleibt also zu zeigen, dass $\mathcal{L}(A_{\equiv_L}) = L$. Dies zeigen wir durch eine (sehr einfache) vollständige Induktion über den Aufbau von w:

$$\text{Für alle } w \in \Sigma^* \text{ ist } \delta(q_0, w) = [w].$$

Die Induktion verläuft wie folgt: Ist $w = \varepsilon$, so gilt

$$\delta(q_0, w) = \delta(q_0, \varepsilon) = q_0 = [\varepsilon] = [w].$$

Angenommen, für ein festes $w \in \Sigma^*$ ist $\delta(q_0, w) = [w]$. Sei nun $a \in \Sigma$. Es gilt:

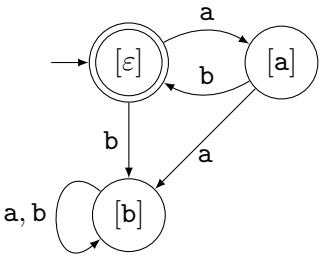
$$\delta(q_0, wa) = \delta\big(\delta(q_0, w), a\big) = \delta([w], a) = [wa].$$

Also ist $\delta(q_0, w) = [w]$ für alle $w \in \Sigma^*$. Nach Definition von F gilt $[w] \in F$ genau dann, wenn $w \in L$. Also ist $\delta(q_0, w) \in F$ genau dann, wenn $w \in L$; somit gilt $\mathcal{L}(A_{\equiv_L}) = L$. □

Beispiel 3.36 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L = \{\mathtt{a} \cdot \mathtt{b}\}^*$ (die Sprache L_3 aus Beispiel 3.32). Wie wir dort bereits erfahren haben, hat $\equiv_L$ die folgenden Äquivalenzklassen:

$$\begin{aligned}
&[\varepsilon] = [\mathtt{ab}] = \{\mathtt{ab}\}^*,\\
&[\mathtt{a}] = \mathtt{a} \cdot \{\mathtt{ba}\}^*,\\
&[\mathtt{b}] = [\mathtt{aa}] = [\mathtt{ba}] = [\mathtt{bb}] = \Sigma^* - ([\varepsilon] \cup [\mathtt{a}])\,.
\end{aligned}$$

Der Äquivalenzklassenautomat $A_{\equiv_L}$ zu L sieht daher wie folgt aus:



$\Diamond$

Wir können Satz 3.35 verwenden, um die Nicht-Regularität von Sprachen zu beweisen: Wenn für eine Sprache L gilt, dass $\text{index}(L) = \infty$, darf diese Sprache nicht regulär sein. Allerdings kann es recht aufwändig sein, alle Äquivalenzklassen von $\equiv_L$ zu bestimmen, wie am Beispiel der Sprache L_4 aus Beispiel 3.32 zu sehen ist. Glücklicherweise kann man sich diesen Aufwand oft sparen, denn um $\text{index}(L) = \infty$ zu beweisen, müssen wir nicht alle Äquivalenzklassen bestimmen; es reicht, wenn wir die Existenz von unendlich vielen unterschiedlichen Äquivalenzklassen beweisen (der Rest kann uns dann auch egal sein). Dabei hilft uns das folgende Lemma:

Lemma 3.37 (Fooling-Set-Lemma) *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Angenommen, es existiert eine unendliche Folge $\big((x_i, z_i)\big)_{i \in \mathbb{N}}$ über $\Sigma^* \times \Sigma^*$ mit*

1. *für alle $i \in \mathbb{N}$ ist $x_i z_i \in L$, und*
2. *für alle $i \in \mathbb{N}$ und alle $j \in \mathbb{N}$ mit $j \neq i$ ist $x_i z_j \notin L$.*

Dann ist L nicht regulär.

Beweis: Für alle $i, j \in \mathbb{N}$ mit $i \neq j$ gilt $z_i \in D_{x_i} L$ und $z_i \notin D_{x_j} L$. Daher ist $x_i \not\equiv_L x_j$, und somit auch $[x_i]_{\equiv_L} \neq [x_j]_{\equiv_L}$. Das bedeutet, dass $\equiv_L$ unendlich viele verschiedene Äquivalenzklassen haben muss. Also ist $\text{index}(L) = \infty$, und gemäß Satz 3.35 ist L nicht regulär. $\square$

Die Anwendung von Lemma 3.37 wird auch als **Fooling-Set-Methode** bezeichnet[11]. Auch wenn diese nicht immer anwendbar ist, erleichtert sie doch viele Beweise von Nicht-Regularität. Wir betrachten dazu ein paar Beispiele:

Beispiel 3.38 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$.

Wir beginnen mit der Sprache $L := \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$ (L_4 aus Beispiel 3.32). Wir definieren für alle $i \in \mathbb{N}$ die Wörter $x_i := \mathtt{a}^i$ und $z_i := \mathtt{b}^i$. Es ist leicht zu sehen, dass $x_i z_j \in L$ genau dann gilt, wenn $i = j$. Also folgt nach Lemma 3.37 dass L nicht regulär ist.

Nun wenden wir uns der Sprache $\text{COPY}_\Sigma := \{xx \mid x \in \Sigma^*\}$. Wir kennen diese Sprache, die Copy-Sprache über Σ, bereits aus Beispiel 3.11. Für alle $i \in \mathbb{N}$ sei $x_i := z_i := \mathtt{a}^i\mathtt{b}$. Wieder gilt $x_i z_j \in \text{COPY}_\Sigma$ genau dann, wenn $i = j$. Also folgt nach Lemma 3.37 dass COPY_Σ nicht regulär ist.

Sei nun $\text{PAL}_\Sigma := \{w \in \Sigma^* \mid w = w^R\}$. PAL ist also die Sprache aller Palindrome über Σ (ein Palindrom ist ein Wort, dass vorwärts wie rückwärts gelesen gleich ist). Wir definieren für jedes $i \in \mathbb{N}$ die Wörter $x_i := \mathtt{a}^i\mathtt{b}$ und $z_i := \mathtt{b}\mathtt{a}^i$. Wieder gilt $x_i z_j \in \text{PAL}_\Sigma$ genau dann, wenn $i = j$. Also ist $\text{PAL}_\Sigma \notin \text{REG}$ gemäß Lemma 3.37 nicht regulär. $\Diamond$

Für andere Sprachen, wie zum Beispiel $\{\mathtt{a}^{i^2} \mid i \in \mathbb{N}\}$, $\{\mathtt{a}^p \mid p \text{ ist eine Primzahl}\}$ oder die Sprache aus Beispiel 3.13 sind das Fooling-Set-Lemma und Satz 3.35 weniger bequem anzuwenden. Hier ist es meist angenehmer, auf das Pumping-Lemma und/oder geeignete Abschlusseigenschaften zurückzugreifen.

[11] Falls man anstelle einer unendlichen Folge eine endliche Folge mit n Folgengliedern angibt, kann man übrigens auf die gleiche Art $\text{index}(L) \geq n$ schließen.

Natürlich ist es gestattet, Abschlusseigenschaften mit dem Pumping-Lemma oder dem Fooling-Set-Lemma zu kombinieren. Meistens ist dies auch die geschickteste Lösung. Ein Beispiel für solch eine Kombination haben wir bereits in Beispiel 3.22 betrachtet.

3.1.4 Minimierung von DFAs

In vielen Anwendungsbereichen von DFAs werden die verwendeten Automaten aus anderen DFAs zusammengesetzt (zum Beispiel mit einer Produktautomatenkonstruktion) oder aus anderen Formalismen abgeleitet (zum Beispiel aus regulären Ausdrücken oder logischen Formeln). Allerdings ist nicht immer garantiert, dass die so erzeugten DFAs keine unnötigen Zustände enthalten (in Beispiel 3.19 haben wir so einen Fall bereits kennengelernt).

Um mit diesen DFAs effizient weiterarbeiten zu können ist es hilfreich, dabei möglichst kompakte DFAs zu haben. Dabei liegt es nahe, die Zahl der Zustände als natürliches Maß für die Größe eines DFAs zu wählen[12]. Dadurch können wir das Konzept eines minimalen DFAs wie folgt definieren:

Definition 3.39 Sei Σ ein Alphabet und $L \subseteq \Sigma^*$. Ein vollständiger DFA $A := (\Sigma, Q, \delta, q_0, F)$ heißt **minimaler vollständiger DFA für** L, wenn $\mathcal{L}(A) = L$ gilt und außerdem jeder vollständige DFA A' mit $\mathcal{L}(A') = \mathcal{L}(A)$ mindestens $|Q|$ Zustände hat.

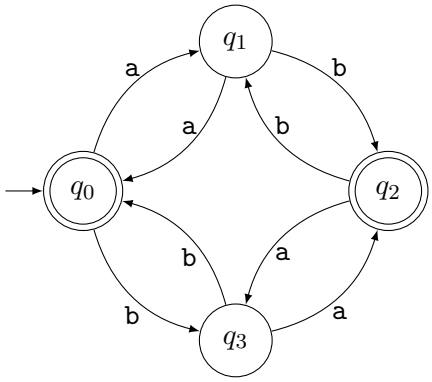
Ein (vollständiger) DFA A heißt **minimal**, wenn er ein minimaler vollständiger DFA für $\mathcal{L}(A)$ ist.

Ein minimaler DFA für eine Sprache L ist also der kleinste vollständige DFA, der L akzeptieren kann.

In Beispiel 3.19 haben wir gesehen, dass der Produktautomat nicht immer zu einer minimalen Zahl von Zuständen führt. Allerdings liegt das in diesem Beispiel daran, dass ein Zustand nicht erreichbar ist, und dass zwei Fallen anstelle von einer entstehen. Wie das folgende Beispiel zeigt, können auch DFAs ohne unerreichbare Zustände und ohne überflüssige Fallen mehr Zustände haben, als zum Akzeptieren ihrer Sprache nötig sind:

Beispiel 3.40 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Der DFA A sei wie folgt definiert:

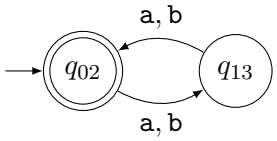
[12] Ein weiterer Kandidat wäre die Zahl der Kanten in der graphischen Darstellung. Da aber in einem vollständigen DFA die Zahl der Kanten von der Zahl der Zustände und der Größe des Alphabets abhängt, und die Größe des Alphabets für jede Sprache fest ist, läuft diese Wahl im Endeffekt auf das Gleiche heraus.



Mit ein wenig Herumprobieren[13] sieht man schnell, dass

$$\mathcal{L}(A) = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid |w| \text{ ist gerade}\}.$$

Allerdings ist A nicht der einzige DFA, der $\mathcal{L}(A)$ akzeptiert, und auch nicht der mit der kleinsten Anzahl von Zuständen. Betrachten wir den DFA A_M:



Wie leicht zu erkennen ist, gilt $\mathcal{L}(A_M) = \mathcal{L}(A)$. Außerdem ist leicht zu sehen, dass kein DFA für $\mathcal{L}(A)$ weniger als zwei Zustände haben kann[14], also ist A_M minimal (und A ist nicht minimal). ◊

Nach dieser Beobachtung stellen sich einige nahe liegende Fragen: Können wir erkennen, ob ein DFA minimal ist? Und können wir einen Algorithmus finden, der uns zu einem DFA einen entsprechenden minimalen DFA berechnet?

Tatsächlich sind deterministische endliche Automaten eines der wenigen Modelle zur Definition von formalen Sprachen, bei denen sich diese Fragen positiv beantworten lassen. Eine zentrale Rolle bei ihrer Beantwortung spielt dabei die Nerode-Relation, die wir in Abschnitt 3.1.3 betrachtet haben:

Satz 3.41 *Sei Σ ein Alphabet und sei $L \subseteq \Sigma^*$ eine reguläre Sprache. Sei $A_{\equiv_L}$ der Äquivalenzklassenautomat zu L. Es gilt:*

1. *$A_{\equiv_L}$ ist minimal.*
2. *Sei $A_M = (\Sigma, Q_M, \delta_M, q_{0,M}, F_M)$ ein minimaler DFA mit $\mathcal{L}(A_M) = L$. Dann gilt für alle $x, y \in \Sigma^*$:*

$$\delta_M(q_{0,M}, x) = \delta_M(q_{0,M}, y) \text{ genau dann, wenn } x \equiv_L y.$$

[13] Sie können dies natürlich auch beweisen, das ist Übung 3.2.
[14] Übung: Warum?

Beweis: Wir zeigen zuerst Aussage 2, da Aussage 1 daraus folgt.

Wir zeigen nun Aussage 2. Sei $A_M = (\Sigma, Q_M, \delta_M, q_{0,M}, F_M)$ ein minimaler DFA mit $\mathcal{L}(A_M) = L$. Die Richtung „$\Rightarrow$“ gilt nach Korollar 3.34 für jeden DFA, wir müssen also nur die Richtung „$\Leftarrow$“ zeigen.

Dazu wählen wir für jede Äquivalenzklasse von $\equiv_L$ ein Wort x_i als Repräsentanten aus. Formaler: Wir definieren eine Menge

$$\mathcal{R} := \{x_1, \ldots, x_{\text{index}(L)}\} \subset \Sigma^*,$$

so dass $x_i \not\equiv_L x_j$ für alle $i, j \in \{1, \ldots, \text{index}(L)\}$ mit $i \neq j$. (Welche Wörter x_i wir genau aussuchen ist egal, solange wir für jede Äquivalenzklasse genau ein Wort x_i wählen.)

Mit Hilfe der Wörter aus $\mathcal{R}$ definieren wir die Menge aller Zustände, die durch sie erreicht werden können:

$$Q_R := \left\{\delta_M(q_{0,M}, x_i) \mid x_i \in \mathcal{R}\right\} \subseteq Q_M.$$

Da $x_i \not\equiv_L x_j$ (für $i \neq j$) wissen wir nach Korollar 3.34, dass A_M beim Lesen von zwei unterschiedlichen Wörtern x_i und x_j auch in unterschiedliche Zustände geraten muss. Also gilt $|Q_R| \geq \text{index}(L)$.

Nehmen wir nun an, dass die Richtung „$\Leftarrow$“ nicht erfüllt ist, das heißt es existieren Wörter $x, y \in \Sigma^*$ mit $x \equiv_L y$ und $\delta_M(q_{0,M}, x) \neq \delta_M(q_{0,M}, y)$. Wir wählen nun dasjenige $x_i \in \mathcal{R}$, für dass $x, y \in [x_i]$. (Da $x \equiv_L y$, existiert ein solches i, und da $\mathcal{R}$ genau einen Repräsentanten jeder Äquivalenzklasse enthält, ist dieses i eindeutig bestimmt.) Es gilt also $x \equiv_L x_i$ und $y \equiv_L x_i$.

Da A_M durch die Wörter x und y in unterschiedliche Zustände gerät, kann A_M durch höchstens eines dieser beiden Wörter in den gleichen Zustand kommen wie durch x_i. Ohne Beeinträchtigung der Allgemeinheit nehmen wir an, dass y in einen anderen Zustand führt, also $\delta_M(q_{0,M}, y) \neq \delta_M(q_{0,M}, x_i)$. Da aber für alle $x_j \in \mathcal{R}$ (mit $j \neq i$) $x_i \not\equiv_L x_j$ und außerdem $x_i \equiv_L y$ gelten, gilt $y \not\equiv_L x_j$ für alle $j \neq i$. Wir stellen fest, dass $\delta_M(q_{0,M}, y) \notin Q_{\mathcal{R}}$, da sonst Korollar 3.34 zu einem Widerspruch führen würde.

Also gilt $|Q_M| \geq |Q_L| + 1 = \text{index}(L) + 1$. Der DFA A_M enthält also mindestens einen Zustand mehr als der Äquivalenzklassenautomat $A_{\equiv_L}$, A_M kann also nicht minimal sein. Fertig. Bitte beachten Sie: Wir benutzen hier nicht die Tatsache dass $A_{\equiv_L}$ minimal ist (das müssen wir nämlich gleich noch beweisen), sondern nur die Tatsache, dass $A_{\equiv_L}$ die gleiche Sprache akzeptiert wie A_M und $\text{index}(L)$ viele Zustände hat.

Nun zu Aussage 1: Aus dem Beweis von Satz 3.35 wissen wir bereits, dass $A_{\equiv_L}$ ein vollständiger DFA mit $\mathcal{L}(A_{\equiv_L}) = L$ ist. Da $\delta_{\equiv_L}([\varepsilon], x) = \delta_{\equiv_L}([\varepsilon], y)$ genau dann, wenn $x \equiv_L y$, folgt dann aus Aussage 2, dass $A_{\equiv_L}$ und jeder minimale DFA A_M gleich viele Zustände haben müssen. □

Anschaulich beschrieben, besagt die zweite Hälfte von Satz 3.41, dass jeder andere minimale DFA für L nicht nur die gleiche Zahl von Zuständen wie $A_{\equiv_L}$ hat, sondern sich auch vollkommen gleich verhält. Der Äquivalenzklassenautomat $A_{\equiv_L}$ und jeder andere minimale DFA für L sind also identisch, abgesehen von einer eventuell notwendigen Umbenennung der Zustände. Ein vollständiger DFA ist also genau dann minimal, wenn er der Äquivalenzklassenautomat für seine Sprache ist.

Diese Beobachtung ist nicht nur eine formale Kuriosität, sondern zentrales Werkzeug bei Minimierung von DFAs: Einen DFA A zu minimieren ist genau die gleiche Aufgabe wie den Äquivalenzklassenautomat für seine Sprache $\mathcal{L}(A)$ zu berechnen.

Wegen Satz 3.41 wissen wir, dass im minimalen DFA für eine Sprache L alle Wörter $x, y \in \Sigma^*$ mit $x \equiv_L y$ zum gleichen Zustand führen müssen. Ist ein reduzierter DFA nicht minimal, dann müssen nerode-äquivalente Wörter existieren, die zu unterschiedlichen Zuständen führen (also Wörter $x, y \in \Sigma^*$ mit $x \equiv_L y$, aber $\delta(q_0, x) \neq \delta(q_0, y)$). Die Hauptidee des Minimierungs-Algorithmus `minimiereDFA`, den wir gleich kennenlernen werden, ist die *Verschmelzung* dieser Zustände: Wenn zwei (oder mehr) unterschiedliche Zustände durch nerode-äquivalente Wörter erreicht werden können, fassen wir diese einfach zu einem Zustand zusammen.

Dazu definieren wir uns eine Äquivalenzrelation $\equiv_A$ auf den Zuständen des DFA, die sich verhält wie die Äquivalenzrelation $\equiv_L$ auf Wörtern.

Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein vollständiger DFA. Für jedes $q \in Q$ definieren wir den DFA A_q durch $A_q := (\Sigma, Q, \delta, q, F)$ (der DFA A_q verwendet also anstelle von q_0 den Zustand q als Startzustand). Zwei Zustände $p, q \in Q$ sind **nicht unterscheidbar**, geschrieben $p \equiv_A q$, wenn $\mathcal{L}(A_p) = \mathcal{L}(A_q)$. Wenn ein Wort $z \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big)$ existiert (und somit $\mathcal{L}(A_p) \neq \mathcal{L}(A_q)$ gilt), heißen p und q **unterscheidbar**. Wir schreiben dies als $p \not\equiv_A q$. Es ist leicht festzustellen, dass $\equiv_A$ eine Äquivalenzrelation auf Q ist[15]. Daher können wir, analog zu $\equiv_L$, Äquivalenzklassen von $\equiv_A$ definieren, und zwar durch

$$[q]_{\equiv_A} := \{p \in Q \mid p \equiv_A q\}.$$

Auch hier schreiben wir, wenn der Kontext klar ist, gelegentlich $[x]$ anstelle von $[x]_{\equiv_A}$. Wie bereits angedeutet, sind die beiden Äquivalenzrelationen $\equiv_A$ und $\equiv_L$ eng miteinander verbunden:

Lemma 3.42 *Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein vollständiger DFA, sei $L := \mathcal{L}(A)$. Dann gilt für alle $x, y \in \Sigma^*$:*

$$x \equiv_L y \text{ genau dann, wenn } \delta(q_0, x) \equiv_A \delta(q_0, y).$$

Beweis: Die Behauptung folgt fast unmittelbar aus den verwendeten Definitionen. Wir können dabei die beiden Richtungen des Beweises gemeinsam behandeln. Für alle Wörter $x, y \in \Sigma^*$ gilt:

$$\begin{aligned}
& x \equiv_L y \\
\Leftrightarrow \quad & D_x L = D_y L && \text{(nach Definition von } \equiv_L) \\
\Leftrightarrow \quad & \mathcal{L}(A_{\delta(q_0,x)}) = \mathcal{L}(A_{\delta(q_0,y)}) && \text{(nach Lemma 3.33)} \\
\Leftrightarrow \quad & \delta(q_0, x) \equiv_A \delta(q_0, y). && \text{(nach Definition von } \equiv_A)
\end{aligned}$$

□

[15] Falls Sie das nicht sofort sehen: Betrachten Sie das als eine Übung.

Lemma 3.42 gibt uns alles, was wir brauchen, um einen vollständigen DFA A in einen minimalen DFA für $\mathcal{L}(A)$ umzuwandeln: Wenn wir in A jeweils die Mengen von Zuständen zusammenlegen, die nicht unterscheidbar sind, erhalten wir (gemäß Lemma 3.42) einen DFA, dessen erreichbare Zustände den Zuständen des Äquivalenzklassenautomaten für $\mathcal{L}(A)$ entsprechen. Es können höchstens noch unerreichbare (und damit überflüssige) Zustände vorhanden sein. Wenn wir diese entfernen, erhalten wir den Äquivalenzklassenautomaten, und dieser ist gemäß Satz 3.41 minimal.

Wir können stattdessen auch gleich im ersten Schritt der Minimierung die nicht erreichbaren Zustände entfernen. Alles, was wir dann tun müssen, ist also die nicht unterscheidbaren Zustände zu ermitteln und diese dann zu verschmelzen. Den entsprechenden Algorithmus nennen wir `minimiereDFA` (siehe Algorithmus 1 auf Seite 39).

Dabei gehen wir „umgekehrt“ vor: Wir berechnen nicht die Relation $\equiv_A$, also welche Zustände nicht unterscheidbar sind, sondern ihr Komplement[16] U. Wir stellen also fest, welche Zustände unterscheidbar sind. Im Endeffekt läuft diese Vorgehen natürlich auf das gleiche Resultat heraus – nämlich eine Liste, welche Zustände unterscheidbar bzw. nicht unterscheidbar sind – aber aus diesem Blickwinkel ist der Algorithmus leichter zu verstehen.

Wir konstruieren dabei U schrittweise aus Mengen U_i, die jeweils die Paare von Zuständen enthalten, die nach (höchstens) i Schritten unterscheidbar sind. Die Menge U_0 enthält also genau die Paare von Zuständen, die durch das leere Wort unterscheidbar sind; das sind genau die Paare, die aus jeweils einem akzeptierenden und einem nicht-akzeptierenden Zustand bestehen.

Anhand von U_i versuchen wir dann in jedem Schritt, weitere Paare von unterscheidbaren Zuständen zu finden. Für jeweils zwei unterschiedliche Zustände p, q, von denen wir noch nicht definitiv wissen, dass sie unterscheidbar sind (also $\{p, q\} \notin U_i$), testen wir, ob durch Einlesen eines Buchstaben ein Paar von unterscheidbaren Zustand erreicht wird. Wir testen also, ob ein $a \in \Sigma$ zu Zuständen $\delta(p, a)$ und $\delta(q, a)$ führt, so dass $\big\{\delta(p, a), \delta(q, a)\big\} \in U_i$. Falls ja, fügen wir das Paar $\{p, q\}$ zu U_{i+1} hinzu (U_{i+1} enthält außerdem U_i). Diesen Prozess wiederholen wir, bis sich die konstruierten Mengen nicht mehr ändern (also $U_i = U_{i+1}$). Danach werden jeweils die Mengen von Zuständen zusammengelegt, die *nicht* unterscheidbar sind.

Beispiele für die Funktionsweise von `minimiereDFA` finden Sie in Abschnitt 3.1.5. Um die Korrektheit von `minimiereDFA` zu zeigen und seine Laufzeit zu analysieren, benötigen wir zuerst noch ein wenig Handwerkszeug. Ein zentraler Punkt unserer späteren Argumentation ist das folgende Lemma:

Lemma 3.43 *Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein reduzierter vollständiger DFA, und sei U_i jeweils die Menge U_i, die `minimiereDFA` bei Eingabe von A im i-ten Durchlauf der while-Schleife berechnet. Dann gilt für alle $i \in \mathbb{N}$ und alle Zustände $p, q \in Q$:*

$$\{p, q\} \in U_i \text{ genau dann, wenn ein Wort } z \in \Sigma^* \text{ existiert, für das } |z| \leq i \text{ und } z \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big).$$

[16] Die Relation $\equiv_A$ kann ja auch als Teilmenge von $Q \times Q$ interpretiert werden, ihr Komplement ist dann also $U := (Q \times Q) - \{(p, q) \mid p \equiv_A q\}$.

Algorithmus 1 : `minimiereDFA`

Eingabe : Ein DFA $A := (\Sigma, Q, \delta, q_0, F)$
Ausgabe : Ein minimaler DFA A' für $\mathcal{L}(A)$

```
1  entferne alle Zustände aus Q, die nicht erreichbar sind;
2  U_0 ← {{p,q} | p ∈ F, q ∈ (Q − F)};      /* U_0 enthält Paare von Zuständen,
   die wir bereits unterscheiden können */
3  fertig ← false;
4  i ← 0;
5  while not fertig do
6      fertig ← true;
7      U_{i+1} ← U_i;
8      foreach p, q ∈ Q mit p ≠ q, {p,q} ∉ U_i do
9          foreach a ∈ Σ do
10             if {δ(p,a), δ(q,a)} ∈ U_i then
                   /* Von p und q kommen wir durch a zu unterscheidbaren
                      Zuständen, also sind p und q unterscheidbar und
                      können in U_{i+1} aufgenommen werden.            */
11                 füge {p,q} zu U_{i+1} hinzu;
12                 fertig ← false;
13     i ← i + 1
14 U ← U_i;                          /* jetzt ist U das Komplement von ≡_A */
15 [q]_{≡_A} := {p | {p,q} ∉ U} für alle q ∈ Q;
16 Q' := {[q]_{≡_A} | q ∈ Q};
17 δ'(q,a) := [δ(q,a)]_{≡_A} für alle q ∈ Q, a ∈ Σ;
18 q'_0 := [q_0]_{≡_A};
19 F' := {[q]_{≡_A} | q ∈ F};
20 return A' := (Σ, Q', δ', q'_0, F')
```

Beweis: Wir zeigen die Behauptung durch Induktion über i.

Induktionsanfang: $i = 0$.

Behauptung: Für alle $p, q \in Q$ gilt:

$$\{p,q\} \in U_0 \text{ genau dann, wenn ein Wort } z \in \Sigma^* \text{ existiert, für das } |z| \leq 0 \text{ und } z \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big).$$

Beweis: Seien $p, q \in Q$. Dann gilt:

$$\begin{aligned} & \{p,q\} \in U_0 \\ \Leftrightarrow \quad & (p \in F \text{ und } q \in (Q-F)) \text{ oder } (q \in F \text{ und } p \in (Q-F)) \\ \Leftrightarrow \quad & \varepsilon \in \big(\mathcal{L}(A_p) - \mathcal{L}(A_q)\big) \text{ oder } \varepsilon \in (\mathcal{L}(A_q) - \mathcal{L}(A_p)) \end{aligned}$$

$$\begin{aligned} &\Leftrightarrow && \varepsilon \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big) \\ &\Leftrightarrow && \text{es existiert ein } z \in \Sigma^* \text{ mit } |z| = 0 \text{ und } z \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big). \end{aligned}$$

Also gilt die Behauptung für den Fall $i = 0$.

Induktionsschritt: Sei $i \in \mathbb{N}$ beliebig.

Induktionsannahme: Für alle $p, q \in Q$ gilt:

$$\{p, q\} \in U_i \text{ genau dann, wenn ein Wort } z \in \Sigma^* \text{ existiert, für das } |z| \leq i \text{ und } z \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big).$$

Behauptung: Für alle $p', q' \in Q$ gilt:

$$\{p', q'\} \in U_{i+1} \text{ genau dann, wenn ein Wort } z' \in \Sigma^* \text{ existiert, für das } |z'| \leq (i+1) \text{ und } z' \in \big(\mathcal{L}(A_{p'}) \triangle \mathcal{L}(A_{q'})\big).$$

Beweis: Seien $p', q' \in Q$.

$\underline{\Rightarrow}$: Sei $\{p', q'\} \in U_{i+1}$. Falls $\{p', q'\} \in U_i$ sind wir fertig (dann existiert ein passendes z' mit $|z'| \leq i \leq i+1$ nach Induktionsannahme), also können wir ohne Beeinträchtigung der Allgemeinheit annehmen, dass $\{p', q'\} \notin U_i$. Also wurde $\{p', q'\}$ im i-ten Durchlauf der while-Schleife in Zeile 5 hinzugefügt. Dazu muss die Bedingung in Zeile 10 erfüllt gewesen sein; das heißt es existieren $\{p, q\} \in U_i$ und ein $a \in \Sigma$ mit $\delta(p', a) = p$ und $\delta(q', a) = q$. Nach der Induktionsannahme existiert außerdem ein Wort $z \in \big(\mathcal{L}(A_p) \triangle \mathcal{L}(A_q)\big)$ mit $|z| \leq i$. Also ist $\delta(p, z) \in F$ und $\delta(q, z) \notin F$ (oder umgekehrt), und somit

$$\delta(p', az) = \delta(p, z) \in F \text{ und } \delta(q', az) = \delta(q, z) \notin F \text{ (oder umgekehrt).}$$

Somit gilt für $z' := az$ sowohl $|z'| = |z| + 1 \geq i + 1$ als auch $z' \in \big(\mathcal{L}(A_{p'}) \triangle \mathcal{L}(A_{q'})\big)$. Die Behauptung stimmt also für diese Richtung.

$\underline{\Leftarrow}$: Sei $z' \in \Sigma^*$ mit $z' \in \big(\mathcal{L}(A_{p'}) \triangle \mathcal{L}(A_{q'})\big)$. Falls $|z'| \leq i$ sind wir gemäß der Induktionsannahme fertig, also nehmen wir an, dass $|z'| = i+1$. Dann existieren ein $a \in \Sigma$ und ein $z \in \Sigma^*$ mit $|z| = i$ und $z' = az$.

Da $z' \in \big(\mathcal{L}(A_{p'}) \triangle \mathcal{L}(A_{q'})\big)$ gilt

$$\begin{aligned} & && \delta(p', az) \in F \text{ und } \delta(q', az) \notin F && \text{(oder umgekehrt),} \\ &\Rightarrow && \delta(\delta(p', a), z) \in F \text{ und } \delta(\delta(q', a), z) \notin F && \text{(oder umgekehrt),} \\ &\Rightarrow && z \in \mathcal{L}(A_{\delta(p',a)}) \text{ und } z \notin \mathcal{L}(A_{\delta(q',a)}) && \text{(oder umgekehrt),} \\ &\Rightarrow && z \in \big(\mathcal{L}(A_{\delta(p',a)}) \triangle \mathcal{L}(A_{\delta(q',a)})\big). \end{aligned}$$

Da $|z| = i$ gilt also gemäß unserer Induktionsannahme $\big\{\delta(p', a), \delta(q', a)\big\} \in U_i$.

Falls $\{p', q'\} \in U_i$, sind wir wegen $U_{i+1} \supseteq U_i$ (siehe Zeile 7) fertig. Nehmen wir also an, dass $\{p', q'\} \notin U_i$. Dann überprüft `minimiereDFA` im $(i+1)$-ten Durchlauf der while-Schleife in Zeile 5, ob $\big\{\delta(p', a), \delta(q', a)\big\} \in U_i$. Wie wir gerade festgestellt haben ist dies der Fall, also fügt der Algorithmus $\{p', q'\}$ zu U_{i+1} hinzu. Die Behauptung stimmt also auch für diese Richtung, und der Induktionsbeweis ist erledigt. □

Lemma 3.43 belegt also, dass `minimiereDFA` nach i Durchläufen der while-Schleife alle Paare von Zuständen ermittelt hat, die man durch Wörter der Länge i unterscheiden kann. Als nächstes stellen wir fest, dass wir nur eine beschränkte Zahl von Durchläufen der while-Schleife benötigen, um alle unterscheidbaren Zustände zu unterscheiden:

Lemma 3.44 *Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein vollständiger DFA, und sei $n := |Q| \geq 2$. Dann wird beim Aufruf von `minimiereDFA` auf A die while-Schleife in Zeile 5 maximal $(n-1)$-mal durchlaufen, und im letzten Schleifendurchlauf werden keine neuen unterscheidbaren Zustände mehr festgestellt.*

Beweis: Zuerst definieren wir für jedes $k \in \mathbb{N}$ eine Relation $\equiv_k \subseteq Q \times Q$ für alle $p, q \in Q$ wie folgt:

$$p \equiv_k q \text{ genau dann, wenn für alle } z \in \Sigma^* \text{ mit } |z| \leq k \text{ ist } (z \in \mathcal{L}(A_p) \Leftrightarrow z \in \mathcal{L}(A_q)).$$

Mit anderen Worten: $p \equiv_k q$ gilt, wenn die Zustände p und q nicht durch Wörter der Länge $\leq k$ unterschieden werden können. Es lässt sich leicht feststellen, dass auch jedes $\equiv_k$ eine Äquivalenzrelation ist. Aufgrund von Lemma 3.43 wissen wir, dass

$$p \equiv_k q, \text{ wenn } \{p, q\} \notin U_k,$$

denn U_k enthält die Zustände, die mit Wörtern der Länge $\leq k$ unterschieden werden können. Für alle $k \in \mathbb{N}$ gilt daher: Sind die Relationen $\equiv_k$ und $\equiv_{k+1}$ identisch, dann sind auch $\equiv_k$ und alle $\equiv_{k+c}$ mit $c \in \mathbb{N}$ identisch.

Es ist leicht zu sehen, dass $p \equiv_A q$ genau dann gilt, wenn ein $k \in \mathbb{N}$ existiert mit $p \equiv_k q$. Da Q endlich ist, muss also ein j existieren, für das $\equiv_k$ und $\equiv_A$ identisch sind. Wir wählen das kleinste solche k.

Nun gilt für alle i mit $0 \leq i < k$, dass $\equiv_i$ und $\equiv_{i+1}$ nicht identisch sein können.

Es ist außerdem leicht zu sehen, dass für alle $p, q \in Q$ aus $p \equiv_{i+1} q$ stets $p \equiv_i q$ folgt[17]. Somit sind die Äquivalenzklassen von $\equiv_{i+1}$ eine Verfeinerung der Äquivalenzklassen von $\equiv_i$, und da die beiden Relationen nicht identisch sind muss $\equiv_{i+1}$ mindestens eine Äquivalenzklasse mehr haben als $\equiv_i$.

Dadurch können wir die Zahl k der Schritte zwischen $\equiv_0$ und $\equiv_k$ (also $\equiv_A$) abschätzen:

Zuerst können wir annehmen, dass $\equiv_0$ genau zwei Äquivalenzklassen hat[18]. Da $\equiv_A$ eine Relation auf Q ist, wissen wir außerdem, dass $\equiv_A$ nicht mehr als $|Q| = n$ Äquivalenzklassen haben kann.

Es gilt also $2 + k \leq n$, und somit $k \leq n - 2$. Da, abgesehen vom letzten, jeder Durchlauf der while-Schleife eine neue Menge U_i erzeugt, und jede dieser Mengen einer der Relationen $\equiv_i$ entspricht, kann die Schleife maximal $(k+1)$-mal durchlaufen werden. Da $k + 1 = n - 1$ folgt hieraus die Behauptung. □

Aus dem Beweis von Lemma 3.44 folgt außerdem:

[17] Wenn p und q nicht durch Wörter der Länge $\leq i + 1$ unterschieden werden können, dann natürlich auch nicht durch Wörter der Länge $\leq i$.

[18] Da U_0 genau die Mengen F und $Q - F$ unterscheidet, hat $\equiv_0$ höchstens diese beiden Äquivalenzklassen. Wenn $\equiv_0$ nur eine Äquivanzklasse hat, sind alle Zustände von A akzeptierend oder alle nicht-akzeptierend, die while-Schleife endet nach einem Durchlauf und die Behauptung gilt ebenfalls.

Korollar 3.45 *Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein reduzierter vollständiger DFA, und seien U die Menge, die `minimiereDFA` bei Eingabe von A berechnet. Dann gilt für alle Zustände $p, q \in Q$:*

$$\{p, q\} \in U \text{ genau dann, wenn } p \not\equiv_A q.$$

Beweis: Folgt unmittelbar aus Lemma 3.43 und der Feststellung, dass $\equiv_i$ und $\equiv_A$ identisch sind. □

Zusammen mit Lemma 3.43 und Lemma 3.44 können wir nun die Korrektheit des Algorithmus `minimiereDFA` beweisen und seine Laufzeit abschätzen:

Satz 3.46 *Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein vollständiger DFA. Sei $k := |\Sigma|$ und $n := |Q|$.*

Der Algorithmus `minimiereDFA` berechnet aus A einen minimalen DFA für $\mathcal{L}(A)$ in maximal $O(kn^3)$ Schritten.

Beweis: Wir zeigen in diesem Beweis die Termination von `minimiereDFA`, die Korrektheit von `minimiereDFA`, und die Korrektheit der Laufzeitabschätzung.

Termination: Lemma 3.44 besagt, dass die while-Schleife höchstens $O(n)$-mal ausgeführt wird. Die foreach-Schleifen iterieren über endliche Mengen. Daher muss der Algorithmus terminieren.

Korrektheit: Sei $A' := (\Sigma, Q', \delta', q_0', F')$ der von `minimiereDFA` aus A berechnete DFA.

Durch Korollar 3.45 wissen wir, dass für alle $p, q \in Q$ gilt:

$$\{p, q\} \in U \text{ genau dann, wenn } p \not\equiv_A q.$$

Umgekehrt ist also

$$\{p, q\} \notin U \text{ genau dann, wenn } p \equiv_A q.$$

Somit ist A' der DFA, der entsteht, wenn in A alle nicht unterscheidbaren Zustände zusammengelegt werden. Also können wir aus Lemma 3.42 schließen, dass A' nichts anderes als der Äquivalenzklassenautomat für $\mathcal{L}(A)$ ist (abgesehen von einer eventuell notwendigen Umbenennung der Zustände). Also ist A' nach Satz 3.35 der minimale DFA für $\mathcal{L}(A)$.

Laufzeitabschätzung: Die Elimination der nicht-erreichbaren Zustände kann anhand einer Tiefensuche innerhalb von $O(nk)$ Schritten erledigt werden. Anhand von Lemma 3.44 wissen wir, dass die while-Schleife maximal $O(n)$ mal ausgeführt wird. In jedem Durchlauf der while-Schleife sind maximal $O(kn^2)$ Kombinationen aus Zuständen $p, q \in Q$ und einem Buchstaben $a \in \Sigma$ zu testen. Es ergibt sich also für die while-Schleife eine Gesamtlaufzeit von $O(kn^3)$. Die anschließenden „Aufräumarbeiten“ sowie die Entfernung der nicht-erreichbaren Zustände werden von dieser Laufzeit dominiert, die Gesamtlaufzeit ist daher $O(kn^3)$. □

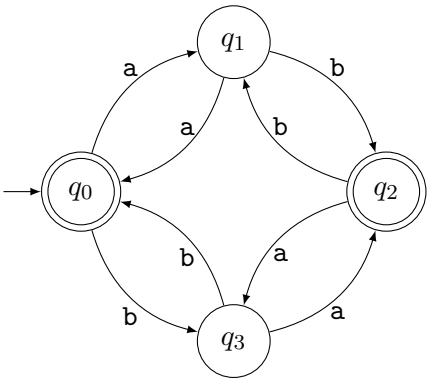
Der Algorithmus `minimiereDFA` ist übrigens nicht der effizienteste bekannte Algorithmus zur Minimierung von DFAs. Viele Lehrbücher, wie zum Beispiel Hopcroft und Ullman [8], stellen einen Algorithmus vor, der Laufzeit $O(kn^2)$ hat (die Grundidee ist, die Tabelle für Unterscheidbarkeitsrelation U geschickter zu konstruieren). Darüber hinaus ist ein Algorithmus bekannt, der in Zeit $O(kn \log n)$ arbeitet. Allerdings sind diese effizienteren Algorithmen schwieriger zu implementieren und schwieriger zu verstehen als `minimiereDFA`, so dass ich mich für diese Vorlesung für `minimiereDFA` entschieden habe.

Einen kurzen Vergleich dieser Algorithmen finden Sie in Abschnitt 3.10 von Shallit [19] („unser“ `minimiereDFA` heißt dort `NAIVE-MINIMIZE`), eine ausführlichere Übersicht über diese und weitere Algorithmen ist in dem Artikel Berstel et al. [2][19] zu finden. Es ist übrigens ein offenes Problem, ob die Laufzeit $O(kn \log n)$ geschlagen werden kann. Später[20] werden wir noch einen weiteren Algorithmus kennen lernen, der zwar nicht effizienter ist, aber aus anderen Gründen interessant ist.

3.1.5 Beispiele für `minimiereDFA`

Im Folgenden betrachten wir drei Beispiele für die Funktionsweise von `minimiereDFA`, ein einfaches (Beispiel 3.47), ein etwas komplexeres (Beispiel 3.48) und ein aufwändiges Beispiel (Beispiel 3.49).

Beispiel 3.47 Wir betrachten noch einmal den DFA A über dem Alphabet $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ aus Beispiel 3.40:



Dieser DFA hat keine nicht erreichbaren Zustände, daher kann `minimiereDFA` gleich die Menge U_0 bestimmen. Da U_0 genau die Paare enthält, die aus einem akzeptierenden und einem nicht-akzeptierenden Zustand bestehen, gilt

$$U_0 = \big\{\{p, q\} \mid p \in \{q_0, q_2\}, q \in \{q_1, q_3\}\big\}$$

[19] Dieser Artikel verwendet übrigens eine etwas andere graphische Notation für DFAs. Dort werden akzeptierende Zustände nicht mit einer doppelten Umrandung markiert, sondern mit Pfeilen, die aus dem Zustand hinaus zeigen (also wie beim Startzustand, nur umgekehrt). In Sakarovitch [13] bezeichnet Sakarovitch diese Darstellung als moderner (und begründet sie auch überzeugend), allerdings ist abzuwarten, ob Sie sich durchsetzen wird. Eventuell werde ich spätere Versionen dieses Skriptes entsprechend umarbeiten.

[20] Abschnitt 3.2.2, ab Satz 3.66.

$$= \big\{\{q_0, q_1\}, \{q_0, q_3\}, \{q_2, q_1\}, \{q_2, q_3\}\big\}.$$

Um uns (und dem Algorithmus `minimiereDFA`) die Arbeit zu erleichtern, halten wir dies in einer Tabelle fest:

q_1	U_0		
q_2		U_0	
q_3	U_0		U_0
	q_0	q_1	q_2

In den Feldern dieser Tabelle ist jeweils vermerkt, für welche Zustandspaare wir bereits die Unterscheidbarkeit festgestellt haben, und in welcher Menge U_i dies geschehen ist. (Da die Paare $\{p, q\}$ ungerichtet sind, müssen wir nur eine „halbe“ quadratische Tabelle aufführen.)

Im ersten Durchlauf der while-Schleife testet `minimiereDFA` alle Paare von ungleichen Zuständen, die nicht in U_0 liegen. Wenn wir das von Hand nachvollziehen, ist es meistens leichter, die Übergangsfunktion von A anhand einer Tabelle festzuhalten:

	`a`	`b`
q_0	q_1	q_3
q_1	q_0	q_2
q_2	q_3	q_1
q_3	q_2	q_0

Wir überprüfen nun die Zustandspaare $\{q_0, q_2\}$ und $\{q_1, q_3\}$. Wir stellen fest:

$$\begin{aligned} \delta(q_0, \mathtt{a}) &= q_1, & \delta(q_0, \mathtt{b}) &= q_3, \\ \delta(q_2, \mathtt{a}) &= q_3, & \delta(q_2, \mathtt{b}) &= q_1, \end{aligned}$$

und

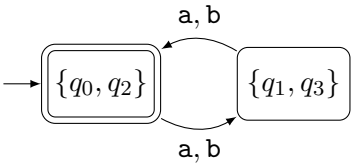
$$\begin{aligned} \delta(q_1, \mathtt{a}) &= q_0, & \delta(q_1, \mathtt{b}) &= q_2, \\ \delta(q_3, \mathtt{a}) &= q_2, & \delta(q_2, \mathtt{b}) &= q_0. \end{aligned}$$

Da weder $\{q_1, q_3\} \in U_0$, noch $\{q_0, q_2\} \in U_0$ werden keine neuen Paare hinzugefügt, die while-Schleife wird also nur ein einziges Mal durchlaufen.

Es gilt also: $U = U_1 = U_0$, und für die Äquivalenzrelation $\equiv_A$ ergeben sich die folgenden Äquivalenzklassen:

$$\begin{aligned} [q_0] &= \{q_0, q_2\}, \\ [q_1] &= \{q_1, q_3\}. \end{aligned}$$

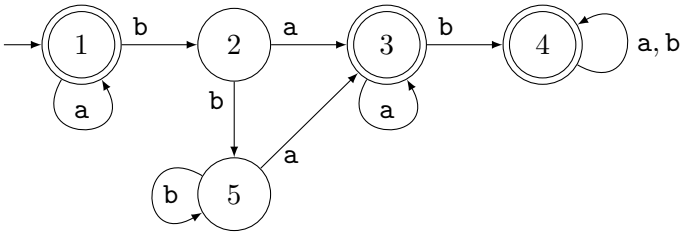
Durch Verschmelzen der äquivalenten Zustände entsteht der minimale DFA A', den wir aus Beispiel 3.40 unter der Bezeichnung A_M kennen (auch wenn dort die Zustände geringfügig anders benannt waren):



◊

Das Beispiel ist vielleicht ein wenig enttäuschend, weil in der while-Schleife nichts passiert. Daher betrachten wir nun ein etwas komplizierteres Beispiel. Danach werden wir mit 3.49 ein noch größeres Beispiel betrachten.

Beispiel 3.48 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir betrachten den folgenden vollständigen DFA A:



Da alle Zustände von A erreichbar sind, wird keiner von `minimiereDFA` entfernt. Die tabellarische Darstellung der Übergangsfunktion von A lautet wie folgt:

	a	b
1	1	2
2	3	5
3	3	4
4	4	4
5	3	5

Anhand der akzeptierenden und nicht-akzeptierenden Zustände berechnet `minimiereDFA` die Menge U_0 wie in der folgenden Tabelle angegeben:

2	U_0			
3		U_0		
4		U_0		
5	U_0		U_0	U_0
	1	2	3	4

Im ersten Durchlauf der while-Schleife testet `minimiereDFA` die vier verbliebenen Zustandspaare, ob sie anhand von U_0 unterschieden werden können. Es gilt:

$$\delta(1, \mathtt{b}) = 2, \qquad \delta(3, \mathtt{b}) = 4, \qquad \delta(4, \mathtt{b}) = 4.$$

Da $\{2, 4\} \in U_0$ kann `minimiereDFA` die Zustandspaare $\{1, 3\}$ und $\{1, 4\}$ unterscheiden und fügt sie zu U_1 hinzu. Bei den Paaren $\{2, 5\}$ und $\{3, 4\}$ kann `minimiereDFA` jeweils

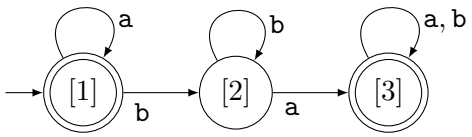
keinen Unterschied feststellen. Unsere Tabelle sieht daher nach dem ersten Durchlauf nun wie folgt aus:

2	U_0			
3	U_1	U_0		
4	U_1	U_0		
5	U_0		U_0	U_0
	1	2	3	4

Im zweiten Durchlauf der Schleife testet `minimiereDFA` nun die verbliebenen zwei Zustandspaare $\{2, 5\}$ und $\{3, 4\}$. Wieder kann kein Unterschied festgestellt werden, daher gilt $U = U_2 = U_1$. Anhand von U können wir (und auch `minimiereDFA`) erkennen, dass die Äquivalenzen $2 \equiv_A 5$ und $3 \equiv_A 4$ gelten. Die Äquivalenzrelation hat also die folgenden Äquivalenzklassen:

$$[1] = \{1\},$$
$$[2] = \{2, 5\},$$
$$[3] = \{3, 4\}.$$

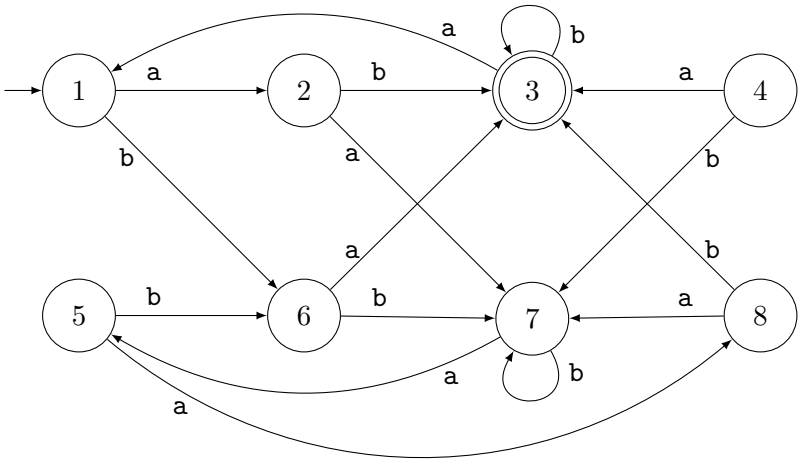
Durch Verschmelzen der äquivalenten Zustände entsteht der minimale DFA A':



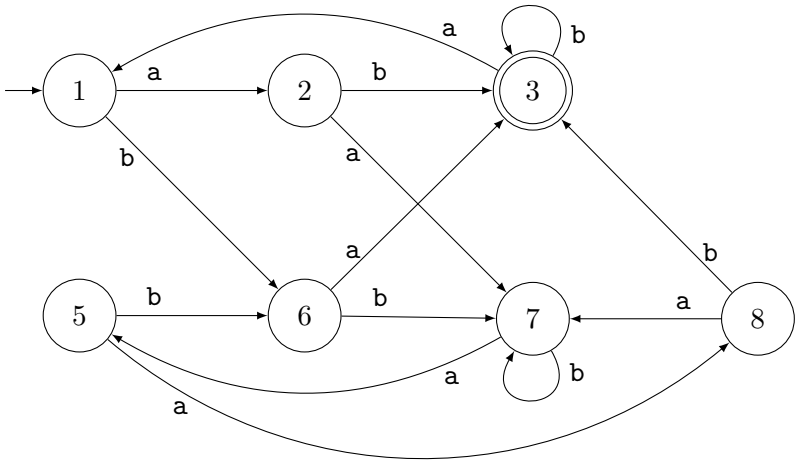
$\Diamond$

Das folgende Beispiel zeigt die Funktionsweise von `minimiereDFA` auf einem etwas größeren Automaten. Wahrscheinlich ist es besser, wenn Sie den Algorithmus zuerst selbst von Hand ausführen und danach Ihre Lösung mit diesem Beispiel vergleichen. Wie Sie an diesem Beispiel sehen können, ist `minimiereDFA` ein Algorithmus, den man oft nicht gerne von Hand ausführt, sondern lieber einem Computer überlässt.

Beispiel 3.49 Wir betrachten wir den folgenden vollständigen DFA A über $\Sigma := \{\mathtt{a}, \mathtt{b}\}$:



Der Algorithmus `minimiereDFA` entfernt zuerst alle Zustände, die von q_0 aus nicht erreicht werden können, das ist in diesem Fall nur der Zustand 4. Es wird also der folgende reduzierte vollständige DFA A_R betrachtet:



Die tabellarische Darstellung der Übergangsfunktion von A_R lautet wie folgt:

	a	b
1	2	6
2	7	3
3	1	3
5	8	6
6	3	7
7	5	7
8	7	3

Danach bestimmt `minimiereDFA` die Menge U_0. Da A_R nur einen einzigen akzeptierenden Zustand hat, nämlich den Zustand 3, ist

$$U_0 = \big\{\{3, q\} \mid q \in \{1, 2, 5, 6, 7, 8\}\big\}.$$

Die Menge U_0 enthält kein Paar der Form $\{p, 4\}$ (mit $1 \leq p \leq 8$), weil wir den Zustand 4 entfernt haben; alle anderen Paare sind nicht enthalten, weil die entsprechenden Zustände nicht anhand von 0 Schritten unterscheidbar sind.

Um bei den Durchläufen der while-Schleife die Übersicht zu behalten, welche Paare wir bereits geprüft haben, können wir eine Tabelle wie die folgende verwenden:

2						
3	U_0	U_0				
5			U_0			
6			U_0			
7			U_0			
8			U_0			
	1	2	3	5	6	7

In den Feldern dieser Tabelle ist jeweils vermerkt, für welche Zustandspaare wir bereits die Unterscheidbarkeit festgestellt haben, und in welcher Menge U_i dies geschehen ist. (Da die Paare $\{p, q\}$ ungerichtet sind, müssen wir nur eine „halbe“ quadratische Tabelle aufführen.)

Nun beginnt `minimiereDFA` mit dem ersten Durchlauf der while-Schleife und setzt $U_1 = U_0$. In der Schleife testet `minimiereDFA` für alle $p, q \in Q$ mit $p \neq q$ und $\{p, q\} \notin U_1$, ob ein $a \in \Sigma$ existiert, für das $\big\{\delta(p, a), \delta(q, a)\big\} \in U_0$.

In diesem Durchlauf sind das die folgenden Paare:

- $\{1, 2\}$, weil $\delta(1, \mathtt{b}) = 6$ und $\delta(2, \mathtt{b}) = 3$, und $\{3, 6\} \in U_0$,
- $\{1, 6\}$, weil $\delta(1, \mathtt{a}) = 2$ und $\delta(6, \mathtt{a}) = 3$, und $\{2, 3\} \in U_0$,
- $\{1, 8\}$, weil $\delta(1, \mathtt{b}) = 6$ und $\delta(8, \mathtt{b}) = 3$, und $\{3, 6\} \in U_0$,
- $\{2, 5\}$, weil $\delta(2, \mathtt{b}) = 3$ und $\delta(5, \mathtt{b}) = 6$, und $\{3, 6\} \in U_0$,
- $\{2, 6\}$, weil $\delta(2, \mathtt{a}) = 7$ und $\delta(6, \mathtt{a}) = 3$, und $\{3, 7\} \in U_0$,
- $\{2, 7\}$, weil $\delta(2, \mathtt{b}) = 3$ und $\delta(7, \mathtt{b}) = 7$, und $\{3, 7\} \in U_0$,
- $\{5, 6\}$, weil $\delta(5, \mathtt{a}) = 8$ und $\delta(6, \mathtt{a}) = 3$, und $\{3, 8\} \in U_0$,
- $\{5, 8\}$, weil $\delta(5, \mathtt{b}) = 6$ und $\delta(8, \mathtt{b}) = 3$, und $\{3, 6\} \in U_0$,
- $\{6, 7\}$, weil $\delta(6, \mathtt{a}) = 3$ und $\delta(7, \mathtt{a}) = 5$, und $\{3, 5\} \in U_0$,
- $\{6, 8\}$, weil $\delta(6, \mathtt{a}) = 3$ und $\delta(8, \mathtt{a}) = 7$, und $\{3, 7\} \in U_0$,
- $\{7, 8\}$, weil $\delta(7, \mathtt{b}) = 7$ und $\delta(8, \mathtt{b}) = 3$, und $\{3, 7\} \in U_0$.

Für alle anderen Paare von ungleichen Zuständen gilt, dass das entsprechende Paar von Folgezuständen nicht in U_0 liegt. Die Menge U_1 enthält also U_0 und jedes der oben aufgeführten Paare. Unsere Tabelle sieht nun wie folgt aus:

2	U_1					
3	U_0	U_0				
5		U_1	U_0			
6	U_1	U_1	U_0	U_1		
7		U_1	U_0		U_1	
8	U_1		U_0	U_1	U_1	U_1
	1	2	3	5	6	7

Der Algorithmus `minimiereDFA` durchläuft nun die while-Schleife zum zweiten Mal. Wie der Tabelle zu entnehmen ist, müssen nur noch die Paare $\{1,5\}$, $\{1,7\}$, $\{2,8\}$ und $\{5,7\}$ getestet werden.

Dabei können wir die folgenden Paare unterscheiden:

- $\{1,7\}$, weil $\delta(1,\mathtt{a}) = 2$ und $\delta(7,\mathtt{a}) = 5$, und $\{2,5\} \in U_1$,
- $\{5,7\}$, weil $\delta(5,\mathtt{a}) = 8$ und $\delta(7,\mathtt{a}) = 5$, und $\{5,8\} \in U_1$.

Die beiden Paare $\{1,5\}$ und $\{2,8\}$ sind nicht unterscheidbar, denn

$$\begin{aligned} \delta(1,\mathtt{a}) &= 2, & \delta(1,\mathtt{b}) &= 6, \\ \delta(5,\mathtt{a}) &= 8, & \delta(5,\mathtt{b}) &= 6, \end{aligned}$$

und

$$\begin{aligned} \delta(2,\mathtt{a}) &= 7, & \delta(2,\mathtt{b}) &= 3, \\ \delta(8,\mathtt{a}) &= 7, & \delta(8,\mathtt{b}) &= 3. \end{aligned}$$

Im Fall von $\{2,8\}$ sind sogar die Nachfolgezustände identisch, dieses Paar kann also unmöglich unterschieden werden. Im Fall von $\{1,5\}$ kann uns nur der Buchstabe `a` eine Unterscheidung ermöglichen, da aber $\{2,8\} \notin U_1$ ist dies frühestens in einem späteren Durchlauf möglich. Da wir aber bereits festgestellt haben, dass $\{2,8\}$ niemals unterscheidbar sein kann, können wir uns diese Arbeit im Prinzip sparen. Der Algorithmus weiß das aber nicht, daher muss er noch einmal die while-Schleife durchlaufen. Davor halten wir noch fest, dass $U_2 = U_1 \cup \big\{\{1,7\},\{5,7\}\big\}$, und unsere Tabelle sieht nun aus wie folgt:

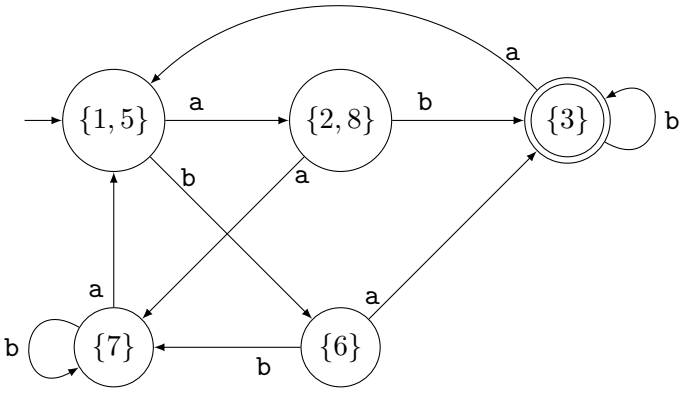
2	U_1					
3	U_0	U_0				
5		U_1	U_0			
6	U_1	U_1	U_0	U_1		
7	U_2	U_1	U_0	U_2	U_1	
8	U_1		U_0	U_1	U_1	U_1
	1	2	3	5	6	7

Es müssen also nur noch die beiden Paare $\{1,5\}$ und $\{2,8\}$ getestet werden. Wie bereits festgestellt sind diese nicht unterscheidbar, also fügt `minimiereDFA` zu U_3 keine weiteren Paare hinzu und beendet die while-Schleife.

Es gilt: $U = U_3 = U_2$. Anhand von U stellen wir fest, dass $1 \equiv_{A_R} 5$ und $2 \equiv_{A_R} 8$ gilt, alle anderen Paare von ungleichen Zuständen sind unterscheidbar. Daher hat $\equiv_{A_R}$ die folgenden Äquivalenzklassen:

$$[1] = \{1,5\},$$
$$[2] = \{2,8\},$$
$$[3] = \{3\},$$
$$[6] = \{6\},$$
$$[7] = \{7\}.$$

Somit konstruiert `minimiereDFA` durch Verschmelzen der unter $\equiv_{A_R}$ äquivalenten Zustände den folgenden minimalen DFA A_M:



◊

Hinweis 3.50 Wenn Sie einen DFA von Hand minimieren wollen[21] ist es oft hilfreich, nicht nur einfach `minimiereDFA` auszuführen, sondern den DFA vorher mit ein wenig Augenmaß zu optimieren.

In Beispiel 3.49 können Sie zum Beispiel feststellen, dass die Zustände 2 und 8 immer die gleichen Folgezustände haben. Diese beiden Zustände können also gefahrlos zu $\{2,8\}$ verschmolzen werden. Im resultierenden Automaten haben dann die Zustände 1 und 5 immer die gleichen Folgezustände (nämlich $\{2,8\}$ und 6); sie können also zu $\{1,5\}$ verschmolzen werden. Auf diese Art erhalten Sie ebenfalls den minimalen Automaten für die Sprache aus Beispiel 3.49, allerdings halt nur nicht so zuverlässig.

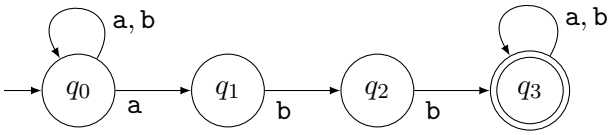
[21] Oder wenn Sie den DFA nicht wirklich minimieren *wollen*, sondern das Gefühl haben, das tun zu *müssen*, zum Beispiel weil Sie dafür in einer Klausur oder Übung Punkte bekommen wollen.

Bei Automaten wie der DFA A in Beispiel 3.40 funktioniert ein ähnlicher Ansatz; allerdings müssen hier die Zustände q_0 und q_2 sowie q_1 und q_3 gleichzeitig verschmolzen werden.

3.2 Nichtdeterministische Endliche Automaten

In diesem Abschnitt befassen wir uns mit einer Verallgemeinerung von DFAs, den *nichtdeterministischen endlichen Automaten*, kurz *NFAs* (vom englischen *non-deterministic finite automaton*).

Beispiel 3.51 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir betrachten in diesem Beispiel einen nichtdeterministischen endlichen Automaten A, der durch die folgende graphische Darstellung gegeben ist:



Dieser NFA ist kein DFA: Im Zustand q_0 kann A den Buchstaben a einlesen und dann sowohl im Zustand q_0 bleiben, als auch in den Zustand q_1 wechseln. Zur Erinnerung: In einem DFA kann jeder Zustand für jeden Buchstaben *höchstens einen* Folgezustand haben (und in einem vollständigen DFA hat jeder Zustand sogar *genau einen* Folgezustand für jeden Buchstaben).

Das nichtdeterministiche Verhalten von A kann man sich auf drei Arten veranschaulichen:

1. Durch Raten: Wenn der NFA A in einen Zustand kommt und einen Buchstaben einliest, zu dem mehrere Folgezustände existieren, rät er einfach welchen Folgezustand er verwendet. Ein Wort w wird von dem NFA A akzeptiert, wenn A beim Abarbeiten der Buchstaben von w durch Raten vom Startzustand zu einem akzeptierenden Zustand gelangen kann. Der NFA A ist dabei nicht gezwungen, in jedem Fall richtig zu raten, es genügt, wenn Raten zum Ziel führen kann.

2. Durch Parallelismus: Wenn der NFA A in einen Zustand kommt und einen Buchstaben einliest, zu dem mehrere Folgezustände existieren, teilt sich A in so viele parallele Kopien von A auf, wie mögliche Folgezustände existieren. Jeder dieser NFAs verwendet einen dieser Folgezustände und rechnet weiter. Ein Wort w wird von dem NFA A akzeptiert, wenn beim Abarbeiten der Buchstaben von w mindestens eine der parallelen Kopien von A zu einem akzeptierenden Zustand gelangt.

3. Durch gleichzeitige Zustände: Wenn der NFA A in einen Zustand kommt und einen Buchstaben einliest, zu dem mehrere Folgezustände existieren, wechselt A zu jedem dieser Zustände. Ein NFA kann also gleichzeitig in mehreren Zuständen sein und rechnet in jedem dieser Zustände unabhängig weiter. Ein Wort w wird von dem NFA A akzeptiert, wenn A beim Abarbeiten der Buchstaben von w in eine Menge von Zuständen gerät, von denen mindestens einer ein akzeptierender Zustand ist.

Alle drei Sichtweisen sind unterschiedliche Blickwinkel auf das gleiche Verhalten; je nachdem, welche Fragestellungen zu oder Aspekte von NFAs man untersucht kann es einfacher sein, eine dieser drei Sichtweisen zu verwenden.

Der NFA A akzeptiert also genau die Sprache $\{\mathtt{a},\mathtt{b}\}^*\mathtt{abb}\{\mathtt{a},\mathtt{b}\}^*$, also die Menge aller Wörter über Σ, die `abb` enthalten. ◊

Nun sind wir bereit, die Definition und das Verhalten von NFAs formal anzugehen:

Definition 3.52 Ein **nichtdeterministischer endlicher Automat (NFA)** A über einem Alphabet Σ wird definiert durch:

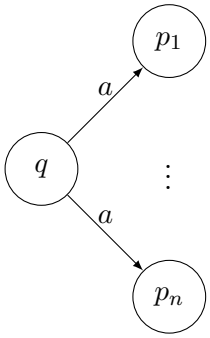
1. eine nicht-leere, endliche Menge Q von **Zuständen**,
2. eine Funktion $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (die **Übergangsrelation**[22]),
3. einen Zustand $q_0 \in Q$ (der **Startzustand**),
4. eine Menge $F \subseteq Q$ von **akzeptierenden Zuständen**.

Wir schreiben dies als $A := (\Sigma, Q, \delta, q_0, F)$.

Der einzige Unterschied zum DFA ist, dass der NFA anstelle der *Übergangsfunktion* eine *Übergangsrelation* verwendet: Einem Zustand q können beim Einlesen eines Buchstaben a mehrere Zustände zugeordnet werden. Natürlich ist es auch gestattet, eine Menge mit genau einem Zustand oder die leere Menge zuzuordnen.

Die graphische Darstellung eines NFAs ist analog zu der graphischen Darstellung eines DFAs definiert. Um Folgezustände zu definieren, verwenden wir den folgenden Ansatz:

- Für jeden Zustand $q \in Q$ und jeden Buchstaben $a \in \Sigma$ gilt: Angenommen, es gilt $\delta(q, a) = \{p_1, \ldots, p_n\}$ mit $n \geq 1$ (also $\delta(q, a) \neq \emptyset$). Dann gibt es in der graphischen Darstellung von A zu jedem p_i (mit $1 \leq i \leq n$) einen Pfeil von q nach p_i, der mit a beschriftet ist.



[22] Streng genommen ist δ eine Relation über $Q \times \Sigma \times Q$, oder eine Funktion von $Q \times \Sigma$ nach $\mathcal{P}(Q)$. Im Gebrauch ist es aber einfacher, von einer Übergangsrelation zu sprechen und diese als Funktion, die in die Potenzmenge von Q abbildet, zu definieren.

Wie bei der Darstellung von DFAs können mehrere Kanten, die die gleichen Zustände verbinden, zusammengefasst werden. Außerdem können natürlich auch Übergagsrelationen anhand einer Übergangstabelle dargestellt werden. Das Verhalten eines NFAs lässt sich analog zum Verhalten eines DFAs (siehe Definition 3.2) durch Erweiterung der Übergangsrelation definieren:

Definition 3.53 Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein NFA. Die Übergangsrelation δ wird zu einer partiellen Funktion $\delta : Q \times \Sigma^* \to \mathcal{P}(Q)$ **erweitert**, und zwar durch die folgende rekursive Definition für alle $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$:

$$\delta(q, \varepsilon) := \{q\},$$
$$\delta(q, wa) := \bigcup_{p \in \delta(q,w)} \delta(p, a).$$

Der NFA A **akzeptiert** ein Wort $w \in \Sigma^*$, wenn $\big(\delta(q_0, w) \cap F\big) \neq \emptyset$. Die von A **akzeptierte Sprache** $\mathcal{L}(A)$ ist definiert als die Menge aller von A akzeptierten Wörter, also

$$\mathcal{L}(A) := \big\{w \in \Sigma^* \mid \big(\delta(q_0, w) \cap F\big) \neq \emptyset\big\}.$$

Anschaulich ausgedrückt, akzeptiert ein NFA also (genau wie ein DFA) alle Wörter, die durch Beschriftungen an den Kanten von Pfaden von q_0 zu einem Zustand aus F gebildet werden können. Anders als ein DFA können bei einem NFA aber auch zu einem Wort mehrere Pfade existieren.

Einer der ersten Schritte bei der Untersuchung eines neuen Modells zur Definition von formalen Sprachen ist der Vergleich der Ausdrucksstärke dieses neuen Modells mit der Ausdrucksstärke bereits bekannter Modelle. Es ist leicht zu sehen, dass jeder DFA durch einen NFA simuliert werden kann. Für viele Anfänger ist es sogar schwieriger, zu sehen, dass ein DFA nicht automatisch ein NFA ist (jedenfalls nicht im strengen Sinn der Definition). DFAs verwenden nämlich eine Übergangsfunktion, während NFAs eine Übergangsrelation verwenden. Wie das folgende Beispiel illustriert ist es aber kein Problem, die Funktion zu einer Relation umzuwandeln:

Beispiel 3.54 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$, Der DFA $A := (\Sigma, Q, \delta, q_0, F)$ sei definiert durch $Q := \{q_0, q_1\}$, $F := \{q_0\}$ und durch die partielle Übergangsfunktion δ:

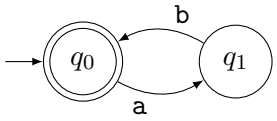
	a	b
q_0	q_1	undef.
q_1	undef.	q_0

Um einen NFA A' anzugeben, der die gleiche Sprache akzeptiert wie der DFA A, verwenden wir die folgende Übergangsrelation δ':

	a	b
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_0\}$

Wann immer δ nicht definiert ist, verwendet δ' die leere Menge, und wann immer δ einen Zustand zurückgibt, gibt δ' die Menge zurück, die genau diesen Zustand enthält. Wir definieren nun den NFA $A' := (\Sigma, Q, \delta', q_0, F)$. Es ist leicht zu sehen, dass $\mathcal{L}(A) = \mathcal{L}(A')$. Wenn Sie wollen, können Sie das anhand einer einfachen Induktion auch beweisen.

Auf Ebene der graphischen Darstellung haben wir das Problem mit der Unterscheidung zwischen Übergangsfunktion und -relation übrigens nicht; der DFA A und der NFA A' haben die gleiche graphische Darstellung:



$\Diamond$

Auch wenn ein DFA gemäß der Definition eigentlich nie ein NFA sein kann, können wir jeden DFA direkt auch als NFA interpretieren; insbesondere, wenn wir den DFA über eine graphische Darstellung definiert haben. Wir wissen also, dass NFAs ausdrucksstark genug sind, um die Klasse REG der regulären Sprachen definieren zu können. Im folgenden Abschnitt widmen wir uns der Frage, ob NFAs mehr als die Klasse definieren können.

3.2.1 NFAs, DFAs und die Potenzmengenkonstruktion

Wenn man noch keine Erfahrungen mit endlichen Automaten sammeln konnte, mag das folgende Resultat überraschend wirken: Auch wenn NFAs das Modell der DFAs verallgemeinern, können sie trotzdem nur reguläre Sprachen definieren:

Satz 3.55 *Sei Σ ein Alphabet und A ein NFA über Σ. Dann gilt: $\mathcal{L}(A)$ ist regulär.*

Beweisidee: Wir zeigen die Behauptung, indem wir einen DFA A' konstruieren, der A simuliert. Um A zu simulieren genügt es, sich zu merken, in welchen Zuständen A beim Lesen sein kann (oder, wenn man A als mehrere parallele NFAs versteht, in welchen Zuständen diese sein können). Dafür gibt es zu jedem Zeitpunkt nicht mehr als $2^{|Q|}$ Kombinationen (für jeden Zustand aus Q gibt es zwei Möglichkeiten, da Q insgesamt $|Q|$ Zustände hat also $2^{|Q|}$). Jede dieser Kombinationen entspricht einer der Mengen aus $\mathcal{P}(Q)$. Der DFA A' verwendet als Zustände diese Mengen und simuliert darauf jeweils A entsprechend der Buchstaben des Eingabewortes w. Nach dem Einlesen von w ist A' in einem Zustand, der einer Menge von Zuständen von A entspricht. Wenn mindestens einer dieser Zustände ein akzeptierender Zustand von A ist, dann akzeptiert A'. □

Beweis: Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein beliebiger NFA über einem beliebigen Alphabet Σ. Wir definieren nun den DFA $A_D := (\Sigma, Q_D, \delta_D, q_{0,D}, F_D)$ wie folgt:

- $Q_D := \mathcal{P}(Q)$
- $q_{0,D} := \{q_0\}$,
- $F_D := \big\{M \in Q_D \mid \big(M \cap F\big) \neq \emptyset\big\}$,

- für alle $M \in Q_D$ und alle $a \in \Sigma$ ist

$$\delta_D(M, a) := \bigcup_{q \in M} \delta(q, a).$$

Da die Zustände von A_D den Mengen aus $\mathcal{P}(Q)$ (der Potenzmenge von Q) entsprechen, bezeichnen wir A_D auch als den **Potenzmengenautomat** (zu A).

Um zu zeigen, dass $\mathcal{L}(A_D) = \mathcal{L}(A)$, zeigen wir zuerst, dass für alle $w \in \Sigma^*$ gilt:

$$\delta_D(q_{0,D}, w) = \delta(q_0, w).$$

Wir zeigen die Behauptung durch eine einfache Induktion über den Aufbau des Wortes w, und beginnen dazu mit dem Fall $w = \varepsilon$. Es gilt:

$$\begin{aligned} \delta_D(q_{0,D}, \varepsilon) &= q_{0,D} \\ &= \{q_0\} \\ &= \delta(q_0, \varepsilon). \end{aligned}$$

Die Behauptung gelte nun für $w \in \Sigma^*$, und sei $a \in \Sigma$. Es gilt:

$$\begin{aligned} \delta_D(q_{0,D}, wa) &= \delta_D\big(\delta_D(q_{0,D}, w), a\big) && \\ &= \delta_D\big(\delta(q_0, w), a\big) && \text{(nach Induktionsann.)} \\ &= \bigcup_{p \in \delta(q_0, w)} \delta(p, a) && \text{(nach Def. } \delta_D\text{)} \\ &= \delta(q_0, wa) && \text{(nach Def. erw. Übergangsrel.).} \end{aligned}$$

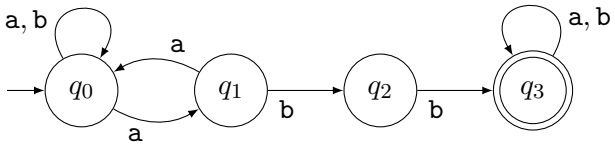
Also gilt $\delta_D(q_{0,D}, w) = \delta(q_0, w)$ für alle $w \in \Sigma^*$. Wir stellen fest:

$$\begin{aligned} & && w \in \mathcal{L}(A_D) \\ \Leftrightarrow \quad & && \delta_D(q_{0,D}, w) \in F_D \\ \Leftrightarrow \quad & && \big(\delta_D(q_{0,D}, w) \cap F\big) \neq \emptyset \\ \Leftrightarrow \quad & && \big(\delta(q_0, w) \cap F\big) \neq \emptyset \\ \Leftrightarrow \quad & && w \in \mathcal{L}(A). \end{aligned}$$

Es gilt $\mathcal{L}(A_D) = \mathcal{L}(A)$, also ist $\mathcal{L}(A)$ regulär. □

Definiert man den Potenzmengenautomaten exakt wie im Beweis von Satz 3.55, so kann dieser nicht-erreichbare Zustände enthalten. In vielen Fällen kann man dies vermeiden, indem man die Übergangstabelle des DFAs schrittweise konstruiert und diese nur für erreichbare Zustände bestimmt. Dieses Vorgehen bezeichnen wir als **Potenzmengenkonstruktion** (im Englischen auch *powerset construction* oder *subset construction* genannt). Wir illustrieren dies durch das folgende Beispiel:

Beispiel 3.56 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Der NFA A sei durch die folgende graphische Darstellung gegeben:



Zuerst stellen wir die Übergangsrelation von A als Tabelle dar (dies vereinfacht das Berechnen der Folgezustandsmengen):

	a	b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\{q_0\}$	$\{q_2\}$
q_2	$\emptyset$	$\{q_3\}$
q_3	$\{q_3\}$	$\{q_3\}$

Um die Tabelle für die Übergangsfunktion des Potenzmengenautomaten zu konstruieren, legen wir eine neue Tabelle an:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

Die erste Zeile (und bisher einzige) enthält den Startzustand $\{q_0\}$ und seine beiden Folgezustandsmengen. Da die ersten dieser beiden Mengen, die Menge $\{q_0, q_1\}$, einem Zustand entspricht der noch keine Zeile in der Tabelle hat, legen wir dafür eine neue Zeile an. Die zweite Menge entspricht dem Startzustand und ist bereits mit einer Zeile in der Tabelle vertreten. Die Tabelle sieht also nun wie folgt aus:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$		

Wir füllen nun die Zeile für $\{q_0, q_1\}$ mit den entsprechenden Folgezustandsmengen:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$		

Da die Menge $\{q_0, q_2\}$ bisher keine Zeile hatte, haben wir für diese eine neue Zeile angefügt. Im nächsten Schritt berechnen wir die Folgezustandsmengen für $\{q_0, q_2\}$:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_3\}$		

Auch hier ist mit $\{q_0, q_3\}$ eine neue Menge aufgetreten, die entsprechende Zeile wurde zur Tabelle hinzugefügt. Nun berechnen wir die entsprechenden Folgezustände:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$
$\{q_0, q_1, q_3\}$		

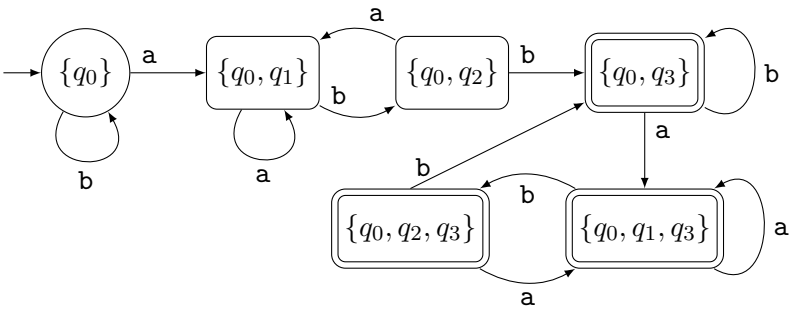
Wieder kam eine Zeile hinzu, die Menge $\{q_0, q_1, q_3\}$ war bisher nicht vertreten. Ihre Folgezustände lauten wie folgt:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$
$\{q_0, q_2, q_3\}$		

Die neue Menge $\{q_0, q_2, q_3\}$ hat ebenfalls eine neue Zeile erhalten. Auch hier berechnen wir wie gewohnt die Folgezustände:

	a	b
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_3\}$
$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$
$\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$
$\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$

Beide Mengen, die als Folgezustandsmengen für $\{q_0, q_2, q_3\}$ auftreten, sind bereits in der Tabelle vertreten. Da keine weiteren unausgefüllten Zeilen vorhanden sind, haben wir alle erreichbaren Zustände des Potenzmengenautomaten abgearbeitet, wir sind also fertig. Der Potenzmengenautomat A_D sieht wie folgt aus:



Wie leicht zu erkennen ist, ist dieser DFA reduziert (wir haben uns ganze zehn nicht-erreichbare Zustände erspart). Allerdings ist auch leicht zu erkennen, dass der DFA nicht minimal ist: Die drei akzeptierenden Zustände können gefahrlos zusammengelegt werden. ◊

Der Potenzmengenautomat erlaubt es uns nicht nur, aus einem DFA einen äquivalenten DFA zu konstruieren, sondern lässt uns außerdem auch die Größe dieses DFAs beschränken:

Korollar 3.57 *Sei Σ ein Alphabet und A ein NFA über Σ. Angenommen, A hat n Zustände. Dann existiert ein vollständiger DFA A_D über Σ mit $\mathcal{L}(A_D) = \mathcal{L}(A)$, und A_D hat höchstens 2^n Zustände.*

Beweis: Folgt direkt aus dem Beweis von Satz 3.55. Gilt für die Zustandsmenge Q des NFAs $n = |Q|$, dann hat der Potenzmengenautomat $|\mathcal{P}(Q)| = 2^n$ Zustände. Außerdem ist der Potenzmengenautomat nach Definition vollständig. □

Betrachtet man eine solche obere Schranke stellt sich natürlich die Frage, ob eine entsprechende untere Schranke existiert. Mit anderen Worten: Gibt es eine Folge von NFAs, so dass für jeden NFA mit n Zuständen der äquivalente vollständiger DFA mindestens 2^n Zustände benötigt? Wie wir gleich sehen werden, lässt sich mit wenig Aufwand eine recht nahe untere Schranke beweisen:

Lemma 3.58 *Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Für $n \geq 2$ definieren wir die Sprache*[23]

$$L_n := \{\mathtt{a}, \mathtt{b}\}^*\{\mathtt{a}\}\{\mathtt{a}, \mathtt{b}\}^{n-1}.$$

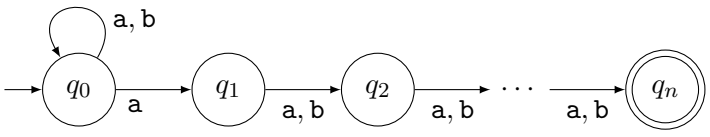
Dann gilt:

1. *Es existiert ein NFA A_n mit $\mathcal{L}(A_n) = L_n$, der $n + 1$ Zustände hat,*
2. *jeder DFA $A_{n,D}$ mit $\mathcal{L}(A_{n,D}) = L_n$ hat mindestens 2^n Zustände.*

Beweis: Wir beweisen zuerst die erste Behauptung. Für $n \geq 2$ definieren wir den NFA $A_n := (\Sigma, Q_n, \delta_n, q_0, \{q_n\})$ durch $Q_n := \{q_0, \ldots, q_n\}$ und

$$\begin{aligned}
\delta(q_0, \mathtt{a}) &:= \{q_0, q_1\}, \\
\delta(q_0, \mathtt{b}) &:= \{q_0\}, \\
\delta(q_i, c) &:= \{q_{i+1}\} && \text{für alle } 1 \leq i < n,\ c \in \Sigma, \\
\delta(q_n, c) &:= \emptyset && \text{für alle } c \in \Sigma.
\end{aligned}$$

Die graphische Darstellung der NFAs A_n lässt sich wie folgt skizzieren:



[23] Anschaulich gesprochen enthält L_n die Wörter über Σ, deren n-letzter Buchstabe ein $\mathtt{a}$ ist.

Offensichtlich hat A_n insgesamt $n+1$ Zustände; außerdem ist leicht zu sehen, dass $\mathcal{L}(A_n) = L_n$ gilt.

Um die zweite Behauptung zu zeigen, beweisen wir, dass $\text{index}(L_n) \geq 2^n$. Wir zeigen dies, indem wir die folgende Behauptung beweisen: Sei $n \geq 2$. Dann gilt für alle $x, y \in \Sigma^n$ (also für alle Wörter x, y über Σ, die genau Länge n haben[24]):

$$\text{Aus } x \neq y \text{ folgt } x \not\equiv_{L_n} y.$$

Sei $n \geq 2$ und seien $x, y \in \Sigma^n$ mit $x \neq y$. Dann existieren Wörter $x', y', z \in \Sigma^*$, so dass entweder

1. $x = x'\mathtt{a}z$ und $y = y'\mathtt{b}z$, oder
2. $x = x'\mathtt{b}z$ und $y = y'\mathtt{a}z$.

O. B. d. A. sei $x = x'\mathtt{a}z$ und $y = y'\mathtt{b}z$. Sei $z' := \mathtt{b}^{n-1-|z|}$. Es gilt:

$$\begin{aligned} xz' &= x'\mathtt{a}zz' = x'\mathtt{a}z\mathtt{b}^{n-1-|z|}, \\ yz' &= y'\mathtt{b}zz' = y'\mathtt{b}z\mathtt{b}^{n-1-|z|}. \end{aligned}$$

Da außerdem

$$|zz'| = |z| + n - 1 - |z| = n - 1$$

gilt

$$\begin{aligned} xz' &\in L_n, \\ yz' &\notin L_n \end{aligned}$$

und somit $z' \in D_x L_n$ und $z' \notin D_y L_n$. Also ist $D_x L_n \neq D_y L_n$ und somit $x \not\equiv_{L_n} y$. Da x, y frei aus Σ^n gewählt wurden, gilt $\text{index}(L_n) \geq |\Sigma^n| \geq 2^n$. □

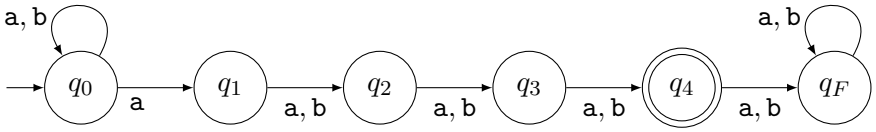
Lemma 3.58 zeigt, dass beim Konvertieren eines NFAs zu einem DFA zumindest in manchen Fällen ein exponentieller Zuwachs in Kauf genommen werden muss. Der Zuwachs von $n+1$ zu 2^n (oder, je nach Blickwinkel, von n zu 2^{n-1} ist übrigens nicht der größte mögliche Zuwachs – mit einem komplizierteren Automaten kann man auch einen Zuwachs von n zu 2^n beweisen.

Anhand einer leichten Abwandlung der Automaten aus dem Beweis von Lemma 3.58 können wir übrigens auch leicht zeigen, dass sich das Komplement einer von einem NFA akzeptierten Sprache nicht so einfach bestimmen lässt wie das Komplement einer von einem DFA akzeptieren Sprache:

Beispiel 3.59 Im Beweis von Lemma 3.14 haben wir gezeigt, dass sich jeder vollständige DFA in einen DFA für das Komplement seiner Sprache umbauen lässt, indem wir einfach akzeptierende und nicht akzeptierende Zustände vertauschen.

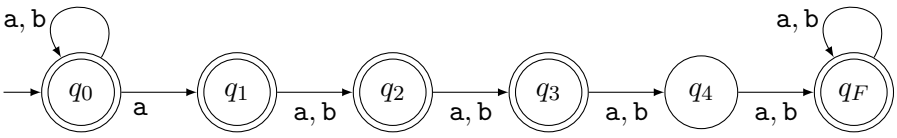
Es lässt sich leicht zeigen, dass dieser Ansatz bei NFAs nicht funktioniert. Über dem Alphabet $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ sei der NFA A definiert wie folgt:

[24] Ich gehe davon aus, dass Sie wissen, dass Σ^n die Menge aller Wörter über Σ ist, die die Länge n haben. Ich weise nur darauf hin, damit Sie auch wirklich beachten, dass wir x und y nicht aus Σ^* wählen, sondern aus Σ^n.



Wie Sie sicher bereits bemerkt haben ist A fast identisch mit dem NFA A_4 aus dem Beweis von Lemma 3.58; der einzige Unterschied ist, dass A noch eine Falle besitzt (dadurch ist sichergestellt, dass jeder Zustand von A zu jedem Buchstaben aus Σ mindestens einen Folgezustand hat). Es gilt also: $\mathcal{L}(A)$ ist die Sprache aller Wörter über Σ, die an viertletzter Stelle ein a haben.

Bilden wir nun zu A analog zum Komplementautomaten für DFAs durch Vertauschen des Akzeptanzverhaltens der Zustände einen „Komplement-NFA“, so erhalten wir den folgenden NFA A':



Allerdings sorgt schon der Zustand q_0 alleine dafür, dass $\mathcal{L}(A') = \Sigma^*$. Also ist $\mathcal{L}(A')$ offensichtlich nicht das Komplement von $\mathcal{L}(A)$. ◊

Um die Sprache eines NFAs zu komplementieren gibt es leider keinen besseren Weg, als den NFA zuerst mittels der Potenzmengenkonstruktion in einen DFA umzuwandeln und dann zu diesem DFA den Komplementautomaten zu konstruieren. Dabei kann die Anzahl der Zustände allerdings exponentiell zunehmen, so dass dies in der Praxis nicht immer eine akzeptable Lösung ist.

Man kann beweisen[25], dass dieser Größenzuwachs nicht zu vermeiden ist: Das heißt es, existieren NFAs A_n so dass A_n jeweils n Zustände hat, aber jeder NFA für die Sprache $\overline{\mathcal{L}(A_n)}$ hat mindestens 2^n Zustände.

Anhand von NFAs können wir auch leicht eine weitere Abschlusseigenschaft beweisen:

Lemma 3.60 *Die Klasse der regulären Sprachen ist abgeschlossen unter Shuffle-Produkt.*

Beweis: Siehe Übung 3.8 □

3.2.2 NFAs mit nichtdeterministischem Startzustand

Anhand der Potenzmengenkonstruktion können wir übrigens auch leicht beweisen, dass sich NFAs noch weiter verallgemeinern lassen, ohne dass dadurch nicht-reguläre Sprachen erzeugt werden können. Wir definieren dazu das folgende Modell:

[25] Bei Gelegenheit werde ich entsprechende Referenzen nachtragen. Falls Sie mich darauf aufmerksam machen, dass Sie dieses Thema interessiert, werde ich mich damit beeilen. Ansonsten kann das etwas dauern.

Definition 3.61 Ein **NFA mit nichtdeterministischem Startzustand** (**NNFA**) A über einem Alphabet Σ wird definiert durch:

1. eine nicht-leere, endliche Menge Q von **Zuständen**,
2. eine Funktion $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ (die **Übergangsrelation**),
3. ein Menge von Zuständen $Q_0 \subseteq Q$ (die **Startzustände**),
4. eine Menge $F \subseteq Q$ von **akzeptierenden Zuständen**.

Wir schreiben dies als $A := (\Sigma, Q, \delta, Q_0, F)$.

Die erweiterte Übergangsrelation eines NNFA ist definiert wie die eines NFA (siehe Definition 3.53). Der NNFA A **akzeptiert** ein Wort $w \in \Sigma^*$, wenn

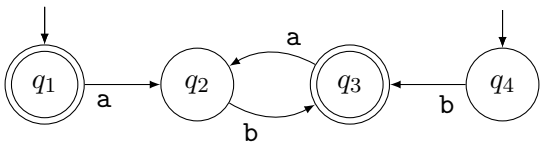
$$\left(\left(\bigcup_{q_0 \in Q_0} \delta(q_0, w) \right) \cap F \right) \neq \emptyset.$$

Die von A **akzeptierte Sprache** $\mathcal{L}(A)$ ist definiert als die Menge aller von A akzeptierten Wörter, also

$$\mathcal{L}(A) := \left\{ w \in \Sigma^* \mid \text{ es existiert ein } q_0 \in Q_0 \text{ mit } \left(\delta(q_0, w) \cap F \right) \neq \emptyset \right\}.$$

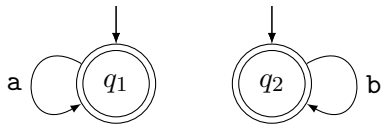
Im Gegensatz zu einem NFA hat ein NNFA mehrere Startzustände, abgesehen davon ist das Akzeptanzverhalten definiert wie beim NFA. Der NNFA akzeptiert also die Wörter, die durch Beschriftungen an den Kanten von Pfaden von *einem* Startzustand $q_0 \in Q_0$ zu einem Zustand aus F gebildet werden können.

Beispiel 3.62 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir betrachten den folgenden NNFA A_2 über Σ:



Es gilt $\mathcal{L}(A_1) = \{\mathtt{a} \cdot \mathtt{b}\}^* \cup \mathtt{b} \cdot \{\mathtt{a} \cdot \mathtt{b}\}^*$, denn durch den Startzustand q_1 akzeptiert A_1 die Sprache $\{\mathtt{a} \cdot \mathtt{b}\}^*$, und durch den Startzustand q_4 die Sprache $\mathtt{b} \cdot \{\mathtt{a} \cdot \mathtt{b}\}^*$.

Natürlich ist es auch möglich, mehrere DFAs oder NFAs zu einem NNFA zusammenzufassen, wie der folgende NNFA A_2 demonstriert:



Ohne den Hinweis, dass es sich hier um einen NNFA handelt, könnte man vermuten, dass die graphische Darstellung einen DFA für die Sprache $\{\mathtt{a}\}^*$ und einen DFA für die Sprache $\{\mathtt{b}\}^*$ zeigt. Der NNFA A_2 akzeptiert die Vereinigung dieser beiden Sprachen, also $\mathcal{L}(A_2) = \{\mathtt{a}\}^* \cup \{\mathtt{b}\}^*$. ◊

Korollar 3.63 *Sei Σ ein Alphabet und A ein NNFA über Σ. Dann gilt: $\mathcal{L}(A)$ ist regulär.*

Beweis: Der Beweis folgt sofort aus einer leichten Modifikation der Potenzmengenkonstruktion: Der Startzustand des Potenzmengenautomaten ist der Zustand, der der Menge der Startzustände von A entspricht. Sei also $A := (\Sigma, Q, \delta, Q_0, F)$. Dann wählen wir $q_0' := Q_0$, und definieren ansonsten den Potenzmengenautomaten wie im Beweis von Satz 3.55. □

Nun können wir leicht eine weitere Abschlusseigenschaft der Klasse REG beweisen:

Lemma 3.64 *Die Klasse* REG *ist abgeschlossen unter dem Reversal-Operator L^R.*

Beweisidee: Wir konstruieren aus einem DFA A für eine reguläre Sprache L einen NNFA A_R für die Sprache L^R, indem wir die akzeptierenden Zustände von A zu Startzuständen von A_R machen, den Startzustand von A als akzeptierenden Zustand von A_R verwenden, und alle Kanten umdrehen.

Beweis: Sei $A := (\Sigma, Q, \delta, q_0, F)$. Wir definieren einen NNFA $A_R := (\Sigma, Q_R, \delta_R, Q_0, F_R)$ wie folgt:

- Sei $Q_R := Q$,
- $Q_0 := F$,
- $F_R := \{q_0\}$,
- für alle $q \in Q$ und alle $a \in \Sigma$ sei

$$\delta_R(q, a) := \{p \in Q \mid \delta(p, a) = q\}.$$

Nun müssen wir noch zeigen, dass $\mathcal{L}(A_R) = \mathcal{L}(A)^R$ gilt. Weil wir schon mehr als genug Induktionen verwendet haben, bedienen wir diesmal eines anderen Ansatzes.

Wir zeigen zuerst, dass $\mathcal{L}(A)^R \subseteq \mathcal{L}(A_R)$, also dass für alle $w \in \Sigma^*$ aus $w \in \mathcal{L}(A)$ stets $w^R \in \mathcal{L}(A_R)$ folgt. Sei $w \in \Sigma^*$ mit $w \in \mathcal{L}(A)$. Sei $n := |w|$, und seien $a_1, \ldots, a_n \in \Sigma$ mit $w = a_1 \cdots a_n$.

Dann existiert eine Folge von Zuständen $q_1, \ldots, q_n \in Q$ mit $\delta(q_{i-1}, a_i) = q_i$ für $1 \leq i \leq n$. Da $w \in \mathcal{L}(A)$ muss $q_n \in F$ gelten. Da $Q_0 = F$ folgt daraus $q_n \in Q_0$. Außerdem gilt nach der Definition von δ_R, dass

$$q_{i-1} \in \delta(q_i, a_i)$$

für $1 \leq i \leq n$. Somit ist $q_0 \in \delta_R(q_n, a_n \cdots a_1)$, und (da $F_R = \{q_0\}$) gilt $a_n \cdots a_1 \in \mathcal{L}(A_R)$. Da $w^R = a_n \cdots a_1$ folgt $w^R \in \mathcal{L}(A_R)$.

Wir zeigen nun, dass $\mathcal{L}(A_R) \subseteq \mathcal{L}(A)^R$, also dass aus $w \in \mathcal{L}(A_R)$ stets $w^R \in \mathcal{L}(A)$ folgt. Sei $w \in \Sigma^*$ mit $w \in \mathcal{L}(A_R)$. Sei $n := |w|$, und seien $a_1, \ldots, a_n \in \Sigma$ mit $w = a_n \cdots a_1$. Dann existiert eine Folge von Zuständen $q_n, \ldots, q_1 \in Q_R$ mit $q_n \in Q_0$ und

$$q_{i-1} \in \delta_R(q_i, a_i)$$

für alle $1 \leq i \leq n$ (insbesondere muss die Folge auf q_0 enden, da q_0 der einzige akzeptierende Zustand von A_R ist). Nach Definition von δ_R folgt daraus

$$\delta(q_{i-1}, a_i) = q_i$$

für $1 \leq i \leq n$. Es gilt also $\delta(q_0, a_1 \cdots a_n) = q_n$ und somit $w^R \in \mathcal{L}(A)$, was gleichbedeutend ist mit $w \in \mathcal{L}(A)^R$. □

Wenn wir mit DFAs anstelle von NFAs arbeiten wollen (oder müssen), kann der Reversal-Operator allerdings vergleichsweise teuer sein. Das folgende Beispiel illustriert dies:

Beispiel 3.65 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Für jedes $n \geq 2$ definieren wir eine Sprache

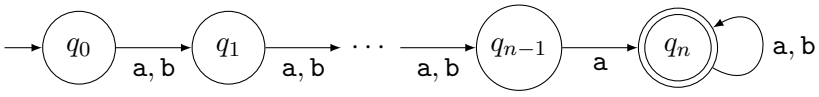
$$L_n := \{\mathtt{a}, \mathtt{b}\}^* \{\mathtt{a}\} \{\mathtt{a}, \mathtt{b}\}^{n-1},$$

und eine Sprache

$$L_{R,n} := \{\mathtt{a}, \mathtt{b}\}^{n-1} \{\mathtt{a}\} \{\mathtt{a}, \mathtt{b}\}^*.$$

Die Sprache $L_{R,n}$ enthält also genau die Wörter über Σ, bei denen an n-ter Stelle der Buchstabe $\mathtt{a}$ steht, und die Sprache L_n die Wörter über Σ, bei denen an n-letzter Stelle der Buchstabe $\mathtt{a}$ steht. Die Sprachen L_n kennen wir bereits aus Lemma 3.58, und für jedes $n \geq 2$ ist $L_{R,n} = (L_n)^R$.

Es ist leicht zu sehen, dass jede der Sprachen $L_{R,n}$ von einem DFA mit $n+1$ Zuständen akzeptiert wird. Wir skizzieren dies durch die folgende graphische Darstellung:



Aus Lemma 3.58 wissen wir, dass jeder DFA für L_n mindestens 2^n Zustände hat. Während die Anwendung des Reversal-Operators auf NNFAs also keine zusätzlichen Zustände erfordert, kann sie auf DFAs zu einem exponentiellen Größenzuwachs führen, der nicht vermeidbar ist. ◊

Wir haben jetzt das notwendige Handwerkszeug, um einen weiteren Minimierungsalgorithmus für DFAs kennenzulernen, der in der Literatur gewöhnlich als *Brzozowskis Algorithmus* bezeichnet wird. Sei A ein NNFA. Wir bezeichnen mit $\mathtt{pot}(A)$ den DFA, der durch Anwendung der Potenzmengenkonstruktion so aus A konstruiert wird, dass alle Zustände von $\mathtt{pot}(A)$ erreichbar sind (siehe Beispiel 3.56).

Sei A ein DFA. Dann bezeichne $\mathtt{rev}(A)$ den NNFA für die Sprache $\mathcal{L}(A)^R$, der anhand der Reversal-Konstruktion aus dem Beweis von Lemma 3.64 aus A konstruiert wird. Es gilt:

Satz 3.66 (Brzozowskis Algorithmus) *Sei Σ ein Alphabet und A ein DFA über Σ. Der DFA A_M sei definiert durch*

$$A_M := \texttt{pot}\,(\texttt{rev}\,(\texttt{pot}\,(\texttt{rev}\,(A))))\,.$$

Dann ist A_M ein minimaler DFA für $\mathcal{L}(A)$.

Der entsprechende Minimierungsalgorithmus ist auch als **Brzozowskis Algorithmus** bekannt[26]. Um Satz 3.66 zu beweisen, zeigen wir zuerst das folgende Lemma:

Lemma 3.67 *Sei A ein vollständiger DFA, in dem jeder Zustand erreichbar ist. Dann ist $\texttt{pot}(\texttt{rev}(A))$ ein minimaler DFA für die Sprache $\mathcal{L}(A)^R$.*

Beweis: Sei Σ ein Alphabet, und sei $A := (\Sigma, Q, \delta, q_0, F)$ ein DFA, in dem jeder Zustand erreichbar ist. Sei $A_R := \texttt{rev}(A)$ mit $A_R = (\Sigma, Q_R, \delta_D, Q_0, F_R)$ und sei $A_D := \texttt{pot}(A_R)$ mit $A_D = (\Sigma, Q_D, \delta_D, q_{0,D}, F_D)$.

Also ist A_R ein NNFA mit $\mathcal{L}(A_R) = \mathcal{L}(A)^R$, und A_D ist ein DFA mit $\mathcal{L}(A_D) = \mathcal{L}(A_R) = \mathcal{L}(A)^R$. Es gilt $Q_R = Q$, $Q_0 = F$ und $F_R = \{q_0\}$. Außerdem entspricht jeder Zustand von A_D einer Menge von Zustände von A_R.

Außerdem wissen wir aufgrund der Konstruktionsvorschriften von `pot` und von `rev`, dass jeder Zustand in A_D erreichbar ist. Um zu zeigen, dass A_D minimal ist, genügt es also, zu zeigen, dass alle Zustände von A_D anhand der Relation $\equiv_{A_D}$ unterscheidbar sind.

Seien also $P_1, P_2 \in Q_D$. Angenommen, es gilt $P_1 \equiv_{A_D} P_2$. Wir zeigen, dass daraus $P_1 = P_2$ folgt. Da die Zustände von A_D Mengen von Zuständen von A_R entsprechen, werden wir sie im Folgenden auch immer wieder wie Mengen behandeln. Um $P_1 \subseteq P_2$ zu zeigen wählen wir frei ein $p \in P_1$. Also gilt $p \in Q_R$ und somit auch $p \in Q$ (p ist also auch ein Zustand von A).

Da jeder Zustand von A erreichbar ist, existiert ein $w \in \Sigma^*$ mit

$$\delta(q_0, w) = p.$$

Somit gilt aber auch (nach Definition von `rev`)

$$q_0 \in \delta_R(p, w^R),$$

und (nach Definition von `pot`)

$$q_0 \in \delta_D(P_1, w^R).$$

Da $F_R = \{q_0\}$ ist $\delta_D(P_1, w^R) \in F_D$. Es gilt also $w^R \in \mathcal{L}(A_{D,P_1})$. Da $P_1 \equiv_{A_D} P_2$ gilt $\mathcal{L}(A_{D,P_2})$, also ist $\delta_D(P_2, w^R) \in F_D$. Somit existiert ein $q \in P_2$ mit $q_0 \in \delta_R(q, w^R)$. Also gilt (nach Definition von `rev`) $q = \delta(q_0, w)$. Somit ist

$$p = \delta(q_0, w) = q$$

und somit $p = q$. Da $q \in P_2$ und $p = q$ folgt $p \in P_2$ und somit $P_1 \subseteq P_2$. Da $\equiv_{A_D}$ symmetrisch ist, können wir auf die gleiche Art $P_2 \subseteq P_1$ schließen und erhalten so $P_1 = P_2$.

[26] Benannt nach Janusz Brzozowski.

Also folgt aus $P_1 \equiv_{A_D} P_2$ stets $P_1 = P_2$; es sind also alle Zustände in A_D unterscheidbar, und A_D ist minimal. □

Beweis (Satz 3.66): Sei Σ ein Alphabet und A ein DFA über Σ. Sei

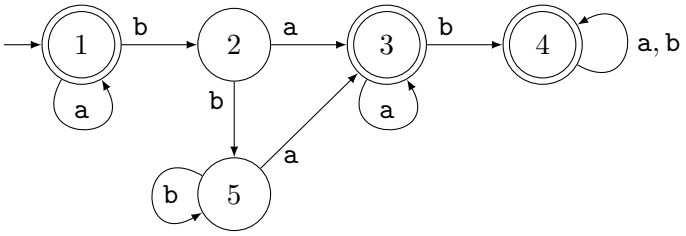
$$\begin{aligned} A_D &:= \texttt{pot}(\texttt{rev}(A)), \\ A_M &:= \texttt{pot}(\texttt{rev}(A_D)) = \texttt{pot}\left(\texttt{rev}\left(\texttt{pot}\left(\texttt{rev}\left(A\right)\right)\right)\right). \end{aligned}$$

Gemäß Definition von `pot` und `rev` ist A_D ein DFA für $\mathcal{L}(A)^R$, in dem jeder Zustand erreichbar ist. Aus Lemma 3.67 folgt, dass A_M ein minimaler DFA für $\mathcal{L}(A_D)^R$ ist, und da $\mathcal{L}(A_D)^R = (\mathcal{L}(A)^R)^R = \mathcal{L}(A)$ folgt hieraus unmittelbar die Behauptung. □

Wie wir bereits in Beispiel 3.65 gesehen haben, kann der Platzaufwand bei der Berechnung von $\texttt{pot}(\texttt{rev}(A))$ exponentiell sein (in der Zahl der Zustände von A). Dadurch können wir auch für die Laufzeit dieses Minimierungsverfahrens eine exponentielle untere Schranke feststellen. Allerdings erlauben uns die verwendeten Konstruktionen, diese Zeit auch von oben zu beschränken: Zu einem DFA mit n Zuständen auf einem Alphabet mit k Buchstaben lässt sich $\texttt{pot}(\texttt{rev}(\texttt{pot}(\texttt{rev}(A))))$ in Zeit $O(kn2^{2n})$ bestimmen[27].

Allerdings ist die Laufzeit von Brzozowkis Minimierungsalgorithmus oft noch vertretbar. Abhängig davon, wie fit Sie bei der Berechnung der Potenzmengenkonstruktion sind, kann dieses Minimierungsverfahren von Hand angenehmer auszuführen sein als `minimiereDFA`[28]. Zum Vergleich betrachten wir ein Beispiel, in dem wir den DFA aus Beispiel 3.48 anhand unseres neuen Minimierungsverfahrens minimieren:

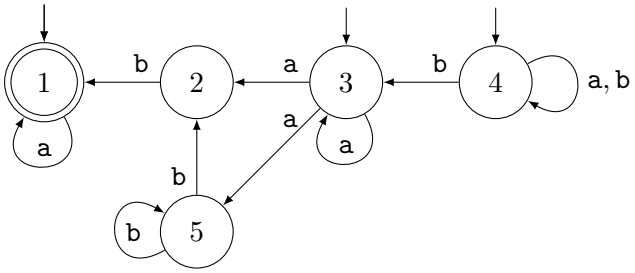
Beispiel 3.68 Sei $\Sigma := \{\texttt{a}, \texttt{b}\}$. Wir betrachten den folgenden vollständigen DFA A (bekannt aus Beispiel 3.48):



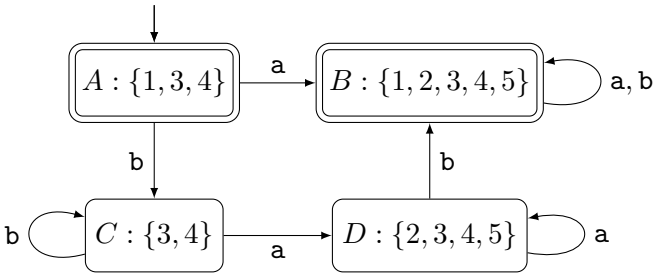
Der entsprechende NNFA $\texttt{rev}(A)$ für die Sprache $\mathcal{L}(A)^R$ sieht wie folgt aus:

[27] Das liegt daran, dass sich `pot` auf einem (N)NFA mit n Zuständen in Zeit $O(kn2^n)$ berechnen lässt. Wir lassen den Beweis dafür hier aus.

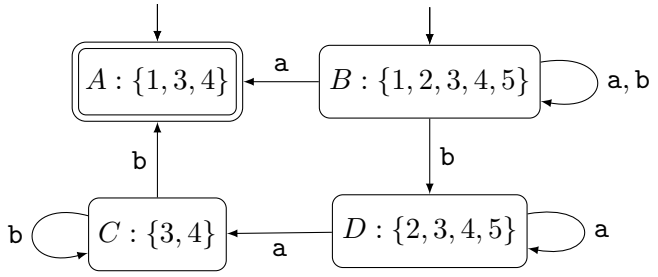
[28] Wahrscheinlich sind Sie in den meisten Fällen mit `minimiereDFA` oder geschicktes Draufstarren deutlich schneller als mit Brzozowskis Algorithmus. Aber vielleicht sind Sie ja richtig schnell im Berechnen der Potenzmengenkonstruktion. Meiner Erfahrung nach ist die Hauptfehlerquelle bei der Anwendung von Brzozowskis Algorithmus, dass beim Berechnen der Reversal-Konstruktion vergessen wird, akzeptierende und Startzustände zu tauschen.



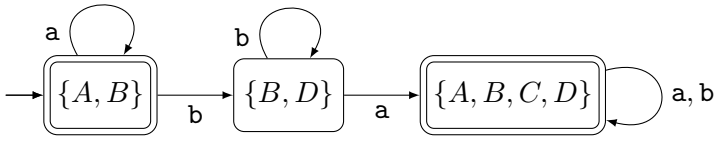
Anhand der Potenzmengenkonstruktion erhalten wir aus $\texttt{rev}(A)$ den folgenden DFA $\texttt{pot}(\texttt{rev}(A))$:



Um die Lesbarkeit der weiteren Konstruktionsschritte zu verbessern sind die Zustände von $\texttt{pot}(\texttt{rev}(A))$ zusätzlich zu den Mengen noch mit Buchstaben A bis D benannt. Durch erneutes Umdrehen erhalten wir den NNFA $\texttt{rev}(\texttt{pot}(\texttt{rev}(A)))$:



Als letzten Schritt führen wir nun noch eine weitere Potenzmengenkonstruktion aus und erhalten so den minimalen DFA $\texttt{pot}(\texttt{rev}(\texttt{pot}(\texttt{rev}(A))))$ (aus Lesbarkeitsgründen verwenden wir hier nur die Buchstaben um die Mengen der Potenzmengenkonstruktion zu bezeichnen):



$\Diamond$

Anhand der in diesem Abschnitt vorgestellten Techniken lässt sich noch eine weitere Abschlusseigenschaft nachweisen:

Lemma 3.69 *Die Klasse der regulären Sprachen ist abgeschlossen unter dem* suffix-*Operator.*

Beweis: Siehe Übung 3.9. □

3.2.3 NFAs mit ε-Übergängen

In diesem Abschnitt lernen wir eine weitere Verallgemeinerung von NFAs kennen. Während sowohl die „normalen“ NFAs als auch die NNFAs in jedem Schritt einen Buchstaben des Eingabewortes abarbeiten, können die sogenannten *NFAs mit ε-Übergängen* auch Zustandsübergänge verwenden, die mit dem leeren Wort beschriftet sind. Auch wenn sich der Sinn dieser Definition vielleicht nicht auf den ersten Blick erschließen mag werden wir doch bald sehen, dass uns diese Erweiterung viele Modellierungsaufgaben erleichtert.

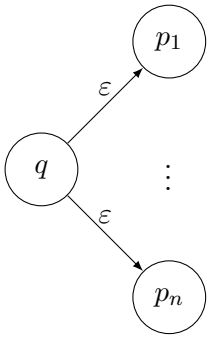
Definition 3.70 Ein **nichtdeterministischer endlicher Automat mit ε-Übergängen (ε-NFA)** A über einem Alphabet Σ wird definiert durch:

1. eine nicht-leere, endliche Menge Q von **Zuständen**,
2. eine Funktion $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ (die **Übergangsrelation**),
3. einen Zustand $q_0 \in Q$ (der **Startzustand**),
4. eine Menge $F \subseteq Q$ von **akzeptierenden Zuständen**.

Wir schreiben dies als $A := (\Sigma, Q, \delta, q_0, F)$.

Von der Definition des Automaten her ist der einzige Unterschied in der Definition also, dass der ε-NFA auch mit ε beschriftete Kanten hat. Diese Kanten bezeichnen wird auch als **ε-Übergänge**. Die graphische Darstellung für solche Übergänge ist entsprechend:

- Für jeden Zustand $q \in Q$ gilt: Angenommen, $\delta(q, \varepsilon) = \{p1, \ldots, p_n\}$ mit $n \geq 1$ (also $\delta(q, \varepsilon) \neq \emptyset$). Dann in der graphischen Darstellung von A zu jedem p_i (mit $1 \leq i \leq n$) einen Pfeil von q nach p_i, der mit ε beschriftet ist.



Um das Akzeptanzverhalten eines ε-NFA zu beschreiben benötigen wir allerdings etwas mehr Aufwand.

Definition 3.71 Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein ε-NFA. Für jeden Zustand $q \in Q$ sei sein **ε-Abschluss** ε-ABSCHLUSS(q) definiert durch

$$\varepsilon\text{-ABSCHLUSS}(q) := \{p \in Q \mid \ p \text{ ist von } q \text{ aus durch } \varepsilon\text{-Übergänge zu erreichen}\}.$$

Insbesondere gilt $q \in \varepsilon$-ABSCHLUSS(q) für alle $q \in Q$. Für jede Menge $P \subseteq Q$ von Zuständen definieren wir

$$\varepsilon\text{-ABSCHLUSS}(P) := \bigcup_{p \in P} \varepsilon\text{-ABSCHLUSS}(p).$$

Anhand der Übergangsrelation δ definieren wir die partielle Funktion $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$, und zwar durch die folgende rekursive Definition für alle $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$:

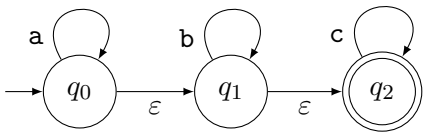
$$\hat{\delta}(q, \varepsilon) := \varepsilon\text{-ABSCHLUSS}(q),$$
$$\hat{\delta}(q, wa) := \varepsilon\text{-ABSCHLUSS}\left(\bigcup_{p \in \hat{\delta}(q,w)} \delta(p, a)\right).$$

Der ε-NFA A **akzeptiert** ein Wort $w \in \Sigma^*$, wenn $\left(\hat{\delta}(q_0, w) \cap F\right) \neq \emptyset$. Die von A **akzeptierte Sprache** $\mathcal{L}(A)$ ist definiert als die Menge aller von A akzeptierten Wörter, also

$$\mathcal{L}(A) := \left\{w \in \Sigma^* \mid \left(\hat{\delta}(q_0, w) \cap F\right) \neq \emptyset\right\}.$$

Auch wenn das auf den ersten Blick furchtbar formal aussieht, ist die Intuition dahinter recht einfach: Wann immer ein ε-NFA durch Lesen eines Buchstaben in einen Zustand q kommt, darf er „kostenlos“ in alle Zustände aus ε-ABSCHLUSS(q) weiterspringen.

Beispiel 3.72 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Der ε-NFA A über Σ sei definiert durch die folgende graphische Darstellung:



In diesem Fall ist auch ohne die formale Definition schon recht klar, welche Sprache der ε-NFA A akzeptiert: Zuerst liest A eine beliebige Zahl von `a`. Irgendwann entscheidet sich A, durch den ersten ε-Übergang in den Zustand q_1 zu wechseln. Dort wird eine beliebige Zahl von `b` abgearbeitet, und anschließend mit dem zweiten ε-Übergang in den

Zustand q_2 gewechselt. Dort kann dann noch c gelesen werden. Es gilt also:

$$\mathcal{L}(A) = \{\mathtt{a}\}^*\{\mathtt{b}\}^*\{\mathtt{c}\}^*.$$

Wir betrachten trotzdem noch einmal die Funktion ε-ABSCHLUSS. Diese hat die folgenden Werte:

$$\begin{aligned}\varepsilon\text{-ABSCHLUSS}(q_0) &= \{q_0, q_1, q_2\},\\ \varepsilon\text{-ABSCHLUSS}(q_1) &= \{q_1, q_2\},\\ \varepsilon\text{-ABSCHLUSS}(q_2) &= \{q_2\}.\end{aligned}$$

Exemplarisch betrachten wir einige der Funktionswerte von $\hat{\delta}$:

$$\begin{aligned}\hat{\delta}(q_0, \varepsilon) &= \varepsilon\text{-ABSCHLUSS}(q_0) = \{q_0, q_1, q_2\},\\ \hat{\delta}(q_0, \mathtt{b}) &= \varepsilon\text{-ABSCHLUSS}(q_1) = \{q_1, q_2\},\\ \hat{\delta}(q_0, \mathtt{abbc}) &= \varepsilon\text{-ABSCHLUSS}(q_2) = \{q_2\},\\ \hat{\delta}(q_0, \mathtt{cb}) &= \varepsilon\text{-ABSCHLUSS}(\emptyset) = \emptyset.\end{aligned}$$ ◊

Unser Ziel ist es nun, uns möglichst wenig mit $\hat{\delta}$ und ε-ABSCHLUSS herumschlagen zu müssen. Dazu zeigen wir, dass sich aus jedem ε-NFA die ε-Übergänge eliminieren lassen, und dass man auf diese Art einen NFA für die gleiche Sprache erhält. Dadurch können wir wann immer wir uns davon einen Vorteil versprechen ε-Übergänge verwenden, und können trotzdem bei Bedarf zu einem einfacher zu verstehenden NFA wechseln. Außerdem stellen wir so auch fest, dass jeder ε-NFA eine reguläre Sprache erzeugt.

Satz 3.73 *Zu jedem ε-NFA A lässt sich ein NFA A_N mit $\mathcal{L}(A) = \mathcal{L}(A_N)$ konstruieren, der genau so viele Zustände hat wie A.*

Beweisidee: Die eigentliche Arbeit bei der Elimination von ε-Übergängen ist schon durch die Definition von $\hat{\delta}$ getan. Wir ersetzen einfach δ durch $\hat{\delta}(q, a)$ für alle $q \in Q$ und alle $a \in \Sigma$. Da $\hat{\delta}(q, a)$ bereits ε-ABSCHLUSS verwendet, simuliert dies alle möglichen ε-Übergänge mit. Einzig und alleine beim Startzustand müssen wir noch ein wenig aufpassen. Betrachten Sie dazu am besten Beispiel 3.74, bevor Sie den folgenden Beweis lesen. □

Beweis: Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein ε-NFA. Wir definieren nun einen NFA $A_N := (\Sigma, Q, \delta_N, q_0, F_N)$ durch

$$F_N := \begin{cases} F \cup \{q_0\} & \text{falls } (\varepsilon\text{-ABSCHLUSS}(q_0) \cap F) \neq \emptyset,\\ F & \text{falls } (\varepsilon\text{-ABSCHLUSS}(q_0) \cap F) = \emptyset.\end{cases}$$

und $\delta_N(q, a) := \hat{\delta}(q, a)$ für alle $q \in Q$ und alle $a \in \Sigma$.

Offensichtlich ist A_N ein NFA, insbesondere enthält A keine ε-Übergänge (dennoch können wir δ wie gewohnt erweitern, so dass $\delta(q, \varepsilon) = \{q\}$ für alle $q \in Q$ gilt). Also müssen wir nur noch $\mathcal{L}(A) = \mathcal{L}(A_N)$ beweisen. Wir zeigen dazu zuerst, dass

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

für alle $w \in \Sigma^+$ gilt[29]. Diese Behauptung beweisen wir durch eine Induktion über den Aufbau von w.

INDUKTIONSANFANG: Sei $w = a \in \Sigma$.
Behauptung: Es gilt $\delta_N(q_0, w) = \hat{\delta}(q_0, w)$.
Beweis: Folgt direkt aus der Definition von δ_N, da $\delta_N(q_0, a) = \hat{\delta}(q_0, a)$.

INDUKTIONSSCHRITT: Seien $w \in \Sigma^+$, $a \in \Sigma$ beliebig.
Induktionsannahme: Es gelte $\delta_N(q_0, w) = \hat{\delta}(q_0, w)$.
Behauptung: Es gilt Es gilt $\delta_N(q_0, wa) = \hat{\delta}(q_0, wa)$.
Beweis: Folgendes gilt:

$$\begin{aligned}
\delta_N(q_0, wa) &= \bigcup_{p \in \delta_N(q_0, w)} \delta_N(p, a), && \text{(nach Def. von } \delta_N\text{)} \\
&= \bigcup_{p \in \hat{\delta}(q_0, w)} \delta_N(p, a) && \text{(nach Induktionsann.)} \\
&= \bigcup_{p \in \hat{\delta}(q_0, w)} \hat{\delta}(p, a) && \text{(nach Def. von } \delta_N\text{)} \\
&= \hat{\delta}(q_0, wa) && \text{(nach Def. von } \hat{\delta}\text{).}
\end{aligned}$$

Dies beendet die Induktion.

Wir zeigen nun, dass $\mathcal{L}(A_N) = \mathcal{L}(A)$, indem wir beweisen, dass für alle $w \in \Sigma^*$ (nun ist also auch $w = \varepsilon$ erlaubt) folgendes gilt:

$$(\delta_N(q_0, w) \cap F_N) \neq \emptyset \text{ genau dann, wenn } \hat{\delta}(q_0, w \cap F) \neq \emptyset.$$

Für $w = \varepsilon$ folgt diese Behauptung unmittelbar aus der Definition von F_N. Angenommen, $w \neq \varepsilon$. Dann existieren ein Wort $v \in \Sigma^*$ und ein Buchstabe $a \in \Sigma$ mit $w = va$. Wir betrachten die beiden Richtungen der Behauptung getrennt.
„$\Leftarrow$“: Angenommen, $\hat{\delta}(q_0, w \cap F) \neq \emptyset$. Da $F_N \supseteq F$ und da $\hat{\delta}(q_0, w) = \delta_N(q_0, w)$ (das haben wir soeben für alle $w \neq \varepsilon$ bewiesen) folgt unmittelbar $(\delta_N(q_0, w) \cap F_N) \neq \emptyset$.

„$\Rightarrow$“: Angenommen, $(\delta_N(q_0, w) \cap F_N) \neq \emptyset$. Um angenehmer argumentieren zu können definieren wir $P := (\delta_N(q_0, w) \cap F_N)$. Falls $q_0 \notin P$ oder falls $q_0 \in F$, folgt ebenfalls direkt die Behauptung. Nehmen wir also an, dass $q_0 \in P$ (und somit $q_0 \in F_N$) und $q_0 \notin F$. Dann existiert nach Definition von F_N ein Zustand $q_N \in (\varepsilon\text{-ABSCHLUSS}(q_0) \cap F)$. Außerdem gilt

$$\hat{\delta}(q_0, w) = \varepsilon\text{-ABSCHLUSS}\left(\delta\left(\hat{\delta}(q_0, v), a\right)\right).$$

Hieraus folgt $q_F \in \hat{\delta}(q_0, w)$ und somit $(\hat{\delta}(q_0, w) \cap F) \neq \emptyset$. □

[29] Den Fall $w = \varepsilon$ lassen wir für diese Behauptung bewusst aus, da $\delta_N(q_0, \varepsilon) = \{q_0\}$ gilt, während $\hat{\delta}(q_0, \varepsilon) = \varepsilon\text{-ABSCHLUSS}(q_0)$.

Wir können uns die Konstruktion von δ_N auch vereinfacht vorstellen. Für jeden Zustand $q \in Q$ und jeden Buchstaben $a \in \Sigma$ gilt nach unserer Definition von $\hat{\delta}$ und von δ_N:

$$\begin{aligned}
\delta_N(q,a) &= \hat{\delta}(q,a) \\
&= \varepsilon\text{-ABSCHLUSS}\left(\bigcup_{p \in \hat{\delta}(q,\varepsilon)} \delta(p,a)\right) \\
&= \varepsilon\text{-ABSCHLUSS}\left(\bigcup_{p \in \varepsilon\text{-ABSCHLUSS}(q)} \delta(p,a)\right) \\
&= \varepsilon\text{-ABSCHLUSS}\left(\left\{\delta(p,a) \mid p \in \varepsilon\text{-ABSCHLUSS}(q_0)\right\}\right).
\end{aligned}$$

Wir können also $\delta_N(q,a)$ mit Hilfe von δ und ε-ABSCHLUSS bestimmen, indem wir zuerst alle Zustände aus ε-ABSCHLUSS(q) auswählen, zu jedem dieser Zustande $p \in \varepsilon$-ABSCHLUSS(q) alle Folgezustände $\delta(p,a)$ berechnen, und von diesen Zuständen wiederum den ε-Abschluss bilden. Anschließend müssen wir nur daran denken, gegebenenfalls q_0 zu einem akzeptierenden Zustand zu machen, falls ε-ABSCHLUSS(q_0) einen akzeptierenden Zustand enthält.

Zusammengefasst lassen sich also ε-Übergänge auf die folgende Art leicht entfernen:

Algorithmus 2 (`entferneEpsilon`) Transformiert ε-NFA in äquivalenten NFA.

Eingabe: Ein ε-NFA $A := (\Sigma, Q, \delta, q_0, F)$.

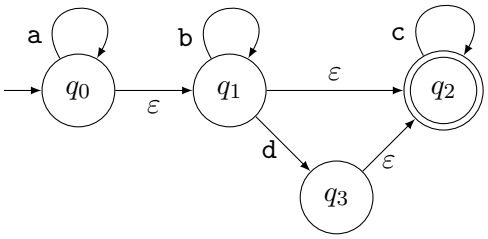
1. Berechne ε-ABSCHLUSS(q) für alle $q \in Q$.
2. Falls $(\varepsilon\text{-ABSCHLUSS}(q_0) \cap F) \neq \emptyset$: Füge q_0 zu F hinzu.
3. Entferne alle ε-Übergänge aus A.
4. Für alle $q \in Q$ und alle $a \in \Sigma$: Füge

   $$\varepsilon\text{-ABSCHLUSS}\left(\left\{\delta(p,a) \mid p \in \varepsilon\text{-ABSCHLUSS}(q_0)\right\}\right)$$

   zu $\delta(q,a)$ hinzu.

Durch diesen Ansatz können wir uns beim Arbeiten mit ε-NFAs die Definition von $\hat{\delta}$ vollkommen ersparen. Das folgende Beispiel soll diese Vorgehensweise illustrieren:

Beispiel 3.74 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. Der ε-NFA A über Σ sei definiert durch die folgende graphische Darstellung:



Die Tabelle der Überangsrelation δ von A lautet wie folgt:

	a	b	c	d	ε
q_0	$\{q_0\}$	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_1\}$
q_1	$\emptyset$	$\{q_1\}$	$\emptyset$	$\{q_3\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_2\}$

Die Zustände von A haben die folgenden ε-Abschlüsse:

$$\begin{aligned}
&\varepsilon\text{-ABSCHLUSS}(q_0) = \{q_0, q_1, q_2\},\\
&\varepsilon\text{-ABSCHLUSS}(q_1) = \{q_1, q_2\},\\
&\varepsilon\text{-ABSCHLUSS}(q_2) = \{q_2\},\\
&\varepsilon\text{-ABSCHLUSS}(q_3) = \{q_2, q_3\}.
\end{aligned}$$

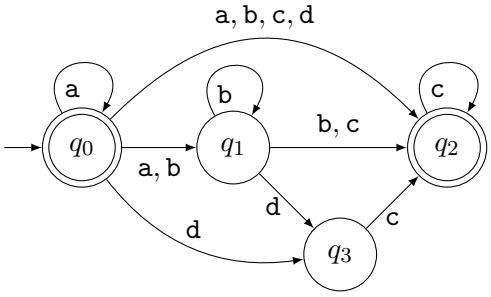
Anhand von δ und ε-ABSCHLUSS können wir nun δ_N berechnen. Wir betrachten dies an einigen Beispielen:

$$\begin{aligned}
\delta(q_0, \mathtt{a}) &= \varepsilon\text{-ABSCHLUSS}\left(\{\delta(p, \mathtt{a}) \mid p \in \varepsilon\text{-ABSCHLUSS}(q_0)\right)\\
&= \varepsilon\text{-ABSCHLUSS}\left(\{\delta(p, \mathtt{a}) \mid p \in \{q_0, q_1, q_2\}\right)\\
&= \varepsilon\text{-ABSCHLUSS}\left(\{q_0\}\right) = \{q_0, q_1, q_2\},\\
\delta(q_1, \mathtt{d}) &= \varepsilon\text{-ABSCHLUSS}\left(\{\delta(p, \mathtt{d}) \mid p \in \varepsilon\text{-ABSCHLUSS}(q_1)\right)\\
&= \varepsilon\text{-ABSCHLUSS}\left(\{\delta(p, \mathtt{d}) \mid p \in \{q_1, q_2\}\right)\\
&= \varepsilon\text{-ABSCHLUSS}\left(\{q_2, q_3\}\right)\\
&= \{q_2, q_3\}.
\end{aligned}$$

Insgesamt können wir auf diese Art δ_N wie in der folgenden Tabelle dargestellt berechnen:

	a	b	c	d
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$
q_1	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$	$\{q_2, q_3\}$
q_2	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\{q_2\}$	$\emptyset$

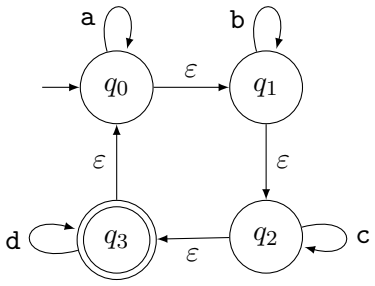
Der dazugehörende NFA A_N hat die folgende graphische Darstellung:



Da $q_2 \in \varepsilon$-ABSCHLUSS(q_0) wurde außerdem q_0 zu einem akzeptierenden Zustand gemacht. ◊

Die Konvertierung eines ε-NFA in einen NFA fügt zwar keine Zustände hinzu; je nach Struktur des ε-NFA kann aber eine beachtliche Zahl an neuen Kanten eingeführt werden. Im schlimmsten Fall kann dies sogar dazu führen, dass jeder Zustand für jeden Buchstaben alle Zustände als Folgezustände hat. Wir betrachten dazu das folgende Beispiel:

Beispiel 3.75 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. Der ε-NFA A sei wie folgt definiert:



Es gilt ε-ABSCHLUSS$(q_i) = \{q_0, q_1, q_2, q_3\}$ für $0 \leq i \leq 3$. Wenn wir nun durch Entfernen der ε-Übergänge den entsprechenden NFA A_N konstruieren, muss $\delta(q_i, a) = \{q_0, q_1, q_2, q_3\}$ für jeden Buchstaben $a \in \Sigma$ und jeden Zustand q_i $(0 \leq i \leq 3)$ gelten. Dadurch ist A_N recht schwer graphisch darzustellen. (Natürlich sind noch unübersichtlichere Beispiele möglich.) ◊

Wie wir festgestellt haben, erlauben auch ε-Übergänge nicht die Definition von nichtregulären Sprachen. Die Beobachtung, dass das Entfernen solcher Übergänge zu schlechter lesbaren Automaten führt, können wir auch positiv deuten: Durch Verwendung von ε-Übergängen können NFAs lesbarer werden, und die Konstruktion von endlichen Automaten kann dadurch deutlich einfacher werden. Wir betrachten dies zuerst anhand einiger neuer Abschlusseigenschaften.

3.2.4 Konstruktionen mit ε-Übergängen

Im Beweis von Korollar 3.20 haben wir bereits gesehen, wie sich aus zwei DFAs A_1, A_2 ein Produktautomat für die Vereinigungssprache $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$ konstruieren lässt. Die

Zahl der Zustände des Produktautomaten entspricht dabei dem Produkt der Anzahlen der Zustände von A_1 und A_2. Anhand von ε-NFAs lässt sich die Vereinigungssprache mit weniger Zuständen konstruieren:

Lemma 3.76 *Sei A_1 ein NFA mit n_1 und A_2 ein NFA mit n_2 Zuständen. Dann existiert ein ε-NFA A_V mit $n_1 + n_2 + 1$ Zuständen, und $\mathcal{L}(A_V) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.*

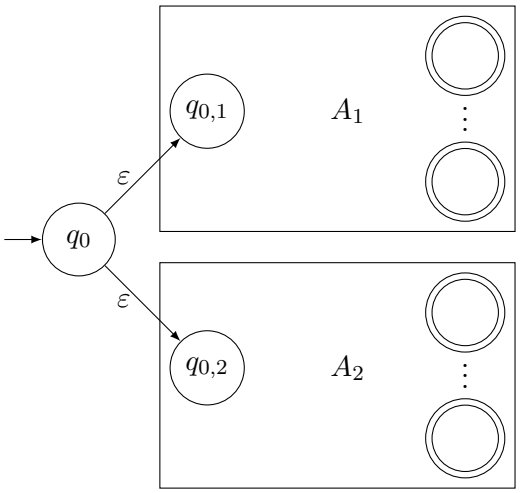
Beweis: Sei $A_1 := (\Sigma, Q_1, \delta_1, q_{0,1}, F_1)$ und $A_2 := (\Sigma, Q_2, \delta_2, q_{0,2}, F_2)$. Ohne Beeinträchtigung der Allgemeinheit gelte $(Q_1 \cap Q_2) = \emptyset$. Sei nun q_0 ein neuer Zustand, es gelte also $q_0 \notin (Q_1 \cap Q_2)$. Wir definieren $A_V := (\Sigma, Q, \delta_V, q_0, F)$, wobei

$$
\begin{aligned}
Q &:= Q_1 \cup Q_2 \cup \{q_0\}, \\
F &:= F_1 \cup F_2, \\
\delta_V(q,a) &:= \begin{cases} \delta_1(q,a) & \text{falls } q \in Q_1, \\ \delta_2(q,a) & \text{falls } q \in Q_2 \end{cases}
\end{aligned}
$$

für alle $a \in \Sigma$, sowie

$$\delta_V(q_0, \varepsilon) := \{q_{0,1}, q_{0,2}\}.$$

Anschaulich gesehen besteht der ε-NFA A_V also aus A_1 und A_2, sowie q_0 als neuem Startzustand. Die graphische Darstellung von A_V lässt sich folgendermaßen skizzieren:



Der ε-NFA A_V kann sich also zu Beginn nichtdeterministisch entscheiden, ob er A_1 oder A_2 simuliert, und arbeitet danach genauso wie der entsprechende ε-NFA.

Die Korrektheit dieser Konstruktion ist eigentlich offensichtlich, trotzdem stellen wir das explizit fest. Für alle $w \in \Sigma^*$ gilt:

$$
\begin{aligned}
& & w \in \mathcal{L}(A_V) \\
\Leftrightarrow & & \big(\delta_V(q_0, w) \cap F\big) \neq \emptyset \\
\Leftrightarrow & & \big(\delta_V(q_0, w) \cap (F_1 \cup F_2)\big) \neq \emptyset \\
\Leftrightarrow & & \big(\big(\delta_V(q_0, w) \cap F_1\big) \neq \emptyset\big) \text{ oder } \big(\big(\delta_V(q_0, w) \cap F_2\big) \neq \emptyset\big).
\end{aligned}
$$

Da jeder Pfad von q_0 zu einem Zustand in F_1 durch $q_{0,1}$ führt, und da analog jeder Pfad zu einem Zustand aus F_2 durch $q_{0,2}$ führt, können wir in der obigen Aussage die beiden Vorkommen von q_0 durch $q_{0,1}$ und $q_{0,2}$ ersetzen. Also gilt:

$$\begin{aligned} & w \in \mathcal{L}(A_V) \\ \Leftrightarrow \quad & \big(\big(\delta_V(q_{0,1}, w) \cap F_1\big) \neq \emptyset\big) \text{ oder } \big(\big(\delta_V(q_{0,2}, w) \cap F_2\big) \neq \emptyset\big) \\ \Leftrightarrow \quad & w \in \mathcal{L}(A_1) \text{ oder } w \in \mathcal{L}(A_2) \\ \Leftrightarrow \quad & w \in (\mathcal{L}(A_1) \cup \mathcal{L}(A_2)). \end{aligned}$$

Somit ist $\mathcal{L}(A_V) = (\mathcal{L}(A_1) \cup \mathcal{L}(A_2))$, unsere Konstruktion ist also korrekt. □

Da die Transformation eines ε-NFA in einen NFA die Zustandszahl nicht verändert, können wir auch einen NFA mit $n_1 + n_2 + 1$ Zuständen konstruieren.

Auf eine ähnliche Art können wir weitere Abschlusseigenschaften zeigen.

Lemma 3.77 *Sei A_1 ein NFA mit n_1 und A_2 ein NFA mit n_2 Zuständen. Dann existiert ein NFA A_K mit $n_1 + n_2$ Zuständen, und $\mathcal{L}(A_K) = \mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$.*

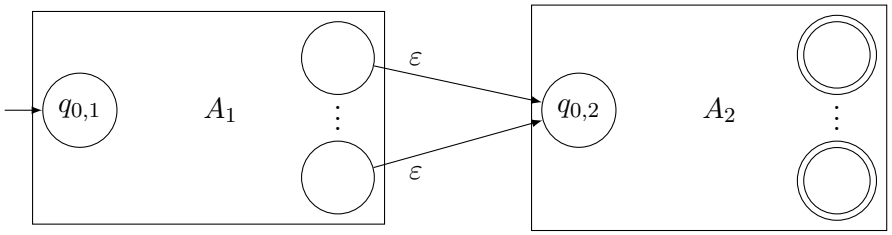
Beweis: Sei $A_1 := (\Sigma, Q_1, \delta_1, q_{0,1}, F_1)$ und $A_2 := (\Sigma, Q_2, \delta_2, q_{0,2}, F_2)$. Ohne Beeinträchtigung der Allgemeinheit gelte $(Q_1 \cap Q_2) = \emptyset$. Wir definieren $A_K := (\Sigma, Q_1 \cup Q_2, \delta_K, q_{0,1}, F_2)$, wobei

$$\delta_K(q, a) := \begin{cases} \delta_1(q, a) & \text{falls } q \in Q_1, \\ \delta_2(q, a) & \text{falls } q \in Q_2 \end{cases}$$

für alle $a \in \Sigma$, sowie

$$\delta_K(q, \varepsilon) := \{q_{0,2}\}$$

für alle $q \in F_1$. Anschaulich gesehen entsteht A_K durch „Hintereinanderschalten“ von A_1 und A_2. Die akzeptierenden Zustände von A_1 sind nicht mehr akzeptierend, stattdessen sind sie durch ε-Übergänge mit dem früheren Startzustand von A_2 verbunden (der nun auch kein Startzustand mehr ist). Die graphische Darstellung von A_K kann folgendermaßen skizziert werden:



Der ε-NFA A_K simuliert also zuerst A_1. Gelangt er in einen Zustand, in dem A_1 akzeptieren würde, kann A_K in den Startzustand von A_2 übergehen.

Nun gilt es noch, die Korrektheit von A_K zu beweisen. Angenommen, $w \in (\mathcal{L}(A_1) \cdot \mathcal{L}(A_2))$. Dann existieren Wörter $w_1 \in \mathcal{L}(A_1)$ und $w_2 \in \mathcal{L}(A_2)$ mit $w = w_1 \cdot w_2$. Also existiert auch ein Zustand $q_1 \in F_1$ mit $q_1 \in \delta_1(q_{0,1}, w_1)$ (also einer der Zustände, mit denen A_1 das Wort w_1) akzeptiert. Also kann A_K beim Einlesen von w_1 den Zustand q_1

erreichen, dann mit einem ε-Übergäng zu $q_{0,2}$ wechseln. Da $w_2 \in \mathcal{L}(A_2)$ ist $(\delta_2(q_{0,2}, w_2) \cap F_2) \neq \emptyset$, also kann auch A_K von $q_{0,2}$ aus einen akzeptierenden Zustand erreichen. Es gilt also $\mathcal{L}(A_K) \supseteq (\mathcal{L}(A_1) \cdot \mathcal{L}(A_2))$.

Angenommen, $w \in \mathcal{L}(A_K)$. Dann existiert ein Pfad in A_K, der von q_0 zu einem Zustand aus F verläuft und mit w beschriftet ist. Gemäß der Definition von A_K passiert dieser Pfad zuerst nur Zustände aus Q_1, und danach nur Zustände aus Q_2. Daher lässt sich w entsprechend in Wörter w_1 und w_2 zerlegen (mit $w = w_1 \cdot w_2$). Der letzte Zustand aus Q_1 muss zur Menge F_1 gehören, also gilt $w_1 \in \mathcal{L}(A_1)$. Der erste Zustand aus Q_2 muss $q_{0,2}$ sein, also gilt $w_2 \in \mathcal{L}(A_2)$ (denn der Pfad endet ja auch in einem Zustand aus F, und $F = F_2$). Somit ist $w \in (\mathcal{L}(A_1) \cdot \mathcal{L}(A_2))$. Also gilt $\mathcal{L}(A_K) \subseteq (\mathcal{L}(A_1) \cdot \mathcal{L}(A_2))$, und damit auch $\mathcal{L}(A_K) = (\mathcal{L}(A_1) \cdot \mathcal{L}(A_2))$. Der Automat A_K ist also korrekt. □

Als direkte Konsequenz von Lemma 3.77 können wir festhalten, dass die Klasse der regulären Sprachen unter Konkatenation abgeschlossen ist. Die gleiche Beweistechnik lässt sich auch verwenden, um den Abschluss unter n-facher Konkatenation zu zeigen:

Lemma 3.78 *Die Klasse der regulären Sprachen ist abgeschlossen unter n-facher Konkatenation.*

Beweis. Angenommen, $n \geq 2$ und $L \in \text{REG}$. Dann existiert ein ε-NFA A mit $\mathcal{L}(A) = L$. Analog zum Beweis von Lemma 3.77 können wir nun n Kopien von A hintereinanderschalten und erhalten so einen ε-NFA für die Sprache $\mathcal{L}(A)^n = L^n$. Somit ist L^n regulär. □

Lemma 3.79 *Sei A ein NFA mit n Zuständen. Dann existiert ein NFA A_S mit $n+1$ Zuständen, und $\mathcal{L}(A_S) = \mathcal{L}(A)^*$.*

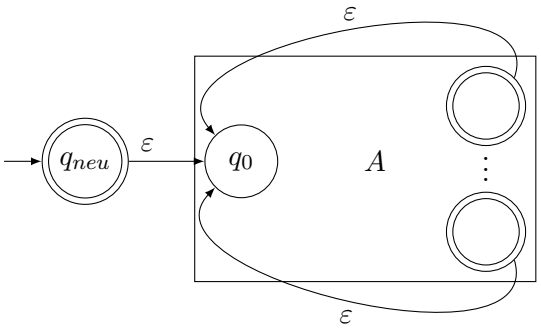
Beweis: Sei $A := (\Sigma, Q, \delta, q_0, F)$, und sei q_{neu} ein neuer Zustand, es gelte also $q_{neu} \notin Q$. Wir definieren nun $A_S := (\Sigma, Q_S, \delta_S, q_{neu}, F_S)$, wobei

$$\begin{aligned} Q_S &:= Q \cup \{q_{neu}\}, \\ F_S &:= F \cup \{q_{neu}\}, \\ \delta_S(q, a) &:= \delta(q, a) \end{aligned}$$

für alle $q \in Q$ und alle $a \in \Sigma$, sowie

$$\delta_S(q, \varepsilon) := \{q_0\}$$

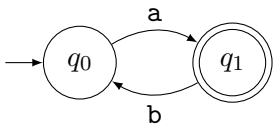
für alle $q \in (F \cup \{q_{neu}\})$. Die graphische Darstellung lässt sich wie folgt skizzieren:



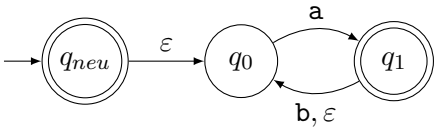
Im Prinzip hat diese Konstruktion die gleiche Grundidee wie die für die Konkatenation (Lemma 3.77), nur dass der NFA A mit sich selbst konkateniert wird. Dadurch entstehen schleifen, die beliebig oft durchlaufen werden können. Der genaue Beweis der Korrektheit sei Ihnen als Übung überlassen. Der neue Startzustand q_{neu} ist notwendig, damit nicht versehentlich zu viele Wörter akzeptiert werden (siehe Beispiel 3.80). □

Wir betrachten nun ein Beispiel, das rechtfertigt, warum in der Konstruktion von Lemma 3.79 ein neuer Startzustand eingeführt wird:

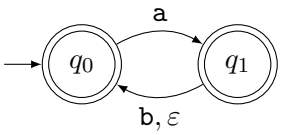
Beispiel 3.80 Gegeben sei der folgende NFA A für die Sprache $L := \{\texttt{a}\} \cdot \{\texttt{ba}\}^*$:



Wir interessieren uns nun für die Sprache L^*. Aus dem Beweis von Lemma 3.79 wissen wir, dass der folgende ε-NFA A_S diese Sprache akzeptiert:



Angenommen, wir nehmen die gleiche Konstruktion, aber statt q_{neu} einzuführen machen wir einfach q_0 zu einem akzeptierenden Zustand. Der so entstehende ε-NFA A_F hat die folgende Darstellung:



Nun gilt aber $\texttt{ab} \in \mathcal{L}(A_F)$, aber $\texttt{ab} \notin L^+$. (Diese Behauptung können wir auf verschiedene Arten zeigen: Zum Beispiel wissen wir, dass $\mathcal{L}(A_S) = L^+$, da $\texttt{ab} \notin \mathcal{L}(A_K)$ gilt

$\texttt{ab} \notin L^+$. Alternativ können wir feststellen, dass kein Wort in L auf $\texttt{b}$ endet. Also endet auch kein Wort in L^+ auf $\texttt{b}$, somit muss $\texttt{ab} \notin L^+$ gelten.) Also ist $\mathcal{L}(A_F)$ nicht korrekt für L^+. ◊

Aus Lemma 3.79 folgt direkt, dass die Klasse der regulären Sprachen unter Kleene-Stern abgeschlossen ist. Den Abschluss unter Kleene-Plus bekommen wir geschenkt:

Korollar 3.81 *Die Klasse der regulären Sprachen ist abgeschlossen unter Kleene-Plus.*

Beweis: Sei $L \in \text{REG}$. Es gilt $L^+ = L \cdot (L^*)$. Da REG unter Kleene-Stern abgeschlossen ist (Lemma 3.79) ist $(L^*) \in \text{REG}$. Da REG unter Konkatenation abgeschlossen ist (Lemma 3.77) ist $(L \cdot (L^*)) \in \text{REG}$. Also ist $L \in \text{REG}$, und wir können feststellen, dass REG unter Kleene-Plus abgeschlossen ist. □

Hinweis 3.82 Die Konstruktionen in Lemma 3.76 (Vereinigung), Lemma 3.77 (Konkatenation) und Lemma 3.79 (Kleene-Stern) verwenden als Eingabe NFAs, keine ε-NFAs. Es ist aber problemlos möglich, diese Konstruktionen auf ε-NFAs zu erweitern. Dabei muss man bei der formalen Definition etwas aufpassen: In der Konstruktion für die Konkatenation (Lemma 3.77) werden alle Zustände, die in A_1 akzeptierende Zustände sind, durch ε-Übergänge mit dem Startzustand von A_2 verbunden. Formal schreiben wir das als

$$\delta(q, \varepsilon) := \{q_{0,2}\} \text{ für alle } q \in F_1.$$

Wenn A_1 kein NFA, sondern ein ε-NFA ist, muss diese Stelle folgendermaßen angepasst werden:

$$\delta(q, \varepsilon) := \{q_{0,2}\} \cup \delta_1(q, \varepsilon) \text{ für alle } q \in F_1,$$

da ansonsten die aus akzeptierenden Zuständen von A_1 hinausgehenden ε-Übergänge verloren gehen würden. Ähnliches gilt für die anderen Konstruktionen. (Für Lemma 3.76 müssen wir nur daran denken, $\delta_V(q, \varepsilon)$ überhaupt zu definieren.)

3.3 Reguläre Ausdrücke

Inzwischen verfügen wir über eine ansehnliche Zahl von Werkzeugen, um mit regulären Sprachen zu arbeiten. Um zu zeigen, dass eine Sprache regulär ist, können wir über den Index ihrer Nerode-Relation argumentieren, oder einen endlichen Automaten angeben. In vielen Fällen können wir uns diese Arbeit aber auch noch erleichtern, indem wir auf eine Vielzahl von Abschlusseigenschaften zurückgreifen.

Für viele Anwendungen ist diese Vorgehensweise allerdings ein wenig zu umständlich. Gerade wenn man eine reguläre Sprache in einer Form spezifizieren will, die von einem Computer verarbeitet werden soll, können Automaten schnell zu umständlich werden; aber auch sonst ist es oft unnötig anstrengend, einen Automaten anzugeben und dann gegebenenfalls noch über Abschlusseigenschaften argumentieren zu müssen. Zu diesem

Zweck führen wir ein weiteres Modell zur Definition von Sprachen ein, das nicht auf Automaten basiert (jedenfalls nicht direkt), die *regulären Ausdrücke*. Diese sind wie folgt definiert:

Definition 3.83 Sei Σ ein Alphabet. Die Menge der **regulären Ausdrücke (über Σ)** und der durch sie definierten Sprachen ist wie folgt rekursiv definiert:

1. $\emptyset$ ist ein regulärer Ausdruck, und $\mathcal{L}(\emptyset) := \emptyset$.
2. ε ist ein regulärer Ausdruck, und $\mathcal{L}(\varepsilon) := \{\varepsilon\}$.
3. Jeder Buchstabe $a \in \Sigma$ ist ein regulärer Ausdruck, und $\mathcal{L}(a) := \{a\}$.
4. **Konkatenation:** Sind α und β reguläre Ausdrücke, dann ist auch $(\alpha \cdot \beta)$ ein regulärer Ausdruck, und $\mathcal{L}((\alpha \cdot \beta)) := \mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$.
5. **Vereinigung:**[30]Sind α und β reguläre Ausdrücke, dann ist auch $(\alpha \mid \beta)$ ein regulärer Ausdruck, und $\mathcal{L}\left((\alpha \mid \beta)\right) := \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$.
6. **Kleene-Stern:** Ist α ein regulärer Ausdruck, dann ist auch α^* ein regulärer Ausdruck, und $\mathcal{L}\left(\alpha^*\right) := \mathcal{L}(\alpha)^*$.

Im Prinzip haben wir bei Schreibweisen wie $\{a \cdot b\}^*$ schon indirekt eine ähnliche Schreibweise benutzt. Wie wir in Abschnitt 3.3.1 sehen werden, können reguläre Ausdrücke genau die Klasse der regulären Sprachen definieren; sie haben also die gleiche Ausdrucksstärke wie die verschiedenen Varianten von endlichen Automaten, die wir bisher kennen gelernt haben. Bevor wir uns dem entsprechenden Beweis zuwenden, vereinfachen wir unsere Notation.

Notation 3.84 Um uns das Lesen und Schreiben von regulären Ausdrücken zu vereinfachen, vereinbaren wir folgende Konventionen:

1. Für alle regulären Ausdrücke α können wir anstelle von $(\alpha \cdot \alpha^*)$ auch α^+ schreiben.
2. Der Konkatenationspunkt $\cdot$ kann weggelassen werden, wir können also $(\alpha\beta)$ anstelle von $(\alpha \cdot \beta)$ schreiben.
3. Bei Ketten gleichartiger Operator können Klammern weggelassen werden, wir schreiben also
   - $(\alpha \mid \beta \mid \gamma)$ statt $((\alpha \mid \beta) \mid \gamma)$ oder $(\alpha \mid (\beta \mid \gamma))$, und
   - $(\alpha \cdot \beta \cdot \gamma)$ statt $((\alpha \cdot \beta) \cdot \gamma)$ oder $(\alpha \cdot (\beta \cdot \gamma))$.
4. Wir vereinbaren die folgenden Präzedenzregeln:

[30]In der Literatur werden anstelle von $\mid$ auch die Symbole $+$ und $\cup$ verwendet.

   a) der Kleene-Stern * bindet stärker als die Konkatenation $\cdot$, und
   b) die Konkatenation $\cdot$ bindet stärker als die Vereinigung $|$.

   Als Merkregel: Stern vor Punkt vor Strich[31].

5. Äußere Klammern, die einen Ausdruck umschließen, können weggelassen werden.
6. Das Setzen zusätzlicher Klammern ist immer erlaubt; allerdings sollte darauf geachtet werden, dass es die Lesbarkeit nicht verschlechtert.

Wir illustrieren diese Notationen durch ein paar Beispiele:

Beispiel 3.85 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}\}$

1. Der reguläre Ausdruck $\mathtt{ab}^* \mid \mathtt{c}^*\mathtt{d} \mid \mathtt{e}$ ist eine verkürzte Schreibweise des Ausdrucks

   $$(\mathtt{ab}^* \mid \mathtt{c}^*\mathtt{d} \mid \mathtt{e})$$

   Aus diesem wiederum erhalten wir (durch Wiedereinfügen der weggelassenen Konkatenationspunkte)

   $$(\mathtt{a} \cdot \mathtt{b}^* \mid \mathtt{c}^* \cdot \mathtt{d} \mid \mathtt{e})$$

   Da $\cdot$ eine höhere Präzedenz hat als $|$ („Punkt vor Strich“) steht dieser Ausdruck für

   $$((\mathtt{a} \cdot \mathtt{b}^*) \mid (\mathtt{c}^* \cdot \mathtt{d}) \mid \mathtt{e}),$$

   den wir wiederum mit zusätzlichen Klammern auch als

   $$\Big(\big(\mathtt{a} \cdot (\mathtt{b}^*)\big) \mid \big((\mathtt{c}^*) \cdot \mathtt{d}\big) \mid \mathtt{e}\Big)$$

   schreiben können.

2. Der Ausdruck $\mathtt{ab}^+$ ist eine alternative Schreibweise für den Ausdruck

   $$\mathtt{a} \cdot (\mathtt{b}^+).$$

   Durch Auflösen der Kurzschreibweise $^+$ erhalten wir

   $$\mathtt{a} \cdot (\mathtt{b} \cdot \mathtt{b}^*),$$

   das $^+$ bezieht also nicht das $\mathtt{a}$ mit ein.

3. Sei $L := \{\mathtt{a}\}\cdot\{\mathtt{a}\cdot\mathtt{b}\}^*$. Diese Sprache wird auch durch den regulären Ausdruck $\mathtt{a}(\mathtt{ab})^*$ beschrieben. Anstelle von $w \in \{\mathtt{a}\}\cdot\{\mathtt{a}\cdot\mathtt{b}\}^*$ können wir nun also auch $w \in \mathcal{L}(\mathtt{a}(\mathtt{ab})^*)$ schreiben. ◊

[31] Hier ist Vorsicht geboten, denn diese Eselsbrücke hat einen kleinen Haken. Sie müssen aufpassen, dass sie nicht mit dem Kleene-Plus $^+$ durcheinander kommen, da $\alpha^+ = \alpha \cdot \alpha^*$ gilt. Das Kleene-Plus ist also kein Strich im Sinne dieser Merkregel.

Es ist leicht zu sehen, dass eine Sprache durch mehrere unterschiedliche reguläre Ausdrücke definiert werden kann. Manchmal ist diese Äquivalenz offensichtlich (zum Beispiel gilt $\mathcal{L}(\mathtt{a} \mid \mathtt{b}) = \mathcal{L}(\mathtt{b} \mid \mathtt{a})$), in anderen Fällen ist sie nicht so einfach zu sehen. In Abschnitt A.3 finden Sie einige Rechenregeln, die im Umgang mit regulären Ausdrücken hilfreich sein können. Die Beweise sind Ihnen als Übung überlassen, um Ihnen dabei zu helfen betrachten wir aber ein paar kleine Beispiele:

Beispiel 3.86 Wir betrachten ein paar Rechenregeln für reguläre Ausdrücke, zusammen mit ihren Beweisen.

1. Für alle regulären Ausdrücke α gilt $\mathcal{L}(\alpha \cdot \emptyset) = \mathcal{L}(\emptyset)$, denn

$$\begin{aligned}
\mathcal{L}(\alpha \cdot \emptyset) &= \mathcal{L}(\alpha) \cdot \mathcal{L}(\emptyset) && \text{(nach Def. von } \mathcal{L}(\alpha \cdot \mathcal{L}(\beta))) \\
&= \mathcal{L}(\alpha) \cdot \emptyset && \text{(da } \mathcal{L}(\emptyset) = \emptyset) \\
&= \{u \cdot v \mid u \in \mathcal{L}(\alpha), v \in \emptyset\} && \text{(nach Def. von } \cdot \text{ auf Sprachen)} \\
&= \emptyset = \mathcal{L}(\emptyset)
\end{aligned}$$

2. Für alle regulären Ausdrücke α, β gilt $\mathcal{L}(\alpha \mid \beta) = \mathcal{L}(\beta \mid \alpha)$, weil

$$\begin{aligned}
\mathcal{L}(\alpha \mid \beta) &= \mathcal{L}(\alpha) \cup \mathcal{L}(\beta) && \text{(nach Def. von } \mathcal{L}(\alpha \cdot \mathcal{L}(\beta))) \\
&= \mathcal{L}(\beta) \cup \mathcal{L}(\alpha) && \text{(weil } \cup \text{ kommutativ ist)} \\
&= \mathcal{L}(\beta \mid \alpha).
\end{aligned}$$

3. Es gilt $\mathcal{L}(\emptyset^*) = \mathcal{L}(\varepsilon)$, wegen

$$\begin{aligned}
\mathcal{L}(\emptyset^*) &= \mathcal{L}(\emptyset)^* && \text{(nach Def. von } \mathcal{L}(\alpha^*)) \\
&= \emptyset^* && \text{(nach Def. von } \mathcal{L}(\emptyset)) \\
&= \bigcup_{n \in \mathbb{N}} \emptyset^n && \text{(nach Def. von } ^*) \\
&= \emptyset^0 \cup \bigcup_{n \in \mathbb{N}_{>0}} \emptyset^n && \text{(betrachten } n = 0 \text{ getrennt)} \\
&= \emptyset^0 \cup \emptyset && \text{(da } \emptyset \cdot \emptyset = \emptyset \text{ ist } \emptyset^n = \emptyset), \\
&= \{\varepsilon\} \cup \emptyset && \text{(nach Def. von } L^0), \\
&= \{\varepsilon\} = \mathcal{L}(\{\varepsilon\}).
\end{aligned}$$

   Andererseits gilt $\mathcal{L}(\emptyset^+) = \mathcal{L}(\emptyset)$, da

$$\begin{aligned}
\mathcal{L}(\emptyset^+) &= \mathcal{L}(\emptyset \cdot \emptyset^*) && \text{(da } \alpha^+ \text{ Kurzschreibweise von } \alpha \cdot \alpha^*) \\
&= \mathcal{L}(\emptyset) \cdot \mathcal{L}(\emptyset^*) && \text{(nach Def. von } \mathcal{L}(\alpha \cdot \mathcal{L}(\beta))) \\
&= \emptyset \cdot \{\varepsilon\} && \text{(wegen der vorigen Rechenregel)} \\
&= \emptyset = \mathcal{L}(\emptyset). && \lozenge
\end{aligned}$$

3.3.1 Reguläre Ausdrücke und endliche Automaten

Wie der Name bereits vermuten lässt, können anhand regulärer Ausdrücke genau die regulären Sprachen definiert werden:

Satz 3.87 *Sei Σ ein Alphabet. Dann gilt für jede Sprache $L \subseteq \Sigma^*$:*

L ist genau dann regulär, wenn ein regulärer Ausdruck α existiert, für den $\mathcal{L}(A) = L$.

Wir werden beide Richtungen dieses Beweises getrennt als Lemmata zeigen; die Hin-Richtung $\Rightarrow$ zeigen wir durch Lemma 3.91, die Rück-Richtung $\Leftarrow$ durch Lemma 3.88. Da wir uns alle Resultate, die wir für den Beweis von Lemma 3.88 benötigen, bereits erarbeitet haben, und da für den Beweis von Lemma 3.91 noch zusätzliches Werkzeug definiert werden muss, beginnen wir mit der Rück-Richtung. Wir zeigen also, dass reguläre Ausdrücke stets reguläre Sprachen definieren.

Lemma 3.88 *Sei Σ ein Alphabet und α ein regulärer Ausdruck über Σ. Dann ist $\mathcal{L}(\alpha)$ regulär.*

Beweis: Diese Behauptung folgt durch eine einfache Induktion über den Aufbau der regulären Ausdrücke.

Induktionsanfang: Sei $\alpha \in (\{\emptyset, \varepsilon\} \cup \Sigma)$ (dies deckt alle Fälle der drei Basisregeln ab). Dann ist $\mathcal{L}(\alpha)$ endlich und somit regulär. (Wenn wir lieber konstruktiv argumentieren wollen, ist ein entsprechender DFA schnell angegeben.)

Induktionsschritt: Wir betrachten die drei möglichen Basisregeln:

1. Seien α, β reguläre Ausdrücke und seien $\mathcal{L}(\alpha), \mathcal{L}(\beta) \in \text{REG}$. Dann ist $(\alpha \cdot \beta)$ ein regulärer Ausdruck mit $\mathcal{L}((\alpha \cdot \beta)) = \mathcal{L}(\alpha) \cdot \mathcal{L}(\beta)$. Gemäß Lemma 3.77 ist $\mathcal{L}((\alpha \cdot \beta))$ regulär.

2. Seien α, β reguläre Ausdrücke und seien $\mathcal{L}(\alpha), \mathcal{L}(\beta) \in \text{REG}$. Dann ist $(\alpha \mid \beta)$ ein regulärer Ausdruck mit $\mathcal{L}((\alpha \mid \beta)) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$. Gemäß Lemma 3.76 (oder Korollar 3.20) ist $\mathcal{L}((\alpha \mid \beta))$ regulär.

3. Sei α ein regulärer Ausdruck und sei $\mathcal{L}(\alpha) \in \text{REG}$. Dann ist α^+ ein regulärer Ausdruck mit $\mathcal{L}(\alpha^*) = \mathcal{L}(\alpha^*)$. Gemäß Lemma 3.79 ist $\mathcal{L}(\alpha^*)$ regulär.

Also ist für jeden regulären Ausdruck α die Sprache $\mathcal{L}(\alpha)$ regulär. □

Anhand der Resultate aus Abschnitt 3.2.4 können wir also nicht nur zeigen, dass reguläre Ausdrücke stets reguläre Sprachen definieren, sondern erhalten auch gleich ein Konstruktionsverfahren, mit dessen Hilfe wir aus einem gegeben regulären Ausdruck α einen ε-NFA für $\mathcal{L}(\alpha)$ konstruieren können. In Abschnitt A.2 im Anhang finden Sie noch einmal einem Übersicht über dieses Konstruktionsverfahren.

Um die andere Richtung von Satz 3.87 zu beweisen, benötigen wir noch etwas zusätzliches Instrumentarium (der Beweis wird dann direkt aus Lemma 3.91 folgen). Aus dem Beweis von Lemma 3.88 wissen wir nicht nur, dass reguläre Ausdrücke stets reguläre

Sprachen beschreiben, sondern auch, dass sie sich direkt in ε-NFAs konvertieren lassen. Wir können daher ε-NFAs gefahrlos erweitern, indem wir an deren Übergängen nicht nur Buchstaben und ε zulassen, sondern beliebige reguläre Ausdrücke:

Definition 3.89 Ein **erweiterter nichtdeterministischer endlicher Automat (ENFA)** A über einem Alphabet Σ wird definiert durch:

1. eine nicht-leere, endliche Menge Q von **Zuständen**,
2. eine Funktion $\delta : (Q \times Q) \to \mathrm{RX}_\Sigma$ (die **Beschriftungsfunktion**), wobei RX_Σ die Menge aller regulären Ausdrücke über Σ bezeichne[32],
3. einen Zustand $q_0 \in Q$ (der **Startzustand**),
4. eine Menge $F \subseteq Q$ von **akzeptierenden Zuständen**.

Wir schreiben dies als $A := (\Sigma, Q, \delta, q_0, F)$.

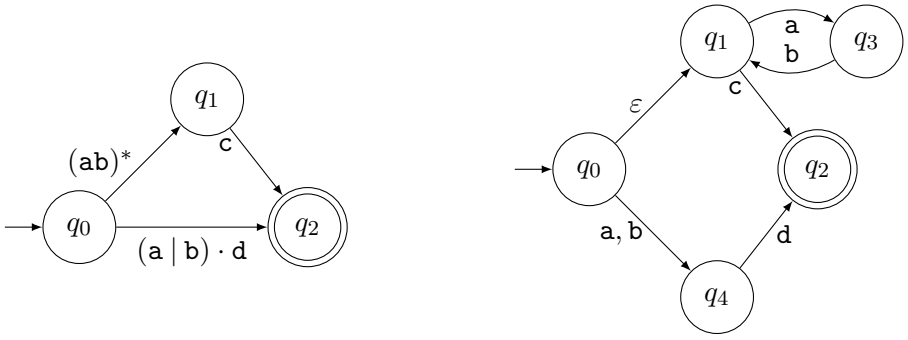
Der ENFA A **akzeptiert** ein Wort $w \in \Sigma^*$, wenn ein $n \geq 0$, eine Folge von Wörtern $w_1, \ldots, w_n \in \Sigma^*$ und eine Folge von Zuständen $q_1, \ldots, q_n \in Q$ existieren, so dass $q_n \in F$, $w = w_1 \cdots w_n$, und

$$w_i \in \mathcal{L}\left(\delta(q_i, q_{i+1})\right)$$

für alle $0 \leq i < n$. Die von A **akzeptierte Sprache** $\mathcal{L}(A)$ ist definiert als die Menge aller von A akzeptierten Wörter.

Wie bereits erwähnt, folgt aus Lemma 3.88 sofort, dass auch ENFAs nur reguläre Sprachen erzeugen.

Beispiel 3.90 Gegeben seien der ENFA A_1 (links) und der ε-NFA A_2 (rechts):



Es gilt $\mathcal{L}(A_1) = \mathcal{L}(A_2)$ (dies zu beweisen ist Ihnen überlassen). Der ε-NFA wurde übrigens nicht direkt anhand der Konstruktionen aus Lemma 3.88 erstellt, sondern zusätzlich ein wenig optimiert, um das Beispiel übersichtlicher zu machen. ◊

[32] Falls wir zwischen zwei Zuständen keine Kante haben wollen, können wir annehmen, dass die Kante mit $\emptyset$ beschriftet ist. Als Konvention nehmen wir an, dass wir δ auch als partielle Funktion gebrauchen können; nicht definierte Kanten werden dann als mit $\emptyset$ beschriftet interpretiert.

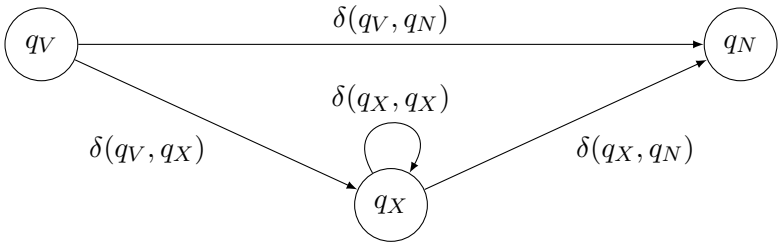
Der Beweis der Hin-Richtung von Satz 3.87 beruht auf dem folgenden Lemma:

Lemma 3.91 *Sei Σ ein Alphabet und sei A ein ENFA über Σ. Dann existiert ein regulärer Ausdruck α mit $\mathcal{L}(\alpha) = \mathcal{L}(A)$.*

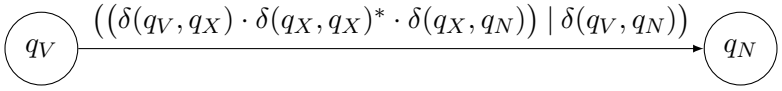
Um dieses Lemma zu beweisen brauchen wir wiederum etwas mehr Handwerkszeug (der Beweis wird dann direkt aus Lemma 3.92 folgen). Die Grundidee ist, den ENFA für L schrittweise in einen ENFA mit genau einem akzeptierenden Zustanden zu transformieren. Dazu verwenden wir ein Technik namens **Zustandselimination**. In jedem Schritt wählen wir einen Zustand q_X aus und bestimmen die Menge aller Vorgängerzustände von q_X, sowie die Menge aller Nachfolgerzustände von q_X. Für jedes Paar (q_V, q_N) aus einem Vorgängerzustand q_V und einem Nachfolgerzustand q_N ersetzen wir die Kante von q_V zu q_N durch eine neue Kante, die mit dem regulären Ausdruck

$$\big(\big(\delta(q_V, q_X) \cdot \delta(q_X, q_X)^* \cdot \delta(q_X, q_N)\big) \mid \delta(q_V, q_N)\big)$$

beschriftet ist. Danach entfernen wir q_X aus dem Automaten. Dies lässt sich leicht graphisch illustrieren. Angenommen, wir wollen einen Zustand q_X mit Vorgängerzustand q_V und Nachfolgerzustand q_N entfernen, dann ist der folgende Teilautomat im ENFA vorzufinden:



Durch Entfernen von q_X und Anpassen der Kante zwischen q_V und q_N erhalten wir den folgenden Teilautomaten:



Diesen Vorgang wiederholen wir, bis nur noch der Startzustand und ein akzeptierender Zustande übrig sind. Diese sind dann durch eine Kante mit dem gesuchten α verbunden, den wir nur noch ablesen müssen. Wir verwenden dazu den folgenden Algorithmus:

Algorithmus 3 (`ENFA2RegEx`) Berechnet aus einem ENFA einen regulären Ausdruck.

Eingabe: Ein ENFA $A := (\Sigma, Q, \delta, q_0, F)$.
Ausgabe: Ein regulärer Ausdruck α mit $\mathcal{L}(\alpha) = \mathcal{L}(A)$.

1. Stelle sicher, dass q_0 keine eingehenden Kanten hat (also dass $\delta(q, q_0) = \emptyset$ für alle $q \in Q$) und nicht akzeptierend ist.

2. Stelle sicher, dass es genau einen akzeptierenden Zustand $q_F \neq q_0$ gibt, und dass dieser keine ausgehenden Kanten hat (also dass $\delta(q_F, q) = \emptyset$ für alle $q \in Q$).
3. Falls $|Q| = 2$: Gib $\delta(q_0, q_F)$ aus.
4. Wähle einen Zustand $q_X \in Q - \{q_0, q_F\}$.
5. Definiere ENFA $A' := (\Sigma, Q', \delta', q_0, F)$ durch:
   a) $Q' := Q - \{q_X\}$,
   b) Für alle $q_V \in Q' - \{q_F\}$ und alle $q_N \in Q' - \{q_0\}$ ist
   $$\delta'(q_V, q_N) := \big(\big(\delta(q_V, q_X) \cdot \delta(q_X, q_X)^* \cdot \delta(q_X, q_N)\big) \mid \delta(q_V, q_N)\big)\,.$$
6. Rufe `ENFA2RegEx` rekursiv auf A' auf und gib das Resultat zurück.

Da jeder DFA, NFA, NNFA oder ε-NFA leicht als ENFA interpretiert werden kann (zur Not fügt man einen neuen Startzustand ein und zieht ein paar ε-Kanten), können auch diese mittels `ENFA2RegEx` in reguläre Ausdrücke konvertiert werden.

Beispiele für die Arbeitsweise von `ENFA2RegEx` finden Sie weiter unten (Beispiel 3.94 und Beispiel 3.95). Vorher beweisen wir noch, dass der Algorithmus auch wirklich das tut, was wir von ihm erwarten:

Lemma 3.92 *Für jeden ENFA A ist `ENFA2RegEx`(A) ein regulärer Ausdruck für die Sprache $\mathcal{L}(A)$.*

Beweis: Sei $A := (\Sigma, Q, \delta, q_0, F)$ ein ENFA. Ohne Beeinträchtigung der Allgemeinheit nehmen wir Folgendes an:

1. Der Startzustand q_0 hat keine eingehenden Kanten und ist nicht akzeptierend, und
2. es gibt genau einen akzeptierenden Zustand q_F, dieser Zustand hat keine ausgehenden Kanten und $q_0 \neq q_F$.

Wir können immer sicherstellen, dass A diese Form hat, indem wir notfalls einen neuen Startzustand einführen und diesen durch eine ε-Kante mit dem alten Startzustand verbinden, oder indem wir q_F neu einführen, alle akzeptierenden Zustände durch eine ε-Kante mit q_F verbinden, und q_F zum einzigen akzeptierenden Zustand machen. Durch die Definition von δ' ist sichergestellt, dass diese Form beim Aufruf von `ENFA2RegEx` erhalten bleibt.

Wir zeigen die Behauptung nun durch Induktion über die Anzahl der Zustände von A.

Induktionsanfang: Sei A ein ENFA mit genau zwei Zuständen. Nach unseren Anforderungen müssen dies die Zustände q_0 und q_F sein. Da q_0 keine eingehenden und q_F keine ausgehenden Kanten hat, ist $\mathcal{L}(A) = \mathcal{L}(\delta(q_0, q_F))$. Also gibt `ENFA2RegEx` mit $\delta(q_0, q_F)$ einen regulären Ausdruck für $\mathcal{L}(A)$ zurück. Dies gilt insbesondere auch, wenn $\delta(q_0, q_F) = \emptyset$.

INDUKTIONSSCHRITT: Die Behauptung gelte nun für ein $n \geq 2$ für alle ENFA mit n Zuständen (bei denen der Startzustand q_0 keine eingehenden Kanten hat, und genau ein akzeptierender Zustand q_F existiert, der keine ausgehenden Kanten hat). Angenommen, A hat $n+1$ Zustände. Sei A' der ENFA, der beim Aufruf von `ENFA2RegEx` auf A erzeugt wird. Dann hat A' genau n Zustände, und es gibt einen Zustand q_X, der aus A entfernt wurde um A' zu erzeugen. Wir zeigen nun, dass $\mathcal{L}(A) = \mathcal{L}(A')$ gilt.

„$\subseteq$“: Angenommen, $w \in \mathcal{L}(A)$. Dann existiert ein $m \geq 0$ und eine Folge $q_0, q_1, \ldots, q_m, q_F$ von Zuständen von A, die A beim Akzeptieren von w durchläuft. Wir unterscheiden nun zwei Fälle:

1. q_X kommt nicht in der Folge $q_0, q_1, \ldots, q_m, q_F$ vor, und
2. q_X kommt in der Folge $q_0, q_1, \ldots, q_m, q_F$ vor.

1. Fall: Angenommen, $q_X \neq q_i$ für $1 \leq i \leq m$. Dann wird w auch von A' akzeptiert, da für alle Zustandspaare p, q von A' der reguläre Ausdruck $\delta'(p, q)$ auch den Ausdruck $\delta(p, q)$ als Teil einer Vereinigung enthält. Also $w \in \mathcal{L}(A')$.

2. Fall: Angenommen, q_X kommt in dieser Folge vor. Der Einfachheit halber definieren wir $q_{m+1} := q_F$. Nun gilt: Für jede Stelle j mit $q_j = q_X$ existiert ein größtes i mit $0 \leq i < j$ und $q_i \neq q_X$, sowie ein kleinstes k mit $j < k \leq m+1$, so dass $q_k \neq q_X$. (Die Folgenglieder q_i und q_k) begrenzen also einen Block von q_X.) Nach Definition von δ' ist

$$\delta'(q_i, q_k) = \big(\big(\delta(q_i, q_X) \cdot \delta(q_X, q_X)^* \cdot \delta(q_X, q_k)\big) \mid \delta(q_i, q_k)\big)\,,$$

also führt jedes Wort, das in A' von q_i über q_X zu q_k führt auch in A von q_i zu q_k. Somit lässt sich jeder Weg, der in A von q_0 zu q_F führt, auch in A' nachvollziehen. Es gilt also $w \in \mathcal{L}(A')$.

„$\supseteq$“: Angenommen, $w \in \mathcal{L}(A')$. Für jedes Zustandspaar q_i, q_j von G' beschreibt $\delta'(q_i, q_j)$ die Wörter, die den ENFA G vom Zustand q_i zu q_j führen, und zwar entweder direkt, oder mit Umweg über q_X. Also ist auch $w \in \mathcal{L}(A)$.

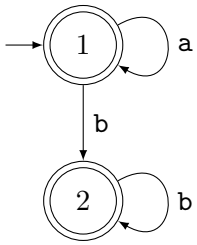
Somit gilt $\mathcal{L}(A') = \mathcal{L}(A)$. Nach Induktionsannahme ist $\alpha :=$ `ENFA2RegEx`(A') ein regulärer Ausdruck für $\mathcal{L}(A')$. Da $\mathcal{L}(A') = \mathcal{L}(A)$ gilt auch $\mathcal{L}(\alpha) = \mathcal{L}(A)$, und somit gibt `ENFA2RegEx` einen regulären Ausdruck für $\mathcal{L}(A)$ zurück. □

Aus Lemma 3.92 folgt sofort Lemma 3.91, und damit auch der fehlende Teil des Beweises von Satz 3.87.

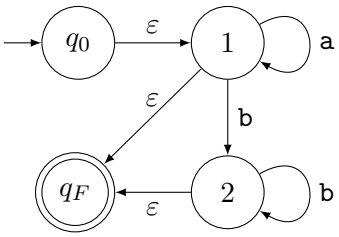
Hinweis 3.93 Beim Ausführen von `ENFA2RegEx` dürfen Sie die Rechenregeln zum Vereinfachen von regulären Ausdrücken verwenden (siehe Abschnitt A.3). Gerade im Zusammenhang mit $\emptyset$ und ε sollten Sie das auf jeden Fall tun.

Um die Arbeitsweise von `ENFA2RegEx` zu veranschaulichen, betrachten wir zwei Beispiele, ein einfaches (Beispiel 3.94) und ein etwas komplexeres (Beispiel 3.95).

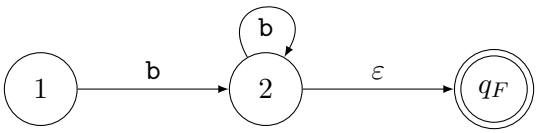
Beispiel 3.94 Wir betrachten den folgenden DFA A:



Wir interpretieren A nun als ENFA und stellen sicher, dass er die gewünschte Form hat. Dazu führen wir einen neuen Startzustand q_0 und einen neuen (einzigen) akzeptierenden Zustand q_F ein, die wir über ε-Kanten passend mit den bereits existierenden Zuständen verbinden. Wir halten den folgenden ENFA:



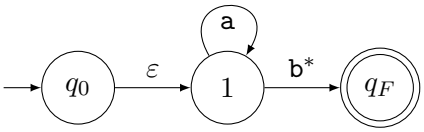
Der Algorithmus `ENFA2RegEx` kann sich nun entscheiden, welchen der beiden Zustände 1 und 2 er zuerst eliminieren möchte. Wir entscheiden uns für den Zustand 2. Da dieser genau einen Vorgängerzustand hat (nämlich 1) und genau einen Nachfolgerzustand (nämlich q_F) haben wir besonders wenig Arbeit. Als Nebenrechnung betrachten wir den folgenden Ausschnitt aus dem EFNA, der nur den zu eliminierenden Zustand (samt Schleife) sowie seine direkten Vorgänger und Nachfolger zeigt (zusammen mit den Kanten, die diese mit 2 verbinden, nicht aber mit eventuell existierenden Kanten zwischen diesen oder anderen Zuständen):



Da der Zustand 2 nur genau einen Vorgänger und genau einen Nachfolger hat, existiert nur ein Pfad von einem Vorgänger zu einem Nachfolger, der über 2 verläuft. Also müssen wir zur Berechnung von δ' nur einen regulären Ausdruck bilden, nämlich

$$\begin{aligned}\delta'(1, q_F) &= \delta(1, q_F) \mid \big(\delta(1,2) \cdot \delta(2,2)^* \delta(2, q_F)\big) \\ &= \varepsilon \mid (\mathtt{b} \cdot \mathtt{b}^* \varepsilon).\end{aligned}$$

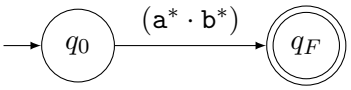
Da $\mathcal{L}(\mathtt{b} \cdot \mathtt{b}^* \varepsilon) = \mathcal{L}(\mathtt{b} \cdot \mathtt{b}^*) = \mathcal{L}(\mathtt{b}^+)$ und da $\mathcal{L}(\varepsilon \mid \mathtt{b}^+) = \mathcal{L}(\mathtt{b}^*)$ können wir zusammengefasst $\delta'(1, q_F) := \mathtt{b}^*$ festlegen. Wir erhalten so den folgenden ENFA A':



Beim Eliminieren des Zustands 1 haben wir nun ein leichtes Spiel: Zwischen q_0 und q_F gibt es keine Kante (bzw., die Kante ist mit $\emptyset$ beschriftet und kann eliminiert werden), der Algorithmus `ENFA2RegEx` berechnet also die folgende neue Kante:

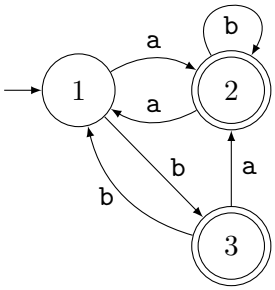
$$\begin{aligned}\delta^{(2)}(q_0, q_F) &= \delta(q_0, 1) \cdot \delta(1, 1)^* \cdot \delta(1, q_F) \\ &= \varepsilon \cdot \mathtt{a}^* \cdot \mathtt{b}^*.\end{aligned}$$

Diesen Ausdruck können wir zu $(\mathtt{a}^* \cdot \mathtt{b}^*)$ vereinfachen. Der berechnete ε-NFA $A^{(2)}$ sieht also folgendermaßen aus:

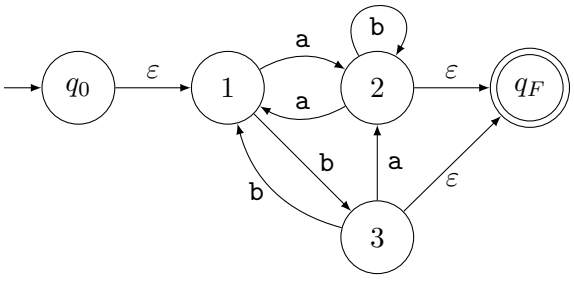


Da nur noch q_0 und q_F vorhanden sind, wird nun $(\mathtt{a}^* \cdot \mathtt{b}^*)$ als regulärer Ausdruck für die Sprache $\mathcal{L}(A)$ ausgegeben. ◊

Beispiel 3.95 Wir betrachten den folgenden DFA A:

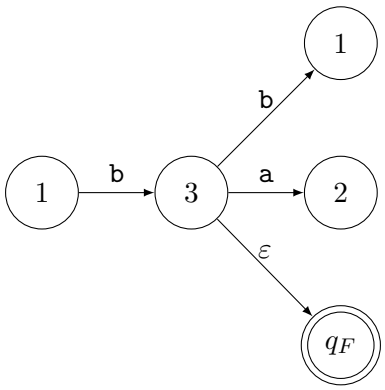


Um A in einen regulären Ausdruck umzuwandeln, fassen wir den DFA als ENFA auf. Beim Durchlauf von `ENFA2RegEx` wird nun sichergestellt, dass der Startzustand keine eingehenden Kanten hat, und dass genau ein akzeptierenden Zustand existiert, der keine ausgehenden Kanten hat. Der resultierende ENFA sieht wie folgt aus:



Denken Sie daran, dass die Beschriftungsfunktion δ des ENFA total ist; alle nicht eingezeichneten Kanten sind mit $\emptyset$ beschriftet. Wir wählen nun einen Zustand $q_X \in \{1, 2, 3\}$ als zu eliminierenden Zustand aus. Hierbei haben wir freie Wahl (wir dürfen nur nicht q_0 oder q_F wählen), und je nachdem, welchen Zustand wir wählen, kann der resultierende Ausdruck unterschiedlich lang ausfallen. In diesem Beispiel entscheiden wir uns für den Zustand 3.

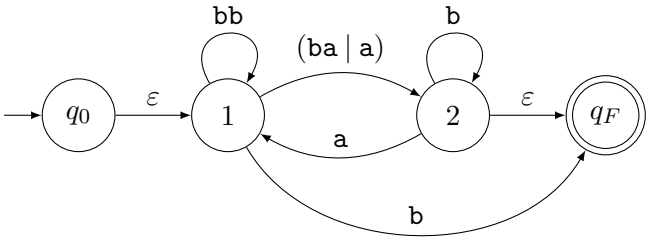
Um uns die Arbeit zu erleichtern, notieren wir uns eine Ausschnitt des ENFA, der nur den Zustand 3 sowie seine direkten Vorgänger und Nachfolger enthält (dabei lassen wir alle weiteren Kanten zwischen diesen Zuständen aus).



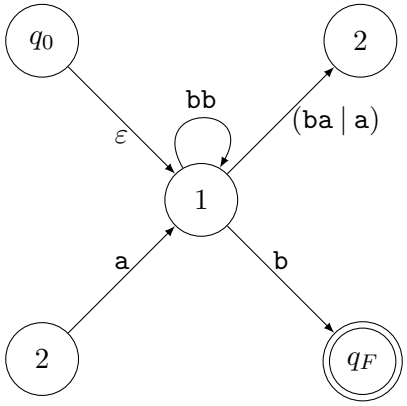
Zuerst stellen wir fest, dass $\delta(3,3) = \emptyset$, da 3 keine Kante zu sich selbst hat. Da der entsprechende Ausdruck $\delta(3,3)^*$ zu ε äquivalent ist, können wir diesen bei der Konstruktion der neuen Kanten einfach weglassen. Es gilt:

$$\begin{aligned}
\delta'(1,1) &= ((\delta(1,3) \cdot \delta(3,1)) \mid \delta(1,1)) \\
&= \mathtt{bb}, \\
\delta'(1,2) &= ((\delta(1,3) \cdot \delta(3,2)) \mid \delta(1,2)) \\
&= (\mathtt{ba} \mid \mathtt{a}), \\
\delta'(1,q_F) &= \delta(1,3) \cdot \delta(3,q_F) \\
&= \mathtt{b}.
\end{aligned}$$

Streng genommen müssten wir (zum Beispiel) $\delta'(1,q_F) = (\mathtt{b} \cdot \emptyset^* \cdot \varepsilon) | \emptyset$ schreiben, allerdings ist dieser regulärer Ausdruck äquivalent zu $\mathtt{b} \cdot \varepsilon \cdot \varepsilon$, und dieser wiederum zu $\mathtt{b}$. Diese einfachen Rechenregeln direkt anzuwenden vereinfacht die Anwendung des Algorithmus deutlich. Durch Einsetzen der neuen Kanten erhalten wir den folgenden ENFA A':



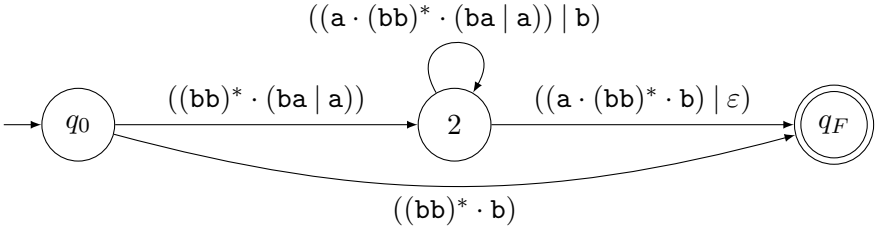
Der Algorithmus `ENFA2RegEx` ruft sich nun rekursiv auf. Nun ist es wieder an der Zeit, einen zu eliminierenden Zustand q_X zu bestimmen. In Frage kommen 1 und 2, wir wählen $q_X = 1$. Analog zum vorigen Fall geben wir einen Ausschnitt aus dem Automaten an. Da der Zustand 1 eine Schleife zu sich selbst hat, ist diese mit angegeben. Ansonsten führen wir in dieser Nebenrechnung nur die Kanten zwischen q_X, seinen Vorgängern und seinen Nachfolgern auf:



Dadurch ergeben sich die folgenden neuen Kanten:

$$\begin{aligned}
\delta^{(2)}(q_0, 2) &= ((\mathtt{bb})^* \cdot (\mathtt{ba} \mid \mathtt{a})),\\
\delta^{(2)}(q_0, q_F) &= ((\mathtt{bb})^* \cdot \mathtt{b}),\\
\delta^{(2)}(2, 2) &= ((\mathtt{a} \cdot (\mathtt{bb})^* \cdot (\mathtt{ba} \mid \mathtt{a})) \mid \mathtt{b}),\\
\delta^{(2)}(2, q_F) &= ((\mathtt{a} \cdot (\mathtt{bb})^* \cdot \mathtt{b}) \mid \varepsilon).
\end{aligned}$$

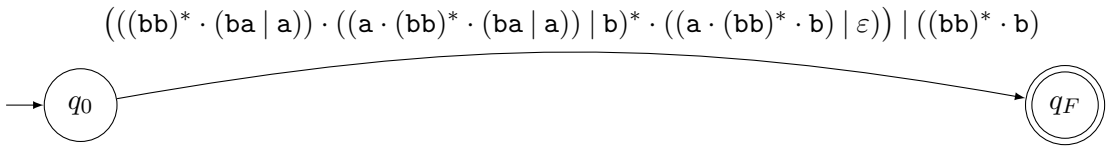
Im Fall von $\delta^{(2)}(q_0, 2)$ und $\delta^{(2)}(q_0, q_F)$ gibt es in A' keine entsprechende Kante, also musste diese hier nicht beachtet werden. Im Gegensatz dazu muss $\delta'(2, q_F)$ sehr wohl in $\delta^{(2)}(2, q_F)$ aufgenommen werden. Der entsprechende ENFA $A^{(2)}$ sieht wie folgt aus:



Nun kann nur noch ein Zustand eliminiert werden, nämlich der Zustand 2. In diesem Fall sparen wir uns eine Nebenrechnung, wir können der regulären Ausdruck fast direkt ablesen. Für diejenigen Pfade von q_0 zu q_F, die über q_F laufen, erhalten wir den folgenden Ausdruck:

$$\underbrace{((\mathtt{bb})^* \cdot (\mathtt{ba} \mid \mathtt{a}))}_{\delta^{(2)}(q_0,2)} \cdot \underbrace{((\mathtt{a} \cdot (\mathtt{bb})^* \cdot (\mathtt{ba} \mid \mathtt{a})) \mid \mathtt{b})^*}_{(\delta^{(2)}(2,2))^*} \cdot \underbrace{((\mathtt{a} \cdot (\mathtt{bb})^* \cdot \mathtt{b}) \mid \varepsilon)}_{\delta^{(2)}(2,q_F)}$$

Zusammen mit $\delta^{(2)}(2, q_F)$ ergibt sich der folgende ENFA $A^{(3)}$:



Eine mögliche Ausgabe des Algorithmus `ENFA2RegEx` mit dem DFA A als Eingabe ist also der folgende reguläre Ausdruck:

$$\big(((\mathtt{bb})^* \cdot (\mathtt{ba} \mid \mathtt{a})) \cdot ((\mathtt{a} \cdot (\mathtt{bb})^* \cdot (\mathtt{ba} \mid \mathtt{a})) \mid \mathtt{b})^* \cdot ((\mathtt{a} \cdot (\mathtt{bb})^* \cdot \mathtt{b}) \mid \varepsilon)\big) \mid ((\mathtt{bb})^* \cdot \mathtt{b}).$$

Abhängig von der Reihenfolge der eliminierten Zustände können auch andere Ausdrücke entstehen, die aber alle die gleiche Sprache beschreiben, nämlich $\mathcal{L}(A)$. ◊

3.3.2 Gezieltes Ersetzen: Substitution und Homomorphismus

Anhand von ENFAs und regulären Ausdrücken können wir nun eine weitere Abschlusseigenschaft betrachten: Die Substitution. Seien Σ_1 und Σ_2 Alphabete[33]. Eine **Substitution (von Σ_1 nach Σ_2)** ist eine Funktion $s\colon \Sigma_1 \to \mathcal{P}(\Sigma_2^*)$ (jedem Buchstaben $a \in \Sigma_1$ wird also eine Sprache $s(a) \subseteq \Sigma_2^*$ zugeordnet). Wir erweitern s von Buchstaben zu Wörtern aus Σ_1^*, indem wir die folgende rekursive Definition verwenden:

$$\begin{aligned} s(\varepsilon) &:= \varepsilon, \\ s(w \cdot a) &:= s(w) \cdot s(a) \end{aligned}$$

für alle $w \in \Sigma_1^*$ und alle $a \in \Sigma_1$. Wie üblich erweitern wir s auf Sprachen, durch

$$s(L) := \bigcup_{w \in L} s(w)$$

für alle $L \subseteq \Sigma_1^*$. Eine Substitution s von Σ_1 nach Σ_2 ist eine **reguläre Substitution**, wenn für jedes $a \in \Sigma_1$ die Sprache $s(a)$ regulär ist.

Beispiel 3.96 Wir betrachten drei Beispiele zu regulären Substitutionen:

1. Sei $L_1 := \{\mathtt{aba}\}$. Wir definieren die Substitution s_1 durch

   $$\begin{aligned} s_1(\mathtt{a}) &:= \{\mathtt{b}\}^*, \\ s_1(\mathtt{b}) &:= \{\mathtt{c}\}. \end{aligned}$$

   Da $s_1(\mathtt{a})$ und $s_1(\mathtt{b})$ reguläre Sprachen sind, ist s_1 eine reguläre Substitution. Es gilt

   $$s_1(L_1) = \mathcal{L}(\mathtt{b}^*\mathtt{c}\mathtt{b}^*)$$

[33] Die beiden Alphabete müssen nicht unterschiedlich sein.

$$= \{\mathtt{b}^i\mathtt{cb}^j \mid i, j \in \mathbb{N}\}.$$

Dies kann man leicht zeigen, indem man $\mathtt{aba}$ als regulären Ausdruck auffasst, und jeden der drei Buchstaben durch einen regulären Ausdruck für sein Bild unter s_1 ersetzt.

2. Sei $L_2 := \{\mathtt{ab}\}^*$. Wir definieren die Substitution s_2 durch

$$s_2(\mathtt{a}) := \{\mathtt{a}\}^*,$$
$$s_2(\mathtt{b}) := \{\mathtt{b}, \mathtt{bb}\}.$$

Da $s_2(\mathtt{a})$ und $s_2(\mathtt{b})$ reguläre Sprachen sind, ist s_2 eine reguläre Substitution, und es gilt $s_2(L_2) = s_2(\{\mathtt{ab}\}^*) = \{\{\mathtt{a}\}^* \cdot \{\mathtt{b}, \mathtt{bb}\}\}^*$. Wir können auch direkt aus L_2 und s_2 einen regulären Ausdruck α_2 konstruieren, und zwar

$$\alpha_2 := (\mathtt{a}^* \cdot (\mathtt{b} \mid \mathtt{bb}))^*.$$

Dann gilt $\mathcal{L}(\alpha_2) = s_2(L_2)$.

3. Sei $L_3 := \{\mathtt{c}^i\mathtt{d}^i \mid i \in \mathbb{N}\}$. Wir definieren die Substitution s_3 durch

$$s_3(\mathtt{c}) := \{\mathtt{a}\},$$
$$s_3(\mathtt{d}) := \{\mathtt{b}, \mathtt{bb}\}.$$

Auch hier ist s_3 eine reguläre Substitution; dass L_3 keine reguläre Sprache ist spielt hier keine Rolle. Es gilt

$$s_3(L_3) = \{\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N}, i \leq j\}.$$

Da L_3 nicht regulär ist, können wir hier $s_3(L_3)$ nicht anhand von regulären Ausdrücken bestimmen.

$\Diamond$

Die Beispiele lassen bereits vermuten, dass wir reguläre Ausdrücke benutzen können, um die Bilder von regulären Sprachen unter regulären Substitutionen zu bestimmen. Dieser Ansatz funktioniert in der Tat immer, wie wir anhand des folgenden Lemmas festhalten:

Lemma 3.97 *Die Klasse der regulären Sprachen ist abgeschlossen unter regulärer Substitution.*

Beweis: Seien Σ_1 und Σ_2 Alphabete, sei $L \in \mathrm{REG}_{\Sigma_1}$ und sei s eine reguläre Substitution von Σ_1 und Σ_2. Da L regulär ist, existiert ein regulärer Ausdruck α mit $\mathcal{L}(\alpha) = L$. Da s eine reguläre Substitution ist, existiert für jedes $a \in \Sigma_1$ ein regulärer Ausdruck β_a (über Σ_2) mit $\mathcal{L}(\beta) = s(a)$. Wir ersetzen nun in α jeden vorkommenden Buchstaben durch den regulären Ausdruck für sein Bild unter s. Formaler definieren wir rekursiv eine Funktion f auf regulären Ausdrücken, durch $f(\varepsilon) = \varepsilon$, $f(\emptyset) = \emptyset$, $f(a) = \beta_a$ für alle $a \in \Sigma_1$, sowie

$$f(\gamma_1 \mid \gamma_2) = f(\gamma_1) \mid f(\gamma_2),$$

$$f(\gamma_1 \cdot \gamma_2) = f(\gamma_1) \cdot f(\gamma_2),$$
$$f(\gamma_1^+) = f(\gamma_1)^+$$

für alle regulären Ausdrücke γ_1, γ_2 über Σ_1. Durch eine Induktion über den Aufbau ist schnell zu sehen, dass $\mathcal{L}(f(\alpha)) = s(L)$. □

Ein Spezialfall von Substitutionen ist der Homomorphismus. Ein **Homomorphismus (von Σ_1 nach Σ_2)** ist eine Funktion $h\colon \Sigma_1^* \to \Sigma_2^*$ mit $h(u \cdot v) = h(u) \cdot h(v)$ für alle $u, v \in \Sigma_1^*$. Wir definieren diesen durch eine Funktion $h\colon \Sigma_1 \to \Sigma_2^*$, diese erweitern wir zu einer Funktion auf Wörtern durch $h(\varepsilon) = \varepsilon$ und $h(w \cdot a) = h(w) \cdot h(a)$ für alle $w \in \Sigma_1^*$ und alle $a \in \Sigma$, sowie zu einer Funktion auf Sprachen durch

$$h(L) := \{h(w) \mid w \in L\}.$$

Da jeder Homomorphismus auch als Substitution aufgefasst werden kann, folgt aus Lemma 3.97 sofort,dass die Klasse der regulären Sprachen auch unter Homomorphismen abgeschlossen ist.

Beispiel 3.98 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\Sigma' := \{\mathtt{0}, \mathtt{1}\}$. Sei $L := \{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \in \mathbb{N}\}$. Der Homomorphismus $h\colon \Sigma^* \to (\Sigma')^*$ sei definiert durch

$$h(\mathtt{a}) = \mathtt{1}, \qquad h(\mathtt{b}) = \mathtt{10}, \qquad h(\mathtt{c}) = \mathtt{100}.$$

Dann ist

$$h(L) = \{\mathtt{1}^n(\mathtt{10})^n(\mathtt{100})^n \mid n \in \mathbb{N}\}.$$ ◊

Zu jedem Homomorphishmus h können wir auch einen **inversen Homomorphismus** oder **umgekehrten Homomorphismus** h^{-1} definieren. Ist $h\colon \Sigma_1^* \to \Sigma_2^*$, so ist seine Umkehrung $h^{-1}\colon \Sigma_2^* \to \mathcal{P}(\Sigma_1^*)$ definiert durch

$$h^{-1}(w) := \{v \in \Sigma_1^* \mid h(v) = w\}.$$

Diese Definition lässt sich auch auf Sprachen erweitern, und zwar wie gewohnt durch

$$h^{-1}(L) := \bigcup_{w \in L} h^{-1}(w).$$

Wir betrachten dazu drei Beispiele:

Beispiel 3.99 Sei $\Sigma_1 := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\Sigma_2 := \{\mathtt{0}, \mathtt{1}\}$. Sei $L := \{\mathtt{011}, \mathtt{101}\}$. Der Homomorphismus $h : (\Sigma_1)^* \to (\Sigma_2)^*$ sei definiert durch

$$h(\mathtt{a}) := \mathtt{0}, \qquad h(\mathtt{b}) := \mathtt{1}, \qquad h(\mathtt{c}) := \mathtt{01}.$$

Dann gilt:

$$\begin{aligned} h^{-1}(L) &= h^{-1}(\{\mathtt{011}, \mathtt{101}\}) \\ &= h^{-1}(\mathtt{011}) \cup h^{-1}(\mathtt{101}) \\ &= \{\mathtt{abb}, \mathtt{cb}\} \cup \{\mathtt{bab}, \mathtt{bc}\} \\ &= \{\mathtt{abb}, \mathtt{cb}, \mathtt{bab}, \mathtt{bc}\}. \end{aligned}$$ ◊

Beispiel 3.100 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{(\mathtt{ab})^n(\mathtt{ba})^n \mid n \in \mathbb{N}\}$. Der Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ sei definiert durch

$$h(\mathtt{a}) := \mathtt{ab}, \qquad h(\mathtt{b}) := \mathtt{ba}.$$

Dann ist

$$h^{-1}(L) = \{\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}\}. \qquad \Diamond$$

Beispiel 3.101 Sei Σ ein beliebiges Alphabet. Der Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ sei definiert durch $h(a) := \varepsilon$ für alle $a \in \Sigma$. Dann ist $h^{-1}(\varepsilon) = \Sigma^*$. $\Diamond$

Hinweis 3.102 Wir definieren Homomorphismen immer über einzelne Buchstaben, also zum Beispiel

$$h(\mathtt{a}) := \mathtt{bb}, \qquad h(\mathtt{b}) := \mathtt{ab}, \qquad h(\mathtt{c}) := \mathtt{cbc}.$$

Etwas salopp ausgedrückt heißt das: Zwischen den Klammern darf *immer* nur ein einzelner Buchstabe stehen. Definition wie $h(\mathtt{aa}) := \mathtt{b}$ sind nicht erlaubt, sondern falsch. Wenn Sie eine Abbildung benötigen, die $\mathtt{aa}$ auf $\mathtt{a}$ abbildet, müssen Sie einen inversen Homomorphismus verwenden. Sie definieren dann zuerst den Homomorphismus, in diesem Beispiel durch $h(\mathtt{a}) := \mathtt{aa}$, und betrachten dann sein Inverses (siehe Beispiel 3.103).

Beispiel 3.103 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $L := \{\mathtt{a}^{2i}\mathtt{b}^i \mid i \in \mathbb{N}\}$. Der Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ sei definiert durch $h(\mathtt{a}) := \mathtt{aa}$ und $h(\mathtt{b}) := \mathtt{b}$. Dann gilt: $h^{-1}(L) := \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. $\Diamond$

Anhand einer einfachen Automatenkonstruktion lässt sich auch die entsprechende Abschlusseigenschaft von REG beweisen:

Lemma 3.104 *Die Klasse der regulären Sprachen ist abgeschlossen unter inversen Homomorphismen.*

Beweis: Seien Σ_1 und Σ_2 Alphabete. Sei $A := (\Sigma_1, Q, \delta, q_0, F)$ ein DFA und $h\colon \Sigma_2^* \to \Sigma_1^*$ ein Homomorphismus. Wir konstruieren nun einen DFA A_I mit $\mathcal{L}(A_I) = h^{-1}(\mathcal{L}(A))$, und zwar durch $A_I := (\Sigma_2, Q, \delta_I, q_0, F)$, wobei

$$\delta_I(q, a) := \delta(q, h(a))$$

für alle $q \in Q$ und alle $a \in \Sigma_2$. Nun lässt sich anhand einer einfachen Induktion leicht zeigen, dass $\delta_I(q_0, w) = \delta(q_0, h(w))$ für alle $w \in \Sigma_2^*$ gilt: Für $w = \varepsilon$ ist $\delta_I(q_0, w) = q_0 = \delta(q_0, h(\varepsilon))$. Angenommen, $\delta_I(q_0, w) = \delta(q_0, h(w))$ gilt für ein $w \in \Sigma_2^*$. Sei $a \in \Sigma_2$. Daraus folgt:

$$\begin{aligned}\delta_I(q_0, wa) &= \delta_I(\delta_I(q_0, w), a)\\ &= \delta(\delta(q_0, h(w)), h(a))\\ &= \delta(q_0, h(w)h(a))\end{aligned}$$

$$= \delta(q_0, h(wa)).$$

Also ist $w \in \mathcal{L}(A_I)$ genau dann, wenn $h(w) \in \mathcal{L}(A)$. Somit ist $\mathcal{L}(A_I) = h^{-1}(\mathcal{L}(A))$. Also ist $h^{-1}(\mathcal{L}(A))$ regulär; und wir können daraus schließen, dass die Klasse der regulären Sprachen unter inversen Homomorphismen abgeschlossen ist. □

Durch den Gebrauch von Homomorphismen und inversen Homomorphismen lassen sich viele Beweise deutlich vereinfachen. Dies gilt besonders für Beweise, dass eine Sprache nicht regulär ist. Wir betrachten dazu ein paar Beispiele:

Beispiel 3.105 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $L := \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \in \mathbb{N}\}$. Um zu zeigen, dass L nicht regulär ist, nehmen wir das Gegenteil an (also $L \in \mathrm{REG}$). Wir definieren einen Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ durch

$$h(\mathtt{a}) = \mathtt{a}, \qquad h(\mathtt{b}) = \mathtt{b}, \qquad h(\mathtt{c}) = \varepsilon.$$

Nun gilt

$$\begin{aligned} h(L) &= h(\{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \in \mathbb{N}\}) \\ &= \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}. \end{aligned}$$

Da die Klasse der regulären Sprachen unter Homomorphismus abgeschlossen ist, ist $h(L)$ regulär. Widerspruch, denn wir wissen, dass diese Sprache nicht regulär ist. Also kann auch L nicht regulär sein. ◊

Beispiel 3.106 Sei $\Sigma := \{\mathtt{a}\}$ und

$$L := \left\{\mathtt{a}^{2i^2} \mid i \in \mathbb{N}\right\}.$$

Wir wollen zeigen, dass L nicht regulär ist. Angenommen, L ist regulär. Wir definieren nun den Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ durch $h(\mathtt{a}) := \mathtt{aa}$. Dann ist

$$\begin{aligned} h^{-1}(L) &= h^{-1}\left(\left\{\mathtt{a}^{2i^2} \mid i \in \mathbb{N}\right\}\right) \\ &= h^{-1}\left(\left\{(\mathtt{aa})^{i^2} \mid i \in \mathbb{N}\right\}\right) \\ &= \left\{\mathtt{a}^{i^2} \mid i \in \mathbb{N}\right\}. \end{aligned}$$

Da reguläre Sprachen unter inversem Homomorphismus abgeschlossen sind, ist $h^{-1}(L)$ regulär. Wir wissen aber bereits, dass diese Sprache nicht regulär ist. Widerspruch. Also kann auch L nicht regulär sein. ◊

Beispiel 3.107 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{(\mathtt{ab})^i(\mathtt{ba})^i \mid i \in \mathbb{N}\}$. Wir wollen zeigen, dass L nicht regulär ist. Angenommen, L ist regulär. Wir definieren einen Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ durch

$$h(\mathtt{a}) = \mathtt{ab}, \qquad h(\mathtt{b}) = \mathtt{ba}.$$

Dann ist

$$\begin{aligned} h^{-1}(L) &= h^{-1}(\{(\mathtt{ab})^i(\mathtt{ba})^i \mid i \in \mathbb{N}\}) \\ &= \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}. \end{aligned}$$

Da reguläre Sprachen unter inversem Homomorphismus abgeschlossen sind, ist $h^{-1}(L)$ regulär. Wir wissen aber bereits, dass diese Sprache nicht regulär ist. Widerspruch. Also kann auch L nicht regulär sein. ◊

Beispiel 3.108 In Beispiel 3.13 haben wir die Sprache

$$L := \left\{b^i a^{j^2} \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N}\right\} \cup \left\{a^k \mid k \in \mathbb{N}\right\}$$

kennengelernt und festgestellt, dass diese Sprache die Eigenschaften des Pumping-Lemmas erfüllt. Daher kann das Pumping-Lemma nicht verwendet werden, um zu zeigen, dass L nicht regulär ist. In Beispiel 3.24 haben wir bereits gesehen, dass sich durch Abschlusseigenschaften ein entsprechender Beweis führen lässt, allerdings kamen wir dort schnell an unsere Grenzen. Jetzt haben wir das nötige Handwerkszeug, um diesen Beweis komplett elegant zu führen.

Angenommen, L ist regulär. Wir definieren

$$\begin{aligned} L' &:= L \cap (\{\mathtt{b}\} \cdot \{\mathtt{a}\}^*) \\ &= \left\{ba^{j^2} \mid j \in \mathbb{N}\right\}. \end{aligned}$$

Die Sprache $\{\mathtt{b}\} \cdot \{\mathtt{a}\}^*$ ist regulär, da sie durch den regulären Ausdruck $(\mathtt{b} \cdot \mathtt{a}^*)$ definiert wird. Da reguläre Sprachen abgeschlossen sind unter Schnitt, ist auch L' regulär. Wir definieren nun den Homomorphismus $h\colon \Sigma^* \to \{\mathtt{a}\}^*$ durch

$$h(\mathtt{a}) = \mathtt{a}, \qquad\qquad h(\mathtt{b}) := \varepsilon.$$

Dann ist

$$\begin{aligned} h(L') &= h\left(\left\{ba^{j^2} \mid j \in \mathbb{N}\right\}\right) \\ &= \left\{a^{j^2} \mid j \in \mathbb{N}\right\}. \end{aligned}$$

Diese Sprache ist bekanntermaßen nicht regulär, also kann L nicht regulär sein. ◊

Diese Beispiel illustrieren hoffentlich, wie ungemein praktisch Homomorphismen und ihre Umkehrungen für sich alleine sein können. Besonders hilfreich sind sie allerdings, wenn man beide Operationen zusammen mit dem Schnitt mit regulären Sprachen benutzt. Dadurch lassen sich gezielt Teile von Wörtern in den zu untersuchenden Sprachen markieren und ersetzen. Wir betrachten dazu ein paar Beispiele:

Beispiel 3.109 Sei $\Sigma := \{\mathtt{a}\}$ und $L := \left\{\mathtt{a}^{i^2+2} \mid i \in \mathbb{N}\right\}$. Wir wollen zeigen, dass diese Sprache nicht regulär ist, und nehmen dazu das Gegenteil an (also $L \in \mathrm{REG}$). Wie

Sie vielleicht ahnen wollen wir einen Widerspruch erhalten, indem wir durch geschickte Anwendung von Abschlusseigenschaften aus der Regularität von L auf die Regularität von $\{a^{i^2} \mid i \in \mathbb{N}\}$ schließen. Offensichtlich gilt

$$\begin{aligned} L &= \left\{ \mathtt{a}^{i^2+2} \mid i \in \mathbb{N} \right\} \\ &= \left\{ \mathtt{a}^{i^2}\mathtt{aa} \mid i \in \mathbb{N} \right\}. \end{aligned}$$

Unser Ziel ist es, die zwei überschüssigen $\mathtt{a}$ am Ende jedes Wortes von L zu entfernen. Dabei können wir allerdings nicht einfach einen Homomorphismus anwenden (dieser würde alle Wörter löschen). Der Trick ist, dass wir einen inversen Homomorphismus und den Schnitt mit einer regulären Sprache benutzen, um die zwei letzten Buchstaben jedes Wortes aus L zu markieren. Dann können wir sie gefahrlos löschen. Dazu definieren zuerst ein Alphabet $\hat{\Sigma} := \{\hat{\mathtt{a}}\}$ und einen Homomorphismus $h : (\Sigma \cup \hat{\Sigma})^* \to \Sigma^*$ durch

$$h(\mathtt{a}) := \mathtt{a}, \qquad\qquad h(\hat{\mathtt{a}}) := \mathtt{a}.$$

Wir betrachten nun die Sprache

$$L' := h^{-1}(L) \cap \mathcal{L}(\mathtt{a}^* \cdot \hat{\mathtt{a}}\hat{\mathtt{a}}).$$

Rein intuitiv gesehen markiert die Anwendung von h^{-1} die Buchstaben der Wörter von L mit dem $\hat{}$ Symbol. Dabei haben wir erst einmal keine Kontrolle darüber, welche Buchstaben markiert werden (zwischen „keiner“ und „alle“ sind alle Kombinationen vertreten). Durch den Schnitt mit der regulären Sprache $\mathcal{L}(\mathtt{a}^* \cdot \hat{\mathtt{a}}\hat{\mathtt{a}})$ bleiben genau die Wörter übrig, bei denen genau die letzten zwei Buchstaben markiert sind. Es gilt also:

$$\begin{aligned} L' &= h^{-1}(L) \cap \mathcal{L}(\mathtt{a}^* \cdot \hat{\mathtt{a}}\hat{\mathtt{a}}) \\ &= \left\{ \mathtt{a}^{i^2}\hat{\mathtt{a}}\hat{\mathtt{a}} \mid i \in \mathbb{N} \right\}. \end{aligned}$$

Jetzt müssen wir nur noch die beiden markierten Buchstaben entfernen. Dazu definieren wir einen Homomorphismus $g : (\Sigma \cup \hat{\Sigma})^* \to \Sigma^*$ durch

$$g(\mathtt{a}) := \mathtt{a}, \qquad\qquad g(\hat{\mathtt{a}}) := \varepsilon.$$

Dann gilt

$$\begin{aligned} g(L') &= g\left(\left\{ \mathtt{a}^{i^2}\hat{\mathtt{a}}\hat{\mathtt{a}} \mid i \in \mathbb{N} \right\} \right) \\ &= \left\{ \mathtt{a}^{i^2} \mid i \in \mathbb{N} \right\}. \end{aligned}$$

Die Sprache L' ist regulär, da L nach unserer Annahme regulär ist, und die Klasse der regulären Sprachen abgeschlossen ist unter inversem Homomorphismus und Schnitt. Da sie auch unter Homomorphismus abgeschlossen ist, ist auch $g(L')$ regulär. Widerspruch, denn wir wissen, dass diese Sprache nicht regulär ist. Also kann L nicht regulär sein. $\Diamond$

Beispiel 3.110 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $L := \left\{\mathtt{a}^i\mathtt{b}^j\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N}\right\}$. Um zu zeigen, dass L nicht regulär ist, nehmen wir das Gegenteil an. Sei nun $\hat{\Sigma} := \{\hat{\mathtt{a}}, \hat{\mathtt{b}}\}$. Der Homomorphismus $h : (\Sigma \cup \hat{\Sigma})^* \to \Sigma^*$ sei definiert durch

$$\begin{aligned} h(\mathtt{a}) &:= \mathtt{a}, & h(\hat{\mathtt{a}}) &:= \mathtt{a} \\ h(\mathtt{b}) &:= \mathtt{b}, & h\left(\hat{\mathtt{b}}\right) &:= \mathtt{b}. \end{aligned}$$

Wir definieren

$$\begin{aligned} L' &:= h^{-1}(L) \cap \mathcal{L}(\mathtt{a}^*\mathtt{b}^+\hat{\mathtt{a}}^*\hat{\mathtt{b}}^+) \\ &= \left\{\mathtt{a}^i\mathtt{b}^j\hat{\mathtt{a}}^i\hat{\mathtt{b}}^j \mid i \in \mathbb{N}, j \in \mathbb{N}_{>0}\right\}. \end{aligned}$$

Wir markieren also den zweiten Block von $\mathtt{a}$ und den zweiten Block von $\mathtt{b}$ (letzteres ist überflüssig, aber auch nicht schädlich. Warum können wir in dem regulären Ausdruck für die Sprache, mit der wir schneiden, nicht $\mathtt{b}^*$ statt $\mathtt{b}^+$ verwenden?) Jetzt müssen wir nur noch alle Vorkommen von $\mathtt{b}$ und $\hat{\mathtt{b}}$ löschen sowie $\hat{\mathtt{a}}$ passend umbenennen. Dazu definieren wir einen Homomorphismus $g : (\Sigma \cup \hat{\Sigma})^* \to \Sigma^*$ durch

$$\begin{aligned} g(\mathtt{a}) &:= \mathtt{a}, & h(\hat{\mathtt{a}}) &:= \mathtt{b} \\ g(\mathtt{b}) &:= \varepsilon, & h\left(\hat{\mathtt{b}}\right) &:= \varepsilon. \end{aligned}$$

Dann gilt

$$\begin{aligned} g(L') &= g\left(\left\{\mathtt{a}^i\mathtt{b}^j\hat{\mathtt{a}}^i\hat{\mathtt{b}}^j \mid i \in \mathbb{N}, j \in \mathbb{N}_{>0}\right\}\right) \\ &= \left\{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\right\}. \end{aligned}$$

Dass $g(L')$ nicht regulär ist, wissen wir bereits. Allerdings folgt aus der Annahme, dass L regulär ist, dass $g(L')$ regulär ist, denn die Klasse der Regulären Sprachen ist abgeschlossen unter inversem Homomorphismus, Schnitt und Homomorphismus. Widerspruch, also ist L nicht regulär. ◊

Hinweis 3.111 Beispiel 3.109 und Beispiel 3.110 arbeiten nach dem selben Grundprinzip:

1. Zuerst werden mit einem inversem Homomorphismus und dem Schnitt mit einer geeigneten regulären Sprache bestimmte Teile aller Wörter der Sprache markiert.
2. Mit einem Homomorphismus werden dann die markierten Stellen ersetzt oder gelöscht.

Dieses Grundprinzip lässt sich in vielen Fällen anwenden und kann natürlich auch mit anderen Abschlusseigenschaften kombiniert werden (achten Sie aber beim Gebrauch von Abschlusseigenschaften immer auf Hinweis 3.25).

Wir können diese Techniken auch verwenden, um eine weiter Abschlusseigenschaft zu beweisen:

Lemma 3.112 *Die Klasse der regulären Sprachen ist abgeschlossen unter Rechts-Quotient. Das heißt: Sind $L, R \in \mathrm{REG}$, dann ist die Sprache (L/R) regulär.*

Beweis: Übung (Aufgabe 3.20). □

3.4 Entscheidungsprobleme

Wir haben im Lauf der Vorlesungen und der Übungen verschiedene Beispiele kennengelernt, in denen formale Sprachen verwendet werden können, um Sachverhalte zu modellieren. Beispielsweise könnten Sie eine regulären Ausdruck benutzen, um in Ihrem Mailprogramm bestimmte Emails (oder Gruppen von Emails) als besonders wichtig oder besonders unwichtig zu markieren. Für jede eingehende Mail muss dann das Programm entscheiden, ob die Adresse des Absenders zu der von dem regulären Ausdruck beschriebenen Sprache passt. Formal gesehen muss also das Wortproblem dieser Sprache entschieden werden.

Statt das Wortproblem nur für eine feste Sprache zu definieren, können wir es auch gleich für eine ganze Klasse von Sprache definieren, zum Beispiel die Klasse der regulären Sprachen. Allerdings haben wir verschiedene Modelle zur Definition von regulären Sprachen kennen gelernt, die unterschiedliche Eigenschaften haben. Daher bietet es sich an, für das Wortproblem für jedes dieser Modelle zu betrachten, wie zum Beispiel für DFAs:

Wortproblem für DFAs

Eingabe: Ein DFA A über einem Alphabet Σ und ein Wort $w \in \Sigma^*$.
Frage: Ist $w \in \mathcal{L}(A)$?

Analog dazu können wir das **Wortproblem für NFAs** (und andere Klassen von endlichen Automaten) sowie das **Wortproblem für reguläre Ausdrücke** definieren. Auch diese beiden Probleme sind vergleichsweise leicht zu lösen. Zuerst stellen wir fest, dass sich das Wortproblem für reguläre Ausdrücke leicht auf das Wortproblem für ε-NFAs reduzieren lässt. Anhand der Konstruktionsvorschriften, die wir in Abschnitt 3.3.1 bereits kennengelernt haben (siehe auch Abschnitt A.2 im Anhang), können wir jeden regulären Ausdruck direkt in einen ε-NFA für die gleiche Sprache umwandeln. Das Wortproblem für die verschiedenen Varianten von NFAs wiederum können wir fast genauso wie beim DFA lösen; der einzige Unterschied ist, dass wir mehrere Zustände gleichzeitig speichern müssen. Ein anderer Ansatz ist, den NFA in einen DFA umzuwandeln. Wenn Effizienz keine Rolle spielt oder der Automat klein genug ist, so dass wir den exponentiellen Größenzuwachs verkraften können, kann dies direkt anhand der Potenzmengenkonstruktion geschehen. In vielen Anwendungsfällen wird der DFA aber nur „just in time“ konstruiert. Statt den ganzen DFA zu berechnen wird also jeweils nur der nächste Zustand anhand der Potenzmengenkonstruktion berechnet.

Zwei weitere Probleme, die im Umgang mit Sprachen immer wieder von Nutzen sind, sind das **Leerheitsproblem** und das **Universalitätsproblem**:

Leerheitsproblem für DFAs

Eingabe: Ein DFA A.
Frage: Ist $\mathcal{L}(A) = \emptyset$?

Universalitätsproblem für DFAs

Eingabe: Ein DFA A über einem Alphabet Σ.
Frage: Ist $\mathcal{L}(A) = \Sigma^*$?

Auch diese Probleme können analog für die nichtdeterministischen endlichen Automaten sowie für reguläre Ausdrücke definiert werden. In all diesen Fällen ist das Leerheitsproblem leicht zu lösen: Für jeden DFA A gilt $\mathcal{L}(A) = \emptyset$, wenn vom Startzustand aus kein akzeptierender Zustand zu erreichen ist. Diese Frage lässt sich durch eine Breiten- oder Tiefensuche leicht beantworten. Dies gilt ebenso für alle NFA-Varianten, die wir betrachtet haben. Bei regulären Ausdrücken ist ebenfalls leicht zu entscheiden, ob diese eine nicht-leere Sprache erzeugen: Gilt $\mathcal{L}(\alpha) = \emptyset$ für einen regulären Ausdruck α, so können wir diese so lange durch Anwenden für Rechenregeln vereinfachen, bis wir $\emptyset$ erhalten (siehe Aufgabe 3.18).

Für DFAs stellt auch das Universalitätsproblem kein Problem dar: Für jede Sprache $L \subseteq \Sigma^*$ gilt $L = \Sigma^*$ genau dann, wenn $\overline{L} = \emptyset$. Da DFAs sich leicht komplementieren lassen, können wir einfach das Leerheitsproblem für den Komplementautomaten von A lösen. Bei NFAs und regulären Ausdrücken haben wir allerdings keine andere Komplementierungsmethode als den Umweg über den entsprechenden DFA, der zu einem exponentiellen Zustandszuwachs führen kann. Man kann sogar zeigen, dass das Universalitätsproblem für NFAs und für reguläre Ausdrücke PSPACE-vollständig ist. Wir müssen also davon ausgehen, dass dieses Problem sich nicht effizient lösen lässt[34].

Zwei weitere Probleme von großer Bedeutung sind das **Inklusionsproblem** und das **Äquivalenzproblem**:

Inklusionsproblem für DFAs

Eingabe: Zwei DFAs A_1 und A_2.
Frage: Ist $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$?

[34] Außer es gilt P = PSPACE. Da dies eine noch stärkere Forderung ist als P = NP wird im allgemeinen davon ausgegangen, dass dies leider nicht der Fall ist.

ÄQUIVALENZPROBLEM FÜR DFAS

Eingabe: Zwei DFAs A_1 und A_2.
Frage: Ist $\mathcal{L}(A_1) = \mathcal{L}(A_2)$?

Auch diese beiden Probleme lassen sich analog für die verschiedenen Varianten von NFAs und für reguläre Ausdrücke definieren. Beide Probleme von theoretischem Interesse, da sie sich mit der Frage befasse, wie unterschiedlich zwei Automaten (oder reguläre Ausdrücke) sein können, die die gleiche Sprache (oder im Fall des Inklusionsproblems vergleichbare Sprachen) beschreiben. Darüber hinaus ist aber gerade das Äquivalenzproblem von praktischem Interesse. Im sogenannten *Model Checking* geht es darum, ein formales Modell für ein System mit der Spezifikation des Systems zu vergleichen und sicherzustellen, dass das System der Spezifikation entspricht. Dabei werden oft logische Formeln verwendet, die sich in endliche Automaten konvertieren lassen.

Ähnlich wie beim Universalitätsproblem können wir das Inklusionsproblem für DFAs leicht anhand des Leerheitsproblems und der bisher vorhandenen Konstruktionen lösen: Seien A_1, A_2 DFAs. Dann gilt $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$ genau dann, wenn $(\mathcal{L}(A_1) - \mathcal{L}(A_2)) = \emptyset$. Also können wir das Inklusionsproblem für zwei DFAs A_1, A_2 lösen, indem wir den DFA für die Sprache $(\mathcal{L}(A_1) - \mathcal{L}(A_2))$ konstruieren und sein Leerheitsproblem lösen.

Das Äquivalenzproblem für DFAs lässt sich auf zwei Arten lösen. Für zwei DFAs A_1, A_2 sind die folgenden drei Bedingungen äquivalent:

1. $\mathcal{L}(A_1) = \mathcal{L}(A_2)$,
2. $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$ und $\mathcal{L}(A_2) \subseteq \mathcal{L}(A_1)$,
3. die minimalen DFAs für $\mathcal{L}(A_1)$ und $\mathcal{L}(A_2)$ sind identisch (abgesehen von einer eventuell nötigen Umbenennung der Zustände).

Um zu testen, ob $\mathcal{L}(A_1) = \mathcal{L}(A_2)$ gilt, können wir also in beiden Richtungen die Inklusion testen, oder wir minimieren die beiden Automaten und überprüfen, ob die so erhaltenen DFAs identisch sind (abgesehen von einer Umbenennung der Zustände)[35].

Untere Schranken für das Äquivalenzproblem sind außerdem hilfreich, um die Schwierigkeit der Minimierung einzuschätzen. Im Fall von NFAs und regulären Ausdrücken können wir von der unteren Schranke für das Universalitätsproblem auch gleich auf entsprechende untere Schranken für das Inklusions- und das Äquivalenzproblem schließen: Da $\Sigma^* \subseteq \mathcal{L}(A)$ genau dann gilt, wenn $\Sigma^* = \mathcal{L}(A)$, können Inklusions- und Äquivalenzproblem nicht leichter sein als das Universalitätsproblem[36]. Außerdem wissen wir dadurch, dass nicht einmal NFAs (oder reguläre Ausdrücke) für die Sprache Σ^* effizient minimiert werden können. Später werden wir mächtigere Modelle kennen lernen, bei denen diese Probleme nicht einmal entscheidbar sind.

[35] Es gibt noch einen weiteren Algorithmus, der das Äquivalenzproblem für DFAs effizienter löst als diese Ansätze. Während der schnellste Minimierungsalgorithmus zu einer Laufzeit von $O(nk \log n)$ führt, kann dieses Problem bei der Verwendung geeigneter Datenstrukturen in $O(nk)$ gelöst werden.

[36] Auch diese Probleme sind für NFAs und reguläre Ausdrücke PSPACE-vollständig.

3.5 Reguläre Grammatiken

Als Abschluss dieses Kapitels befassen wir uns mit einem weiteren Modell zur Definition von regulären Sprachen, den *regulären Grammatiken*. Diese dienen vor allem dazu, das nächste Kapitel vorzubereiten. Die Syntax der regulären Grammatiken ist wie folgt definiert:

Definition 3.113 Eine **reguläre Grammatik** G über einem Alphabet Σ wird definiert durch

1. eine Alphabet V von **Variablen** (auch **Nichtterminale** oder **Nichtterminalsymbole** genannt), wobei $(\Sigma \cap V) = \emptyset$,
2. eine endliche Menge $P \subseteq V \times (\{\varepsilon\} \cup (\Sigma \cdot V))$ von **Regeln** (auch **Produktionen** genannt),
3. eine Variable $S \in V$ (dem **Startsymbol**).

Wir schreiben dies als $G := (\Sigma, V, P, S)$. Regeln der Form $(\alpha, \beta) \in P$ schreiben wir auch als $\alpha \to \beta$. Alle Regeln in P haben also die Form $A \to \varepsilon$ oder $A \to bC$ mit $A, C \in V$ und $b \in \Sigma$.

In der Literatur wird V auch oft mit N bezeichnet (für Nichtterminalalphabet), und Σ mit T (für Terminalalphabet). Die Buchstaben von Σ werden auch als **Terminale** bezeichnet.

Notation 3.114 Sei $G := (\Sigma, V, P, S)$ eine reguläre Grammatik. Für jede Regel $(\alpha, \beta) \in P$ oder $\alpha \to \beta$ bezeichnen wir α als **linke Seite** und β als rechte Seite der Regel. Wenn mehrere Regeln die gleiche linke Seite haben und wir Regeln mit einem Pfeil schreiben, so können wir diese Regeln zusammenfassen und die rechten Seiten durch ein | Symbol verbinden. Die Regeln $\alpha \to \beta_1$ bis $\alpha \to \beta_n$ können wir dann verkürzt schreiben als

$$\alpha \to \beta_1 \mid \cdots \mid \beta_n.$$

Die Semantik der regulären Grammatiken definieren wir wie folgt:

Definition 3.115 Sei $G := (\Sigma, V, P, S)$ eine reguläre Grammatik. Ein Wort $\alpha \in (\Sigma \cup V)^*$ bezeichnen wir auch als **Satzform** von G. Auf der Menge aller Satzformen von G definieren wir die Relation $\Rightarrow_G$ wie folgt:

Für alle $\alpha, \beta \in (\Sigma \cup V)^*$ gilt $\alpha \Rightarrow_G \beta$ genau dann, wenn Satzformen $\alpha_1, \alpha_2 \in (\Sigma \cup V)^*$, eine Variable $A \in V$ und eine Regel $(A, \gamma) \in P$ mit $\gamma \in (\{\varepsilon\} \cup (\Sigma \cdot V))$ existieren, so dass

$$\alpha = \alpha_1 \cdot A \cdot \alpha_2, \qquad \text{und} \qquad \beta = \alpha_1 \cdot \gamma \cdot \alpha_2.$$

Wir erweitern die Relation $\Rightarrow_G$ für jedes $n \in \mathbb{N}$ zu einer Relation $\Rightarrow_G^n$ auf $(\Sigma \cup V)^*$ für alle $\alpha, \beta \in (\Sigma \cup V)^*$ wie folgt:

- $\alpha \Rightarrow_G^0 \beta$ genau dann, wenn $\beta = \alpha$, und
- $\alpha \Rightarrow_G^{n+1} \beta$ genau dann, wenn ein γ existiert mit $\alpha \Rightarrow_G^n \gamma$ und $\gamma \Rightarrow_G \beta$.

Die **Ableitungsrelation** $\Rightarrow_G^*$ definieren wir für alle $\alpha, \beta \in (\Sigma \cup V)^*$ durch

$$\alpha \Rightarrow_G^* \beta \text{ genau dann, wenn ein } n \in \mathbb{N} \text{ existiert, für das } \alpha \Rightarrow_G^n \beta.$$

Gilt $\alpha \Rightarrow_G^* \beta$, so sagen wir, dass β (in G) **aus** α **ableitbar** ist. Eine Satzform β heißt **ableitbar** (in G), wenn sie aus dem Startsymbol S ableitbar ist. Eine **Ableitung** (von β in G) ist eine Folge $\gamma_0, \ldots, \gamma_n$ von Satzformen mit $\gamma_0 = S$, $\gamma_n = \beta$ und $\gamma_i \Rightarrow_G \gamma i + 1$ für $0 \leq i < n$.

Die von G **erzeugte Sprache** $\mathcal{L}(G)$ definieren wir als

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

Es gilt also $\alpha \Rightarrow_G \beta$, wenn β aus α erzeugt werden kann, indem man in α eine Variable entsprechend einer Regel aus P ersetzt. Eine Satzform α kann zu einer anderen Satzform β abgeleitet werden, wenn eine Folge von solchen Ersetzungen existiert, die α in β umschreibt. Die Grammatik G erzeugt also die Sprache aller Wörter, die nur aus Terminalen bestehen und sich aus dem Startsymbol von G ableiten lassen.

Beispiel 3.116 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Die reguläre Grammatik $G := (\Sigma, V, P, S)$ sei definiert durch $V := \{S, B\}$ und

$$P := \{S \to \mathtt{a}S \mid \mathtt{b}B \mid \varepsilon,\\ B \to \mathtt{b}B \mid \varepsilon\}.$$

Es gilt $\mathcal{L}(G) = \mathcal{L}(\mathtt{a}^*\mathtt{b}^*)$. Dies kann man sich leicht veranschaulichen, indem man die möglichen Ableitungen betrachtet. Das Startsymbol S kann gelöscht werden (durch $S \to \varepsilon$), beliebig oft durch $\mathtt{a}S$ ersetzt werden, oder durch $S \to \mathtt{b}B$ ersetzt werden. Sobald wir von S zu B gewechselt sind, können wir nie wieder ein S erhalten.

Es gilt also $S \Rightarrow_G^i \mathtt{a}^i B$ und $S \Rightarrow_G^* \mathtt{a}^i$ mit $i \in \mathbb{N}$. Danach kann B beliebig of durch $\mathtt{b}B$ ersetzt werden, bis die Ableitung durch $B \to \varepsilon$ beendet wird. Es gilt also $S \Rightarrow_G^i \mathtt{a}^i B \Rightarrow_G^j \mathtt{a}^i\mathtt{b}^j B \Rightarrow_G \mathtt{a}^i\mathtt{b}^j$ mit $j \in \mathbb{N}_{>0}$. Somit ist $\mathcal{L}(G) \subseteq \mathcal{L}(\mathtt{a}^*\mathtt{b}^*)$. Andererseits lässt sich auf die gleiche Art argumentieren, dass jedes Wort aus $\mathcal{L}(\mathtt{a}^*\mathtt{b}^*)$ aus S abgeleitet werden kann, eben durch $S \Rightarrow_G^i \mathtt{a}^i B \Rightarrow_G^j \mathtt{a}^i\mathtt{b}^j B \Rightarrow_G \mathtt{a}^i\mathtt{b}^j$ für $j \in \mathbb{N}_{>0}$, oder durch $S \Rightarrow_G^i \mathtt{a}^i S \Rightarrow_G \mathtt{a}^i$ für $j = 0$. Also ist auch $\mathcal{L}(\mathtt{a}^*\mathtt{b}^*) \subseteq \mathcal{L}(G)$. ◊

Wie leicht zu erkennen ist, benötigen wir in der Definition von $\Rightarrow_G$ die Satzform α_2 eigentlich nicht, denn jede Satzform α mit $S \Rightarrow_G^* \alpha$ kann höchstens eine Variable enthalten, und diese muss stets am rechten Ende von α stehen:

Lemma 3.117 *Sei $G := (\Sigma, V, P, S)$. Dann gilt für jede Satzform $\alpha \in (\Sigma \cup V)^*$ mit $S \Rightarrow_G^* \alpha$:*

$$\alpha \in \Sigma^* \text{ oder } \alpha \in (\Sigma^* \cdot V).$$

Beweis: Wir zeigen dies durch eine vollständige Induktion über die Zahl der Ableitungsschritte.

Induktionsanfang: Angenommen, $\alpha \in (\Sigma \cup V)^*$ und $S \Rightarrow_G^0 \alpha$ Dann gilt $\alpha = S$, und somit $\alpha \in V \subset (\Sigma^* \cdot V)$.

Induktionsschritt: Sei $n \in \mathbb{N}$. Angenommen, für alle $\alpha \in (\Sigma \cup V)^*$ mit $S \Rightarrow_G^n$ gilt $\alpha \in \Sigma^*$ oder $\alpha \in (\Sigma^* \cdot V)$.

Sei nun $\beta \in (\Sigma \cup V)^*$, und $S \Rightarrow_G^{n+1} \beta$. Dann existiert ein $\gamma \in (\Sigma \cup V)^*$ mit $S \Rightarrow_G^n \gamma \Rightarrow_G \beta$. Nach Induktionsvoraussetzung ist $\gamma \in \Sigma^*$ oder $\gamma \in (\Sigma^* \cdot V)$. Da Regeln nur auf Satzformen angewendet werden können, die eine Variable enthalten, ist $\gamma \in (\Sigma^* \cdot V)$. Jede Regeln in P kann eine Variable entweder durch ε ersetzten, oder durch eine Satzform aus $\Sigma \cdot V$. Im ersten Fall folgt $\beta \in \Sigma^*$, im zweiten $\beta \in (\Sigma^* \cdot V)$. □

Satz 3.118 *Sei Σ ein Alphabet. Dann gilt für jede Sprache $L \subseteq \Sigma^*$:*

L ist genau dann regulär, wenn eine reguläre Grammatik G existiert, für die $\mathcal{L}(G) = L$.

Beweisidee: Dieser Satz benötigt keine besonderen Kunstgriffe: Aus jedem DFA A lässt sich direkt eine reguläre Grammatik für $\mathcal{L}(A)$ ablesen, und jede reguläre Grammatik G kann direkt in einen NFA für $\mathcal{L}(G)$ umgewandelt werden. Im Prinzip sind reguläre Grammatiken nur andere Schreibweisen für NFAs. □

Beweis: Wir betrachten Hin- und Rück-Richtung getrennt.
$\Rightarrow$: Sei $L \in$ REG. Dann existiert ein DFA $A := (\Sigma, Q, \delta, q_0, F)$ mit $\mathcal{L}(A) = L$. Wir definieren nun eine reguläre Grammatik $G := (\Sigma, V, P, S)$ durch

$$\begin{aligned} V &:= Q, \\ S &:= q_0, \\ P &:= \{p \to aq \mid p, q \in Q, a \in \Sigma, \delta(p, a) = q\} \cup \{p \to \varepsilon \mid p \in F\}. \end{aligned}$$

Dann gilt für alle $w \in \Sigma^*$:

$$\begin{aligned} & & w \in \mathcal{L}(A) \\ \Leftrightarrow & & \delta(q_0, w) \in F \\ \Leftrightarrow & & \delta(q_0, w) = q \text{ für ein } q \in F \\ \Leftrightarrow & & q_0 \Rightarrow_G^* w \cdot q \text{ für ein } q \in F \\ \Leftrightarrow & & q_0 \Rightarrow_G^* w \cdot q \Rightarrow_G w \text{ für ein } q \in F \\ \Leftrightarrow & & q_0 \Rightarrow_G^* w \\ \Leftrightarrow & & w \in \mathcal{L}(G). \end{aligned}$$

Somit gilt $\mathcal{L}(G) = \mathcal{L}(A) = L$, und der Beweis für die Hin-Richtung ist fertig.
$\Leftarrow$: Sei $G := (\Sigma, V, P, S)$. Wir definieren einen NFA $A := (\Sigma, Q, \delta, q_0, F)$ durch

$$Q := V,$$

$$\begin{aligned} q_0 &:= S, \\ F &:= \{A \in Q \mid (A, \varepsilon) \in P\}, \\ \delta(A, b) &= \{C \in Q \mid (A, bC) \in S \end{aligned}$$

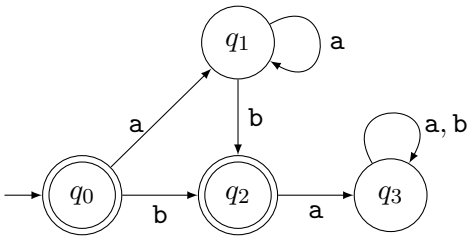
für alle $A \in Q$, $b \in \Sigma$. (Da zu jeder linken Seite mehrere Regeln mit dem gleichen Terminal auf der rechten Seite existieren können, würden wir es uns hier unnötig schwer machen, wenn wir darauf bestehen, dass A ein DFA ist.) Es gilt für alle $w \in \Sigma^*$:

$$\begin{aligned} & & w \in \mathcal{L}(G) \\ \Leftrightarrow & & S \Rightarrow_G^* w \\ \Leftrightarrow & & S \Rightarrow_G^* wA \Rightarrow_G^* w \text{ für ein } A \in V \\ \Leftrightarrow & & S \Rightarrow_G^* wA \Rightarrow_G^* w \text{ für ein } A \in V, \text{ und } (A, \varepsilon) \in P \\ \Leftrightarrow & & A \in \delta(S, w) \text{ für ein } A \in V, \text{ und } A \in F \\ \Leftrightarrow & & \delta(S, w) \in F \\ \Leftrightarrow & & w \in \mathcal{L}(A). \end{aligned}$$

Also gilt $\mathcal{L}(A) = \mathcal{L}(G)$, woraus wir schließen, dass $L \in \text{REG}$. Dies beendet den Beweis der Rück-Richtung. □

Selbstverständlich betrachten wir für jede der beiden Richtungen des Beweises ein kleines Beispiel:

Beispiel 3.119 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Der DFA A über Σ sei wie folgt definiert:



Anhand der Konstruktion aus dem Beweis von Satz 3.118 erhalten wir die reguläre Grammatik $G_A = (\Sigma, V, P, S)$, wobei $V = \{q_0, \ldots, q_3\}$, $S = q_0$, und

$$\begin{aligned} P = \{&q_0 \to \mathtt{a}q_1 \mid \mathtt{b}q_2 \mid \varepsilon, \\ &q_1 \to \mathtt{a}q_1 \mid \mathtt{b}q_2, \\ &q_2 \to \mathtt{a}q_3 \mid \varepsilon, \\ &q_3 \to \mathtt{a}q_3 \mid \mathtt{b}q_3\}. \end{aligned}$$

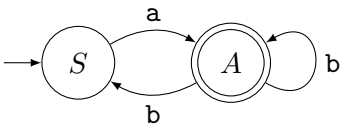
Wie Sie leicht sehen können, ist A nicht vollständig, und außerdem ist der Zustand in q_3 eine Sackgasse. Beide Phänomene stellen für die Grammatik G_A kein Problem dar: Kanten, die in A nicht vorhanden sind, kommen ebenfalls nicht als Regel vor; und die Variable q_3, die dem gleichnamigen Sackgassenzustand entspricht, kann niemals zu einem

Terminalwort abgeleitet werden. Eine Satzform, die q_3 enthält, verhält sich also genauso wie eine Sackgasse.

Für die Rück-Richtung betrachten wir nun die reguläre Grammatik $G := (\Sigma, V, P, S)$ mit $V := \{S, A\}$ und

$$P := \{S \to \mathtt{a}A, \\ A \to \mathtt{b}S \mid \mathtt{b}A \mid \varepsilon\}.$$

Der entsprechende NFA sieht wie folgt aus:



$\Diamond$

Anstelle der regulären Grammatiken betrachten wir nun eine Verallgemeinerung:

Definition 3.120 Eine **rechtslineare Grammatik** G über einem Alphabet Σ wird definiert durch

1. eine Alphabet V von **Variablen** (auch **Nichtterminal** oder **Nichtterminalsymbol** genannt), wobei $(\Sigma \cap V) = \emptyset$,
2. eine Menge $P \subseteq V \times (\Sigma^* \cup (\Sigma^* \cdot V))$ von **Regeln** (auch **Produktionen** genannt),
3. eine Variable $S \in V$ (dem **Startsymbol**).

Wir schreiben dies als $G := (\Sigma, V, P, S)$. Regeln der Form $(\alpha, \beta) \in P$ schreiben wir auch als $\alpha \to \beta$. Alle Regeln in P haben also die Form $A \to w$ oder $A \to wB$ mit $A, B \in V$ und $w \in \Sigma^*$.

Die Definitionen aus Definition 3.115 gelten analog. Es ist leicht zu sehen, dass jede reguläre Grammatik auch eine rechtslineare Grammatik ist. Umgekehrt gilt dies zwar nicht, allerdings haben beide Grammatikmodelle die gleiche Ausdrucksstärke:

Lemma 3.121 *Sei G eine rechtslineare Grammatik. Dann ist $\mathcal{L}(G) \in$ REG.*

Beweis: Übung. (Aufgabe 3.19) □

3.6 Aufgaben

Aufgabe 3.1 Zeigen Sie: FIN $\subseteq$ REG. Zeigen Sie dazu: Für jedes Alphabet Σ ist $\text{FIN}_\Sigma \subseteq \text{REG}_\Sigma$. Verwenden Sie dazu nur DFAs; *nicht* NFAs, reguläre Ausdrücke oder ähnliches.

Hinweise: Geben Sie an, wie man zu einer endlichen Sprache $L \subset \Sigma^*$ einen DFA A mit $\mathcal{L}(A) = L$ konstruiert. Sie können sich Ihre Arbeit deutlich einfacher machen, indem Sie die Zustände entsprechend aussagekräftig benennen. Beweisen Sie die Korrektheit Ihrer Konstruktion.

Aufgabe 3.2 Sei A der in Beispiel 3.40 definierte DFA. Zeigen Sie:

$$\mathcal{L}(A) = \{w \in \{\mathtt{a}, \mathtt{b}\}^* \mid |w| \text{ ist gerade}\}.$$

Aufgabe 3.3 Sei Σ ein Alphabet und sei $A := (\Sigma, Q, \delta, q_0, F)$ ein DFA. Sei $n \geq 2$ und seien $w_1, \ldots, w_n \in \Sigma$. Zeigen Sie:

$$\delta\big(\delta(q, w_1 \cdots w_{n-1}), w_n\big) = \delta\big(\delta(q, w_1), w_2 \cdots w_n\big).$$

Aufgabe 3.4 Welche der folgenden Sprachen sind regulär, welche nicht?

- $L := \{xx \mid x \in \{\mathtt{a}, \mathtt{b}\}^*, |x| \leq 5172342\}$,
- $L := \{\mathtt{a}^i\mathtt{b}^{2i} \mid i \in \mathbb{N}\}$,
- $\{\mathtt{a}^p \mid p \text{ ist eine Primzahl}\}$,
- $\{\mathtt{a}^n \mid n \text{ ist keine Primzahl}\}$.

Beweisen Sie jeweils Ihre Antwort.

Aufgabe 3.5 Sei die Sprache L definiert wie in Beispiel 3.13, also

$$L := \{b^i a^{j^2} \mid i \in \mathbb{N}_{>0}, j \in \mathbb{N}\} \cup \{a^k \mid k \in \mathbb{N}\}.$$

Zeigen Sie: Es existiert eine Pumpkonstante $n_L \in \mathbb{N}_{>0}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n_L$ folgende Bedingung erfüllt ist: Es existieren Wörter $u, v, w \in \Sigma^*$ mit $uvw = z$, $|v| \geq 1$, $|uv| \leq n_L$ und $uv^iw \in L$ für alle $i \in \mathbb{N}$.

Aufgabe 3.6 Im Beweis von Lemma 3.14 setzen wir voraus, dass der DFA A vollständig ist. Warum ist dies notwendig?

Aufgabe 3.7 Beweisen Sie Korollar 3.17. Zeigen Sie: Die Klasse REG ist abgeschlossen unter der Operation prefix.

Hinweis: Zeigen sie dazu, wie man einen DFA A so in einen DFA A' umwandeln kann, dass $\mathcal{L}(A') = \text{prefix}\big(\mathcal{L}(A)\big)$. Beweisen Sie die Korrektheit Ihrer Konstruktion.

Aufgabe 3.8 Beweisen Sie Lemma 3.60. Zeigen Sie: Die Klasse der regulären Sprachen ist abgeschlossen unter Shuffle-Produkt.

Hinweise: Nehmen Sie an, dass zwei DFAs $A_1 = (\Sigma, Q_1, \delta_1, q_{1,0}, F_1)$ mit $\mathcal{L}(A_1) = L_1$ und $A_2 = (\Sigma, Q_2, \delta_2, q_{2,0}, F_2)$ mit $\mathcal{L}(A_2) = L_2$ gegeben sind. Konstruieren Sie daraus einen NFA $A_3 = (\Sigma, Q_3, \delta_3, q_{3,0}, F_3)$ mit $\mathcal{L}(A_3) = \text{shuffle}(L_1, L_2)$ und begründen Sie die Korrektheit Ihrer Konstruktion.
Für eine effiziente Konstruktion von A_3 erhält man $|Q_3| = |Q_1| \cdot |Q_2|$.

Aufgabe 3.9 Beweisen Sie Lemma 3.69. Zeigen Sie: Die Klasse der regulären Sprachen ist abgeschlossen unter dem suffix-Operator.

Aufgabe 3.10 Die Klasse REG ist unter Vereinigung abgeschlossen; d. h., die Vereinigung von zwei regulären Sprachen ist ebenfalls eine reguläre Sprache. Es ist leicht zu sehen, dass die Klasse auch unter „dreistelliger Vereinigung“ abgeschlossen ist: Sind $L_1, L_2, L_3 \in \text{REG}$, dann gilt für $L_{123} := (L_1 \cup L_2 \cup L_3)$ auch $L_{123} \in \text{REG}$, denn $L_{123} = (L_1 \cup L_2) \cup L_3$, und $(L_1 \cup L_2) \in \text{REG}$ gilt wegen des Abschlusses von REG unter Vereinigung. Dieses Argument lässt sich natürlich auch auf jede andere *endliche Vereinigung* ausweiten; sind $L_1 \ldots, L_n \in \text{REG}$ (mit $n \in \mathbb{N}$), dann ist auch $\bigcup_{i=1}^{n} L_i$ eine reguläre Sprache.

Analog dazu lässt sich die *unendliche* Vereinigung definieren: Sei $\big(L_i\big)_{i\in\mathbb{N}}$ eine unendliche Folge von regulären Sprachen. Dann ist die unendliche Vereinigung dieser Sprachen definiert als $\bigcup_{i\in\mathbb{N}} L_i$.

Beweisen oder widerlegen Sie die folgende Behauptung: Jede unendliche Vereinigung von regulären Sprachen ist eine reguläre Sprache. (Mit anderen Worten: Sei $\big(L_i\big)_{i\in\mathbb{N}}$ eine unendliche Folge von regulären Sprachen. Dann ist auch die Sprache $\bigcup_{i\in\mathbb{N}} L_i$ regulär.)

Aufgabe 3.11 Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Seien $x, y \in \Sigma^*$ mit $x \equiv_L y$. Zeigen Sie:

$$x \in L \text{ genau dann, wenn } y \in L.$$

Aufgabe 3.12 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{ww^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$. Zeigen Sie: Für alle $x \in \Sigma^*$ gilt $[x]_{\equiv_L} = \{x\}$.

Aufgabe 3.13 (Fortgeschritten) Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und

- Sei
$$L_1 := \{\mathtt{a}^i y \mid i \in \mathbb{N}_{>0}, y \in \Sigma^*, |y|_{\mathtt{a}} \geq i\}.$$
Beweisen oder widerlegen Sie: $L_1 \in \text{REG}$.

- Sei
$$L_2 := \{\mathtt{a}^i y \mid i \in \mathbb{N}_{>0}, y \in \Sigma^*, |y|_{\mathtt{a}} < i\}.$$
Beweisen oder widerlegen Sie: $L_2 \in \text{REG}$.

Aufgabe 3.14 (Fortgeschritten) Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und

$$L := \{w \in \Sigma^* \mid \text{die Wörter } \mathtt{ab} \text{ und } \mathtt{ba} \text{ kommen gleich oft in } w \text{ vor}\}.$$

Beweisen oder widerlegen Sie: $L \in \text{REG}$.

Aufgabe 3.15 [Schwer] Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und

$$L := \{xyyz \mid x, y, z \in \Sigma^+\}.$$

Beweisen oder widerlegen Sie: $L \in \text{REG}$.

Aufgabe 3.16 Beweisen Sie die Korrektheit der Rechenregeln für reguläre Ausdrücke aus Abschnitt A.3 im Anhang. Sie können sich an den Beweisen in Beispiel 3.86 orientieren.

Aufgabe 3.17 Für $n \geq 1$ betrachten wir das Alphabet $\Sigma_n := \{a_1, \ldots, a_n\}$ und den NFA $A := (\Sigma_n, Q_n, \delta_n, q_0, F)$, wobei

- $Q_n := \{q_0, q_1, \ldots, q_n, q_F\}$,
- $F := \{q_F\}$,
- $\delta_n(q, a_i) := \{q_i, q_F\}$ für alle $q \in Q - \{q_F\}$ und alle $1 \leq i \leq n$, sowie $\delta(q_F, a) = \emptyset$ für alle $a \in \Sigma$.

1. Geben Sie die graphische Darstellung von A_3 an und wandeln Sie diesen NFA anhand von `ENFA2RegEx` (Algorithmus 3) in einen äquivalenten regulären Ausdruck um.
2. Sei $n \geq 1$. Wie viele Zeichen aus Σ_n enthält der reguläre Ausdruck, den `ENFA2RegEx` aus A_n berechnet? Wie viele Zeichen aus Σ_n enthält der kürzeste reguläre Ausdruck für $\mathcal{L}(A_n)$?

Aufgabe 3.18 Beweisen Sie die folgende Aussage: Für jeden regulären Ausdruck α gilt:

1. $\mathcal{L}(\alpha) = \emptyset$, oder
2. es existiert ein regulärer Ausdruck β mit $\mathcal{L}(\beta) = \mathcal{L}(\alpha)$, und β enthält nicht das Zeichen $\emptyset$.

Aufgabe 3.19 Beweisen Sie Lemma 3.121. Zeigen Sie also: Für jede rechtslineare Grammatik G ist $\mathcal{L}(G) \in \text{REG}$.

Aufgabe 3.20 Beweisen Sie Lemma 3.112. Zeigen Sie dazu: Sind $L, R \in \text{REG}$, dann ist die Sprache (L/R) regulär.

3.7 Bibliographische Anmerkungen

Dieser Abschnitt ist momentan nur ein Platzhalter. In Kürze werden hier einige Kommentare zu den verwendeten Quellen und weiterführendem Lesematerial zu finden sein.

4 Kontextfreie Sprachen

Wir haben im vorigen Kapitel verschiedene Modelle zur Definition regulärer Sprachen kennen gelernt und deren Eigenschaften ausführlich untersucht. Dabei haben wir einerseits erfahren, dass reguläre Sprachen vergleichsweise handhabbar sind, andererseits haben wir aber auch gesehen, dass sich viele Sprachen in diesen Modellen nicht ausdrücken. In vielen Bereichen der Informatik ist eine größere Ausdrucksstärke notwendig. Beispielsweise können die Syntax von Programmiersprachen oder von XML-Dokumenten nicht anhand regulärer Sprachen beschrieben werden. Selbst für eingeschränkte Mechanismen wie zum Beispiel reguläre Ausdrücke ist die Syntax zu komplex, um Sie anhand einer regulären Sprache definieren zu können:

Beispiel 4.1 Da reguläre Ausdrücke selbst nur Wörter über einem Alphabet sind, hindert uns nichts daran, die Sprache aller syntaktisch korrekten regulären Ausdrücke zu definieren. Um unnötigen Ärger mit unbegrenzt großen Alphabeten zu vermeiden, beschränken wir uns für dieses Beispiel auf reguläre Ausdrücke über einem zweibuchstabigem Alphabet. Allerdings verwenden wir zur Definition von regulären Ausdrücken noch weitere Symbole (wie zum Beispiel ε und $|$). Diese müssen natürlich auch in unserem Alphabet vorkommen. Um unnötige Verwirrung zu vermeiden, benennen wir diese Zusatzzeichen daher um[37].

Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, e, \oslash, \mathtt{(}, \mathtt{)}, \mathtt{|}, \mathtt{.}, \mathtt{*}\}$. Die Sprache L_{RX} sei wie folgt rekursiv definiert:

1. Es gilt $\mathtt{a} \in L_{\mathrm{RX}}$, $\mathtt{b} \in L_{\mathrm{RX}}$, $e \in L_{\mathrm{RX}}$ und $\oslash \in L_{\mathrm{RX}}$.
2. Sind $u, v \in L_{\mathrm{RX}}$, so gilt auch
   - $\mathtt{(}u \mathtt{\ |\ } v\mathtt{)} \in L_{\mathrm{RX}}$,
   - $\mathtt{(}u \mathtt{\ .\ } v\mathtt{)} \in L_{\mathrm{RX}}$ und
   - $u\mathtt{*} \in L_{\mathrm{RX}}$.

Dass diese Sprache den syntaktisch korrekten regulären Ausdrücken über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ entspricht, ergibt sich direkt aus Definition 3.83.

Dabei stellt sich natürlich die Frage, ob diese Sprache selbst regulär ist. Leider ist das nicht der Fall, da L_{RX} zum Beispiel die Wörter

$$\mathtt{(a\ .\ a)}, \mathtt{((a\ .\ a)\ .\ a)}, \mathtt{(((a\ .\ a)\ .\ a)\ .\ a)}, \ldots$$

[37] Und zwar e statt ε, $\oslash$ statt $\emptyset$, (und) statt (und), . statt $\cdot$, | statt |, sowie * statt $+$. Allerdings ist die exakte Umbenennung nebensächlich, da wir uns in diesem Beispiel nur die Syntax der regulären Ausdrücke ansehen, nicht aber ihre Semantik. In der Praxis haben reguläre Ausdrücke übrigens meistens kein Symbol für die Konkatenation, und das leere Wort wird gewöhnlich durch eine Stringkonstante wie "" abgebildet. Die leere Menge kann in praktischen Ausdrücken meist gar nicht definiert werden.

enthält. Anschaulich gesprochen muss L_{RX} sicherstellen, dass alle Klammern auch bei beliebigen Schachtelungstiefen sicherstellen korrekt geöffnet und geschlossen werden. Anhand geeigneter Abschlusseigenschaften ist schnell gezeigt, dass L_{RX} nicht regulär ist. ◊

Um die Syntax regulärer Ausdrücke zu beschreiben benötigen wir also einen Mechanismus, der mächtiger ist als die regulären Ausdrücke. Ähnliches gilt für XML-Dokumente:

Beispiel 4.2 Wir betrachten eine Liste von Cocktailrezepten[38] im XML-Format:

```
<cocktails>
 <cocktail>
  <name>Whisky Sour</name>
  <zutaten>
   <zutat><name>Bourbon</name><menge>4 cl</menge></zutat>
   <zutat><name>Zitronensaft</name><menge>2 cl</menge></zutat>
   <zutat><name>Gomme-Sirup</name><menge>1 cl</menge></zutat>
  </zutaten>
 </cocktail>
 <cocktail>
  <name>Brandy Alexander</name>
  <zutaten>
   <zutat><name>Weinbrand</name><menge>3 cl</menge></zutat>
   <zutat><name>Creme de Cacao</name><menge>3 cl</menge></zutat>
   <zutat><name>Sahne</name><menge>3 cl</menge></zutat>
  </zutaten>
 </cocktail>
 <cocktail>
  <name>Golden Nugget</name>
  <zutaten>
   <zutat><name>Maracujanektar</name><menge>12 cl</menge></zutat>
   <zutat><name>Zitronensaft</name><menge>2 cl</menge></zutat>
   <zutat><name>Limettensirup</name><menge>2 cl</menge></zutat>
  </zutaten>
 </cocktail>
 <cocktail>
  <name>Martins wilde Party</name>
  <zutaten>
   <zutat><name>Mineralwasser (still)</name><menge>10 cl</menge></zutat>
   <zutat><name>Leitungswasser</name><menge>10 cl</menge></zutat>
  </zutaten>
 </cocktail>
</cocktails>
```

[38] Dieses Beispiel soll nicht als Aufforderung zum Alkoholkonsum missverstanden werden. Wenn Sie Alkohol konsumieren, dann bitte in Maßen und verantwortungsvoll, erst nach dem Besuch der Vorlesung und bevorzugt auch erst nach dem Bearbeiten der Übungen.

Jedes XML-Dokument besteht aus ineinander verschachtelten *Elementen*. In diesem Beispiel haben wir ein Element mit dem Namen `cocktails`, dieses enthält vier Elemente namens `cocktail`, von denen jedes wiederum jeweils ein Element mit dem Namen `name` und dem Namen `zutaten` enthält. Innerhalb der `zutaten`-Elemente sind wiederum andere Elemente enthalten.

Dabei wir jedes Element durch sogenannte *Tags* geöffnet und geschlossen, die den Namen des Elements enthalten. Beispielsweise wird ein Element mit dem Namen `zutaten` durch das Tag `<zutaten>` geöffnet und durch das Tag `</zutaten>` geschlossen. Für jedes syntaktisch korrekte XML-Dokument (also auch außerhalb dieses Beispiels) ist es wichtig, dass verschiedenen Tags nicht überlappen dürfen. Konstruktionen wie beispielsweise `<a><b></a></b>` sind also ausdrücklich ausgeschlossen, da hier keine korrekte Verschachtelung vorhanden ist, im Gegensatz zu (zum Beispiel) `<a><b></b></a>`. Wir können die Menge aller syntaktisch korrekten XML-Dokumente als eine Sprache auffassen. Es lässt sich leicht zeigen, dass diese Sprache nicht regulär ist. In nächsten Beispiel behandeln wir diese Frage etwas abstrakter. ◊

XML-Dokumente bestehen im Kern aus geschachtelten Paaren von verschiedenen Klammern. Dieses Phänomen tritt in verschiedenen formalen Sprachen auf. Daher lohnt es sich, dies etwas allgemeiner zu betrachten, indem man die folgende Sprache (bzw. Familie von Sprachen) untersucht:

Beispiel 4.3 Sei Σ ein Alphabet. Wir definieren eine markierte Kopie von Σ durch $\hat{\Sigma} := \{\hat{a} \mid a \in \Sigma\}$ (hierbei seien Σ und $\hat{\Sigma}$ disjunkt). Die **Dyck-Sprache[39] (zu Σ)** ist durch die folgenden Regeln rekursiv definiert:

1. Für alle $a \in \Sigma$ ist $a\hat{a} \in D_\Sigma$.
2. Sind $u, v \in D_\Sigma$, $a \in \Sigma$, so gilt auch
   - $(a \cdot u \cdot \hat{a}) \in D_\Sigma$, und
   - $(u \cdot v) \in D_\Sigma$.

Im Prinzip beschreiben die Dyck-Sprachen die gleichen Strukturen wie XML-Dokumente. Die Buchstaben aus Σ entsprechen dabei öffnenden Tags, die dazugehörenden Buchstaben aus $\hat{\Sigma}$ die entsprechenden schließenden Tags. Für Beispiel 4.2 würden wir beispielsweise Σ definieren als

$$\Sigma := \{\texttt{cocktail}, \texttt{cocktails}, \texttt{menge}, \texttt{name}, \texttt{zutat}, \texttt{zutaten}\}.$$

Es ist leicht zu sehen, dass D_Σ selbst dann nicht regulär ist, wenn Σ nur einen einzigen Buchstaben enthält: Sei $\Sigma := \{\texttt{a}\}$, und angenommen, D_Σ ist regulär. Dann ist die Sprache $D_\Sigma \cap \mathcal{L}(\texttt{a}^*(\hat{\texttt{a}})^*) = \{\texttt{a}^i\hat{\texttt{a}}^i \mid i \in \mathbb{N}\}$ ebenfalls regulär. Widerspruch. ◊

[39] Benannt nach Walther von Dyck, mit vollem Namen Walther Franz Anton Ritter von Dyck. Laut einer Festschrift der TU München soll er laut seiner Enkelin folgendes gesagt haben: „Mathematik ist nicht schwer, man muss sie nur richtig erklären“. Diese Überlieferung aus dritter Hand ist zwar nicht unbedingt zuverlässig; aber das Zitat ist zu schön, um es auszulassen.

Gleiches gilt für nahezu alle üblichen Programmiersprachen: Da diese eigentlich immer verschiedenen Arten von Klammerpaaren verwenden und noch dazu rekursiv definiert sind, stoßen reguläre Sprachen hier schnell an ihre Grenzen. In diesem Kapitel untersuchen wir Modelle, die uns aus dieser unangenehmen Situation retten.

4.1 Kontextfreie Grammatiken und kontextfreie Sprachen

Die *kontextfreien Grammatiken*[40] sind eine Erweiterung der regulären Grammatiken. Der einzige Unterschied in der Definition ist hierbei, dass auf der rechten Seite jeder Regel beliebige (auch leere) Kombinationen aus Variablen und Terminalen stehen können:

Definition 4.4 Eine **kontextfreie Grammatik (CFG)** G über einem Alphabet Σ wird definiert durch

1. eine Alphabet V von **Variablen** (auch **Nichtterminal** oder **Nichtterminalsymbol** genannt), wobei $(\Sigma \cap V) = \emptyset$,
2. eine endliche Menge $P \subseteq (V \times (\Sigma \cup V)^*$ von **Regeln** (auch **Produktionen** genannt),
3. eine Variable $S \in V$ (dem **Startsymbol**).

Wir schreiben dies als $G := (\Sigma, V, P, S)$. Wie bei den regulären Grammatiken schreiben wir Regeln der Form $(\alpha, \beta) \in P$ auch als $\alpha \to \beta$. Alle Regeln in P haben also die Form $A \to \beta$ mit $\beta \in (\Sigma \cup V)^*$.

Wie bei den regulären Grammatiken bezeichnen wir die beiden Teile einer Regel als rechte und linke Seite (siehe Notation 3.114), und auch die Relationen $\Rightarrow_G$, $\Rightarrow_G^n$ und $\Rightarrow_G^*$ sowie die Begriffe *Satzform* und *ableitbar* sind analog zu Definition 3.115 definiert. Ebenso ist es mit der von einer kontextfreien Grammatik G erzeugten Sprache:

Definition 4.5 Sei $G := (\Sigma, V, P, S)$ eine CFG. Die von G **erzeugte Sprache** $\mathcal{L}(G)$ definieren wir als

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

Eine Sprache $L \subseteq \Sigma^*$ heißt **kontextfrei**, wenn eine kontextfreie Grammatik G existiert, für die $\mathcal{L}(G) = L$. Wir bezeichnen[41] die **Klasse aller kontextfreien Sprachen** über dem Alphabet Σ mit CFL_Σ, und definieren die Klasse aller kontextfreien Sprachen $\mathrm{CFL} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{CFL}_\Sigma$.

Im Gegensatz zu regulären Grammatiken können kontextfreie Grammatiken auf den rechten Seiten ihrer Regeln beliebige Wörter aus $(\Sigma \cup V)^*$ enthalten. Dadurch können auch nicht reguläre Sprachen erzeugt werden. Es gilt:

[40] Im Singular abgekürzt als CFG, vom Englischen *context-free grammar*.
[41] Die Abkürzung CFL stammt von *context-free language(s)*.

Satz 4.6 REG $\subset$ CFL

Beweis: Jede reguläre Grammatik ist auch eine kontextfreie Grammatik, daher folgt REG $\subseteq$ CFL direkt aus der Definition.

Sei nun Σ ein Alphabet mit mindestens zwei Buchstaben[42]. O. B. d. A. sei $\Sigma \supseteq \{\mathtt{a}, \mathtt{b}\}$. Die kontextfreie Grammatik $G := (\Sigma, V, P, S)$ über Σ sei definiert durch $V := \{S\}$ und $P := \{S \to \mathtt{a}S\mathtt{b}, S \to \varepsilon\}$. Es gilt $\mathcal{L}(G) = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Da G eine kontextfreie Grammatik ist, ist $\mathcal{L}(G)$ eine kontextfreie Sprache. Wie wir bereits wissen, ist $\mathcal{L}(G)$ nicht regulär; somit ist REG $\neq$ CFL. $\square$

In dem Beweis von Satz 4.6 haben wir bereits erfahren, dass die Sprache $\{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$ kontextfrei ist. Wir betrachten noch ein paar weitere Beispiele:

Beispiel 4.7 Sei Σ ein Alphabet. Die Sprache PAL_Σ, Sprache aller Palindrome über Σ, sei definiert als $\mathrm{PAL}_\Sigma := \{w \in \Sigma^* \mid w = w^R\}$. In Beispiel 3.38 haben wir bereits erfahren, dass PAL_Σ nicht regulär ist, wenn $|\Sigma| \geq 2$. Sei nun $G := (\Sigma, V, P, S)$ mit $V := \{S\}$ und

$$P := \{S \to aSa \mid a \in \Sigma\} \cup \{S \to a \mid a \in \Sigma\} \cup \{S \to \varepsilon\}.$$

Dann ist $\mathcal{L}(G) = \mathrm{PAL}_\Sigma$. Dies zu beweisen ist ein wenig mühselig. Bevor wir uns an diesem Beweis wagen, stellen wir zur Veranschaulichung fest, dass sich die Regeln aus P in zwei unterschiedliche Gruppen unterteilen lassen, nämlich Regeln der Form $S \to aSa$, die auf der rechten Seite genau eine Variable haben, und Regeln der Form $S \to a$ oder $S \to \varepsilon$, die rechts keine Variable haben. Dadurch ist sichergestellt, dass jede Satzform, die aus S abgeleitet werden kann, höchstens eine Variable enthält. Außerdem können nur anhand der ersten Art von Regeln beliebig lange Ableitungen erzeugt werden; die Regeln der zweiten Art dienen nur dazu, die Ableitung zu beenden. Sie können schnell feststellen, dass diese Ableitungen immer die folgende Form haben:

$$S \Rightarrow_G a_1Sa_1 \Rightarrow_G a_1a_2Sa_2a_1 \Rightarrow_G \cdots \Rightarrow_G wSw^R,$$

wobei $w \in \Sigma^*$. Diese Satzformen sind also alle Palindrome (über dem Alphabet $\Sigma \cup V$), und durch Anwendung der Regel $S \to \varepsilon$ oder einer Regel $S \to a$ kann daraus ein Palindrom (gerade oder ungerader Länge) über Σ abgeleitet werden.

Wir schreiten nun zum Beweis, dass $\mathcal{L}(G) = \mathrm{PAL}_\Sigma$.
$\mathcal{L}(G) \subseteq \mathrm{PAL}_\Sigma$: Da die rechte Seite jeder Regel von P höchstens eine Variable enthält, kann jede aus S ableitbare Satzform höchstens eine Variable enthalten. Ist also $S \Rightarrow_G^* \alpha$ mit $\alpha \notin \Sigma^*$, müssen $u, v \in \Sigma^*$ existieren mit $\alpha = uSv$. Wir zeigen nun durch Induktion über die Zahl n der Ableitungsschritte, dass für alle $w \in \Sigma^*$ folgendes gilt:

$$\text{Ist } S \Rightarrow_G^n uSv \text{ mit } u, v \in \Sigma^*, \text{ dann ist } v^R = u.$$

Wir beginnen mit $n = 0$. Nach Definition ist$S \Rightarrow_G^n uSv$ nur für $u = v = \varepsilon$ erfüllt; da $\varepsilon = \varepsilon^R$, gilt $u = v^R$, und die Behauptung stimmt.

Angenommen, die Behauptung gilt für ein $n \in \mathbb{N}$. Seien nun $u', v' \in \Sigma^*$ mit $S \Rightarrow_G^{n+1} u'Sv'$. Also existieren Wörter $u, v \in \Sigma^*$ mit

$$S \Rightarrow_G^n uSv \Rightarrow_G u'Sv'.$$

[42] Enthält Σ übrigens nur einen Buchstaben, so ist $\mathrm{REG}_\Sigma = \mathrm{CFL}_\Sigma$. Wir werden das in Satz 4.23 beweisen.

Also existiert ein $a \in \Sigma$ mit $u' = ua$ und $v' = av$. Gemäß unserer Induktionsvoraussetzung ist $u = v^R$, und somit

$$\begin{aligned} u' &= ua \\ &= v^R a \\ &= (av)^R = (v')^R. \end{aligned}$$

Also gilt $u' = (v')^R$, und der Beweis des Induktionsschrittes ist bewiesen.

Sei nun $w\mathcal{L}(G)$. Dann existiert ein $n \in \mathbb{N}$ mit $S \Rightarrow_G^n w$, und Wörter $u, v \in \Sigma^*$ mit also $S \Rightarrow_G^{n-1} uSv \Rightarrow_G w$. Außerdem ist entweder $w = uv$, oder es existiert ein $a \in \Sigma$ mit $w = uav$. Gemäß der Behauptung, die wir soeben bewiesen haben, ist $u = v^R$. Also ist entweder $w = u \cdot u^R$, oder $w = uau^R$. In beiden Fällen gilt $w = w^R$, also $w \in \mathrm{PAL}_\Sigma$.

$\mathcal{L}(G) \supseteq \mathrm{PAL}_\Sigma$: Wir zeigen zuerst, dass für jedes Wort $w \in \mathrm{PAL}_\Sigma$ einer der beiden folgenden Fälle erfüllt ist:

1. $w \in (\Sigma \cup \{\varepsilon\})$, oder
2. es existieren ein Wort $v \in \mathrm{PAL}_\Sigma$ und ein Buchstabe $a \in \Sigma$ mit $w = ava$.

Sei $w \in \mathrm{PAL}_\Sigma$. Angenommen, $|w| \leq 1$. Dann ist $w \in (\Sigma \cup \{\varepsilon\})$, der erste Fall der Behauptung ist erfüllt. Angenommen, $|w| \geq 2$. Dann existieren ein $w' \in \Sigma^*$ und ein $a \in \Sigma$ mit $w = aw'$. Da $|w| \geq 2$ ist $|w'| \geq 1$, und da $w = w^R$ muss der letzte Buchstabe von w' ein a sein. Also existiert ein $u \in \Sigma^*$ mit $w' = ua$. Also ist $w = aua$. Wir müssen nun noch zeigen, dass $u \in \mathrm{PAL}_\Sigma$. Da $w = w^R$, ist

$$\begin{aligned} aua &= (aua)^R \\ &= (ua)^R \cdot a \\ &= a \cdot u^R \cdot a. \end{aligned}$$

Wir können von dieser Gleichung an beiden Enden der Wörter das a abspalten und erhalten $u = u^R$, also $u \in \mathrm{PAL}_\Sigma$.

Wir können also PAL_Σ folgendermaßen rekursiv definieren:

- Basisregeln: $\varepsilon \in \mathrm{PAL}_\Sigma$ und $a \in \mathrm{PAL}_\Sigma$.
- Rekursive Regeln: Ist $w \in \mathrm{PAL}_\Sigma$, dann ist $awa \in \mathrm{PAL}_\Sigma$ für alle $a \in \Sigma$.

Wir zeigen nun durch Induktion über diese rekursive Definition, dass $\mathrm{PAL}_\Sigma \subseteq \mathcal{L}(G)$. Für jede der Basisregeln haben wir eine entsprechende Regel $S \to w$ (mit $w \in \{\varepsilon\} \cup \Sigma$) in P; also gilt $S \Rightarrow_G w$ und somit $w \in \mathcal{L}(G)$. Angenommen, für ein $w \in \mathrm{PAL}_\Sigma$ gilt $w \in \mathcal{L}(G)$. Sei $a \in \Sigma$. Da $w \in \mathcal{L}(G)$, gilt $S \Rightarrow_G^* w$. Zusammen mit der Regel $S \to aSa$ stellen wir fest:

$$S \Rightarrow_G aSa \Rightarrow_G^* awa,$$

also $awa \in \mathcal{L}(G)$. Also ist $\mathrm{PAL}_\Sigma \subseteq \mathcal{L}(G)$. $\Diamond$

Wie Sie an Beispiel 4.7 sehen können, ist es bei der Konstruktion von kontextfreien Grammatiken von Vorteil, wenn wir bereits über eine rekursive Definition der zu erzeugenden Sprache verfügen. Anhand dieser Definition ist es oft leicht, zu zeigen, dass die konstruierte Grammatik alle Wörter der Zielsprache erzeugen kann. Fehlt eine solche rekursive Definition, kann der entsprechende Beweis recht mühsam werden. Gleiches gilt für den Beweis der umgekehrten Richtung.

Um ein weiteres Beispiel der Konstruktion von regulären Grammatiken aus rekursiven Definitionen zu untersuchen greifen wir auf unsere Beispielsprache der syntaktisch korrekten regulären Ausdrücke zurück:

Beispiel 4.8 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, e, \oslash, \mathtt{(}, \mathtt{)}, \mathtt{|}, \mathtt{.}, \mathtt{*}\}$. Die Sprache L_{RX} über Σ sei definiert wie in Beispiel 4.1. Wir definieren die CFG $G := (\Sigma, V, P, S)$ mit $V := \{S\}$, wobei P genau die folgenden Regeln enthalte:

1. $S \to \mathtt{a} \mid \mathtt{b} \mid e \mid \oslash$,
2. $S \to \mathtt{(}S \mathtt{\ |\ } S\mathtt{)}$,
3. $S \to \mathtt{(}S \mathtt{\ .\ } S\mathtt{)}$ und
4. $S \to S\mathtt{*}$.

Es gilt: $\mathcal{L}(G) = L_{\mathrm{RX}}$. Dass $L_{\mathrm{RX}} \subseteq \mathcal{L}(G)$ gilt, lässt sich leicht anhand einer Induktion über den Aufbau von L_{RX} beweisen. Die andere Richtung ist anstrengender und mit unseren Werkzeugen nicht leicht anzugehen. Wenn wir Ableitungsbäume eingeführt haben können wir diese Richtung durch eine Induktion über die Höhe der Ableitungsbäume zeigen.

Diese CFG ist für unsere bisherigen Bedürfnisse ausreichend; allerdings werden wir in Beispiel 4.57 sehen, wie wir eine CFG konstruieren können, die auch die beim Verzicht auf Klammern notwendigen Präzedenzregeln ausdrücken kann. ◊

Auch für jede der Dyck-Sprachen können wir eine CFG angeben:

Beispiel 4.9 Sei Σ ein Alphabet, und die Dyck-Sprache über Σ sei definiert wie in Beispiel 4.3. Wir definieren nun eine CFG $G := (\Sigma_D, V, P, S)$ durch

- $\Sigma_D := \Sigma \cup \hat{\Sigma}$,
- $V := \{S\}$, und
- $P := \{S \to a\hat{a} \mid a \in \Sigma\} \cup \{S \to aS\hat{a} \mid a \in \Sigma\} \cup \{S \to SS\}$.

Dann ist $\mathcal{L}(G) = D_\Sigma$ (dies ist durch zwei Induktionen leicht nachzuweisen). ◊

Ein Konzept, das im Umgang mit CFGs eine große Rolle spielt, ist der **Ableitungsbaum** (engl. **parse tree**):

Definition 4.10 Sei $G := (\Sigma, V, P, S)$ eine CFG. Jede Ableitung $S \Rightarrow_G^* w$ (mit $w \in \Sigma^*$) lässt sich als gerichteter Baum darstellen, bei dem

- jeder Knoten mit einem Symbol aus $\Sigma \cup V \cup \{\varepsilon\}$ markiert ist, und
- die Kinder jedes Knotens eine festgelegte Reihenfolge haben. In der Zeichnung eines Ableitungsbaums werden diese Kinder von links nach rechts dargestellt; das erste Kind eines Knotens steht also ganz links, das letzte ganz rechts.

Die Wurzel des Baums ist mit dem Startsymbol S markiert. Jeder Knoten mit seinen Kindern repräsentiert die Anwendung einer Regel aus P:

- Die Anwendung einer Regel $A \to \beta$ mit $A \in V$, $\beta = \beta_1 \cdots \beta_n$ mit $n \in \mathbb{N}_{>0}$ und $\beta_i \in (\Sigma \cup V)$ für $1 \leq i \leq n$ wird im Ableitungsbaum repräsentiert durch einen Knoten, der mit dem Symbol A markiert ist und n Kinder hat, so dass das i-te Kind dem Symbol β_i entspricht.
- Die Anwendung einer Regel $A \to \varepsilon$ wird im Ableitungsbaum repräsentiert durch einen Knoten, der mit dem Symbol A markiert ist und genau ein Kind hat, das mit ε markiert ist.

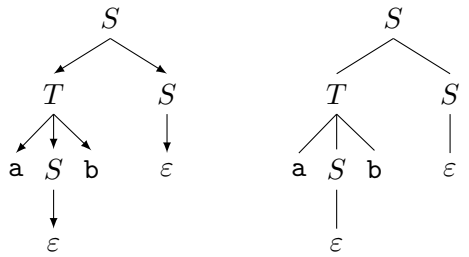
Ist $S \Rightarrow_G \gamma_1 \Rightarrow_G \cdots \Rightarrow_G \gamma_{n-1} \Rightarrow_G w$ eine Ableitung für w (in G), so bezeichnen wir den entsprechenden Ableitungsbaum als **Ableitungsbaum für w (in G)**.

Notation 4.11 Nach unserer Definition sind Ableitungsbäume gerichtete Bäume. Streng formal gesehen müssen sie daher auch gerichtet gezeichnet werden; das heißt, dass die Knoten eigentlich durch Pfeile statt durch einfache Kanten verbunden sein müssten. Wir vereinbaren aber als Konvention, dass die Pfeilspitzen auch weggelassen werden können.

Beispiel 4.12 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $G := (\Sigma, \{S, T\}, P, S)$ mit

$$P := \{S \to TS \mid \varepsilon, T \to \mathtt{a}S\mathtt{b}\}.$$

Das Wort $\mathtt{ab}$ lässt sich durch die Ableitung $S \Rightarrow_G TS \Rightarrow_G \mathtt{a}S\mathtt{b}S \Rightarrow_G \mathtt{ab}S \Rightarrow_G \mathtt{ab}$ aus S ableiten. Der Ableitungsbaum für diese Ableitung lässt sich wie folgt darstellen (links in der formal sauberen, rechts in der vereinfachten Darstellung):



Die oben angegebene Ableitung ist übrigens nicht die einzige, die diesem Ableitungsbaum entspricht. Die beiden folgenden Ableitungen führen zu dem gleichen Ableitungsbaum:

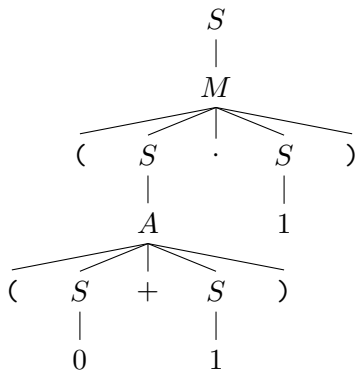
1. $S \Rightarrow_G TS \Rightarrow_G T \Rightarrow_G \mathtt{a}S\mathtt{b} \Rightarrow_G \mathtt{ab}$,
2. $S \Rightarrow_G TS \Rightarrow_G \mathtt{a}S\mathtt{b}S \Rightarrow_G \mathtt{a}S\mathtt{b} \Rightarrow_G \mathtt{ab}$. ◊

In vielen Anwendungsfeldern von kontextfreien Grammatiken sind Ableitungsbäume von entscheidender Bedeutung. Bei der Definition von Progammiersprachen wird beispielsweise nicht nur die Syntax, sondern auch die Semantik (zumindest teilweise) über Ableitungsbäume definiert.

Beispiel 4.13 Sei $\Sigma := \{0, 1, +, \cdot, (,)\}$. Wir betrachten nun die folgende CFG $G := (\Sigma, V, P, S)$ mit $V := \{S, M, A\}$, wobei P genau diese Regeln enthalte:

1. $S \to 0 \mid 1 \mid M \mid A$
2. $A \to (S + S)$,
3. $M \to (S \cdot S)$.

Die Sprache $\mathcal{L}(G)$ beschreibt also arithmetische Ausdrücke über den Zahlen 0 und 1. Wir betrachten nun einen Ableitungsbaum für das Wort $((0 + 1) \cdot 1)$:



Anhand dieses Ableitungsbaumes kann der Computer den Ausdruck dann auch entsprechend auswerten. Wir werden später sehen, wie man mit einer geeigneten Grammatik sicherstellen kann, dass bei Ausdrücken ohne Klammern Präzedenzregeln wir „Punkt vor Strich“ beachtet werden. ◊

Unser XML-Dokument aus Beispiel 4.2 lässt sich auf die gleiche Art untersuchen. Dazu müssen wir aber erst eine geeignete Grammatik angeben:

Beispiel 4.14 Sei Σ die Menge aller ASCII-Zeichen[43]. Die Grammatik $G := (\Sigma, V, P, S)$ sei definiert durch

$$V := \{\mathit{Bs}, \mathit{Cocktail}, \mathit{Cocktails}, \mathit{CListe}, \mathit{Menge}, \mathit{Name}, \mathit{Text}, \mathit{Zutat}, \mathit{Zutaten}, \mathit{ZListe}\},$$

[43] Für XML können wir im Prinzip auch Unicode verwenden, für dieses Beispiel ist die Unterscheidung aber ohnehin irrelevant.

$S := \mathit{Cocktails}$,

sowie

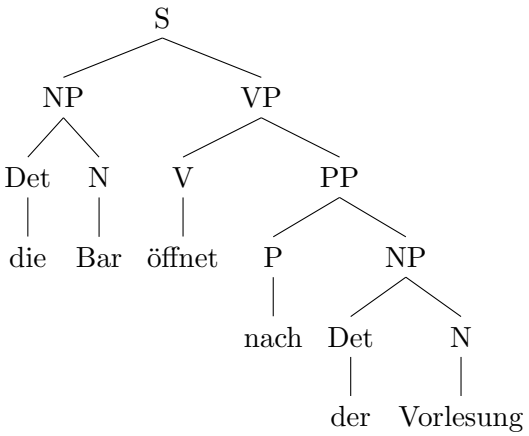
$$
\begin{aligned}
P := \Big\{ \qquad \mathit{Cocktails} &\to \texttt{<cocktails>}\mathit{CListe}\texttt{</cocktails>},\\
\mathit{CListe} &\to \mathit{Cocktail} \cdot \mathit{CListe} \mid \mathit{Cocktail},\\
\mathit{Cocktail} &\to \texttt{<cocktail>}\mathit{Name} \cdot \mathit{Zutaten}\texttt{</cocktail>},\\
\mathit{Name} &\to \texttt{<name>}\mathit{Text}\texttt{</name>},\\
\mathit{Zutaten} &\to \texttt{<zutaten>}\mathit{ZListe}\texttt{</zutaten>},\\
\mathit{ZListe} &\to \mathit{ZListe} \cdot \mathit{Zutat} \mid \mathit{Zutat},\\
\mathit{Zutat} &\to \texttt{<zutat>}\mathit{Name} \cdot \mathit{Menge}\texttt{</zutat>},\\
\mathit{Menge} &\to \texttt{<menge>}\mathit{Text}\texttt{</menge>},\\
\mathit{Text} &\to \mathit{Bs} \cdot \mathit{Text} \mid \varepsilon,\\
\mathit{Bs} &\to \texttt{0} \mid \cdots \mid \texttt{9} \mid \texttt{a} \mid \cdots \texttt{Z} \mid \texttt{␣}\Big\}.
\end{aligned}
$$

Das Zeichen ␣ steht hierbei für ein Leerzeichen. Anhand dieser CFG können Sie sich nun als Übung einen Ableitungsbaum für das XML-Dokument aus Beispiel 4.2 erstellen (ignorieren Sie dazu die Einrückungen und Zeilenumbrüche, und leiten Sie unterhalb der *Text*-Variablen nicht weiter ab).

Anhand dieses Ableitungsbaumes kann man nun die Struktur des XML-Dokuments untersuchen und dabei die Tags vollkommen ignorieren. Im Grunde geht jeder XML-Parser ähnlich vor: Aus den Tags wird eine Baumstruktur rekonstruiert. ◊

Auch in der Linguistik und Computer-Linguistik finden Ableitungsbäume Verwendung, um die Abhängigkeiten verschiedener Satzteile zu modellieren:

Beispiel 4.15 Angenommen, wir verfügen über eine CFG für die Sprache „Deutsch[44]“. Dann kann bei der Analyse eines Beispielsatzes der folgende Ableitungsbaum entstehen:



[44] Wie auch immer Sie diese genau definieren wollen. Glücklicherweise ist dies eine linguistische Fragestellung und keine der theoretischen Informatik, so dass wir uns nicht darum kümmern müssen.

Der Ableitungsbaum zeigt dann nicht nur, dass der Satz in grammatischer Satz der Sprache ist, sondern auch, welche Funktionen die einzelnen Satzteile haben. Wir werden uns dies aber nicht genauer ansehen[45]. ◊

Um uns die Arbeit mit kontextfreien Grammatiken zu erleichtern, betrachten wir eine Reihe von Abschlusseigenschaften. Die verwendeten Konstruktionen illustrieren außerdem eine Reihe von Tricks, die beim Bau von CFGs hilfreich sein können.

Satz 4.16 *Die Klasse* CFL *ist abgeschlossen unter*

1. *Vereinigung,*
2. *Konkatenation,*
3. *n-facher Konkatenation,*
4. *Kleene-Stern,*
5. *Kleene-Plus,*
6. *Homomorphismus,*
7. *regulärer Substitution,*
8. *dem Reversal-Operator.*

Beweis: In diesem Beweis wird die Korrektheit der meisten Konstruktionen nicht gezeigt. Dies sei Ihnen als Übung überlassen. Sei Σ ein beliebiges Alphabet.
Zu 1: Seien $L_1, L_2 \subseteq \Sigma^*$ kontextfreie Sprachen. Dann existieren CFGs $G_1 := (\Sigma, V_1, P_1, S_1)$ und $G_2 := (\Sigma, V_2, P_2, S_2)$ mit $\mathcal{L}(G_1) = L_1$ und $\mathcal{L}(G_2) = L_2$. Ohne Beeinträchtigung der Allgemeinheit gelte $(V_1 \cap V_2) = \emptyset$. Wir definieren nun eine CFG $G_V := (\Sigma, V, P, S)$ durch:

- $V := V_1 \cup V_2 \cup S$, wobei $S \notin (V_1 \cup V_2 \cup \Sigma)$,
- $P := P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}$.

Zu 2: Seien $L_1, L_2 \subseteq \Sigma^*$ kontextfreie Sprachen. Dann existieren CFGs $G_1 := (\Sigma, V_1, P_1, S_1)$ und $G_2 := (\Sigma, V_2, P_2, S_2)$ mit $\mathcal{L}(G_1) = L_1$ und $\mathcal{L}(G_2) = L_2$. Ohne Beeinträchtigung der Allgemeinheit gelte $(V_1 \cap V_2) = \emptyset$. Wir definieren nun eine CFG $G_K := (\Sigma, V, P, S)$ durch:

- $V := V_1 \cup V_2 \cup S$, wobei $S \notin (V_1 \cup V_2 \cup \Sigma)$,
- $P := P_1 \cup P_2 \cup \{S \to S_1 \cdot S_2\}$.

Zu 3: Folgt direkt aus dem Abschluss unter Konkatenation.
Zu 4: Sei $L \subseteq \Sigma^*$ eine kontextfreie Sprache. Dann existiert eine CFG $G := (\Sigma, V, P, S)$ mit $\mathcal{L}(G) = L$. Wir definieren nun eine CFG $G_* := (\Sigma, V_*, P_*, S_*)$ durch:

- $V := V \cup \{S_*\}$, wobei $S_* \notin (V \cup \Sigma)$,
- $P_* := P \cup \{S_* \to S_* S_* \mid \varepsilon\}$.

[45] Darüber hinaus bin ich mir nicht einmal sicher, ob dieser Baum aus linguistischer Sicht korrekt ist. Dieses Beispiel ist entstanden, indem ich einfach in einem Beispiel aus der Dokumenation des LaTeX-Pakets `tikz-qtree` die Wörter passend ersetzt habe.

Zu 5: Folgt direkt aus dem Abschluss unter Kleene-Stern und Konkatenation, da die Gleichung $L^+ = L \cdot (L^*)$ für alle Sprachen L gilt..
Zu 6: Dies folgt zwar direkt aus dem Abschluss unter regulärer Substitution, allerdings ist dieser Beweis einfacher zu verstehen. Wir zeigen daher den Abschluss unter Homomorphismus unabhängig vom allgemeineren Fall. Sei $L \subseteq \Sigma^*$ eine kontextfreie Sprache. Sei Σ_2 ein Alphabet, und sei $h : \Sigma^* \to (\Sigma_2)^*$ ein Homomorphismus. O. B. d. A. gelte $(\Sigma_2 \cap V) = \emptyset$. Wir erweitern nun h zu einem Homomorphismus $h' : (\Sigma \cup V)^* \to (\Sigma_2 \cup V)^*$ durch $h'(A) := A$ für alle $A \in V$, und $h'(a) := h(a)$ für alle $a \in \Sigma$. Wir definieren eine CFG $G_h := (\Sigma_2, V, P_h, S)$ durch

$$P_h := \{(A, h'(\beta)) \mid (A, \beta) \in P\}.$$

In jeder Regel wird also jedes vorkommende Terminal a durch $h(a)$ ersetzt.
Zu 7: Sei $L \subseteq \Sigma^*$ eine kontextfreie Sprache. Sei Σ_2 ein Alphabet, und sei $s : \Sigma^* \to (\Sigma_2)^*$ eine reguläre Substitution. Dann existiert eine CFG $G := (\Sigma, V, P, S)$ mit $\mathcal{L}(G) = L$. Außerdem ist $s(a)$ für jedes $a \in \Sigma$ eine reguläre Sprache; also existiert zu jedem $a \in \Sigma$ eine reguläre Grammatik $G_a := (\Sigma_2, V_a, P_a, S_a)$ mit $\mathcal{L}(G_a) = s(a)$. Ohne Beeinträchtigung der Allgemeinheit sind die Mengen V, Σ_2 und alle V_a paarweise disjunkt.

Wir definieren nun eine CFG $G_S := (\Sigma, V_S, P_S, S)$ wie folgt:

- $V_S := V \cup \bigcup_{a \in \Sigma} V_a$,
- $P_S := P' \cup \bigcup_{a \in \Sigma} P_a$, wobei die Menge P' wie folgt definiert ist. Der Homomorphismus $h : (V_S \cup \Sigma)^* \to V_S^*$ sei definiert durch $h(A) := A$ für alle $A \in V_S$, und $h(A) := S_a$ für alle $a \in A$. Dann ist $P' := \{(A, h(\beta)) \mid (A, \beta) \in P\}$.

Die Idee der Konstruktion ist, dass wir jedes Terminal a in einer Regel aus P durch das Startsymbol der Grammatik G_a ersetzen. Anstelle eines Wortes w kann nun aus S die Satzform $h(w)$ abgeleitet werden.
Zu 8: Übung. □

Hinweis 4.17 Bei der Konstruktion von kontextfreien Grammatiken gibt es verschiedene Ansätze, die uns die Arbeit deutlich erleichtern können. Einerseits können wir oft eine rekursive Definition direkt in die entsprechende Grammatik umbauen. Andererseits können wir sowohl endliche Automaten als auch reguläre Ausdrücke in kontextfreie Grammatiken konvertieren. Darüber hinaus können wir die bekannten Abschlusseigenschaften anhand verwenden, um kontextfreie Grammatiken modular aus einfacheren kontextfreien Grammatiken zu definieren.

4.1.1 Das Pumping-Lemma für kontextfreie Sprachen

Wie Sie vielleicht bereits vermutet haben, sind nicht alle Sprachen kontextfrei. Wir verfügen zwar bereits über eine ansehnliche Sammlung von Abschlusseigenschaften, aber wenn wir beweisen wollen, dass eine Sprache nicht kontextfrei ist, nutzen uns diese nichts, wenn wir keine Sprache haben, von der wir bereits wissen, dass sie nicht kontextfrei ist.

Dazu erweitern wir das Pumping-Lemma zu einem Pumping-Lemma für kontextfreie Sprachen. Dieses ist auch unter den Namen *Lemma von Bar-Hillel*[46] oder *uvwxy-Lemma* bekannt:

Lemma 4.18 (Pumping-Lemma für kontextfreie Sprachen) *Sei Σ ein Alphabet. Für jede kontextfreie Sprache $L \subseteq \Sigma^*$ existiert eine* ***Pumpkonstante*** *$n_L \in \mathbb{N}_{>0}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n_L$ folgende Bedingung erfüllt ist: Es existieren Wörter $u, v, w, x, y \in \Sigma^*$ mit*

1. *$uvwxy = z$,*
2. *$|vx| \geq 1$,*
3. *$|vwx| \leq n_L$, und*
4. *für alle $i \in \mathbb{N}$ ist $uv^iwx^iy \in L$.*

Beweisidee: Da L eine kontextfreie Sprache ist, existiert eine CFG G mit $\mathcal{L}(G) = L$. Wenn nun ein Wort z lang genug ist (wir werden dieses später genauer definieren), dann müssen in jedem Ableitungsbaum für z in G auf einem Pfad von der Wurzel zu einem Blatt Variablen mehrfach vorkommen. Wir wählen uns eine geeignete mehrfach vorkommende Variable A aus und betrachten die Teilbäume, die an zwei Vorkommen von A verwurzelt sind. Beim Aufpumpen ersetzen den kleineren Baum durch eine Kopie des größeren, beim Abpumpen umgekehrt. □

Beweis: Sei $L \subseteq \Sigma^*$ eine kontextfreie Sprache. Dann existiert eine CFG $G := (\Sigma, V, P, S)$ mit $\mathcal{L}(G) = L$. Sei b die größtmögliche Zahl von Symbolen, die auf einer rechten Seite einer Regel aus P auftreten, das heißt

$$b := \max\{|\beta| \mid (\alpha, \beta) \in P\}.$$

Angenommen, $b = 1$. Dann ist G eine rechtslineare Grammatik, und somit ist L eine reguläre Sprache. Dann folgt die Behauptung mit $x := y := \varepsilon$ und dem Pumping-Lemma für reguläre Sprachen. Wir können also ohne Beeinträchtigung der Allgemeinheit annehmen, dass $b \geq 2$ ist

Gemäß der Definition von b hat in jedem Ableitungsbaum für G jeder Knoten höchstens b Kinder. Somit sind höchstens b Blätter genau einen Schritt von der Wurzel entfernt, und höchstens b^2 Blätter sind genau zwei Schritte von der Wurzel entfernt. Diese Beobachtung lässt sich fortsetzen: Wenn ein Ableitungsbaum für G die Höhe h hat, kann dieser Baum höchstens b^h Blätter haben und entspricht daher einem Terminalwort mit einer Länge von höchstens b^h.

Wir definieren nun $n_L := b^{|V|+2}$. Da $b \geq 2$ wissen wir, dass $n_L > b^{|V|+1}$. Ist also $z \in L$ und $|z| \geq n_L$, muss jeder Ableitungsbaum für z in G mindestens die Höhe $|V|+2$ haben.

Nun sei $z \in L$ und $|z| \geq n_L$. Sei τ ein Ableitungsbaum für z in G. Falls z mehrere Ableitungsbäume hat, wählen wir τ so, dass die Zahl der Knoten von τ minimal ist (also hat kein anderer Ableitungsbaum für z in G weniger Knoten als τ). Wir wählen nun

[46] Benannt nach Yehoshua Bar-Hillel.

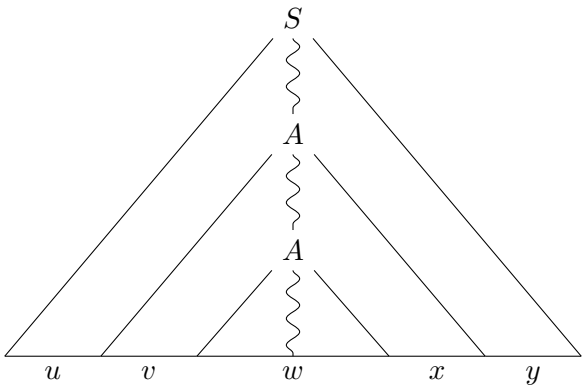


Abbildung 4.1: Eine Illustration der Definition der Teilbäume τ_1 und τ_2 im Beweis von Lemma 4.18. Die gekurvte Linie stellt den gewählten längsten Pfad von der Wurzel zu einem Blatt dar. Oben sehen sie das Startsymbol S, darunter das zweite gewählte Vorkommen von A (die Wurzel von τ_2), darunter wiederum das erste gewählte Vorkommen von A (die Wurzel von τ_1). Der Teilbaum τ_1 erzeugt das Wort w, der Teilbaum τ_2 das Wort vwx, und der gesamte Baum τ das Wort $uvwxy = z$.

einen längsten Pfad von der Wurzel von τ zu einem Blatt. Da $|z| \geq n_L$ hat τ eine Höhe von mindestens $|V| + 2$; also hat der gewählte Pfad eine Länge von mindestens $|V| + 2$. Da alle Knoten auf diesem Pfad mit Variablen beschriftet sind (Terminale können nur an den Blättern auftreten), sind auf diesem Pfad mindestens $|V| + 1$ Variablen anzutreffen. Also kommt mindestens eine Variable $A \in V$ auf diesem Pfad doppelt vor. Um unsere Argumentation später zu vereinfachen, wählen wir A so, dass A innerhalb der unteren $|V| + 1$ Knoten des Pfades mindestens zweimal vorkommt.

Wir definieren nun im Ableitungsbaum τ Teilbäume τ_1 und τ_2 auf die folgende Art: Wir wählen zuerst dasjenige Vorkommen von A auf dem Pfad, das am nächsten an dem Blatt ist. Den Teilbaum von τ, der dieses Vorkommen von A als Wurzel hat, bezeichnen wir mit τ_1. Dann betrachten wir das nächste Vorkommen von A an diesem Pfad und bezeichnen den daran verwurzelten Teilbaum mit τ_2. Also ist τ_1 ein Teilbaum von τ_2. Eine Illustration dieser Definition finden Sie in Abbildung 4.1. Wir definieren nun eine Zerlegung von $z = uvwxy$ folgendermaßen:

- Das Terminalwort, das an den Blättern von τ_1 steht, bezeichnen wir mit w.
- Im Teilbaum τ_2 kommt τ_1 als Teilbaum vor. Also existieren Wörter $v, x \in \Sigma^*$, so dass τ_2 an seinen Blättern mit vwx beschriftet ist.
- Da τ den Baum τ_2 als Teilbaum enthält, existieren Wörter $u, y \in \Sigma^*$ mit $uvwxy = z$.

Auch dieser Zusammenhang wird in Abbildung 4.1 illustriert. Wir sind nun bereit, die vier Bedingungen dieses Lemmas zu überprüfen.

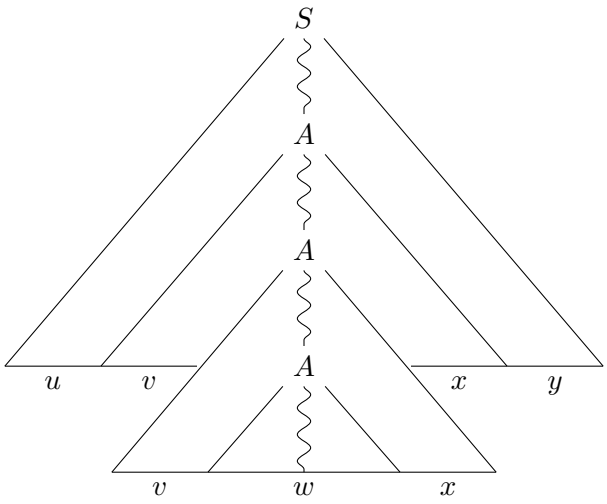


Abbildung 4.2: Eine Illustration des Aufpumpens im Beweis von Lemma 4.18 im Fall $i = 2$. Der Teilbaum τ_1 wurde durch eine Kopie des Teilbaums τ_2 ersetzt.

Zu 1: Es gilt $z = uvwxy$ gemäß unserer Wahl von u, v, w, x und y.
Zu 4: Wir zeigen zuerst, dass $uv^iwx^iy \in \mathcal{L}(G)$ für alle $i \geq 2$ gilt. Da τ_1 und τ_2 die selbe Wurzel haben (nämlich A) können wir den Teilbaum τ_1 durch eine Kopie von τ_2 ersetzen (siehe Abbildung 4.2). Wenn wir diesen Schritt $i-1$ mal durchführen, ersetzen wir die Teilwörter v und x durch v^i und x^i und erhalten so insgesamt uv^iwx^iy. Für den Fall $i = 0$ können wir analog verfahren, indem wir den Teilbaum τ_2 durch τ_1 ersetzen (siehe Abbildung 4.3). Dadurch entfernen wir die Teilwörter v und x und erhalten $uwy = uv^0wx^0y$. Etwas formaler: Gemäß unserer Wahl von u, v, w, x, y wissen wir von der Existenz von Ableitungen mit

1. $S \Rightarrow_G^* uAy$ (dies entspricht τ, ohne τ_1 oder τ_2 abzuleiten),
2. $A \Rightarrow_G^* vAx$ (dies entspricht τ_2, aber ohne τ_1 abzuleiten) und
3. $A \Rightarrow_G^* w$ (dies entspricht τ_1).

Also existiert auch für jedes $i \geq 2$ eine Ableitung mit $A \Rightarrow_G^* v^iAx^i \Rightarrow_G^* v^iwx^i$, und somit ist $S \Rightarrow_G^* uAy \Rightarrow_G^* uv^iwx^iy$ Analog gilt für den Fall $i = 0$, dass $S \Rightarrow_G^* uAy \Rightarrow_G^* uwy = uv^0wx^0y$. Somit ist $uv^iwx^iy \in L$ für alle $i \in \mathbb{N}$.
Zu 2: Um zu zeigen, dass $|vx| \geq 1$, nehmen wir das Gegenteil an. Sei also $|vx| = 0$, dann ist $v = x = \varepsilon$. Also können wir wie beim Abpumpen (siehe Abbildung 4.3) den Teilbaum τ_2 durch τ_1 ersetzen und erhalten so einen Ableitungsbaum für das Wort $uwy = uvwxy = z$ (da $v = x = \varepsilon$) der weniger Knoten hat als τ. Da wir τ so gewählt haben, dass kein anderer Ableitungsbaum für z weniger Knoten hat als τ, ist dies ein Widerspruch. Also ist $v \neq \varepsilon$ oder[47] $x \neq \varepsilon$ und $|vx| \geq 1$.

[47] Der Fall, dass entweder $v = \varepsilon$ oder $x = \varepsilon$ ist, kann allerdings eintreten.

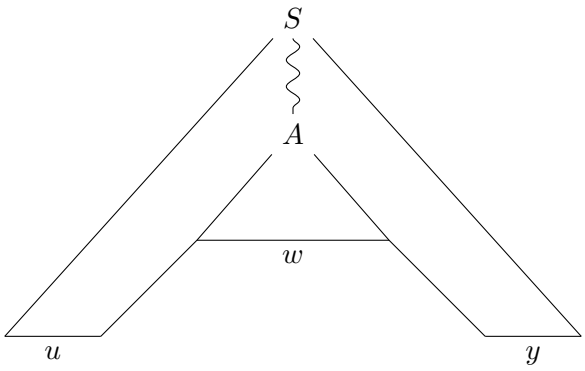


Abbildung 4.3: Eine Illustration des Abumpens im Beweis von Lemma 4.18, also des Falls $i = 0$. Der Teilbaum τ_2 wurde durch den Teilbaum τ_1 ersetzt.

Zu 3: Zu zeigen ist $|vwx| \leq n_L$. Wir hatten die beiden Vorkommen von A so gewählt, dass diese unter den letzten (untersten) $|V| + 1$ Variablen auf dem Pfad von der Wurzel von τ zu dem Blatt liegen. Also hat τ_2 eine Höhe von maximal $|V| + 2$. Somit hat (gemäß unserer Wahl von b und unserer zu Beginn des Beweises getroffenen Überlegungen) das von τ_2 erzeugte Terminalwort vwx höchstens eine Länge von $b^{|V|+2} = n_L$. □

Nun können wir anfangen, eine kleine Sammlung von Sprachen zu erstellen, die nicht kontextfrei sind.

Beispiel 4.19 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $L := \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \in \mathbb{N}\}$. Angenommen, $L \in$ CFL. Sei n_L die Pumpkonstante von L. Wir wählen nun das Wort $z := \mathtt{a}^{n_L}\mathtt{b}^{n_L}\mathtt{c}^{n_L}$. Es gilt $|z| = 3n_L > n_L$, und offensichtlich gilt auch $z \in L$. Sei nun u, v, w, x, y eine Zerlegung von z, die das Pumping-Lemma erfüllt.

Nun muss $v, x \in \mathcal{L}(\mathtt{a}^* \mid \mathtt{b}^* \mid \mathtt{c}^*)$ gelten. Sobald v oder x zwei unterschiedliche Terminale enthält, ist uv^2wx^2y nicht in der Sprache $\mathcal{L}(\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*)$ enthalten und somit auch nicht in L.

Da $v, x \in \mathcal{L}(\mathtt{a}^* \mid \mathtt{b}^* \mid \mathtt{c}^*)$ gilt, existiert ein Buchstabe $d \in \Sigma$, der niemals gepumpt werden kann. Es gilt also $|uv^iwx^iy|_d = |z|_d$ für alle $i \in \mathbb{N}$. Da aber $|vx| > 1$ existiert ein $e \in \Sigma$, das gepumpt werden kann. Es gilt also $|uv^iwx^iy|_e \neq |z|_e$ für alle $i \neq 1$. Insbesondere gilt $|uv^iwx^iy|_d \neq |uv^iwx^iy|_e$ für $i \neq 1$, und somit $uv^iwx^iy \notin L$. Widerspruch. ◊

Beispiel 4.20 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{\mathtt{a}^i\mathtt{b}^j\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N}\}$. Angenommen, $L \in$ CFL. Sei n_L die Pumpkonstante von L. Wir wählen nun das Wort $z := \mathtt{a}^{n_L}\mathtt{b}^{n_L}\mathtt{a}^{n_L}\mathtt{b}^{n_L}$. Das Wort z besteht also aus vier *Blöcken* der Länge n_L, wobei jeder Block komplett aus as oder bs besteht.

Offensichtlich gilt $|z| > n_L$, also existiert eine Zerlegung von z in Wörter u, v, w, x, y, die das Pumping-Lemma erfüllt. Angenommen, v oder x enthält sowohl Vorkommen von a, als auch von b. Dann erhalten wir durch Aufpumpen ein Wort uv^2wx^2y, das *nicht* in der Sprache $\mathcal{L}(\mathtt{a}^*\mathtt{b}^*\mathtt{a}^+\mathtt{b}^*) \supset L$ ist, also auch nicht in L. Sowohl v als auch x liegt daher komplett in einem der vier Blöcke (die jeweils ausschließlich aus as oder bs bestehen).

Da $|vwx| \leq n_L$ müssen v und x in dem selben oder in zwei direkt aufeinander folgenden

Blöcken liegen. In beiden Fällen erzeugt Aufpumpen ein Wort, in dem ein Block von as oder bs mehr Symbole enthält als der andere Block mit dem gleichen Buchstaben. In jedem Fall erhalten wir also $uv^2wx^2y \notin L$. Somit kann L nicht kontextfrei sein. $\Diamond$

Hinweis 4.21 Hinweis 3.25 zur korrekten Anwendung von Abschlusseigenschaften bei regulären Sprachen gilt natürlich ebenfalls für kontextfreie Sprachen.

Leider sind die kontextfreien Sprachen nicht so handhabbar wie die regulären Sprachen. Ein Beispiel dafür ist das Fehlen einiger Abschlusseigenschaften:

Lemma 4.22 *Die Klasse* CFL *ist nicht abgeschlossen unter Schnitt, Komplementierung und Mengendifferenz.*

Beweis: Wir zeigen zuerst den Nicht-Abschluss unter Schnitt, dann unter Komplementierung und Mengendifferenz.
Schnitt: Angenommen, CFL ist unter Schnitt abgeschlossen. Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Wir definieren nun die Sprachen $L_1, L_2 \subseteq \Sigma^*$ durch

$$L_1 = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^j \mid i, j \in \mathbb{N}\},$$
$$L_2 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^j \mid i, j \in \mathbb{N}\}.$$

Bei Sprachen sind kontextfrei. Wir können dies entweder zeigen, indem wir direkt jeweils eine CFG angeben, oder indem wir die beiden Sprachen aus der Sprache $\{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$ (kontextfrei laut Beweis zu Satz 4.6) und der Sprache $\{\mathtt{a}\}^*$ unter Verwendung des Abschlusses unter Konkatenation und Homomorphismus zusammenbauen (wir können auch L_2 durch Anwendung des Reversal-Operators und eines Homomorphismus aus L_1 erhalten). Nun gilt:

$$L_1 \cap L_2 = \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \in \mathbb{N}\}.$$

Aus Beispiel 4.19 wissen wir, dass diese Sprache nicht kontextfrei ist. Widerspruch.
Komplementierung: Angenommen, CFL ist unter Komplementierung abgeschlossen. Für alle Sprachen L_1, L_2 gilt:

$$\begin{aligned}\overline{\overline{L_1} \cup \overline{L_2}} &= \overline{\overline{L_1}} \cap \overline{\overline{L_2}} \\ &= L_1 \cap L_2.\end{aligned}$$

Da CFL unter Vereinigung abgeschlossen ist, wäre die Klasse dann auch unter Schnitt abgeschlossen. Dass dies nicht gilt, haben wir gerade erst gezeigt. Widerspruch.
Mengendifferenz: Da Σ^* eine kontextfreie Sprache ist, und da $\overline{L} = \Sigma^* - L$, folgt dies direkt aus dem Nicht-Abschluss unter Komplementierung. $\square$

Diese schlechte Nachricht lässt sich ein wenig abschwächen. Wir werden bald zeigen, dass die Klasse CFL immerhin unter Schnitt mit regulären Sprachen abgeschlossen ist (das heißt, dass für jede kontextfreie Sprache $L \in$ CFL und jede reguläre Sprache $R \in$ REG gilt, dass $(L \cap R) \in$ REG). Dadurch werden wir später einige der Beweistechniken,

die den Abschluss unter Schnitt verwenden, auch auf kontextfreie Sprachen anwenden können.

Wir betrachten nun eine weitere Anwendung von Lemma 4.18. In Satz 4.6 haben wir gesehen, dass REG $\subset$ CFL. Allerdings haben wir dies nicht für alle Alphabete bewiesen, denn unser Beispiel für eine kontextfreie Sprache, die nicht regulär ist, benötigt mindestens zwei unterschiedliche Terminalbuchstaben. Wir können beweisen, dass *jedes* solche Beispiel mindestens zwei Terminalbuchstaben benötigt:

Satz 4.23 *Sei Σ ein Alphabet mit $|\Sigma| = 1$. Dann ist $\mathrm{REG}_\Sigma = \mathrm{CFL}_\Sigma$.*

Beweis: Sei $\Sigma := \{a\}$. Da $\mathrm{REG}_\Sigma \subseteq \mathrm{CFL}_\Sigma$ gemäß Definition gilt, müssen wir nur beweisen, dass $\mathrm{CFL}_\Sigma \subseteq \mathrm{REG}_\Sigma$. Sei also $L \subseteq \Sigma^*$ eine kontextfreie Sprache. Dann existiert eine Pumpkonstante n_L, für die L das Pumping-Lemma für kontextfreie Sprachen erfüllt. Der Lesbarkeit halber sei $n := n_L$. Bevor wir den eigentlich Beweis beginnen, zeigen wir die folgende Behauptung:

Behauptung 1 *Für alle $z \in L$ mit $|z| > n$ gilt $\{z\} \cdot \left\{a^{n!}\right\}^* \subseteq L$.*

Beweis: Da $z \in L$ und $|z| > n$ existiert eine Zerlegung von z in Wörter u, v, w, x, y, die das Pumping-Lemma für kontextfreie Sprachen erfüllt. Also gilt $uv^iwx^iy \in L$ für alle $i \geq 0$. Da Σ nur einen Buchstaben enthält, können wir die aufgepumpten Wörter umsortieren und erhalten so $uvwxyv^ix^i = z(vx)^i \in L$ für alle $i \geq 0$. Mit anderen Worten: Es gilt $z \cdot \{vx\}^* \subseteq L$. Sei nun $k := \frac{n!}{|vx|}$. Da $|vx| > 0$ und da $|vx| \leq |vwx| \leq n$ ist diese Definition erlaubt (insbesondere wird $n!$ von $|vx|$ geteilt). Außerdem ist

$$(vx)^k = \left(a^{|vx|}\right)^k = a^{|vx|k} = a^{\frac{|vx|n!}{|vx|}} = a^{n!},$$

und somit ist $\{a^{n!}\}^* \subseteq \{vx\}^*$. Hieraus folgt unmittelbar $\{z\} \cdot \{a^{n!}\}^* \subseteq z \cdot \{vx\}^*$ und, wegen $z \cdot \{vx\}^* \subseteq L$, auch $\{z\} \cdot \{a^{n!}\}^* \subseteq L$. □(Behauptung 1)

Wir betrachten nun die Sprachen

$$L_\leq := \{z \in L \mid |z| \leq n\},$$
$$L_> := \{z \in L \mid |z| > n\}.$$

Offensichtlich ist $L = L_\leq \cup L_>$. Die Sprache $L_\leq$ ist endlich und somit regulär. Wir müssen also nur noch zeigen, dass $L_>$ ebenfalls regulär ist. Dazu partitionieren wir $L_>$ in disjunkte Mengen M_j mit $0 \leq j < n!$, die definiert sind durch

$$\begin{aligned} M_j &:= L_> \cap \left(\left\{a^j\right\} \cdot \left\{a^{n!}\right\}^*\right) \\ &= \{z \in L \mid |z| > n, |z| \bmod n! = j\}. \end{aligned}$$

Die Menge M_j enthält also die Wörter aus L, die mindestens eine Länge von $n+1$ haben, und deren Länge bei Teilung durch $n!$ einen Rest von j ergibt. Für alle $0 \leq j < n!$ mit $M_j \neq \emptyset$ sei w_j das kürzeste Wort von M_j (da $|\Sigma| = 1$ ist jedes dieser w_j eindeutig definiert). Für jedes $0 \leq j < n!$ sei

$$L_j := \begin{cases} \emptyset & \text{falls } M_j = \emptyset, \\ \{w_j\} \cdot \{a^{n!}\}^* & \text{falls } M_j \neq \emptyset. \end{cases}$$

Offensichtlich ist jede dieser Sprachen L_j regulär[48]. Wir behaupten nun:

Behauptung 2

$$L_> = \bigcup_{0 \leq j < n!} L_j.$$

Beweis: Wir zeigen die Äquivalenz der beiden Sprachen durch doppelte Inklusion.

„$\supseteq$“: Für alle $0 \leq j < n!$ folgt $L_> \supseteq L_j$ direkt aus Behauptung 1, da $|w_j| > n$ gemäß Definition gilt.

„$\subseteq$“: Wir zeigen zuerst, dass $M_j \subseteq L_j$ für alle $0 \leq j < n!$ gilt. Falls $M_j = \emptyset$, so gilt $L_j = \emptyset = M_j$. Sei also $M_j \neq \emptyset$. Dann gilt (gemäß der Definition von M_j) $|v| \bmod n! = j$ für alle $v \in M_j$. Da w_j definitionsgemäß das kürzeste Wort aus M_j ist, existiert ein $k \geq 0$ mit $|v| = |w_j| + kn!$. Also ist $v = w_j \cdot (a^{n!})^k$, und somit $v \in \{w_j\} \cdot \{a^{n!}\}^* = L_j$. Daher ist $M_j \subseteq L_j$ für alle $0 \leq j < n!$, und wir folgern

$$\bigcup_{0 \leq j < n!} L_j \supseteq \bigcup_{0 \leq j < n!} M_j = L_>.$$

Auch diese Inklusion ist also erfüllt. □(Behauptung 2)

Gemäß Behauptung 2 ist $L_>$ eine endliche Vereinigung von regulären Sprachen und somit selbst regulär. Wie wir bereits festgestellt haben, ist $L_\leq$ endlich und somit ebenfalls regulär. Wegen $L = (L_\leq \cup L_>)$ ist auch $L \in \mathrm{REG}_\Sigma$. Da $L \in \mathrm{CFL}_\Sigma$ frei gewählt wurde, gilt $\mathrm{CFL}_\Sigma \subseteq \mathrm{REG}_\Sigma$ und somit auch $\mathrm{CFL}_\Sigma = \mathrm{REG}_\Sigma$. □

Dank Satz 4.23 können wir nun einfach feststellen, dass Sprachen wie $\{a^{n^2} \mid n \in \mathbb{N}\}$ nicht nur nicht regulär sind, sondern auch nicht kontextfrei. Außerdem können wir einige Beweise für Nicht-Kontextfreiheit vereinfachen, indem wir diese auf Nicht-Regularität reduzieren und ausnutzen, dass wir bei REG über einen größeren Werkzeugkasten verfügen:

Beispiel 4.24 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei

$$L_1 := \{\mathtt{a}^p\mathtt{b} \mid p \text{ ist eine Primzahl}\},$$
$$L_2 := \{\mathtt{a}^n\mathtt{b} \mid n \text{ ist keine Primzahl}\}.$$

Beide Sprachen sind zwar über einem zweielementigen Alphabet definiert, aber anhand des Homomorphimus $h : \Sigma^* \to \{\mathtt{a}\}^*$, der definiert ist durch $h(\mathtt{a}) := \mathtt{a}$, $h(\mathtt{b}) := \varepsilon$ können wir sie in folgende Sprachen überführen:

$$h(L_1) := \{\mathtt{a}^p \mid p \text{ ist eine Primzahl}\},$$
$$h(L_2) := \{\mathtt{a}^n \mid n \text{ ist keine Primzahl}\}.$$

Wir wissen bereits, dass $h(L_1)$ nicht regulär ist (siehe zum Beispiel Schweikardt [16], oder Sie zeigen dies einfach schnell selbst mit dem Pumping-Lemma für reguläre Sprachen). Da REG unter Komplementierung abgeschlossen ist, kann auch $h(L_2)$ nicht regulär sein.

Angenommen, L_1 (oder L_2) ist kontextfrei. Dann ist auch $h(L_1)$ (bzw. $h(L_2)$) kontextfrei, und somit auch (nach Satz 4.23) regulär. Widerspruch. ◊

[48] Nicht vergessen: n ist fest.

Kontexfreie Sprachen auf unären Terminalalphabeten haben also keine größere Ausdruckskraft als die Modelle für reguläre Sprachen, die wir bisher kennengelernt haben. Allerdings können Sie diese Sprachen manchmal deutlich kompakter beschreiben:

Beispiel 4.25 Sei $\Sigma := \{\mathtt{a}\}$ und sei $n \in \mathbb{N}_{>0}$. Die CFG $G_n := (\Sigma, V_n, P_n, A_n)$ sei definiert durch $V := \{A_1, \ldots, A_n\}$ und

$$\begin{aligned} P_n := &\{A_i \to A_{i-1}A_{i-1} \mid 2 < i \leq n\} \\ &\cup \{A_1 \to \mathtt{aa}\}. \end{aligned}$$

Dann enthält die Sprache $\mathcal{L}(G_n)$ genau ein Wort, nämlich $\mathtt{a}^{2^n}$. Während G_n nur n Variablen und n Regeln (mit einer rechten Seite der Länge 2) benötigt, um diese Sprache zu erzeugen, benötigt jeder DFA (und jeder NFA) für $\mathcal{L}(G_n)$ mindestens 2^n Zustände (und jede reguläre Grammatik hat ebenfalls eine exponentielle Größe). ◊

Ähnlich wie beim Pumping-Lemma für reguläre Sprachen lassen sich nicht alle Beweise für die Nicht-Kontextfreiheit einer Sprache mit Lemma 4.18 durchführen, und auch die Verwendung von Abschlusseigenschaften hilft nicht weiter. Ein Beispiel für solch einen schweren Fall ist die folgende Sprache:

Beispiel 4.26 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und

$$L := \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j, k \in \mathbb{N}, i \neq j, i \neq k, j \neq k\}.$$

Die Sprache L ist nicht kontextfrei, allerdings ist es meiner Meinung nach nicht möglich, dies mit unseren Mitteln zu beweisen. Falls Sie einen Beweis finden, dass diese Sprache L nicht kontextfrei ist, der nur Lemma 4.18 und die Abschlusseigenschaften der Klasse CFL verwendet, wäre ich daran sehr interessiert. ◊

Um solche Sprachen zu bezwingen greift man gewöhnlich auf eine Verallgemeinerung des Pumping-Lemmas für kontextfreie Sprachen zurück, die als *Ogden's Lemma* bekannt ist[49]. Wir werden uns nicht damit befassen, weitere Informationen finden Sie zum Beispiel in Hopcroft und Ullman [8], Shallit [19] oder Schnitger [14]. Eine weitere Verallgemeinerung von Ogden's Lemma finden Sie am Ende von Abschnitt 5.5 in Kudlek [10]. Eine andere Alternative zum Pumping-Lemma stellt das *Interchange Lemma* dar, das zum Beispiel in [19] beschrieben wird. Aber auch diese Lemmata haben ihre Grenzen. Selbst bei manchen einfach zu definierenden Sprachen ist immer noch unbekannt, ob diese kontextfrei sind oder nicht. Ein Beispiel dafür ist die folgende Sprache:

Beispiel 4.27 Sei Σ ein Alphabet. Ein Wort $w \in \Sigma^+$ heißt **primitiv**, wenn für alle $u \in \Sigma^+$ und alle $n \in \mathbb{N}$ aus $u^n = w$ stets $u = w$ und $n = 1$ folgt. Zum Beispiel ist das Wort $\mathtt{aba}$ primitiv, das Wort $\mathtt{abab}$ aber nicht, denn $\mathtt{abab} = (\mathtt{ab})^2$. Wir definieren nun die Sprache

$$\mathrm{PRIM}_\Sigma := \{w \in \Sigma^+ \mid w \text{ ist primitiv}\}.$$

Es wird vermutet, dass PRIM_Σ für alle Alphabet mit $|\Sigma| \geq 2$ nicht kontextfrei ist. Allerdings war bisher noch niemand in der Lage, dies auch wirklich zu beweisen. Diese

[49] Benannt nach William F. Ogden.

Frage hat sich als ein so schweres Problem herausgestellt, dass sich ein ganzes Buch mit Lösungsansätzen beschäftigt, nämlich Dömösi und Ito [5]. ◊

Fragestellungen wie diese werden natürlich nicht unbedingt wegen ihrer unmittelbaren praktischen Relevanz untersucht. Stattdessen ist es eher eine Frage des Prinzips: Auf der einen Seite haben wir einen großen Werkzeugkasten an Methoden, um Sprachen auf Kontextfreiheit zu untersuchen. Auf der anderen Seite haben wir eine vergleichsweise einfach zu definierende Sprache wie PRIM_Σ, die all diesen Werkzeugen trotzt. Dies kann man als ein Zeichen verstehen, dass der vorhandene Werkzeugkasten noch nicht groß genug ist, und dass vielleicht eine komplett neue Idee oder Technik notwendig ist, um diese Lücke zu schließen.

4.1.2 Die Chomsky-Normalform

In vielen Fällen kann es hilfreich sein, einen Beweis nicht für *alle* kontextfreien Grammatiken durchführen zu müssen, sondern nur für Grammatiken, in denen die Regeln eine bestimmte Form haben. Ein Beispiel dafür ist die sogenannte Chomsky-Normalform[50], die wir ausgiebig nutzen werden:

Definition 4.28 Eine kontextfreie Grammatik $G := (\Sigma, V, P, S)$ ist in **Chomsky-Normalform (CNF)**, wenn jede Regel aus P eine der folgenden Formen hat:

$$\begin{aligned} &A \to BC && \text{mit } A \in V,\ B, C \in (V - \{S\}), \\ &A \to a && \text{mit } A \in V,\ a \in \Sigma, \\ &S \to \varepsilon. \end{aligned}$$

In einer CFG in Chomsky-Normalform werden also Variablen auf einzelne Terminale oder auf genau zwei Variablen abgebildet. Außerdem darf das Startsymbol bei keiner Regel auf der rechten Seite vorkommen, und außer dem Startsymbol kann keine Variable zu ε abgeleitet werden.

Beispiel 4.29 Sei $V := \{S, A, B, C\}$ und $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Sei $G := \{\Sigma, V, P, S\}$ eine CFG. Jede der folgenden Regeln führt dazu, dass G *nicht* in Chomsky-Normalform ist:

1. $A \to BS$ (denn S darf nicht auf einer rechten Seite vorkommen),
2. $B \to \mathtt{a}C$ (hier enthält rechte Seite sowohl ein Terminal als auch eine Variable),
3. $B \to \mathtt{aa}$ (die rechte Seiten enthält mehr als ein Terminal),
4. $B \to \varepsilon$ (eine andere Variable als S wird auf ε abgebildet),
5. $S \to A$ (die rechte Seite enthält zu wenige Variablen),
6. $S \to AAA$ (die rechte Seite enthält zu viele Variablen). ◊

[50] Benannt nach Noam Chomsky.

Wie wir sehen werden, kann jede kontextfreie Grammatik zu einer Grammatik umgeformt werden, die in Chomsky-Normalform ist und die gleiche Sprache erzeugt. Wir verwenden dazu den folgenden Algorithmus[51]:

Algorithmus 4 (`CNF`) Berechnet aus einer CFG eine CFG in CNF.

Eingabe: Eine CFG $G := (\Sigma, V, P, S)$.
Ausgabe: Eine CFG G_C in Chomsky-Normalform mit $\mathcal{L}(G_C) = \mathcal{L}(G)$.

1. Führe ein neues Startsymbol S_0 ein.
2. Ersetze alle ε-Regeln.
3. Ersetze alle Ein-Variablen-Regeln.
4. Zerlege alle rechten Seiten, die die falsche Form haben.
5. Gib Grammatik als G_C aus.

Die einzelnen Schritte von `CNF` sind dabei wie folgt definiert:

Schritt 1 (neues Startsymbol): Wir verwenden eine neue Variable S_0 als Startsymbol. Außerdem fügen wir S_0 zu V sowie zu P die Regel $S_0 \to S$ hinzu. Dadurch ist sicher gestellt, dass das Startsymbol S_0 auf keiner rechten Seite einer Regel von P erscheint.

Schritt 2 (Ersetzen der ε-Regeln): In diesem Schritt sollen Regeln der Form $A \to \varepsilon$ entfernt werden[52]. Wir entfernen dazu aus P eine Regel $A \to \varepsilon$, wobei $A \in V$ und $A \neq S_0$ (existiert keine solche Regeln, ist dieser Schritt fertig). Dazu schreiben wir jede Regel $B \to \beta$ aus P (mit $\beta \in (\Sigma \cup V)^*$), in der A auf der rechten Seite (also β) vorkommt, in mehrere Regeln $B \to \beta_1 \mid \cdots \mid \beta_n$ ($n \geq 1$) um, indem wir für *jedes Vorkommen* von A in β jeweils zwischen den Möglichkeiten „Löschen“ und „nicht löschen“ wählen. Eine Regel $B \to \gamma_1 A \gamma_2 A \gamma_3$ (wobei $\gamma_1, \ldots, \gamma_3$ die Variable A nicht enthalten) wird also umgeschrieben zu

$$B \to \gamma_1 A \gamma_2 A \gamma_3 \mid \gamma_1 \gamma_2 A \gamma_3 \mid \gamma_1 A \gamma_2 \gamma_3 \mid \gamma_1 \gamma_2 \gamma_3.$$

Kommt A also insgesamt n mal in β vor, können wir bis zu 2^n verschiedene rechte Seiten erhalten Dabei dürfen wir aber gleiche rechte Seiten zusammenfassen. Wird zum Beispiel das Symbol B aus der Regel $A \to BB$ entfernt, so erhalten wir $A \to B \mid \varepsilon$ (also nur zwei verschiedene rechte Seiten, anstelle von 2^2 möglichen). *Wichtig:* Entsteht auf

[51] Dieser Algorithmus ist nicht optimal, weder in Bezug auf die Laufzeit, noch auf die Größe der berechneten Grammatik in CNF. Genau genommen ist er nicht einmal effizient, da er zu einem exponentiellen Größenzuwachs führen kann. Einen weitaus effizienteren Ansatz finden Sie z. B. in Wegener [22].

[52] Mit Ausnahme von $S_0 \to \varepsilon$. Diese Regel ist zwar zu Beginn des Algorithmus nicht in P enthalten, kann aber in diesem Schritt eingeführt werden.

diese Art eine ε-Regel, die wir bereits entfernt haben, so wird diese nicht wieder in P aufgenommen.

Dieser Schritt wird wiederholt, bis (außer möglicherweise dem Startsymbol S_0) keine Regel $A \to \varepsilon$ mehr in P vorhanden ist.

Schritt 3 (Ersetzen der Ein-Variablen-Regeln): In diesem Schritt sollen Regeln der Form $A \to B$ entfernt werden. Wir entfernen dazu eine Regel der Form $A \to B$ aus P (mit $A, B \in V$). Existiert keine solche Regeln, ist dieser Schritt fertig. Für jede Regel der Form $B \to \beta$ in P (mit $\beta \in (\Sigma \cup V)^*$) fügen wir nun eine Regel $A \to \beta$ zu P hinzu; *aber nur* wenn diese Regel $A \to \beta$ *nicht* eine bereits entfernte Regel ist. Wir wiederholen diesen Schritt, bis keine Regel der Form $A \to B$ mehr vorhanden ist.

Schritt 4 (Zerlegen der rechten Seiten): Wir betrachten nun jede Regel $A \to \beta$ (mit $A \in V$, $\beta \in (\Sigma \cup V)^+$). Da $\beta \in (\Sigma \cup V)^+$ existieren ein $n \in \mathbb{N}_{>0}$ und $\beta_1, \ldots, \beta_n \in (\Sigma \cup V)$ mit $\beta = \beta_1 \cdots \beta_n$. Wir behandeln diese Regeln wie folgt:

1. Ist $n \geq 2$, so ersetzen wir die Regel $A \to \beta$ durch Regeln
$$\begin{aligned} A &\to \beta_1 A_1, \\ A_1 &\to \beta_2 A_2, \\ &\vdots \\ A_{n-2} &\to \beta_{n-1}\beta_n, \end{aligned}$$
wobei $A_1, \ldots, A_{n-2}$ neue Variablen sind.
2. Ist $n = 2$ so ersetzen wir jedes Terminal a in β durch eine neue Variable N_a und fügen die Regel $N_a \to a$ zu P hinzu.
3. Ist $n = 1$ muss $\beta = \beta_1 \in \Sigma$ gelten. Diese Regel wird nicht verändert.

(Diese Ersetzungsvorschriften werden auch auf die neuen Regeln angewendet.)

Hinweis 4.30 Beim Ausführen von `CNF` dürfen Sie mit Geschick und Augenmaß vorgehen. Zum Beispiel können sie in jedem Schritt Variablen, die nicht aus dem Startsymbol abgeleitet werden können, aus V entfernen (und die entsprechenden Regeln aus P). Ebenso können Sie mit Variablen verfahren, die nicht zu Wörtern aus Σ^* abgeleitet werden können. In jedem Schritt können Sie Regeln der Form $A \to A$ sofort entfernen. Außerdem können Sie die Variablen und die Regeln in Schritt 4 wieder verwenden. Streng nach Wortlaut müssten Sie zum Beispiel die Regel $A \to \texttt{bbbb}$ eigentlich zuerst in $A \to \texttt{a}A_1$, $A_1 \to \texttt{a}A_2$ und $A_2 \to \texttt{aa}$ umwandeln, und dann für jedes der vier Vorkommen von `a` eine eigene Variable $N_{\texttt{a}}$ mit Regel $N_{\texttt{a}} \to \texttt{a}$ einführen. Stattdessen dürfen (und sollten) Sie optimieren und folgende Regeln verwenden:

$$A \to A_1 A_1,\ A_1 \to N_{\texttt{a}} N_{\texttt{a}},\ N_{\texttt{a}} \to \texttt{a}.$$

Dadurch sparen Sie Zeit, Schreiberei und Nerven.

Beispiel 4.31 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $G := (\Sigma, V, P, S)$ eine CNF mit $V := \{A, S\}$, und P enthalte die folgenden Regeln:

$$S \to AS \mid A, \qquad A \to \mathtt{a}A\mathtt{b}A \mid \varepsilon.$$

Wie leicht zu sehen ist, ist G nicht in CNF. Also rufen wir den Algorithmus `CNF` auf.

Schritt 1 (neues Startsymbol): Wir fügen zu V das Symbol S_0 hinzu, ernennen dieses zu unserem neuen Startsymbol und nehmen zu P die Regel $S_0 \to S$ auf. Die Menge P enthält nun die folgenden Regeln:

$$S_0 \to S, \qquad S \to AS \mid A, \qquad A \to \mathtt{a}A\mathtt{b}A \mid \varepsilon.$$

Ansonsten ist in diesem Schritt nichts zu tun.

Schritt 2 (Ersetzen der ε-Regeln): Derzeit enthält P genau eine ε-Regel, nämlich $A \to \varepsilon$. Wir entfernen diese. Dazu müssen wir alle Regeln betrachten, bei denen A auf der rechten Seite vorkommt, also $S \to AS$, $S \to A$ und $A \to \mathtt{a}A\mathtt{b}A$. Diese Regeln können wir wie folgt umschreiben:

- $S \to AS$ wird zu $S \to AS \mid S$, da es nur eine Möglichkeit gibt, A zu entfernen. Die so neu entstehende Regel $S \to S$ ist eigentlich überflüssig (und wird später garantiert entfernt), wir könnten sie also gleich wieder löschen. Aus didaktischen Gründen löschen wir sie jetzt noch nicht.
- $S \to A$ wird zu $S \to A \mid \varepsilon$. Auch hier gibt es nur eine Möglichkeit, und diese Regel $S \to \varepsilon$ ist neu und wird zu P hinzugefügt.
- $A \to \mathtt{a}A\mathtt{b}A$ wird zu $A \to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}$. Da A zweimal vorkommt, gibt es vier mögliche Kombinationen, und wir erhalten so drei neue Regeln.

Wir erhalten also die neue Regelmenge

$$S_0 \to S, \qquad S \to AS \mid A \mid S \mid \varepsilon, \qquad A \to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}.$$

Leider haben wir nun eine neue ε-Regel hinzugewonnen; so dass wir diesen Schritt noch einmal durchführen müssen. Wir löschen nun $S \to \varepsilon$. Die Variable S kommt bei drei Regeln auf der rechten Seite vor:

- $S_0 \to S$ wird zu $S_0 \to S \mid \varepsilon$. Wir gewinnen also eine neue ε-Regel hinzu.
- $S \to AS$ müssten wir eigentlich zu $S \to AS \mid A$ umschreiben. Die Regel $S \to A$ ist allerdings bereits vorhanden, so dass wir hier einfach nichts tun.
- $S \to S$ wird zu $S \to S \mid \varepsilon$. Die Regel $S \to \varepsilon$ haben wir allerdings bereits gelöscht, so dass wir sie nicht wieder hinzufügen dürfen!

Unsere Regelmenge enthält nun also die folgenden Regeln:

$$S_0 \to S \mid \varepsilon, \qquad S \to AS \mid A \mid S, \qquad A \to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}.$$

Die einzige ε-Regel, die noch vorhanden ist, ist $S_0 \to \varepsilon$. Da S_0 unser neues Startsymbol ist, muss diese Regel unverändert bleiben. Wir sind also mit diesem Schritt fertig.

Schritt 3 (Ersetzen der Ein-Variablen-Regeln): In P sind momentan drei Regeln vorhanden, die auf der rechten Seite genau eine Variable enthalten: $S_0 \to S$, $S \to A$ und $S \to S$. Regeln, bei denen eine Variable auf sich selbst abgebildet wird, können wir einfach so entfernen, ohne sonst etwas tun zu müssen. Wir arbeiten also mit folgenden Regeln weiter:

$$S_0 \to S \mid \varepsilon, \qquad S \to AS \mid A, \qquad A \to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}.$$

Wir entfernen nun zuerst die Ein-Variablen-Regel $S \to A$. Dazu fügen wir für jede Regel $A \to \beta$ eine Regel $S \to \beta$ hinzu. Auf diese Art erhalten wir die folgende Menge (aus Übersichtsgründen sind die Regeln von nun an übereinander aufgelistet):

$$\begin{aligned} S_0 &\to S \mid \varepsilon, \\ S &\to AS \mid \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}, \\ A &\to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}. \end{aligned}$$

Wir entfernen nun die Ein-Variablen-Regel $S_0 \to S$. Auch hier müssen wir für jede Regel $S \to \beta$ eine Regel $S_0 \to \beta$ einführen:

$$\begin{aligned} S_0 &\to \varepsilon \mid AS \mid \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}, \\ S &\to AS \mid \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}, \\ A &\to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab}. \end{aligned}$$

Nun sind keine Ein-Variablen-Regeln mehr vorhanden, und dieser Schritt ist beendet.

Schritt 4 (Zerlegen der rechten Seiten): Wir betrachten zuerst die Regeln für S_0. Drei von diesen Regeln haben eine rechte Seite, die zu lang ist, außerdem stehen bei der Regel $S_0 \to \mathtt{ab}$ zwei Terminale auf der rechten Seite. Wir ersetzen diese Regeln wie folgt:

- $S_0 \to \mathtt{a}A\mathtt{b}A$ wird zuerst zerlegt in $S_0 \to \mathtt{a}A_1$, $A_1 \to AA_2$, $A_2 \to \mathtt{b}A$ (dazu benötigen wir die neuen Variablen A_1, A_2). Da auf den rechten Seiten dieser Regeln Terminale vorkommen, schreiben wir diese neuen Regeln um zu $S_0 \to N_{\mathtt{a}}A_1$, $A_1 \to AA_2$, $A_2 \to N_{\mathtt{b}}A$ und fügen außerdem noch die Regeln $N_{\mathtt{a}} \to \mathtt{a}$ und $N_{\mathtt{b}} \to \mathtt{b}$ hinzu (mit den neuen Variablen $N_{\mathtt{a}}$ und $N_{\mathtt{b}}$).
- $S_0 \to \mathtt{ab}A$ wird zerlegt zu $S_0 \to N_{\mathtt{a}}A_3$, $A_3 \to N_{\mathtt{b}}A$.
- $S_0 \to \mathtt{a}A\mathtt{b}$ wird zerlegt zu $S_0 \to N_{\mathtt{a}}A_4$, $A_4 \to AN_{\mathtt{b}}$.
- $S_0 \to \mathtt{ab}$ wird umgeschrieben zu $N_{\mathtt{a}}N_{\mathtt{b}}$.

Insgesamt erhalten wir also die folgende Regelmenge:

$$\begin{aligned}
S_0 &\to \varepsilon \mid AS \mid N_{\mathtt{a}}A_1 \mid N_{\mathtt{a}}A_3 \mid N_{\mathtt{a}}A_4 \mid N_{\mathtt{a}}N_{\mathtt{b}},\\
S &\to AS \mid \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab},\\
A &\to \mathtt{a}A\mathtt{b}A \mid \mathtt{ab}A \mid \mathtt{a}A\mathtt{b} \mid \mathtt{ab},\\
A_1 &\to AA_2,\\
A_2 &\to N_{\mathtt{b}}A,\\
A_3 &\to N_{\mathtt{b}}A,\\
A_4 &\to AN_{\mathtt{b}},\\
N_{\mathtt{a}} &\to \mathtt{a},\\
N_{\mathtt{b}} &\to \mathtt{b}.
\end{aligned}$$

Im Prinzip müssten wir die gleiche Zerlegung auf die entsprechenden Regeln von S und A anwenden. Da dort aber genau die gleichen rechten Seiten zu zerlegen sind, wie es bei S_0 der Fall war, können wir auch genau die gleichen zerlegten Regeln verwenden. Dadurch erhalten wir die folgende Menge P_C von Regeln:

$$\begin{aligned}
S_0 &\to \varepsilon \mid AS \mid N_{\mathtt{a}}A_1 \mid N_{\mathtt{a}}A_3 \mid N_{\mathtt{a}}A_4 \mid N_{\mathtt{a}}N_{\mathtt{b}},\\
S &\to AS \mid N_{\mathtt{a}}A_1 \mid N_{\mathtt{a}}A_3 \mid N_{\mathtt{a}}A_4 \mid N_{\mathtt{a}}N_{\mathtt{b}},\\
A &\to N_{\mathtt{a}}A_1 \mid N_{\mathtt{a}}A_3 \mid N_{\mathtt{a}}A_4 \mid N_{\mathtt{a}}N_{\mathtt{b}},\\
A_1 &\to AA_2,\\
A_2 &\to N_{\mathtt{b}}A,\\
A_3 &\to N_{\mathtt{b}}A,\\
A_4 &\to AN_{\mathtt{b}},\\
N_{\mathtt{a}} &\to \mathtt{a},\\
N_{\mathtt{b}} &\to \mathtt{b}.
\end{aligned}$$

Keine dieser Regeln verletzt die Bedingungen der Chomsky-Normalform; die Grammatik $G_C := (\Sigma, V_C, P_C, S_0)$ mit $V_C := \{S_0, S, A, A_1, \ldots, A_4, N_{\mathtt{a}}, N_{\mathtt{b}}\}$ ist also in CNF. Da die einzelnen Schritte die Sprache der Grammatik nicht verändern erzeugt G_C die gleiche Sprache wie die Ursprungsgrammatik (auch wenn ihr dies nicht unbedingt direkt anzusehen ist). ◊

Beispiel 4.32 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $G := (\Sigma, V, P, S)$ eine CNF mit $V := \{A, B, S\}$, und P enthalte die folgenden Regeln:

$$S \to ASA \mid \mathtt{a}B, \qquad A \to B \mid S, \qquad B \to \mathtt{b} \mid \varepsilon.$$

Offensichtlich ist G nicht in CNF. Wir rufen nun den Algorithmus `CNF` auf.

Schritt 1 (neues Startsymbol): Wir fügen zu V das Symbol S_0 hinzu, ernennen dieses zu unserem neuen Startsymbol und nehmen zu P die Regel $S_0 \to S$ auf. Die Menge P enthält nun die folgenden Regeln:

$$S_0 \to S, \qquad S \to ASA \mid \mathtt{a}B, \qquad A \to B \mid S, \qquad B \to \mathtt{b} \mid \varepsilon.$$

Ansonsten kann in diesem Schritt nichts passieren.

Schritt 2 (Ersetzen der ε-Regeln): In ihrer momentanen Form enthält unsere Regelmenge eine ε-Regel, nämlich $B \to \varepsilon$. Wir entfernen diese und berechnen betrachten alle Regeln, in denen B auf der rechten Seite vorkommt, also diese Regeln:

$$S \to \mathtt{a}B, \qquad\qquad A \to B.$$

Die erste dieser beiden Regeln wird umgeformt zu $S \to \mathtt{a}B \mid \mathtt{a}$ (wir erhalten also eine neue Regel $S \to \mathtt{a}$), die zweite zu $A \to B \mid \varepsilon$. Wir erhalten also eine neue Regel $A \to \varepsilon$. Diese Regel ist zwar eine ε-Regel; aber da wir diese vorher nicht entfernt hatten wird sie in die Menge mit aufgenommen. Dadurch enthält P nun die folgenden Regeln (neue Regeln sind hervorgehoben):

$$S_0 \to S, \qquad S \to ASA \mid \mathtt{a}B \mid \mathtt{a}, \qquad A \to B \mid S \mid \varepsilon, \qquad B \to \mathtt{b}.$$

Da P noch die ε-Regel $A \to \varepsilon$ enthält, müssen wir diesen Schritt noch einmal ausführen, um diese Regel zu löschen. Glücklicherweise kommt A nur auf einer rechten Seite vor, nämlich in der Regel $S \to ASA$. Diese ersetzen wir durch die Regeln $S \to ASA|SA|AS|S$, und erhalten so die folgende Menge P:

$$S_0 \to S, \qquad S \to ASA \mid \mathbf{SA} \mid \mathbf{AS} \mid \mathbf{S} \mid \mathtt{a}B \mid \mathtt{a}, \qquad A \to B \mid S, \qquad B \to \mathtt{b}.$$

Schritt 3 (Ersetzen der Ein-Variablen-Regeln): Zuallererst entfernen wir die Regel $S \to S$ (solche Selbstabbildungen können immer direkt entfernt werden). Wir erhalten:

$$S_0 \to S, \qquad S \to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \qquad A \to B \mid S, \qquad B \to \mathtt{b}.$$

Als nächstes entfernen wir die Regel $S_0 \to S$ und erhalten folgende Menge P:

$$S_0 \to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \quad S \to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \quad A \to B \mid S, \quad B \to \mathtt{b}.$$

Leider sind immer noch Ein-Variablen-Regeln vorhanden. Wir entfernen als erstes $A \to B$ und erhalten die folgenden Regeln:

$$S_0 \to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \quad S \to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \quad A \to \mathtt{b} \mid S, \quad B \to \mathtt{b}.$$

Inzwischen ist nur noch eine Ein-Variablen-Regel verblieben, nämlich $A \to S$. Wir entfernen auch diese; dadurch wird die Regelmenge so groß, dass wir die Regeln von jetzt an übereinander darstellen:

$$\begin{aligned} S_0 &\to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \\ S &\to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a}, \\ A &\to ASA \mid SA \mid AS \mid \mathtt{a}B \mid \mathtt{a} \mid \mathtt{b}, \\ B &\to \mathtt{b}. \end{aligned}$$

Nun sind keine Ein-Variablen-Regeln mehr vorhanden, so dass wir mit diesem Schritt fertig sind.

Schritt 4 (Zerlegen der rechten Seiten): Drei Regeln haben rechte Seiten, die mehr als zwei Zeichen enthält, nämlich $S_0 \to ASA$, $S \to ASA$ und $A \to ASA$. Streng genommen müssten wir für jede dieser drei Regeln eine eigene neue Variable einführen; da die rechten Seiten aber identisch sind kürzen wir dies ab. Wir ersetzen diese Regeln durch die Regeln $S_0 \to AA_1$, $S \to AA_1$ und $A \to AA_1$ und fügen die neue Regel $A_1 \to SA$ hinzu.

Unter den Regeln, bei denen die rechte Seite eine Länge von 2 hat, sind nur die drei Regeln $S_0 \to \mathtt{a}B$, $S \to \mathtt{a}B$ und $A \to \mathtt{a}B$ problematisch. Hierfür ersetzen wir das $\mathtt{a}$ durch eine neue Variable $N_\mathtt{a}$ und fügen eine neue Regel $N_\mathtt{a} \to \mathtt{a}$ hinzu (auch hier müssten wir eigentlich drei verschiedene Variablen einführen, wir kürzen dies aber ab). Wir erhalten die folgenden Regeln:

$$\begin{aligned}
S_0 &\to AA_1 \mid SA \mid AS \mid N_\mathtt{a}B \mid \mathtt{a},\\
S &\to AA_1 \mid SA \mid AS \mid N_\mathtt{a}B \mid \mathtt{a},\\
A &\to AA_1 \mid SA \mid AS \mid N_\mathtt{a}B \mid \mathtt{a} \mid \mathtt{b},\\
A_1 &\to SA,\\
N_\mathtt{a} &\to \mathtt{a},\\
B &\to \mathtt{b}.
\end{aligned}$$

Hiermit endet Schritt 4 und somit der Algorithmus `CNF`. Die berechnete Regelmenge P wurde bereits angegeben. Es gilt $V = \{S_0, S, A, A_1, N_\mathtt{a}, B\}$. Wie leicht zu erkennen ist, ist die berechnete CFG tatsächlich in CNF. Da die einzelnen Schritte die Sprache der Grammatik nicht verändern erzeugt sie die gleiche Sprache wie die Ursprungsgrammatik. ◊

Satz 4.33 *Zu jeder kontextfreien Grammatik G berechnet der Algorithmus `CNF` eine kontextfreie Grammatik G_C in Chomsky-Normalform mit $\mathcal{L}(G_C) = \mathcal{L}(G)$.*

Beweis: Wir gehen hier nicht auf die Details ein, sondern skizzieren stattdessen die Argumentation. Die Termination von `CNF` ist sichergestellt, da die Schleifen in Schritt 2 und Schritt 3 nur begrenzt oft ausgeführt werden (in beiden Schritten ist explizit erwähnt, dass bereits gelöschte Regeln nicht wieder zu P hinzugefügt werden). Außerdem verkürzt die Schleife in Schritt 4 jeweils die Regeln, so dass diese auch nur begrenzt oft ausgeführt werden kann.

Außerdem ist in jedem der Schritte sichergestellt, dass die Veränderungen die von der Grammatik erzeugte Sprache nicht verändern. Darüber hinaus haben die Schritte folgende Funktionen:

- Schritt 1: Nach diesem Schritt ist sichergestellt, dass das Startsymbol auf keiner rechten Seite einer Regel mehr vorkommt.
- Schritt 2: Nach diesem Schritt ist sichergestellt, dass jede rechte Seite mindestens ein Symbol aus V oder Σ enthält.

- Schritt 3: Nach diesem Schritt ist sichergestellt, dass jede rechte Seite entweder aus einem Terminal besteht, oder mindestens eine Länge von 2 hat.
- Schritt 4: Nach diesem Schritt ist sichergestellt, dass jede rechte Seite entweder aus genau einem Terminal oder aus genau zwei Variablen besteht.

Es ist leicht zu sehen, dass diese direkt nach einem Schritt garantierten Eigenschaften der Grammatik auch nach allen späteren Schritten gelten. Zusammen stellen diese vier Schritte sicher, dass die Grammatik in Chomsky-Normalform ist. □

Korollar 4.34 *Jede kontextfreie Sprache wird von einer kontextfreien Grammatik in Chomsky-Normalform erzeugt.*

Unsere erste Anwendung der Chomsky-Normalform ist der Beweis des folgenden Resultats:

Lemma 4.35 *Die Klasse* CFL *ist abgeschlossen unter Schnitt mit regulären Sprachen. Mit anderen Worten: Für jede kontextfreie Sprache* $L \in \text{CFL}$ *und jede reguläre Sprache* $R \in \text{REG}$ *ist* $(L \cap R) \in \text{CFL}$.

Beweisidee: Wir konstruieren eine Grammatik G_{LR}, die sowohl eine CFG G für L als auch einen DFA A für R gleichzeitig simuliert. Dabei verwenden wir als Variablen ein neues Startsymbol S_{LR} sowie Tripel aus der Menge $Q \times V \times Q$. Aus einer Variablen $[q, A, p]$ können wir alle Wörter ableiten, die in der Grammatik G aus A abgeleitet werden können, und die außerdem im Automaten A vom Zustand q zum Zustand p führen können. Für jeden akzeptierenden Zustand $q_f \in F$ gibt es daher eine Regel $S_{LR} \to [q_0, S, q_f]$.

Bei Variablen der Form $[p, A, q]$ gibt es jeweils zwei Arten von Regeln: Ist $A \to b$ eine Regel in P, und geht A beim Lesen des Buchstaben b von p in q über (d. h. es ist $\delta(p, b) = q$), so enthält unsere neue Grammatik eine Regel $[p, A, q] \to b$. Außerdem können wir die längeren Pfade zwischen p und q noch zerlegen: Deswegen fügen wir für jeden Zustand $q' \in Q$ und jede Regel in P, die die Form $A \to BC$ hat, noch eine Regel $[p, A, q] \to [p, B, q'][q', C, q]$ zu P_{LR} hinzu. Wir zerteilen also die Wörter, die von p zu q führen (bwz. die Wörter, die aus A ableitbar sind) in diese entsprechenden Hälften. Dabei können Variablen entstehen, aus denen kein Terminalwort abgeleitet werden kann, oder die nicht aus dem Startsymbol abgeleitet werden können. Da diese die Sprache aber nicht verändern ist das kein Problem. □

Beweis: Sei $L \in \text{CFL}$ und $R \in \text{REG}$. Dann existieren eine CFG $G := (\Sigma, V, P, S)$ und ein DFA $A := (\Sigma, Q, \delta, q_0, F)$ mit $\mathcal{L}(G) = L$ und $\mathcal{L}(A) = R$. Wegen Korollar 4.34 können wir davon ausgehen, dass G in Chomsky-Normalform ist.

Wir betrachten zuerst den Sonderfall, dass $\varepsilon \notin L$. Wir definieren nun eine kontextfreie Grammatik $G_{LR} := (\Sigma, V_{LR}, P_{LR}, S_{LR})$ wie folgt[53]:

$$V_{LR} := \{S_{LR}\} \cup (Q \times V \times Q),$$

[53] Der Lesbarkeit halber und aus Tradition schreiben wir die Tripel, aus denen die Variablen dieser Grammatik bestehen, mit eckigen Klammern.

$$\begin{aligned} P_{LR} := &\big\{S_{LR} \to [q_0, S, q_f] \mid q_f \in F\big\} \\ &\cup \big\{[p, A, q] \to [p, B, q'][q', C, q] \mid q' \in Q, A \to BC \in P\big\} \\ &\cup \big\{[p, A, q] \to b \mid \delta(p, b) = q, A \to b \in P\big\}. \end{aligned}$$

Um die Korrektheit dieser Grammatik zu zeigen, beweisen wir zuerst die folgende Behauptung:

Behauptung 1 *Für alle $A \in V$, alle $p, q \in Q$ und alle $w \in \Sigma^+$ gilt:*

$$[p, A, q] \Rightarrow^*_{G_{LR}} w \text{ genau dann, wenn } A \Rightarrow^*_G w \text{ und } \delta(p, w) = q.$$

Beweis: Wir zeigen diese Behauptung durch Induktion über die Länge von w. Da $w \neq \varepsilon$ (wir fordern $w \in \Sigma^+$) beginnt die Induktion bei $|w| = 1$.

Induktionsanfang: Sei $|w| = 1$, es ist also $w = a$ für ein $a \in \Sigma$.

Behauptung: Für alle $p, q \in Q$ und alle $A \in V$ gilt $[p, A, q] \Rightarrow^*_{G_{LR}} a$ genau dann, wenn $A \Rightarrow^*_G a$ und $\delta(p, a) = q$.

Beweis: Dieser Beweis folgt unmittelbar aus den verwendeten Definitionen, daher müssen wir den Beweis nicht in zwei Richtungen aufteilen. Es gilt:

$$\begin{aligned} & [p, A, q] \Rightarrow^*_{G_{LR}} a \\ \text{gdw.} \quad & [p, A, q] \to a \text{ ist Regel in } P_{LR} \\ \text{gdw.} \quad & \delta(p, a) = q, \text{ und } A \to a \text{ ist Regel in } P \\ \text{gdw.} \quad & \delta(p, a) = q, \text{ und } A \Rightarrow^*_G a. \end{aligned}$$

Für den Induktionsanfang stimmt die Behauptung also.

Induktionsschritt: Sei $n \in \mathbb{N}_{>0}$ und sei $w \in \Sigma^+$ mit $|w| = n + 1$.

Induktionsannahme: Für alle $p, q \in Q$, alle $A \in V$ und alle $\hat{w} \in \Sigma^+$ mit $|\hat{w}| \leq n$ gelte $[p, A, q] \Rightarrow^*_{G_{LR}} \hat{w}$ genau dann, wenn $A \Rightarrow^*_G \hat{w}$ und $\delta(p, \hat{w}) = q$.

Behauptung: Für alle $p, q \in Q$ und alle $A \in V$ gilt $[p, A, q] \Rightarrow^*_{G_{LR}} w$ genau dann, wenn $A \Rightarrow^*_G w$ und $\delta(p, w) = q$.

Beweis: Für den Induktionsschritt unterscheiden wir die beiden Richtungen des Beweises. Wir beginnen mit der Rück-Richtung $\Leftarrow$: Sei $A \Rightarrow^*_G w$ und $\delta(p, w) = q$. Da $|w| = n+1 \geq 2$ existieren $B, C \in V$ und eine Regel $A \to BC$ in P mit $A \Rightarrow_G BC \Rightarrow^*_G w$. Somit lässt sich w in Wörter $u, v \in \Sigma^+$ zerlegen, mit

$$B \Rightarrow^*_G u \qquad \text{und} \qquad C \Rightarrow^*_G v.$$

Da $w = uv$ gilt $\delta(p, w) = \delta(\delta(p, u), v)$. Wir definieren nun $q' := \delta(q, u)$, und stellen fest, dass

$$\delta(p, u) = q' \qquad \text{und} \qquad \delta(q', v) = q.$$

Nach Definition von G_{LR} enthält P_{LR} die Regel $[p, A, q] \to [p, B, q'][q', C, q]$. Nach der Induktionsannahme gilt nun:

$$[p, B, q'] \Rightarrow^*_{G_{LR}} u \qquad \text{und} \qquad [q', C, q] \Rightarrow^*_{G_{LR}} v.$$

Also ist $[p, A, q] \Rightarrow_{G_{LR}} [p, B, q'][q', C, q] \Rightarrow^*_{G_{LR}} uv = w$. Dies beendet den Beweis der Rück-Richtung.

Für die Hin-Richtung $\Rightarrow$ nehmen wir an, dass $[p, A, q] \Rightarrow^*_{G_{LR}} w$. Da $|w| = n + 1 \geq 2$ müssen (gemäß Definition von G_{LR}) nun geeignete $q' \in Q$ und $B, C \in V$ existieren, so dass

$$[p, A, q] \Rightarrow_{G_{LR}} [p, B, q'][q', C, q] \Rightarrow^*_{G_{LR}} w.$$

Also können wir w in Wörter $u, v \in \Sigma^+$ zerlegen, so dass $w = uv$, und außerdem

$$[p, B, q'] \Rightarrow^*_{G_{LR}} u \qquad \text{und} \qquad [q', C, q] \Rightarrow^*_{G_{LR}} v.$$

Nach Induktionsannahme gilt

1. $B \Rightarrow^*_G u$ und $\delta(p, u) = q'$, sowie
2. $C \Rightarrow^*_G v$ und $\delta(q', v) = q$.

Nach Definition von P_{LR} muss P die Regel $A \to BC$ enthalten, also ist

1. $A \Rightarrow_G BC \Rightarrow^*_G uv = w$, und
2. $\delta(p, w) = \delta(\delta(p, u), v) = \delta(q', v) = q$.

Somit gilt auch die Rück-Richtung der Behauptung. □(Behauptung 1)

Für alle $w \in \Sigma^+$ gilt nun:

$$\begin{aligned}
& w \in \mathcal{L}(G_{LR}) \\
\text{gdw.} \quad & S \Rightarrow^*_{G_{LR}} w \\
\text{gdw.} \quad & S \Rightarrow_{G_{LR}} [q_0, S, q_f] \Rightarrow^*_{G_{LR}} w \text{ für ein } q_f \in F \\
\text{gdw.} \quad & [q_0, S, q_f] \Rightarrow^*_{G_{LR}} w \text{ für ein } q_f \in F \\
\text{gdw.} \quad & S \Rightarrow^*_G w \text{ und } \delta(q_0, w) = q_f \text{ für ein } q_f \in F \\
\text{gdw.} \quad & S \Rightarrow^*_G w \text{ und } \delta(q_0, w) \in F \\
\text{gdw.} \quad & w \in \mathcal{L}(G) \text{ und } w \in \mathcal{L}(A) \\
\text{gdw.} \quad & w \in (\mathcal{L}(G) \cap \mathcal{L}(A)) \\
\text{gdw.} \quad & w \in L \cap R.
\end{aligned}$$

Also gilt $\mathcal{L}(G_{LR}) = L \cap R$. Die Grammatik G_{LR} ist also in der Tat eine korrekte Grammatik für die Sprache $L \cap R$.

Für den Fall, dass $\varepsilon \in L$, fügen wir zu P_{LR} noch die Regel $S_{LR} \to \varepsilon$ hinzu. □

Die Aussage von Lemma 4.35 ist eine gute Nachricht: Für viele unserer Beweise ist nämlich der Schnitt mit einer regulären Sprache vollkommen ausreichend. Wir betrachten dazu ein Beispiel:

Beispiel 4.36 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $\text{COPY}_\Sigma := \{ww \mid w \in \Sigma^*\}$. Angenommen, $\text{COPY}_\Sigma \in \text{CFL}$. Dann ist

$$\begin{aligned} L' &:= \text{COPY}_\Sigma \cap \mathcal{L}(\mathtt{a}^*\mathtt{b}^*\mathtt{a}^*\mathtt{b}^*) \\ &= \{\mathtt{a}^i\mathtt{b}^j\mathtt{a}^i\mathtt{b}^j \mid i, j \in \mathbb{N}\}. \end{aligned}$$

Gemäß Lemma 4.35 ist L' eine kontextfreie Sprache. Aus Beispiel 4.20 wissen wir, dass $L' \notin \text{CFL}$. Widerspruch. Also ist $\text{COPY}_\Sigma \notin \text{CFL}$. ◊

4.1.3 Wortproblem und Syntaxanalyse

Für kontextfreie Grammatik in Chomsky-Normalform lässt sich das Wortproblem leicht lösen. Wir verwenden dazu Algorithmus 5, bekannt als **Cocke-Younger-Kasami-Algorithmus**[54], oder auch nur **CYK-Algorithmus**.

Algorithmus 5 : `CYK`

Eingabe : Eine CFG $G := (\Sigma, V, P, S)$ in CNF und ein Wort $w = a_1 \cdots a_n$, $n \geq 1$
Ausgabe : true, falls $w \in \mathcal{L}(G)$

```
1  for i := 1 to n do
2      V[i,i] := {A ∈ V | A → a_i ist Regel in P};
3  for d := 1 to n − 1 do
4      for i := 1 to n − d do
5          j := i + d;
6          V[i,j] := ∅;
7          for k := i to j − 1 do
8              V_neu := {A ∈ V | A → BC ist Regel in P, B ∈ V[i,k], C ∈ V[k+1,j]};
9              V[i,j] := V[i,j] ∪ V_neu;
10 if S ∈ V[1,n] then return true;
11 else return false;
```

Die Grundidee des CYK-Algorithmus ist, die Struktur der Regeln einer CFG in Chomsky-Normalform auszunutzen. Sei $G := (\Sigma, V, P, S)$ eine CFG in CNF, und sei $w \in \Sigma^+$. Wenn w die Länge 1 hat, ist leicht zu überprüfen, ob $w \in \mathcal{L}(G)$ (wir suchen einfach nach einer entsprechenden Regel $S \to w$).

Hat w zwei oder mehr Buchstaben, so muss w passend zerlegbar sein. Wir sehen nun genauer an, was das bedeutet: Angenommen, $|w| \geq 2$. Sei $w = a_1 \ldots a_n$ mit $n := |w|$ und $a_i \in \Sigma$ für $1 \leq i \leq n$. Es gilt $w \in \mathcal{L}(G)$ genau dann, wenn ein k mit $1 \leq k < n$ existiert, so dass

$$S \to BC \in P,\ B \Rightarrow_G^* a_1 \cdots a_k \text{ und } C \Rightarrow_G^* a_{k+1} \cdots a_n.$$

[54] Benannt nach John Cocke, Daniel Younger und Tadao Kasami.

Das heißt, wir können die Frage, ob $S \Rightarrow_G^* w$ gilt, zerlegen in die Fragen $B \Rightarrow_G^* a_1 \cdots a_k$ und $C \Rightarrow_G^* a_{k+1} \cdots a_n$ (für jede passende Zerlegung und alle passenden Regeln $S \to BC$). Dies können wir beliebig fortsetzen, bis alle zu testenden Stücke von w nur noch die Länge 1 haben.

Der Trick des CYK-Algorithmus ist, dieses Problem durch dynamische Programmierung zu lösen, und zwar nicht von S aus, sondern von den einzelnen a_i aus. Für alle $1 \leq i \leq n$ definieren wir

$$V[i,i] := \{A \mid A \to a_i \in P\},$$

also die jeweils die Menge aller Variablen, aus denen sich a_i erzeugen lässt. Anschließend definieren wir für i, j mit $1 \leq i \leq j \leq |w|$ jeweils eine Menge $V[i,j] \subseteq V$ mit der Eigenschaft, dass

$$V[i,j] = \{A \in V \mid A \Rightarrow_G^* a_i \cdots a_j\}.$$

Die Menge $V[i,j]$ enthält also die Variablen, die das Teilstück von w von der Position i bis zur Position j erzeugen kann. Da G in CNF ist, können wir dies durch dynamische Programmierung lösen. Für $1 \leq i < j \leq n$ gilt:

$$V[i,j] = \bigcup_{i \leq k < j} \{A \mid A \to BC \in P, B \in V[i,k], C \in V[k+1,j]\}.$$

Mit anderen Worten: Es gilt $A \Rightarrow_G^* a_i \cdots a_j$, wenn eine Trennstelle k mit $i \leq k \leq j$ und eine Regel $A \to BC$ existiert, so dass sich $a_i \cdots a_k$ aus B ableiten lässt, und $a_{k+1} \cdots a_j$ aus C. Eine bildliche Darstellung dieses Zusammenhangs finden Sie in Abbildung 4.4. Der Algorithmus `CYK` berechnet also lediglich alle diese Mengen $V[i,j]$ und speichert sie als Zwischenwerte in einer Tabelle. Dazu beginnt er mit allen Mengen $V[i,i]$, dann allen Mengen $V[i,i+1]$, dann alle $V[i,i+2]$, und so weiter, bis er schließlich die letzte Menge $V[1,n]$ berechnet. Genau dann, wenn diese das Startsymbol S enthält, lässt sich $w = a_1 \cdots a_n$ aus S ableiten. Dies ist nicht nur eine korrekte Lösung, sondern auch eine vergleichsweise effiziente:

Satz 4.37 *Sei $G := (\Sigma, V, P, S)$ eine kontextfreie Grammatik in Chomsky-Normalform, und sei $w \in \Sigma^+$. Dann entscheidet `CYK` in Zeit $O(mn^3)$ ob $w \in \mathcal{L}(G)$, wobei $m := |P|$ und $n := |w|$.*

Beweis: Die Korrektheit des Algorithmus folgt direkt aus den Vorüberlegungen.

Die erste Schleife wird maximal $O(n)$-mal durchlaufen und überprüft in jedem Durchlauf maximal m Regeln. Hier ergibt sich also eine Laufzeit von $O(mn)$.

Die zweite Schleife ist verschachtelt, die innerste Schleife wird maximal $O(n^3)$ mal durchlaufen und testet höchstens $O(m)$ Regeln. Hier ergibt sich also eine Laufzeit von $O(mn^3)$.

Insgesamt dominiert die Laufzeit der zweiten Schleife die der ersten, wir erhalten also eine Gesamtlaufzeit von $O(mn^3)$. □

Hinweis 4.38 Die Version von `CYK`, die wir in Algorithmus 5 betrachten, kann nicht entscheiden, ob $\varepsilon \in \mathcal{L}(G)$. Da G in Chomsky-Normalform ist, lässt sich `CYK` schnell

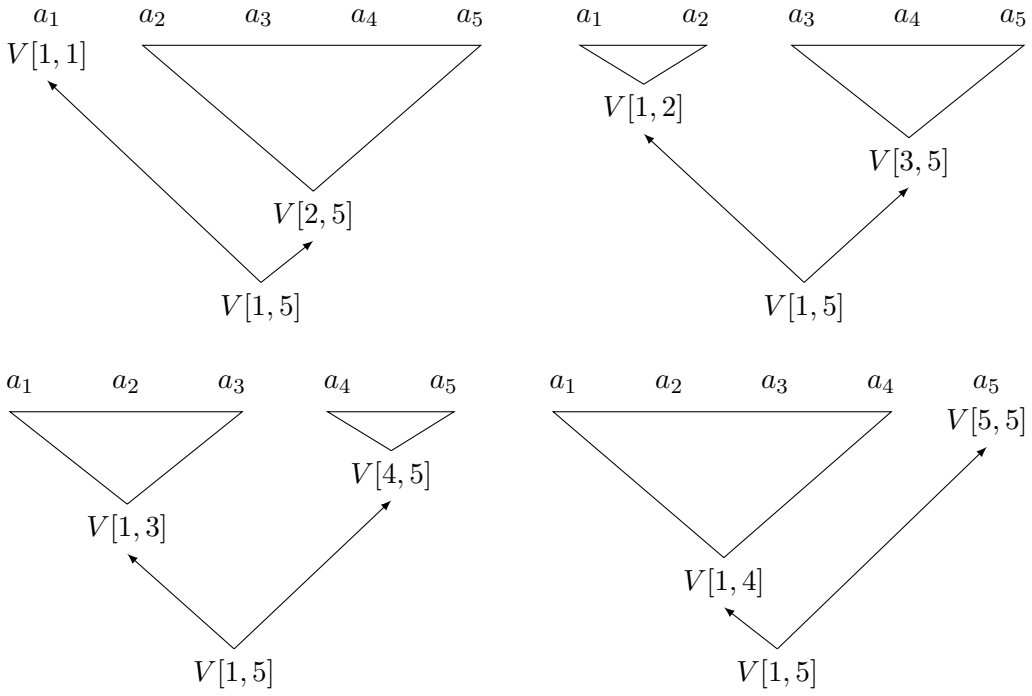


Abbildung 4.4: Eine schematische Darstellung der vier möglichen Zerlegungen, die der Algorithmus `CYK` (Algorithmus 5) beim Berechnen einer Menge $V[1,5]$ berücksichtigt. An den mit Dreiecken dargestellten Mengen (wie z. B. $V[2,5]$) werden ebenfalls entsprechende Zerlegungen vorgenommen. Einzig und alleine die Mengen $V[i,i]$ werden nicht weiter zerlegt.

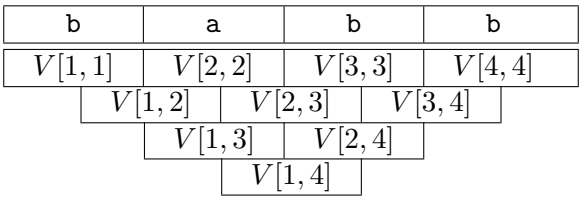
erweitern: Für $w = \varepsilon$ gilt $w \in \mathcal{L}(G)$ genau dann, wenn $S \to \varepsilon \in P$.

Zum besseren Verständnis werden im Folgenden zwei Beispiele aufgeführt: Ein nicht allzu großes (Beispiel 4.39) und ein etwas größeres (Beispiel 4.40).

Beispiel 4.39 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Die CFG $G := (\Sigma, V, P, S)$ sei definiert durch $V := \{S, A, B, C\}$ und durch die folgende Regelmenge P:

$$\begin{aligned} S &\to AB \mid BC, \\ A &\to BC \mid \mathtt{a}, \\ B &\to CA \mid \mathtt{b}, \\ C &\to AB \mid \mathtt{a}. \end{aligned}$$

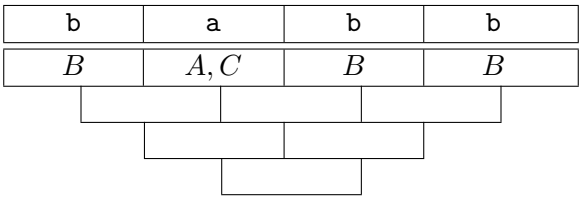
Es ist leicht zu sehen, dass G in Chomsky-Normalform ist. Wir untersuchen die Arbeitsweise von `CYK` auf dem Wort $w := \mathtt{babb}$. Dazu werden wir die berechneten Werte $V[i, j]$ in einer Tabelle darstellen, die wie folgt organisiert ist:



Die erste Zeile der Tabelle wird vom Algorithmus `CYK` in der Schleife in Zeile 1 erzeugt. Dazu werden die folgenden Mengen betrachtet:

- $V[1, 1]$: Es gilt $a_1 = \mathtt{b}$. Das Terminal `b` kann nur durch eine Regel erzeugt werden, nämlich durch $B \to \mathtt{b}$. Also ist $V[1, 1] := \{B\}$.
- $V[2, 2]$: Es gilt $a_2 = \mathtt{a}$. Das Terminal `a` kann durch zwei verschiedene Regeln erzeugt werden, nämlich durch $A \to \mathtt{a}$ und $C \to \mathtt{a}$. Also ist $V[2, 2] := \{A, C\}$.
- $V[3, 3]$ und $V[4, 4]$: Analog zu $V[1, 1]$ gilt $V[3, 3] = V[4, 4] = \{B\}$.

Wir tragen diese Mengen in unsere Tabelle ein:

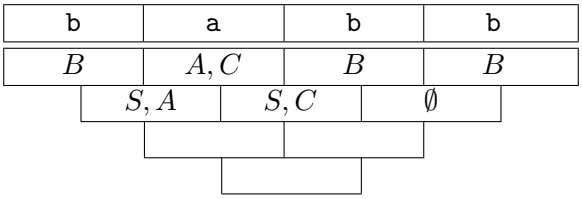


Streng genommen müssten wir die Einträge der Tabelle mit Mengenklammern schreiben, da es sich bei den $V[i, j]$ ja um Mengen handelt. Allerdings tragen diese Klammern nicht zu Lesbarkeit bei, so dass wir sie in Zukunft stillschweigend unterschlagen.

Im nächsten Schritt berechnet `CYK` die zweite Zeile der Tabelle, also alle $V[i, i+1]$. Jede dieser Mengen entspricht einem Teilwort der Länge 2, und kann wegen der CNF jeweils nur aus $V[i, i]$ und $V[i+1, i+1]$ berechnet werden. Im Einzelnen ergeben sich die folgenden Mengen:

- $V[1, 2]$: Hierzu suchen wir alle Regeln, die eine Satzform aus $V[1, 1] \cdot V[2, 2] = \{BA, BC\}$ erzeugen können. Dabei kommt BA auf keiner rechten Seite vor, während BC bei den Regeln $S \to BC$ und $A \to BC$ erscheint. Also ist $V[1, 2] = \{S, A\}$.
- $V[2, 3]$: Es gilt $V[2, 2] \cdot V[3, 3] = \{AB, CB\}$. Da CB auf keiner rechten Seite vorkommt hat dieses Wort keinen Einfluss, AB kommt in den Regeln $S \to AB$ und $C \to AB$ vor. Daher gilt $V[2, 3] = \{S, C\}$.
- $V[3, 4]$: Hier ist nur BB zu untersuchen. Dies kommt auf keiner rechten Seite vor, also ist $V[3, 4] = \emptyset$.

Wir fassen unsere Beute in der folgenden Tabelle zusammen:



Wir wagen uns nun an die dritte Zeile, also alle Mengen $V[i, i+2]$. Hierbei sind jeweils zwei mögliche Zerlegungen zu berücksichtigen, und zwar folgendermaßen:

- $V[1, 3]$: Die zu berücksichtigenden Kombinationen ergeben sich aus der folgenden Mengen:

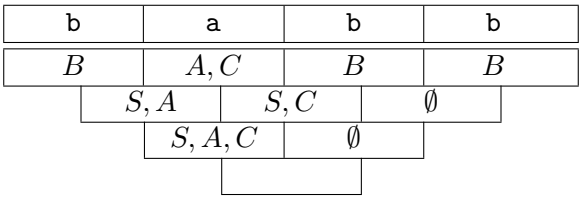
  $$V[1, 1] \cdot V[2, 3] = \{B\} \cdot \{S, C\} = \{BS, BC\},$$
  $$V[1, 2] \cdot V[3, 3] = \{S, A\} \cdot \{B\} = \{SB, AB\}.$$

  Dabei wird AB von den Regeln $S \to AB$ und $C \to AB$ erzeugt, und BC aus $A \to BC$. Es gilt also $V[1, 3] = \{S, A, C\}$.
- $V[2, 4]$: Wir betrachten dazu die beiden folgenden Mengen:

  $$V[2, 2] \cdot V[3, 4] = \{A, C\} \cdot \emptyset = \emptyset,$$
  $$V[2, 3] \cdot V[4, 4] = \{S, C\} \cdot \{B\} = \{SB, CB\}.$$

  Keine der beiden Satzformen SB und BC kommt in einer Regel aus P vor, also ist $V[2, 4] = \emptyset$.

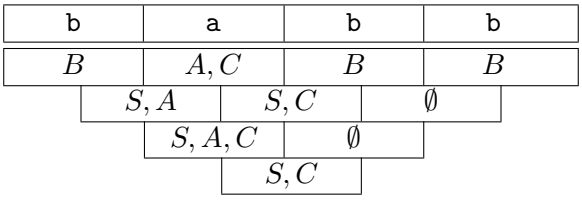
Damit wir nicht durcheinander geraten, tragen wir auch diese Werte in unsere (fast volle) Tabelle ein:



Auf zum Endspurt! Wir untersuchen nun die letzte Zeile, also $V[1,4]$. Dazu müssen wir ganze drei mögliche Zerlegungen beachten. Es gilt:

$$V[1,1] \cdot V[2,4] = \{B\} \cdot \emptyset = \emptyset,$$
$$V[1,2] \cdot V[3,4] = \{S,A\} \cdot \emptyset = \emptyset,$$
$$V[1,3] \cdot V[4,4] = \{S,A,C\} \cdot \{B\} = \{SB, AB, AC\}.$$

Aufgrund der Regeln $S \to AB$ und $C \to AB$ ist $V[1,4] = \{S,C\}$ (für A und B existieren keine passenden rechten Seiten). Wir betrachten zuerst zufrieden die vollendete Tabelle:

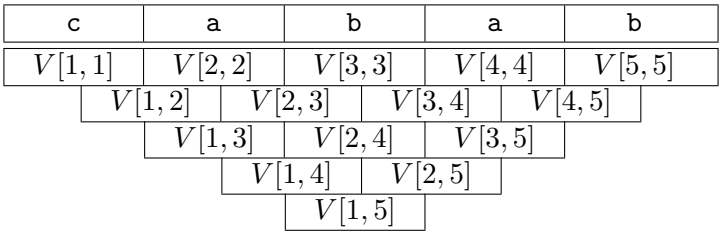


Nun müssen wir nur noch entscheiden, ob $w \in \mathcal{L}(G)$. Durch einen kurzen Kontrollblick stellen wir fest, dass $S \in V[1,4]$. Das heißt, $w \in \mathcal{L}(G)$. ◊

Beispiel 4.40 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Wir betrachten die CFG $G := (\Sigma, V, P, S)$ mit $V := \{S, A, B, C\}$, wobei P die folgenden Regeln enthalte:

$$S \to AB \mid \mathtt{b},$$
$$A \to CB \mid AA \mid \mathtt{a},$$
$$B \to AD \mid \mathtt{b},$$
$$C \to BD \mid \mathtt{c},$$
$$D \to AB \mid \mathtt{b}.$$

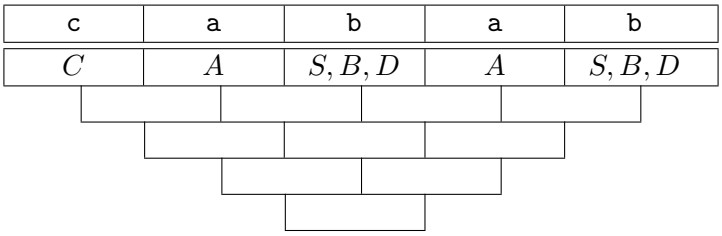
Offensichtlich ist G in Chomsky-Normalform. Wir betrachten nun das Wort $w := \mathtt{cabab}$. Wir stellen dabei die berechneten Werte $V[i,j]$ in einer Tabelle dar, die die folgende Form haben wird:



Wir beginnen mit der ersten Zeile der Tabelle. Diese wird vom Algorithmus `CYK` in der Schleife in Zeile 1 erzeugt. Wir betrachten die folgenden Mengen:

- $V[1,1]$: Es gilt $a_1 =$ `c`. Da das Terminal `c` nur durch die Regel $C \to$ `c` erzeugt werden kann, ist $V[1,1] = \{C\}$.
- $V[2,2]$: Es gilt $a_2 =$ `a`. Auch das Terminal `a` kann nur durch eine einzige Regel erzeugt werden, nämlich $A \to$ `a`. Also gilt $V[2,2] = \{A\}$.
- $V[3,3]$: Es gilt $a_3 =$ `b`. Hier müssen wir drei Regeln beachten, nämlich $S \to$ `b`, $B \to$ `b` und $D \to$ `b`. Also gilt $V[3,3] = \{S, B, D\}$.
- $V[4,4]$: Es gilt $a_4 =$ `a`. Analog zu $V[2,2]$ schließen wir $V[4,4] = \{A\}$-
- $V[5,5]$: Es gilt $a_5 =$ `b`, und somit $V[5,5] = \{S, B, D\}$.

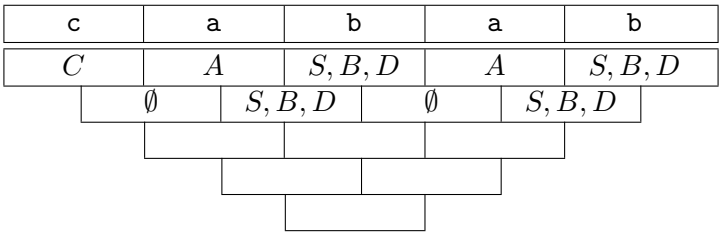
Daraus ergibt sich die folgende Tabelle:



Im nächsten Schritt müssen wir die zweite Zeile betrachten, und so alle $V[i, i+1]$ konstruieren. Wir gehen wie folgt vor:

- $V[1,2]$: Wörter der Länge 2 können nur auf eine Art entstehen; wir müssen also nur Regeln suchen, die auf der rechten Seite eine Satzform aus $V[1,1] \cdot V[1,2]$ haben. Es gilt $V[1,1] \cdot V[2,2] = CA$. Diese Kombination von Variablen kommt auf keiner rechten Seite einer Regel aus P vor, also ist $V[1,2] = \emptyset$.
- $V[2,3]$: Es gilt $V[2,2] \cdot V[3,3] = \{AB, AD, AS\}$. Diese können aus den Variablen S und D (mit $S \to AB$ und $D \to AB$) und B (mit $B \to AD$) erzeugt werden. Also ist $V[2,3] = \{S, B, D\}$
- $V[3,4]$: Es gilt $V[3,3] \cdot V[4,4] = \{SA, BA, DA\}$. Keine dieser Kombinationen taucht auf einer rechten Seite auf, also ist auch diese Menge leer.
- $V[4,5]$: Es gilt $V[4,4] \cdot V[5,5] = \{AB, AD, AS\}$. Analog zu $V[2,3]$ gilt $V[4,5] = \{S, B, D\}$.

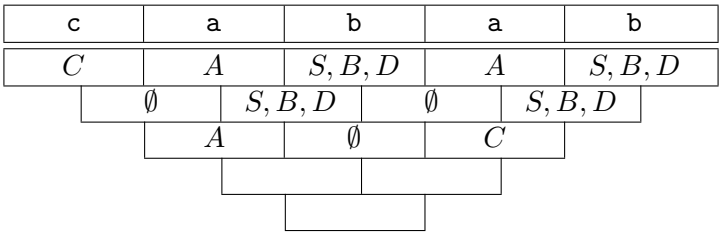
Dadurch erhalten wir die folgende Tabelle:



Im dritten Schritt wollen wir nun alle Mengen $V[i, i+2]$ konstruieren. Hier müssen wir ein bisschen mehr beachten: Während Wörter der Längen 1 und 2 aufgrund der Chomsky-Normalform eindeutig zerlegt werden können, bestehen bei Wörtern der Länge 3 im Allgemeinen zwei Möglichkeiten, die Terminale zu „verteilen“: Es können jeweils die ersten zwei oder die letzten zwei Terminale eine gemeinsame Elternvariable haben. Daher sind in diesem Schritt für jedes $V[i, i+2]$ immer zwei mögliche Konkatenationen der Mengen zu betrachten, nämlich $V[i, i+1] \cdot V[i+2, i+2]$ und $V[i, i] \cdot V[i+1, i+2]$ (im ersten Fall haben die ersten zwei Terminale eine gemeinsame Elternvariable, im zweiten Fall die letzten zwei Terminale). Wir erhalten die folgenden Mengen:

- $V[1, 3]$: Es gilt $V[1, 2] \cdot V[3, 3] = \emptyset$ und $V[1, 1] \cdot V[2, 3] = \{CS, CB, CD\}$. Von diesen drei Kombinationen kommt nur CB auf der rechten Seite einer Regel vor, nämlich der Regel $A \to CB$. also ist $V[1, 3] = \{A\}$.
- $V[2, 4]$: Es gilt $V[2, 3] \cdot V[4, 4] = \{SA, BA, DA\}$ und $V[2, 2] \cdot V[3, 4] = \emptyset$. Hier trifft keine Regel zu, also ist $V[2, 4] = \emptyset$.
- $V[3, 5]$: Wegen $V[3, 4] = \emptyset$ betrachten wir nur $V[3, 3] \cdot V[4, 5] = \{S, B, D\} \cdot \{S, B, D\}$. Von den neun Wörtern aus dieser Menge kommt nur BD auf der rechten Seite einer Regel vor (nämlich $C \to BD$), also ist $V[3, 5] = \{C\}$.

Wir beenden diesen Schritt mit einem Blick auf den aktuellen Stand der Tabelle:



Im vierten Schritt müssen wir nun alle Mengen $V[i, i+3]$ konstruieren. Die Zahl der möglichen Zerlegungen steigt wieder an: Für jede Menge $V[i, i+3]$ müssen drei mögliche Zerlegungen betrachtet werden. Es gilt:

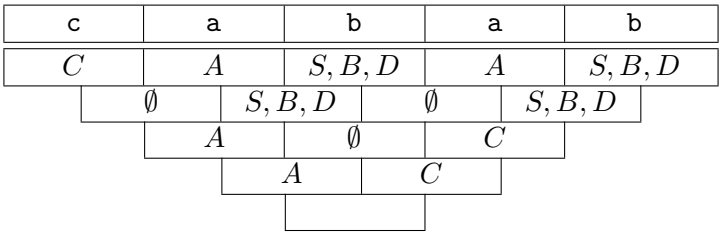
- $V[1, 4]$: Wir betrachten die Mengen $V[1, 1] \cdot V[2, 4]$, $V[1, 2] \cdot V[3, 4]$ und $V[1, 3] \cdot V[4, 4]$. Die einzige dieser drei Mengen, die nicht nicht leer ist, ist $V[1, 3] \cdot V[4, 4] = \{AA\}$. Die Satzform AA kann nur durch die Regel $A \to AA$ erzeugt werden, also ist $V[1, 4] = \{A\}$.

- $V[2,5]$: Wir betrachten die Mengen

$$V[2,2] \cdot V[3,5] = \{AC\},$$
$$V[2,3] \cdot V[4,5] = \{S,B,D\}\{S,B,D\},$$
$$V[2,4] \cdot V[5,5] = \emptyset.$$

  Dabei kommt AC auf keiner rechten Seite vor, und von den Elementen der Menge $\{S,B,D\} \cdot \{S,B,D\}$ kann nur BD erzeugt werden, und zwar durch $C \to BD$ (den Fall hatten wir bereits im vorigen Schritt). Also ist $V[2,5] = \{C\}$.

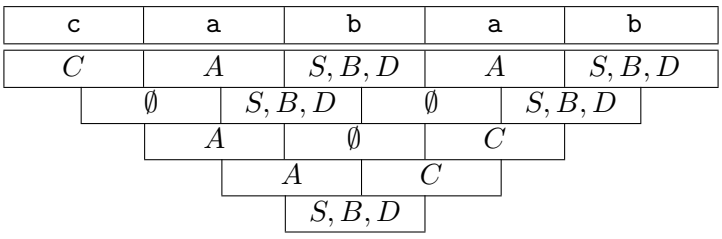
Unsere Tabelle hat nun folgende Einträge:



Wir führen nun den letzten Schritt aus, die Berechnung von $V[1,5]$. Dazu müssen wir vier mögliche Kombinationen betrachten, die den vier möglichen Zerlegungen von w entsprechen. Das sind:

$$V[1,1] \cdot V[2,5] = \{C\} \cdot \{C\} = \{CC\},$$
$$V[1,2] \cdot V[3,5] = \emptyset \cdot \{C\} = \emptyset,$$
$$V[1,3] \cdot V[4,5] = \{A\} \cdot \{S,B,D\} = \{AS, AB, AD\},$$
$$V[1,4] \cdot V[5,5] = \{A\} \cdot \{S,B,D\} = \{AS, AB, AD\}.$$

Dabei kommen CC und AS auf keiner rechten Seite einer Regel vor. Für AB kommen die Regeln $S \to AB$ und $D \to AB$ in Frage, für AD die Regel $B \to AD$. Also ist $V[1,5] = \{S,B,D\}$. Der Vollständigkeit halber führen wir die komplette Tabelle auf:



Die entscheidende Frage ist nun, ob $S \in V[1,5]$. Dies ist der Fall, also gilt $w \in \mathcal{L}(G)$. $\Diamond$

Hinweis 4.41 Da das Startsymbol gemäß der Definition der Chomsky-Normalform bei keiner Regel auf der rechten Seite vorkommen kann, können Sie die Erstellung der Tabelle ein wenig optimieren. Alle Elemente einer Menge $V[i,k] \cdot V[k+1,j]$, die ein Vorkommen des Startsymbols enthalten, können bei der Berechnung von $V[i,j]$ keine Rolle spielen. In Beispiel 4.40 kommt zum Beispiel zweimal die Menge $\{S,B,D\} \cdot \{S,B,D\}$. Anstelle nun für jedes der neun Elemente dieser Menge nach Regeln zu suchen, die dieses Element auf der rechten Seite haben, können wir uns auf die vier Elemente aus $\{B,D\} \cdot \{B,D\}$ beschränken.

Hinweis 4.42 Denken Sie bitte daran: Das zu untersuchende Wort $a_1 \cdots a_n$ gehört nur dann zur Sprache, wenn das Startsymbol in der Menge $V[1,n]$ enthalten ist. Es genügt *nicht*, dass $V[1,n] \neq \emptyset$.

Noch einmal: $a_1 \cdots a_n \in \mathcal{L}(G)$ gilt genau dann, wenn $S \in V[1,n]$.

In praktischen Fällen ist oft nicht nur die Lösung des Wortproblems für CFGs interessant, sondern auch das Berechnen der dazu gehörenden Ableitungsbäume. Dieser Vorgang nennt sich **Parsing** oder auch **Syntaxanalyse**. Durch eine leichte Modifikation lässt sich der CYK-Algorithmus in einen Parsing-Algorithmus umbauen. Dazu speichern wir (für $j \geq 2$) in den Mengen $V[i,j]$ nicht nur die einzelnen Variablen A, sondern 4-Tupel aus $(V \times V \times [1,\ldots,n] \times V)$. Enthält $V[i,j]$ ein 4-Tupel $A\colon B,k,C$ (der Lesbarkeit halber schreiben wir die Tupel auf diese Art), so bedeutet dies:

- In der Regelmenge P existiert eine Regel $A \to BC$,
- $B \in V[i,k]$, und
- $C \in V[k+1,j]$.

Wir speichern also alle Informationen, die bei der Konstruktion von V_{neu} in Zeile 9 von `CYK` verwendet worden sind (während `CYK` nur das A speichert). Jedes 4-Tupel $A\colon B,k,C$ in einer Tabellenzelle $V[i,j]$ kodiert also die folgenden Informationen

- Die Regel $A \to BC$, der das Tupel entspricht, und
- welcher Teil des Wortes $a_i \cdots a_j$ von B bzw. C erzeugt wird; nämlich $a_i \cdots a_k$ von B, und $a_{k+1} \cdots a_j$ von C.

Anhand dieser Informationen können anschließend aus der Tabelle alle möglichen Ableitungsbäume für w rekonstruiert werden. Angenommen, wir haben die Tabelle mit den oben beschriebenen Zusatzinformationen für eine kontextfreie Grammatik $G := (\Sigma, V, P, S)$ in CNF und ein Wort $w \in \Sigma^+$ erzeugt. Dann beginnen wir in der Tabellenzelle $V[1,|w|]$ und betrachte dort alle Einträge der Form $S\colon (B,k,C)$ (mit $B,C \in V$, $1 \leq k \leq |w|$). Für jede dieser Einträge erzeugen wir (mindestens) einen Ableitungsbaum.

Dazu schreiben wir S an die Wurzel, nehmen B als linkes Kind und C als rechtes Kind. Dann konstruieren wir jeweils den Baum anhand der Zellen $V[1,k]$ (unterhalb von B) und $V[k+1,|w|]$ (unterhalb von C) weiter.

Allgemein betrachten wir also ein beliebiges Nichtterminal A zusammen mit einer Zelle $V[i,j]$. Ist $i = j$, so können wir den Baum an dieser Stelle anhand der Regel $A \to a_i$ beenden (da die Tabelle von `CYK` konstruiert wurde, muss diese Regel existieren). Spannender ist der Fall $i < j$. Hier betrachten wir alle Einträge aus $V[i,j]$, die die Form $A\colon B, j, C$ haben. Sind mehrere Einträge vorhanden, so müssen wir mit entsprechend vielen Kopien des Baumes weiterarbeiten. In jedem der Fälle gehen wir wie vorher beschrieben vor: Das linke Kind des aktuellen Knoten beschriften wir mit B, das rechte mit C; und unterhalb von B konstruieren wir einen Baum mit B aus $V[i,k]$, sowie unterhalb von C einen Baum mit C aus $V[k+1,j]$.

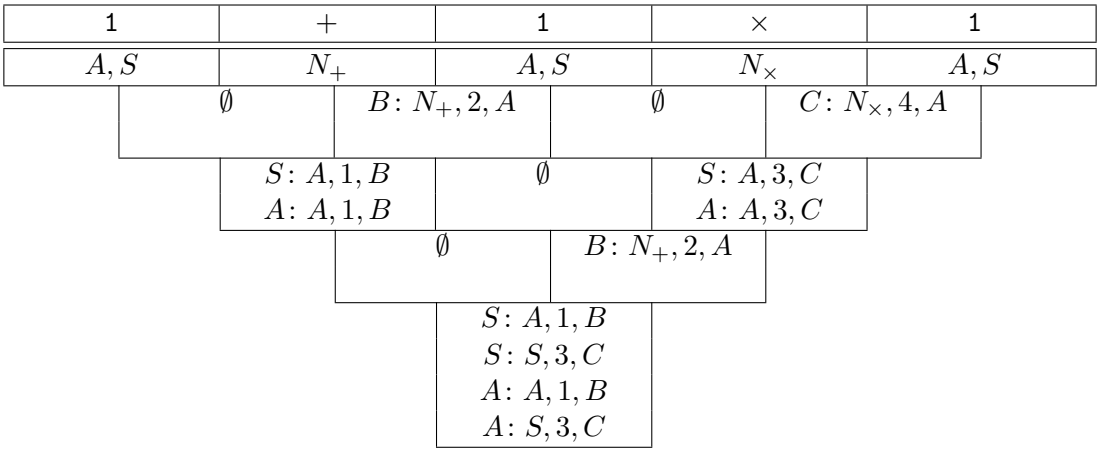
Wir betrachten dieses Vorgehen an einem kleinen Beispiel:

Beispiel 4.43 Sei $\Sigma := \{\mathtt{0}, \mathtt{1}, +, \times\}$. Wir betrachten die CFG $G := (\Sigma, V, P, S)$ mit $V := \{S, A, B, C, N_+, N_\times\}$ und der folgenden Regelmenge P:

$$\begin{aligned} S &\to AB \mid AC \mid \mathtt{0} \mid \mathtt{1}, \\ A &\to AB \mid AC \mid \mathtt{0} \mid \mathtt{1}, \\ B &\to N_+ A, \\ C &\to N_\times A, \\ N_+ &\to +, \\ N_\times &\to \times. \end{aligned}$$

Diese Grammatik erzeugt ungeklammerte arithmetische Ausdrücke über den Zahlen 0 und 1 (hier geschrieben als `0` und `1`). Wir betrachten nun das Wort $w := \mathtt{1} + \mathtt{1} \times \mathtt{1}$.

Wir gehen nun wie in den letzten Beispielen vor, speichern aber zusätzlich die Informationen aus Zeile 9. Dadurch erhalten wir die folgende Tabelle:



Im Feld $V[1,5]$ sind zwar vier Einträge vorhanden; wir interessieren uns aber nur für die beiden, die sich auf das Startsymbol beziehen (da wir nur an Ableitungen $S \Rightarrow_G^* w$ inter-

essiert sind). Aus jedem dieser beiden Einträge können wir nun einen Ableitungsbaum erzeugen:

- $S\colon (A, 1, B)$. Dieser Eintrag lässt sich als Ableitung $S \Rightarrow_G AB$ mit $A \Rightarrow_G^* a_1$ und $B \Rightarrow_G^* a_2 \cdots a_5 = +\mathtt{1} \times \mathtt{1}$ interpretieren (da k=1). Für die Ableitung $A \Rightarrow_G^* a_1 = \mathtt{1}$ müssen wir nichts weiter nachsehen, daher betrachten wir die Ableitung aus B. Wir sehen dazu im Tabellenfeld $V[2, 5]$ nach. Dort ist nur ein Eintrag vorhanden, nämlich $B\colon N_+, 2, A$. Also gilt $C \Rightarrow_G^* N_+A$ mit $N_+ \Rightarrow_G^* a_2$ und $A \Rightarrow_G^* a_3 \cdots a_5$. Auch hier ist klar, wie N_+ abgeleitet werden muss, also wenden wir uns der Ableitung von $a_3 \cdots a_5$ aus A zu. Die entsprechende Tabellenzelle $V[3, 5]$ enthält zwar zwei Zeilen, aber uns interessiert nur die, die mit A beginnt (denn immerhin wollen wir A ableiten). Die Zeile ist also $A\colon A, 3, C$. Es gilt also $A \Rightarrow_G AC$, mit $A \Rightarrow_G^* a_3$ und $C \Rightarrow_G^* a_4a_5$. Aus der Tabellenzelle $V[4, 5]$ entnehmen wir, dass $C \Rightarrow_G^* N_\times, 4, A$. Hier muss also die Regel $C \to N_\times A$ verwendet werden (da C gar nicht anders abgeleitet werden kann, hätten wir uns den Blick in die Tabelle hier auch sparen können). Für alle verbleibenden Ableitungen gibt es jeweils nur eine Möglichkeit (jeweils von einer Variable auf ein Terminal), und somit können wir den gesamten Baum abgeben (Abbildung 4.5a).

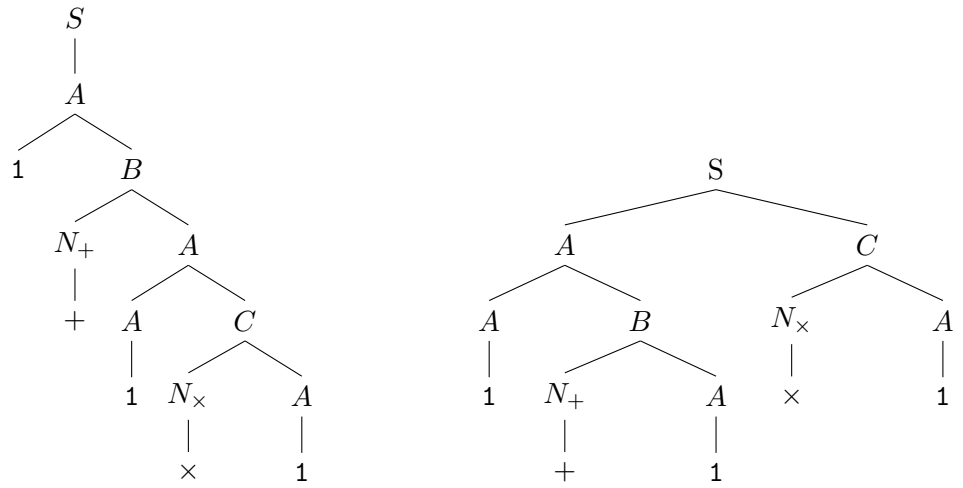
- $S\colon (A, 3, C)$. Aus diesem Eintrag schließen wir auf $S \Rightarrow_G AC$ mit $A \Rightarrow_G^* a_1a_2a_3$ und $C \Rightarrow_G^* a_4a_5$. Die Ableitung aus A können wir aus Tabellenzelle $V[1, 3]$ entnehmen, die aus C aus Tabellenzelle $V[4, 5]$.
  - $V[1, 3]$: Hier trifft für A nur der Eintrag $A\colon (A, 1, B)$ zu. Also ist $A \Rightarrow_G AB$ mit $A \Rightarrow_G^* a_1$ und $B \Rightarrow_G^* a_2a_3$. Für die erste dieser Ableitungen gibt es nur eine Möglichkeit, für die zweite betrachten wir $V[2, 3]$ und lesen dort $B : N_+, 2, A$. Die Ableitungen von N_+ auf a_2 und von A auf a_3 sind eindeutig bestimmt.
  - $V[4, 5]$: Hier ist nur ein Eintrag vorhanden, nämlich $C\colon N_\times, 4, A$. Die entsprechenden Ableitungen ergeben sich sofort.

  Der Ableitungsbaum für diesen Fall ist in Abbildung 4.5b dargestellt.

Da sonst keine möglichen Wege durch die Tabelle vorhanden sind, haben wir alle möglichen Ableitungsbäume für w in G konstruiert. Insbesondere wissen wir sicher, dass keine anderen Ableitungsbäume für w in G existieren. ◊

Hinweis 4.44 Lassen Sie sich nicht von Beispiel 4.43 in die Irre führen: Es ist möglich, dass zu einem Wort w mehrere Ableitungsbäume existieren, während $V[1, |w|]$ nur einen einzigen Eintrag hat. Abhängig von der betrachteten Grammatik können die Mehrdeutigkeiten auch erst in späteren Tabellenzellen auftreten.

Für unterschiedliche Anwendungszwecke existiert eine Reihe von Alternativen zum CYK-Algorithmus; eine kleine Übersicht finden Sie beispielsweise in Kapitel 5 von Shallit [19]. Insbesondere existieren Algorithmen, die nicht voraussetzen, dass die Grammatik in CNF ist.



(a) Ableitungsbaum für $S\colon A, 1, B$ **(b)** Ableitungsbaum für $S\colon A, 3, C$

Abbildung 4.5: Die beiden in Beispiel 4.43 möglichen Ableitungsbäume.

Wir haben gesehen, dass `CYK` in Bezug auf die Länge von w eine kubische Laufzeit hat. Dies ist nicht optimal; Leslie Valiant hat gezeigt, dass sich der gleiche Exponent wie bei der Matrixmultiplikation verwenden lässt[55]. Die bekannten Algorithmen zu „effizienten“ Matrixmultiplikation haben zwar eine Laufzeit mit Exponenten, der kleiner als 3 ist, allerdings sind die entsprechenden Konstanten so grausam hoch, dass dieser Ansatz im Allgemeinen nicht von praktischer Relevanz ist. Allerdings ist auch die kubische Laufzeit von `CYK` für viele Anwendungen nicht akzeptabel; daher werden gewöhnlich Algorithmen für Unterklassen von CFL verwendet und die Grammatiken entsprechend eingeschränkt. Wir werden uns mit diesen nur am Rande befassen. Stattdessen wenden wir uns einem anderen Problemfeld zu, das wir schon indirekt angeschnitten haben.

4.1.4 Mehrdeutigkeit

Wie wir unter anderem in Beispiel 4.43 gesehen haben, können zu einem Wort für eine Grammatik mehrere Ableitungsbäume existieren. Wie bereits in der Einleitung dieses Kapitels (ab Beispiel 4.13) diskutiert, werden Ableitungsbäume in der Praxis oft verwendet, um die Semantik von Wörtern (oder ganzen Programmen) zu codieren. Bei Programmiersprachen ist es normalerweise wünschenswert, dass jedes Programm genau eine Bedeutung hat. Aus Sicht der formalen Sprachen bedeutet dies, dass es zu dem Wort (also dem Code des Programms) genau einen Ableitungsbaum gibt. Um Fragestellungen zu diesem Themenkomplex zu formalisieren, führen wir die folgende Definition ein:

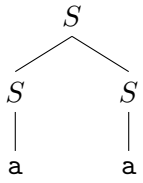
[55] Eine Referenz trage ich bei Gelegenheit nach.

Definition 4.45 Sei $G := (\Sigma, V, P, S)$ eine kontextfreie Grammatik. Ein Wort $w \in \mathcal{L}(G)$ heißt **eindeutig (in G)**, wenn für w in G genau ein Ableitungsbaum existiert. Existiert mehr als ein Ableitungsbaum für w, so heißt w **mehrdeutig (in G)**. Wir bezeichnen die Grammatik G als **mehrdeutig**, wenn mindestens ein Wort $w \in \mathcal{L}(G)$ mehrdeutig ist, und wir nennen G **eindeutig**, wenn G nicht mehrdeutig ist.

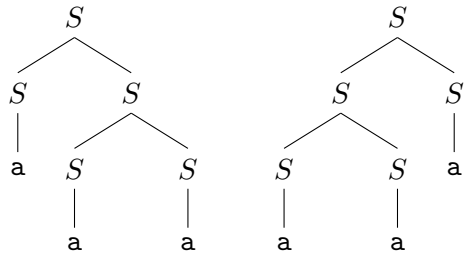
Wir illustrieren diese Definition an einem kleinen Beispiel:

Beispiel 4.46 Sei $\Sigma := \{\mathtt{a}\}$, $G := (\Sigma, \{S\}, P, S)$ mit $P := \{S \to SS \mid \mathtt{a}\}$.

Wir betrachten die Wörter $w_1 := \mathtt{aa}$ und $w_2 := \mathtt{aaa}$. Da für w_1 nur eine einzige Ableitung existiert, existiert genau ein Ableitungsbaum für w_1 in G, nämlich der folgende:



Im Gegensatz dazu existieren für w_2 zwei verschiedene Ableitungsbäume:



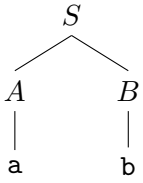
Also ist w_1 eindeutig (in G), während w_2 mehrdeutig ist. Da $\mathcal{L}(G)$ mindestens ein mehrdeutiges Wort enthält, ist G mehrdeutig. ◊

Oft ist es hilfreich, Mehrdeutigkeit nicht über Ableitungsbäume, sondern direkt über die Ableitungen zu definieren. Allerdings können wir dazu nicht einfach über die Existenz mehrere Ableitung argumentieren, wie das folgende Beispiel zeigt:

Beispiel 4.47 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $G := (\Sigma, V, P, S)$ mit $V := \{S, A, B\}$ und

$$P := \{S \to AB, \qquad A \to \mathtt{a}, \qquad B \to \mathtt{b}\}.$$

Dann gilt $\mathcal{L}(G) = \{\mathtt{ab}\}$. Die Sprache $\mathcal{L}(G)$ enthält nur ein einziges Wort, und dieses ist eindeutig (in G), da nur der folgende Ableitungsbaum möglich ist:



Allerdings können zwei verschiedene Ableitungen für ab angegeben werden, nämlich die folgenden:

$$S \Rightarrow_G AB \Rightarrow_G \mathtt{a}B \Rightarrow_G \mathtt{ab},$$
$$S \Rightarrow_G AB \Rightarrow_G A\mathtt{b} \Rightarrow_G \mathtt{ab}.$$

Wir können uns also entscheiden, ob wir zuerst A oder B ableiten. In beiden Fällen ergibt sich der oben dargestellte Ableitungsbaum. ◊

Um eine Definition von Mehrdeutigkeit anhand von Grammatiken zu erhalten, verfeinern wir den Ableitungsbegriff noch ein wenig:

Definition 4.48 Sei $G := (\Sigma, V, P, S)$ eine kontextfreie Grammatik, sei $w \in \mathcal{L}(G)$ und sei $\Gamma := \gamma_0, \ldots, \gamma_n$ ($n \in \mathbb{N}$) eine Ableitung von w in G. Es gelte also $\gamma_0 = S$, $\gamma_n = w$ und $\gamma_i \Rightarrow_G \gamma_{i+1}$ für alle $0 \leq i < n$.

Wir bezeichnen einen Ableitungsschritt $\gamma_i \Rightarrow_G \gamma_{i+1}$ als **Linksableitungsschritt**, wenn in γ_i die am weitesten links stehende Variable ersetzt wird. Wir bezeichnen die Ableitung Γ als **Linksableitung**, wenn jeder Ableitungsschritt von Γ ein Linksableitungsschritt ist.

Dabei drücken Ableitungsbäume und Linksableitungen den gleichen Sachverhalt aus:

Lemma 4.49 *Sei G eine kontextfreie Grammatik, und sei $w \in \mathcal{L}(G)$. Dann gilt:*

1. *Jeder Ableitungsbaum für w in G entspricht genau einer Linksableitung von w in G.*
2. *Jede Linksableitung von w in G entspricht genau einem Ableitungsbaum für w in G.*

Beweisidee: Folgt direkt aus den Definitionen. □

Lemma 4.49 besagt also, dass Ableitungsbäume und Linksableitungen unterschiedliche Darstellungen der selben Sache sind. Je nachdem, was gerade praktischer ist, können wir über Ableitungsbäume oder Linksableitungen argumentieren.

Beispiel 4.50 Sei G definiert wie in Beispiel 4.46. Die beiden Ableitungsbäume für w_2 entsprechen den folgenden Linksableitungen:

$$S \Rightarrow_G SS \Rightarrow_G \mathtt{a}S \Rightarrow_G \mathtt{a}SS \Rightarrow_G \mathtt{aa}S \Rightarrow_G \mathtt{aaa},$$
$$S \Rightarrow_G SS \Rightarrow_G SSS \Rightarrow_G \mathtt{a}SS \Rightarrow_G \mathtt{aa}S \Rightarrow_G \mathtt{aaa}.$$

Dabei entspricht die erste Linksableitung dem linken Ableitungsbaum für w_2, und die zweite Linksableitung dem rechten Ableitungsbaum für w_2. ◊

Bisher haben wir uns nur mit der Eindeutigkeit (bzw. Mehrdeutigkeit) von Wörtern und kontextfreien Grammatiken beschäftigt. Ein ähnliches Konzept lässt sich auch für kontextfreie Sprachen definieren:

Definition 4.51 Eine kontextfreie Sprache L heißt **inhärent mehrdeutig**, wenn jede CFG G mit $\mathcal{L}(G) = L$ mehrdeutig ist. Existiert eine eindeutige CFG G mit $\mathcal{L}(G) = L$, so ist L **nicht inhärent mehrdeutig**.[56]

Hinweis 4.52 Denken Sie immer daran: Wir haben die Begriffe *mehrdeutig* und *eindeutig* nur für Wörter und Grammatiken definiert. Sprachen hingegen sind *inhärent mehrdeutig* oder *nicht inhärent mehrdeutig*. Bitte werfen Sie diese Begriffe nicht durcheinander.

Wir betrachten nun zuerst eine Sprache, die nicht inhärent mehrdeutig ist:

Beispiel 4.53 Die Grammatik G aus Beispiel 4.46 erzeugt die Sprache $\mathcal{L}(G) = \{\mathtt{a}\}^+$. Wir haben dort bereits gesehen, dass G mehrdeutig ist.

Allerdings lässt sich zu $\mathcal{L}(G)$ eine eindeutige kontextfreie Grammatik angeben, nämlich $G' := (\Sigma, \{S\}, P', S)$ mit $P' := \{S \to \mathtt{a}S \mid \mathtt{a}\}$. Die Sprache $\mathcal{L}(G)$ ist also nicht inhärent mehrdeutig. ◊

Wie wir in Beispiel 4.53 gesehen haben, ist es also durchaus möglich, zu einer Sprache, die nicht inhärent mehrdeutig ist, eine mehrdeutige Grammatik anzugeben. Das folgende Resultat liefert uns eine große Klasse von Sprachen, die allesamt nicht inhärent mehrdeutig sind:

Satz 4.54 *Jede reguläre Sprache ist nicht inhärent mehrdeutig.*

Beweis: Sei $L \in \mathrm{REG}$. Dann existiert ein DFA A mit $\mathcal{L}(A) = L$. Wie im Beweis von Satz 3.118 beschreiben lässt sich nun aus A eine reguläre Grammatik G mit $\mathcal{L}(G) = \mathcal{L}(A) = L$ konstruieren (insbesondere ist G auch eine kontextfreie Grammatik). Für jedes Wort $w \in L$ existiert genau ein Pfad in A, der mit w beschriftet ist. Da jeder Pfad in A einer Ableitung in G entspricht, existiert also auch genau eine Ableitung von w in G. Daher kann auch nur genau eine Linksableitung von w in G existieren, und somit auch nur genau ein Ableitungsbaum für w in G. Somit ist jedes $w \in \mathcal{L}(G)$ eindeutig, und daher auch G. Da G eine eindeutige CFG für L ist, ist L nicht inhärent mehrdeutig. □

Zusammen mit Satz 4.23 können wir also folgern, dass über unären Terminalalphabeten (d. h. $|\Sigma| = 1$) alle kontextfreien Sprachen nicht inhärent mehrdeutig sind. Hier stellt sich natürlich die Frage, ob dies auch für größere Terminalalphabete gilt, und ob überhaupt inhärent mehrdeutige Sprachen existieren. Schließlich könnte es ja prinzipiell möglich sein, dass aus Mehrdeutigkeit aus jeder kontextfreien Grammatik eliminiert werden kann. Allerdings ist dies nicht möglich, da tatsächlich inhärent mehrdeutige Sprachen existierten:

[56] Die Bezeichnung *inhärent eindeutig* (statt „nicht inhärent mehrdeutig“) klingt zwar sinnvoll, ist aber meines Wissens nach nicht gebräuchlich. In der Literatur wird gelegentlich auch die Bezeichnung *eindeutig* verwendet. Dagegen habe ich bewusst entschieden, da dies die Trennung zwischen den Begriffen für Wörter und Grammatiken auf der einen und Sprachen auf der anderen Seite verwässern würde. Beachten Sie bitte Hinweis 4.52.

Beispiel 4.55 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Wir betrachten die beiden folgenden Sprachen:

$$L_1 := \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j \in \mathbb{N}, i = j \text{ oder } j = k\},$$
$$L_2 := \{\mathtt{a}^m\mathtt{b}^m\mathtt{a}^n\mathtt{b}^n \mid m, n \in \mathbb{N}_{>0}\} \cup \{\mathtt{a}^m\mathtt{b}^n\mathtt{a}^n\mathtt{b}^m \mid m, n \in \mathbb{N}_{>0}\}.$$

Beide Sprachen sind kontextfrei und inhärent mehrdeutig. Zu zeigen, dass $L_1, L_2 \in \text{CFL}$ ist eine einfache Übung[57]. Der eigentliche Beweis (bzw. die beiden Beweise), dass die Sprachen inhärent mehrdeutig sind, ist lang und anstrengend. Wir verzichten daher darauf und verweisen auf Hopcroft und Ullman [8] (Kapitel 4.7).

Allerdings lässt sich intuitiv recht leicht veranschaulichen, warum die beiden Sprachen inhärent mehrdeutig sein müssen: Jede der beiden Sprachen ist eine Vereinigung von zwei kontextfreien Sprachen, die nicht inhärent mehrdeutig sind. Eine CFG für L_1 oder L_2 lässt sich also jeweils (mittels unserer Konstruktion aus Satz 4.16) aus den CFGs für die beiden Teilsprachen erzeugen.

Dabei haben aber die beiden Teilsprachen jeweils eine Schnittmenge, die nicht kontextfrei ist (für L_1 ist diese Schnittmenge $\{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i, j \in \mathbb{N}, i = j = k\}$, für L_2 ist sie $\{\mathtt{a}^n\mathtt{b}^n\mathtt{a}^n\mathtt{b}^n \mid n \in \mathbb{N}_{>0}\}$). Um zu verhindern, dass Wörter aus dieser Schnittmenge mehrdeutig generiert werden, müsste die Grammatik eine Art „Synchronisierung“ verwenden, die einem auch die Erzeugung dieser nicht-kontextfreien Sprachen erlauben würde. ◊

Da inhärent mehrdeutige Sprachen existieren, kann kein Umwandlungsverfahren existieren, dass zuverlässig mehrdeutige CFGs in eindeutige CFGs für die selbe Sprache umwandelt. Wir werden später sehen, dass nicht einmal entscheidbar ist, ob eine CFG eindeutig ist oder nicht. In manchen Fällen ist es allerdings möglich, eine mehrdeutige CFG in eine eindeutige CFG umzubauen. Wir betrachten dies anhand eines Beispiels:

Beispiel 4.56 Sei $\Sigma := \{0, 1, +, \cdot, (,)\}$. In Beispiel 4.13 hatten wir eine Sprache von geklammerten arithmetischen Ausdrücke über den Zahlen 0 und 1 betrachtet. Die dort definierte Grammatik G verwendete die folgenden Regeln:

$$S \to 0 \mid 1 \mid M \mid A,$$
$$A \to (S + S),$$
$$M \to (S \cdot S).$$

Unser erster Ansatz ist, die Klammern aus den Regeln für A und M wegzulassen, und stattdessen eine weitere Regel $S \to (S)$ hinzuzufügen. Allerdings ist leicht zu sehen, dass die so entstehende Grammatik mehrdeutig ist (z. B. anhand des Wortes $1 + 1 \cdot 0$). Auch wenn wir anstelle von $S \to (S)$ Regeln wie $A \to (A)$ verwenden löst dies das Problem. Wir wollen diese Mehrdeutigkeit aufhaben, indem wir der Grammatik Präzedenzregeln für die Operatoren beibringen. Wir entscheiden uns hier für das klassische Punkt vor Strich.

Der Trick ist, Regeln zu verwenden, die die Variablen entsprechend der Reihenfolge der Operatoren zu ordnen. Dazu definieren wir die CFG $G_E := (\Sigma, V_E, P_E, A)$ mit der

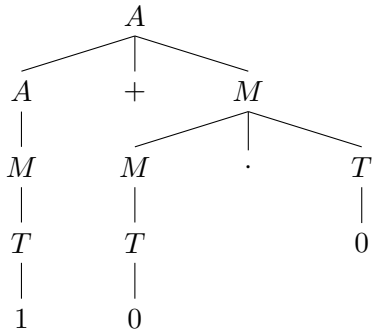
[57] Wirklich. Schreiben Sie sich die Grammatiken schnell auf einen Schmierzettel oder, falls Sie dieses Skript ausgedruckt haben, auf den Rand.

Variablenmenge $V_E := \{A, M, T\}$. Die Regelmenge P_E enthalte genau die folgenden Regeln:

$$\begin{aligned} A &\to A + M \mid M, \\ M &\to M \cdot T \mid T, \\ T &\to (A) \mid 0 \mid 1. \end{aligned}$$

Dabei ist die Variable A hauptsächlich für Additionen, M hauptsächlich für Multiplikationen und T für Klammerungen und die beiden Terminale 0 und 1 zuständig. Die einzige Art, aus M oder T eine Satzform mit A abzuleiten ist die Verwendung der Regel $T \to (A)$, die neue Klammerungen einführt. Aus M und T sind also keine ungeklammerten Vorkommen von A ableitbar. Ebenso kann aus T kein ungeklammertes Vorkommen von M abgeleitet werden. Wir haben also die Operatoren $+$, $\cdot$ und die Klammerung in eben dieser Reihenfolge geordnet.

Wie Sie nun leicht überprüfen können, ist zum Beispiel das Wort $1 + 1 \cdot 0$ in G_E eindeutig. Es ergibt sich der folgende Ableitungsbaum:



Wie Sie leicht erkennen können, enstpricht die Baumstruktur der Klammerung $1+(0\cdot 0)$. Es wird also tatsächlich die Regel „$\cdot$ vor $+$“ modelliert. Auch wenn es nun zu mühsam es, das im Detail zu überprüfen, so stellen wir doch fest, dass G_E eindeutig ist und die Präzedenzregeln darstellt. ◊

Wir können diese Vorgehensweise auch auf die Sprache der syntaktisch korrekten regulären Ausdrücke übertragen:

Beispiel 4.57 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, e, \oslash, \mathtt{(}, \mathtt{)}, \mathtt{|}, \mathtt{.}, \mathtt{*}\}$. In Beispiel 4.8 hatten wir eine CFG für die Sprache $L_{\mathrm{RX}} \subset \Sigma^*$ aller syntaktisch korrekten regulären Ausdrücke über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ betrachtet. Die dort angegebene Grammatik ist eindeutig, allerdings müssen die regulären Ausdrücke dort geklammert sein. Wir versuchen, eine CFG zu konstruieren, die auch ungeklammerte reguläre Ausdrücke zulässt (so, wie dies auch bei praktischen Anwendungen möglich ist). Analog zum ersten Ansatz in Beispiel 4.56 verwendet unser erster Ansatz die folgenden Regeln:

$$\begin{aligned} S &\to \mathtt{a} \mid \mathtt{b} \mid e \mid \oslash, \\ S &\to \mathtt{(}S\mathtt{)}, \end{aligned}$$

$$S \to S \texttt{ | } S,$$
$$S \to S \texttt{ . } S,$$
$$S \to S\texttt{*}.$$

Wir stoßen auf das gleiche Problem wie in Beispiel 4.56: Die so definierte Grammatik erzeugt zwar wirklich genau die gewünschte Sprache, allerdings ist sie nicht eindeutig, wie man zum Beispiel an dem Wort `a . b*` erkennt (dies sei Ihnen als Übung überlassen). Wir definieren nun eine CFG $G_E := (\Sigma, V_E, P_E, A_|)$ mit $V_E := \{A_|, A_., A_*, T\}$ und den folgenden Regeln:

$$A_| \to A_| \texttt{ | } A_. \mid A_.,$$
$$A_. \to A_. \texttt{ . } A_* \mid A_*,$$
$$A_* \to A_* \texttt{*} \mid \texttt{(} A_| \texttt{)} \mid T,$$
$$T \to \texttt{a} \mid \texttt{b} \mid e \mid \oslash.$$

Auch hier verwenden wir für jeden Operator eine Variable, und haben diese in die entsprechende Ordnung gebracht. Die Grammatik G_E ist eindeutig und bildet die Operatorpräzedenzen entsprechend ab. ◊

Für Programmiersprachen in der Praxis werden Präzedenzregeln allerdings oft anders umgesetzt; und zwar durch explizite Definition der Präzedenzregeln zusätzlich zur Grammatik (mehr dazu können Sie zum Beispiel in Aho et al. [1] erfahren).

4.2 Kellerautomaten

In diesem Abschnitt entwickeln wir ein Automatenmodell für die Klasse der kontextfreien Sprachen. Um die Motivation hinter diesem Modell zu verstehen, betrachten wir zuerst noch einmal endliche Automaten. Diese lassen sich auch wie folgt beschreiben: Ein endlicher Automat besteht aus einer endliche Kontrolle, und einem Eingabeband. Die endliche Kontrolle besteht aus den Zuständen und Übergängen, sie ist also der eigentliche Automat. Wir können diesen Zusammenhang wie in Abbildung 4.6 schematisch darstellen. Bei DFAs und NFAs wird das Eingabeband in jedem Schritt des Automaten um einen Schritt weiterbewegt, während ε-NFAs die Möglichkeit haben, durch ε-Übergänge die Eingabe zu ignorieren und ihren Zustand ändern können, ohne das Eingabeband bewegen zu müssen.

Die wohl wichtigste Eigenschaft von endlichen Automaten ist, dass sie mit einem endlichen Speicher arbeiten müssen, da sie nur in ihren Zuständen speichern können. Ein Automatenmodell, das alle kontextfreien Sprachen erkennen kann, muss daher über eine Art von Speicher verfügen. Es liegt daher nahe, endliche Automaten um einen Speicher zu erweitern. Allerdings ist hierbei Vorsicht geboten, denn Turing-Maschinen sind nichts anderes als endliche Automaten mit einem zusätzlichen Speicherband. Wenn wir unseren Speicher nicht einschränken, erhalten wir also schnell ein Modell, das nicht mehr handhabbar ist.

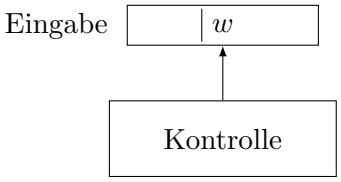


Abbildung 4.6: Eine schematische Darstellung eines endlichen Automaten. Der noch nicht gelesene Teil des Eingabebands enthält das Wort w, der Zustand ist diesem Bild nicht zu entnehmen.

Die Automaten, mit denen wir uns im Folgenden beschäftigen werden, heißen **Kellerautomaten**, kurz **PDAs** (vom Englischen *pushdown automaton*). Diese sind im Prinzip ε-NFAs, die um einen **Kellerspeicher** oder **Keller** erweitert wurden (dieser wird auch **Stapel**, **Stack** oder **Pushdown** genannt). In diesem ist eine Folge von Kellersymbolen gespeichert. Auf den Keller kann mit zwei Operationen zugegriffen werden:

- „Auskellern“, hier wird das oberste Symbol vom Keller genommen, und
- „Einkellern“, hier werden endlich viele Symbole oben auf den Keller gelegt.

Der Keller arbeitet also nach dem *Last-In-First-Out-Prinzip*, darf aber beliebig voll sein. Ein PDA betrachtet in jedem Schritt nicht nur den aktuellen Buchstaben der Eingabe (oder ignoriert ihn durch einen ε-Übergang), sondern nimmt gleichzeitig auch den den obersten Buchstaben vom Keller. Aus seinem aktuellen Zustand, dem aktuellen Buchstaben der Eingabe und dem ausgekellerten Buchstaben werden der Folgezustand und eine (möglicherweise leere) Folge von Kellersymbolen berechnet, die eingekellert werden. Wie gewohnt definieren wir zuerst das Automatenmodell selbst; das eigentliche Verhalten werden wir in Definition 4.59 betrachten.

Definition 4.58 Ein **Kellerautomat (PDA)** A über einem Alphabet Σ wird definiert durch:

1. ein Alphabet Γ von **Kellersymbolen**
2. eine nicht-leere, endliche Menge Q von **Zuständen**,
3. eine Funktion $\delta : (Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \to \mathcal{P}_F(Q \times \Gamma^*)$ (die **Übergangsrelation**[58]),
4. einen Zustand $q_0 \in Q$ (der **Startzustand**),
5. einem Kellersymbol $K_0 \in \Gamma$ (das **Startsymbol**),
6. eine Menge $F \subseteq Q$ von **akzeptierenden Zuständen**.

Wir schreiben dies als $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, F)$. Eine **Konfiguration** von A ist ein 3-Tupel aus $Q \times \Sigma^* \times \Gamma^*$. Wir bezeichnen die Komponenten einer Konfiguration als

Zustand, **verbleibende Eingabe** und **Kellerinhalt**. Für jedes $w \in \Sigma^*$ bezeichnen wir die Konfiguration (q_0, w, K_0) als **Startkonfiguration (mit Eingabe w)**.

In einer Startkonfiguration (q_0, w, K_0) ist der PDA also im Startzustand q_0, auf der Eingabe befindet sich das Wort w, und im Keller steht nur das Startsymbol K_0.

Eine Konfiguration der Form $(q, w, A\beta)$ können wir folgendermaßen interpretieren: Der PDA ist im Zustand q, die noch zu lesende Eingabe ist w, und der Keller enthält das Wort $A\beta$, wobei A der oberste Buchstabe ist. Eine schematische Darstellung dieser Konfiguration finden Sie in Abbildung 4.7.

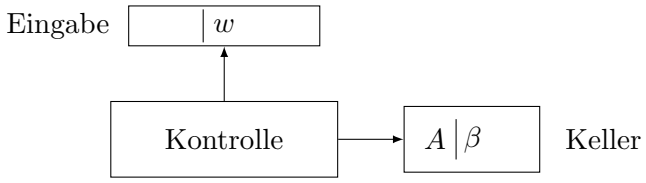
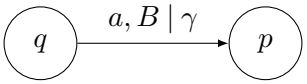


Abbildung 4.7: Eine schematische Darstellung eines PDA in der Konfiguration $(q, w, A\beta)$. Der noch nicht gelesene Teil des Eingabebands enthält das Wort w, der Keller enthält $A\beta$ (mit oberstem bzw. vordersten Buchstaben A), und auch hier ist der Zustand q dem Bild nicht zu entnehmen.

Die graphische Darstellung eines PDA ist fast identisch mit der eines ε-NFA. Der einzige Unterschied liegt in der Beschriftung der Kanten zwischen den Knoten. Ist $(p, \gamma) \in \delta(q, a, B)$, so beschriften wir die Kante von q nach p mit $a, B \mid \gamma$:



Auch hier können mehrere Kanten zwischen den selben Zuständen zu einer Kante zusammengefasst werden. Um das Verhalten (und insbesondere die Akzeptanz) von PDAs zu definieren, benötigen wir noch einige Definitionen:

Definition 4.59 Sei $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, F)$ ein PDA. Wir definieren die Relation $\vdash_A$ auf Konfigurationen von A wie folgt: Für alle $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$, $A \in \Gamma$, $\beta \in \Gamma^*$ gilt:

1. $(q, aw, A\beta) \vdash_A (p, w, \gamma\beta)$, wenn $(p, \gamma) \in \delta(q, a, A)$, und
2. $(q, w, A\beta) \vdash_A (p, w, \gamma\beta)$, wenn $(p, \gamma) \in \delta(q, \varepsilon, A)$.

Wir erweitern die Relation $\vdash_A$ für jedes $n \in \mathbb{N}$ zu einer Relation $\vdash_A^n$ für alle Konfigurationen C, C' von A wie folgt:

[58] Hier gelten die gleichen Hinweise zu den Begriffen Funktion und Relation wie in Definition 3.52.

1. $C \vdash_A^0 C$, und
2. $C \vdash_A^{n+1} C'$, wenn ein $n \in \mathbb{N}$ und eine Konfiguration C'' von A existieren, so dass $C \vdash_A^n C''$ und $C'' \vdash_A C'$.

Die **Erreichbarkeitsrelation** $\vdash_A^*$ definieren wir für alle Konfigurationen C, C' von A durch:

$$C \vdash_A^* C', \text{ wenn ein } n \in \mathbb{N} \text{ existiert, für das } C \vdash_A^n C'.$$

Gilt $C \vdash_A^* C'$, so sagen wir, dass C' **von** C **erreichbar** ist. Eine Konfiguration C heißt **erreichbar**, wenn sie von einer Startkonfiguration aus erreichbar ist. Eine **Berechnung** (mit Eingabe $w \in \Sigma^*$) ist eine Folge $C_0, \ldots, C_n$ von Konfigurationen von A mit $\gamma_0 = (q_0, w, K_0)$ und $C_i \vdash_A C_{i+1}$ für $0 \leq i < n$.

Kellerautomaten arbeiten also nichtdeterministisch. In jedem Schritt wird das oberste Symbol vom Keller genommen. Dann kann der PDA ein Eingabesymbol lesen (Übergänge der Form $\delta(q, a, B)$) oder darauf verzichten (Übergänge der Form $\delta(q, \varepsilon, B)$ – analog zum ε-NFA bezeichnen wir als **ε-Übergänge**). Anschließend wird ein Wort γ auf den Keller gelegt, und der PDA wechselt in die nächste Konfiguration. Um die von einem PDA akzeptierte Sprache zu definieren, gibt es zwei verschiedene Möglichkeiten:

Definition 4.60 Sei $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, F)$ ein PDA. Die von A akzeptierten Sprachen sind definiert als:

$$\mathcal{L}_Z(A) := \{w \in \Sigma^* \mid (q_0, w, K_0) \vdash_A^* (q, \varepsilon, \gamma), q \in F, \gamma \in \Gamma^*\},$$
$$\mathcal{L}_K(A) := \{w \in \Sigma^* \mid (q_0, w, K_0) \vdash_A^* (q, \varepsilon, \varepsilon), q \in Q\}.$$

Dabei ist $\mathcal{L}_Z(A)$ die Sprache, die von A mit akzeptierenden Zuständen akzeptiert wird, und $\mathcal{L}_K(A)$ die Sprache, die von A mit leerem Keller akzeptiert wird. Wir definieren außerdem $\mathcal{L}(A) := \mathcal{L}_Z(A)$.

Bevor wir geeignete Beispiele betrachten, halten wir noch eine Sammlung hilfreiche Beobachtungen fest, die direkt aus den Definitionen folgen:

Hinweis 4.61 Eine Reihe von Hinweisen, die im Umgang mit PDAs hilfreich oder zu beachten sind:

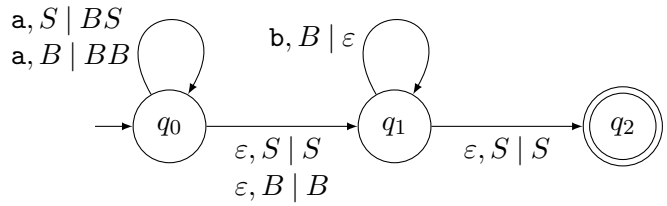
- Wenn Sie das Akzeptanzverhalten nicht angeben, gehen wir von Akzeptanz durch Zustände aus (da $\mathcal{L}(A) := \mathcal{L}_Z(A)$).
- Im Allgemeinen gilt *nicht* $\mathcal{L}_K(A) = \mathcal{L}_Z(A)$, allerdings kann dies durchaus vorkommen.
- Bei Akzeptanz mit leerem Keller spielen die akzeptierenden Zustände keine

Rolle, F kann also leer sein. Um nicht unnötig zu verwirren empfiehlt es sich in diesem Fall sogar $F := \emptyset$ zu wählen.

- Bei beiden Akzteptanzverhalten akzeptiert der PDA nur, wenn das gesamte Eingabewort gelesen worden ist. Abgesehen davon hat ein PDA aber keine Möglichkeit das Ende seiner Eingabe zu erkennen.
- Wenn der Keller leer ist, kann ein PDA nicht weiterarbeiten. Falls die Eingabe noch nicht abgearbeitet wurde, stürzt der Automat ab (und akzeptiert nicht). Wurde die Eingabe komplett abgearbeitet, so hängt das Verhalten vom Akzeptanzmodus ab: Bei Akzeptanz mit leerem Keller akzeptiert der PDA auf jeden Fall, bei Akzeptanz mit Zuständen akzeptiert er nur, wenn er in einem akzeptierenden Zustand ist.
- Da PDAs ε-Übergänge verwenden können (wenn $\delta(q, \varepsilon, B) \neq \emptyset$), ist es möglich in Endlosschleifen zu geraten.
- Die Alphabete Σ und Γ müssen nicht disjunkt sein. Im Allgemeinen trägt es aber zu besserer Lesbarkeit bei, wenn Sie mit disjunkten Alphabeten arbeiten.

Wir betrachten nun Beispiele für die Arbeitsweise und das Akzeptanzverhalten von Kellerautomaten:

Beispiel 4.62 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir definieren den folgenden PDA A mit Kelleralphabet $\Gamma := \{S, B\}$ und Startsymbol S:



Wir betrachten nun die Arbeitsweise des PDA A. Eine graphische Veranschaulichung finden Sie in Abbildung 4.8. Zuerst liest A im Zustand q_0 Vorkommen von $\mathtt{a}$ ein. Für jedes Vorkommen legt er ein B auf den Stapel. Streng genommen nimmt er im ersten Schritt das Symbol S vom Stapel (da jeder PDA in jede Schritt ein Symbol auskellern muss) und legt dann BS auf den Stapel, was insgesamt auf das Hinzufügen von einem B hinausläuft. In den weiteren Schritten wird dann für jedes gelesene $\mathtt{a}$ ein B auf den Stapel gelegt.

An einem beliebigen Punkt wechselt A nichtdeterministisch in den Zustand q_1, ohne den Keller zu verändern (auch hier wird eigentlich das oberste Symbol ausgekellert und gleich wieder eingekellert, so dass insgesamt keine Veränderung geschieht). Im Zustand q_1 wird dann für jedes eingelesene $\mathtt{b}$ ein B vom Keller entfernt. Wenn alle B entfernt wurden, liegt wieder das Symbol S oben auf dem Keller. Dann kann A in den akzeptierenden Zustand q_2 wechseln. Es gilt also $\mathcal{L}(A) = \mathcal{L}_Z(A) = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$.

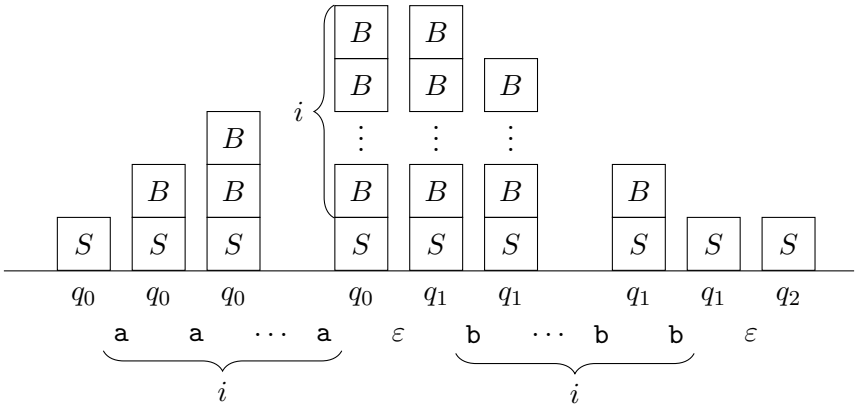


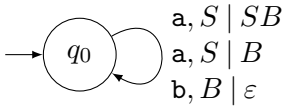
Abbildung 4.8: Eine Veranschaulichung der Arbeitsweise des PDA A aus Beispiel 4.62. Im Zustand q_0 werden für i Vorkommen von a insgesamt i Vorkommen von B auf den Stapel gelegt. Dann wechselt a nichtdeterministisch in den Zustand q_1. Dort wird nun für jedes b ein B vom Stapel genommen. Sind alle B aufgebraucht, wechselt A in den Zustand q_2.

Die einzelnen Konfigurationen der akzeptierenden Berechnung für A sehen dabei folgendermaßen aus:

$$(q_0, \mathtt{a}^i\mathtt{b}^i, S) \vdash_A^i (q_0, \mathtt{b}^i, B^iS) \vdash_A (q_1, \mathtt{b}^i, B^iS) \vdash_A^i (q_1, \varepsilon, S) \vdash_A (q_2, \varepsilon, S).$$

Die Arbeitsweise des PDA A lässt sich also in zwei Phasen unterteilen, die durch die Zustände unterschieden werden. Im Zustand q_0 (der ersten Phase) werden die Vorkommen von a gezählt, indem entsprechend viele B auf den Keller gelegt werden. In der zweiten Phase werden dann Vorkommen von b gezählt, indem die B wieder vom Keller entfernt werden.

Da das Symbol S nie vom Keller entfernt wird, kann A den Keller niemals leer räumen. Also ist $\mathcal{L}_K(A) = \emptyset$. Wenn wir aber den Übergang von q_1 nach q_2 in $\varepsilon, S \mid \varepsilon$ ändern (an Stelle von $\varepsilon, S \mid S$), so erhalten wir einen PDA A' mit $\mathcal{L}_K(A') = \mathcal{L}_Z(A') = \mathcal{L}(A)$. Stattdessen können wir aber auch den folgenden PDA A_K mit Kelleralphabet $\Gamma_K := \{S, B\}$ und Startsymbol S definieren:



Ähnlich wie der PDA A, den wir zuerst betrachtet haben, arbeitet A_K in zwei Phasen: In der ersten Phase werden Vorkommen von a gezählt indem B auf den Keller gelegt werden; in der zweiten Phase werden Vorkommen von b gezählt, und zwar durch Entfernen von B. Während bei A die verschiedenen Phasen an den Zuständen zu erkennen sind, geschieht dies bei A_K durch das oberste Kellersymbol. In der ersten Phase (dem Hoch-Zählen)

ist stets das Symbol S oben auf dem Keller (in jedem Schritt wird S ausgekellert und gleich darauf SB eingekellert). Wie auch A rät A_K nichtdeterministisch, wann in die zweite Phase (dem Herunter-Zählen) gewechselt wird und legt das S nicht wieder auf den Keller. An diesem Punkt wurden i Vorkommen von `a` eingelesen, und der Keller enthält i Vorkommen des Symbols B. Jetzt liest A_K Vorkommen von `b` ein und entfernt für jedes von diesen ein B vom Keller. Dieser ist leer, wenn i Vorkommen von `b` eingelesen wurden. Wenn nun das Eingabewort komplett eingelesen wurde akzeptiert A_K. Es gilt:

$$(q_0, \mathtt{a}^i\mathtt{b}^i, S) \vdash_{A_K}^{i-1} (q_0, \mathtt{a}\mathtt{b}^i, SB^{i-1}) \vdash_{A_K} (q_0, \mathtt{b}^i, B^i) \vdash_{A_K}^{i} (q_0, \varepsilon, \varepsilon).$$

Eine graphische Illustration der Arbeitsweise von A_K finden Sie in Abbildung 4.9.

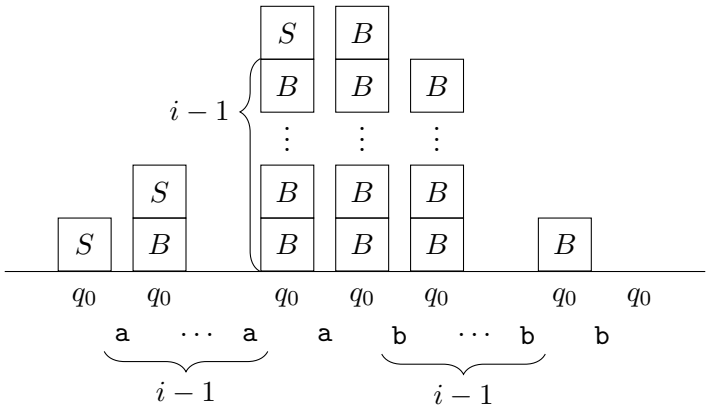


Abbildung 4.9: Eine Veranschaulichung der Arbeitsweise des PDA A_K aus Beispiel 4.62. Solange das oberste Kellersymbol S ist, legt A_K für jedes `a` ein B unter das S. An einer Stelle (nach i Vorkommen von `a`) beschließt A_K nichtdeterministisch, das S nicht auf den Keller zu legen und wechselt so in die zweite Phase, in der anhand des Kellers i Vorkommen von `b` abgezählt werden.

Sowohl A als auch A_K können so umgebaut werden, dass der Wechsel von der ersten in die zweite Phase ohne nichtdeterministisches Raten geschieht. Dies sei Ihnen als Übung überlassen. ◊

In diesem Beispiel wurde der Keller der PDAs überwiegend als Zähler benutzt (im Fall von A_K wurde auch die aktuelle Phase darin codiert). Allerdings ist es auch möglich, mehr Informationen als eine Anzahl und die aktuelle Phase im Keller zu speichern; zum Beispiel können wir ganze Wörter im Keller aufbewahren. Wir betrachten dies anhand eines weiteren Beispiels:

Beispiel 4.63 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir betrachten den PDA A mit Kelleralphabet $\Gamma := \{S, A, B\}$ und Startsymbol S, der wie in Abbildung 4.10 definiert ist.

Es gilt: $\mathcal{L}_Z(A) = \mathcal{L}_K(A) = \{ww^R \mid w \in \Sigma^*\}$; warum, werden wir im Folgenden betrachten. Die Arbeitsweise von A lässt sich in zwei Phasen zerlegen: In der ersten Phase wir w eingelesen, und auf dem Keller wird w^R abgelegt (`a` wird als A gespeichert, `b` als B).

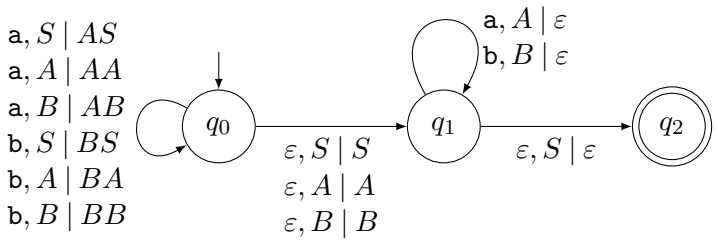


Abbildung 4.10: Der in Beispiel 4.63 verwendete PDA A.

Der PDA rät nichtdeterministisch die Mitte der Eingabe (also den Übergang zwischen w und w^R), indem er in den Zustand q_1 und somit in die zweite Phase wechselt. Hier wird überprüft, ob die restliche Eingabe w^R entspricht, indem nur Terminale eingelesen werden können, die dem obersten Kellersymbol entsprechen (also $\mathtt{a}$ für A und $\mathtt{b}$ für B). Ist genau w^R abgearbeitet worden, so liegt auf dem Keller das Symbol S. Der PDA wechselt nun in den akzeptierenden Zustand q_2 und nimmt S vom Keller. ◊

In Beispiel 4.62 und Beispiel 4.63 haben wir Sprachen gesehen, die sich mit jedem der beiden Akzeptanzverhalten durch einen PDA definieren lassen. Dabei stellt sich natürlich die Frage, ob die Wahl des Akzeptanzverhaltens einen Einfluss auf die Ausdrucksstärke von PDAs hat. Wir werden gleich zeigen, dass dies nicht der Fall: Jede Sprache, die von einem PDA mit einem der beiden Akzeptanzverhalten akzeptiert wird, wird auch von einem PDA mit dem anderen Akzeptanzverhalten akzeptiert. Es gilt:

Satz 4.64 *Sei A ein PDA. Dann existieren PDAs A_K und A_Z mit:*

- $\mathcal{L}_Z(A) = \mathcal{L}_K(A_K)$, *und*
- $\mathcal{L}_K(A) = \mathcal{L}_Z(A_Z)$.

Beweis: Sei $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, F)$ ein PDA. Wir beweisen die beiden Behauptungen des Satzes getrennt.

Behauptung 1 *Es existiert ein PDA A_K mit $\mathcal{L}_K(A_K) = \mathcal{L}_Z(A)$.*

Beweisidee: Wir konstruieren A_K, indem wir zu A einen zusätzlichen Zustand q_L hinzufügen, in dem der Keller leergeräumt wird. Von jedem Zustand, in dem A akzeptiert, fügen wir einen ε-Übergang zu q_L hinzu. Um zu verhindern, dass A_K versehentlich akzeptiert wenn A den Keller leerräumen sollte, fügen wir ein zusätzliches Kellersymbol hinzu, das nur in q_L entfernt werden kann. Dafür benötigen wir einen neuen Startzustand q_0^K. □

Beweis: Wir definieren $A_K := (\Sigma, \Gamma_K, Q_K, \delta_K, q_0^K, K_K, \emptyset)$, wobei

- $\Gamma_K := \Gamma \cup \{K_K\}$ (wobei $K_K \notin \Gamma$),
- $Q_K := Q \cup \{q_0^K, q_L\}$ (wobei $q_0^K, q_L \notin Q$),
- für alle $q \in Q$, alle $a \in (\Sigma \cup \{\varepsilon\})$ und alle $B \in \Gamma$ ist $\delta_K(q, a, B) := \delta(q, a, B)$,

- für alle $q \in F$ und alle $A \in \Gamma_K$ ist $\delta_K(q, \varepsilon, A) := \{(q_L, A)\}$,
- es gilt $\delta_K(q_0^K, \varepsilon, K_K) := \{(q_0, K_0 K_K)\}$,
- außerdem ist $\delta_K(q_L, \varepsilon, A) := \{(q_L, \varepsilon)\}$ für alle $A \in \Gamma_K$

Da K_K nur im Zustand q_L vom Keller entfernt werden kann, gilt $w \in \mathcal{L}(A_K)$ für alle $w \in \Sigma^*$ genau dann, wenn

$$(q_0^K, w, K_K) \vdash_{A_K}^* (q_L, \varepsilon, \varepsilon).$$

Nach Definition von A_K kann A_K im Zustand q_0^K nichts tun, außer ein K_0 auf das K_K zu legen und in den Zustand q_0 zu wechseln; also gilt dies genau dann, wenn

$$(q_0^K, w, K_K) \vdash_{A_K} (q_0, w, K_0 K_K) \vdash_{A_K}^* (q_L, \varepsilon, \varepsilon).$$

Da in q_L keine Eingabe mehr verarbeitet wird, ist dies genau dann der Fall, wenn ein $q_F \in F$ und $\gamma_K \in \Gamma^*$ existieren mit

$$(q_0^K, w, K_K) \vdash_{A_K} (q_0, w, K_0 K_K) \vdash_{A_K}^* (q_F, \varepsilon, \gamma K_K) \vdash_{A_K} (q_L, \varepsilon, \gamma K_K) \vdash_{A_K}^* (q_L, \varepsilon, \varepsilon).$$

Da A_K auf den Zuständen aus Q und den Kellersymbolen aus Γ genauso arbeitet wie A, existieren ein solcher Zustand q_F und ein solches Kellerwort γ genau dann, wenn

$$(q_0, w, K_0) \vdash_A^* (q_F, \varepsilon, \gamma).$$

Ein $q_F \in F$ und ein $\gamma \in \Gamma^*$ mit $(q_0, w, K_0) \vdash_A^* (q_F, \varepsilon, \gamma)$ existieren genau dann, wenn $w \in \mathcal{L}_Z(A)$. Somit ist $\mathcal{L}_K(A_K) = \mathcal{L}_Z(A)$. □(Behauptung 1)

Behauptung 2 *Es existiert ein PDA A_Z mit $\mathcal{L}_Z(A_Z) = \mathcal{L}_K(A)$.*

Beweisidee: Wir konstruieren A_Z aus A, indem wir zuerst ein neues Kellersymbol K_L einführen, das wir unter K_0 legen (wie K_K im Beweis der vorigen Behauptung). Der PDA A_Z arbeitet nun genau wie A. Genau dann, wenn A einen leeren Keller hat, ist bei A_Z das Symbol K_L auf dem Keller. In diesem Fall wechselt A_Z in den neuen akzeptierenden Zustand q_A. □

Beweis: Wir definieren $A_Z := (\Sigma, \Gamma_Z, Q_Z, \delta_Z, q_0^Z, K_L, F_Z)$, wobei

- $\Gamma_Z := \Gamma \cup \{K_L\}$ (wobei $K_L \notin \Gamma$),
- $Q_Z := Q \cup \{q_0^Z, q_A\}$ (wobei $q_0^Z, q_A \notin Q$),
- für alle $q \in Q$, alle $a \in (\Sigma \cup \{\varepsilon\})$ und alle $B \in \Gamma$ ist $\delta_Z(q, a, B) := \delta(q, a, B)$,
- für alle $q \in Q$ ist $\delta_Z(q, \varepsilon, K_L) = \{(q_A, K_L)\}$,
- $\delta_Z(q_0^Z, \varepsilon, K_L) := \{(q_0, K_0 K_L)\}$,
- $F_Z := \{q_A\}$.

Für alle $w \in \Sigma^*$ gilt $w \in \mathcal{L}_Z(A_Z)$ genau dann, wenn

$$(q_0^Z, w, K_L) \vdash_{A_Z}^* (q_A, \varepsilon, K_L).$$

Da A_Z in q_0^Z nichts weiter tun kann, als K_0 auf den Keller zu legen und in den Zustand q_0 zu wechseln, gilt dies genau dann, wenn

$$(q_0^Z, w, K_L) \vdash_{A_Z} (q_0, w, K_0K_L) \vdash_{A_Z}^* (q_A, \varepsilon, K_L).$$

Da der Zustand q_A nur durch einen Übergang der Form $\delta_Z(q, \varepsilon, K_L) = \{(q_A, K_L)\}$ erreicht werden kann, gilt dies genau dann, wenn

$$(q_0^Z, w, K_L) \vdash_{A_Z} (q_0, w, K_0K_L) \vdash_{A_Z}^* (q, \varepsilon, K_L) \vdash_{A_Z} (q_A, \varepsilon, K_L)$$

für ein $q \in Q$. Da A_Z auf den Zuständen von Q und den Kellersymbolen aus Γ genauso arbeitet wie A, gilt dies genau dann, wenn

$$(q_0, w, K_0) \vdash_A^* (q, \varepsilon, \varepsilon)$$

für ein $q \in Q$. Dies ist genau dann der Fall, wenn $w \in \mathcal{L}_K(A)$. Es gilt also wie gewünscht $\mathcal{L}_Z(A_Z) = \mathcal{L}_K(A)$. □(Behauptung 2)

Aus diesen beiden Behauptungen folgt sofort die Korrektheit des Satzes. □

Wenn wir mit PDAs arbeiten, können wir uns also dasjenige Akzeptanzverhalten aussuchen, das für die jeweilige Aufgabenstellung am besten geeignet ist.

4.2.1 Kellerautomaten und kontextfreie Sprachen

Nun haben wir auch das nötige Handwerkszeug, um die Klasse der Sprachen, die von PDAs akzeptiert werden, mit der Klasse der kontextfreien Sprachen zu vergleichen. Unser Ziel bei der Definition der Kellerautomaten war es ein Automatenmodell für die Klasse CFL zu erhalten. Glücklicherweise erfüllen die Kellerautomaten diesen Zweck auch tatsächlich:

Satz 4.65 *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$. Es gilt:*

L ist kontextfrei genau dann, wenn ein PDA A existiert mit $\mathcal{L}(A) = L$.

Wir zerlegen den Beweis von Satz 4.65 in zwei Teilbehauptungen: Die Hin-Richtung zeigen wir in Lemma 4.66, die Rück-Richtung in Lemma 4.70. In beiden Fällen sind die Beweise konstruktiv: Wir zeigen sowohl, wie man eine CFG in einen PDA umwandeln kann, als auch die andere Richtung.

Lemma 4.66 *Sei $L \in$ CFL. Dann existiert ein PDA A mit $\mathcal{L}(A) = L$.*

Beweisidee: Wir zeigen, wie zu einer CFG G in CNF ein PDA A konstruiert werden kann, für den $\mathcal{L}_K(A) = \mathcal{L}(G)$. Dabei hat A genau einen Zustand; die Übergänge sind so gewählt, dass jeder Schritt von A einer Linksableitung von G entspricht. Im Keller von A wird dabei in jedem Schritt der Teil der Satzform gespeichert, der aus Variablen besteht. □

Beweis: Sei Σ ein Alphabet, und sei $L \subseteq \Sigma^*$ eine kontextfreie Sprache. Dann existiert[59] eine CFG $G := (\Sigma, V, P, S)$ in Chomsky-Normalform mit $\mathcal{L}(G) = L$. Wir definieren nun $A := (\Sigma, \Gamma, Q, \delta, q, S, \emptyset)$, wobei

- $\Gamma := V$,
- $Q := \{q\}$,
- für alle $a \in \Sigma$ und alle $B \in \Gamma$ ist

$$\delta(q, a, B) := \begin{cases} \{(q, \varepsilon)\} & \text{falls } B \to a \in P, \\ \emptyset & \text{falls } B \to a \notin P, \end{cases}$$

- für alle $A \in \Gamma$ ist

$$\delta(q, \varepsilon, A) := \{BC \mid A \to BC \in P\}.$$

- Ist $S \to \varepsilon \in P$, so enthält $\delta(q, \varepsilon, S)$ außerdem (q, ε).

Wir führen den Korrektheitsbeweis für den Fall durch, dass $\varepsilon \notin L$. Die Korrektheit für den Fall $\varepsilon \in L$ folgt danach unmittelbar aus der CNF von G und der Definition von A.

Um zu zeigen, dass G auch wirklich korrekt ist, beweisen wir die folgende Behauptung:

Behauptung 1 *Für alle $w \in \Sigma^+$, alle $A \in \Gamma$ und alle $n \in \mathbb{N}_{>0}$ gilt $(q, w, A) \vdash_A^n (q, \varepsilon, \varepsilon)$ genau dann, wenn $A \Rightarrow_G^n w$.*

Beweis: Wir zeigen dies durch Induktion über n.

Induktionsanfang: Sei $n = 1$.

Behauptung: Für alle $w \in \Sigma^+$ und alle $A \in \Gamma$ gilt $(q, w, A) \vdash_A (q, \varepsilon, \varepsilon)$ genau dann, wenn eine Linksableitung existiert mit $A \Rightarrow_G w$.

Beweis: Wir beginnen mit der Hin-Richtung: Angenommen, $(q, w, A) \vdash_A (q, \varepsilon, \varepsilon)$. Nach Definition von A gilt $w \in \Sigma$, und P enthält eine Regel $A \to w$. Es gilt also $A \Rightarrow_G w$. Die Rückrichtung funktioniert analog: Angenommen, $A \Rightarrow_G w$. Dann ist $w \in \Sigma$, und $A \to w \in P$. Also ist $\delta(q, w, A) = \{(q, \varepsilon)\}$, und es gilt $(q, w, A) \vdash_A (q, \varepsilon, \varepsilon)$.

Induktionsschritt: Sei $n \in \mathbb{N}_{>0}$.

Induktionsannahme: Die Behauptung gelte für alle $i \leq n$.

Behauptung: Für alle $w \in \Sigma^+$ und alle $A \in \Gamma$ gilt $(q, w, A) \vdash_A^{n+1} (q, \varepsilon, \varepsilon)$ genau dann, wenn eine Linksableitung existiert mit $A \Rightarrow_G^{n+1} w$.

Beweis: Wir beginnen mit der Hin-Richtung: Sei $(q, w, A) \vdash_A^{n+1} (q, \varepsilon, \varepsilon)$. Dann existiert eine Konfiguration (q, w', γ) von A mit $(q, w, A) \vdash (q, w', \gamma) \vdash_A^n (q, \varepsilon, \varepsilon)$. Nun muss $w' = w$ gelten, denn andernfalls gilt $\gamma = \varepsilon$, und die Berechnung ist vorzeitig beendet. Also muss A im ersten Schritt dieser Berechnung einen ε-Übergang ausführen und zwei Symbole

[59] Zur Erinnerung: Wir haben in Satz 4.33 gezeigt, dass jede CFG in eine äquivalente CFG in CNF umgewandelt werden kann.

auf den Keller legen. Es existieren also $B, C \in V$ mit $\gamma = BC$, und P enthält die Regel $A \to BC$. Um die Konfiguration $(q, \varepsilon, \varepsilon)$ erreichen zu können, muss der PDA A nun zuerst B und dann C abarbeiten. Wir können also w in Wörter $u, v \in \Sigma^+$ zerlegen, so dass $w = uv$, $(q, uv, BC) \vdash_A^* (q, v, C) \vdash_A^* (q, \varepsilon, \varepsilon)$. Wir wählen u so, dass das untere Kellersymbol C bei der Berechnung $(q, uv, BC) \vdash_A^* (q, v, C)$ keine Rolle spielt. Es gilt also $(q, u, B) \vdash_A^* (q, \varepsilon, \varepsilon)$. Es existieren also $i, j \in \mathbb{N}_{>0}$ mit $1 \leq i, j < n$ und $i + j = n$, so dass

$$(q, u, B) \vdash_A^i (q, \varepsilon, \varepsilon) \qquad \text{und} \qquad (q, u, C) \vdash_A^j (q, \varepsilon, \varepsilon).$$

Gemäß der Induktionsannahme gilt nun

$$B \Rightarrow_G^i u \qquad \text{und} \qquad C \Rightarrow_G^j v.$$

Wie wir eingangs festgestellt haben, ist $A \to BC$ in P und daher auch $A \Rightarrow_G BC$. Durch Zusammensetzen dieser Ableitungen erhalten wir

$$A \Rightarrow_G BC \Rightarrow_G^i uC \Rightarrow_G^j uv = w$$

und somit $A \Rightarrow_G^{n+1} w$, da $n = i + j$. Die Hin-Richtung trifft also zu.

Für den Beweis der Rück-Richtung gelte $A \Rightarrow_G^{n+1} w$. Da $n + 1 \geq 2$ müssen wir im ersten Schritt eine Regel $A \to BC$ (mit $B, C \in V$) verwenden, da sonst nicht weiter abgeleitet werden kann. Also existieren Wörter $u, v \in \Sigma^+$ und $i, j \in \mathbb{N}_{>0}$ mit $uv = w$, $i + j = n$ und

$$A \Rightarrow_G BC \Rightarrow_G^i uC \Rightarrow_G^j uv.$$

Gemäß der Induktionsannahme gilt $(q, u, B) \vdash_A^i (q, \varepsilon, \varepsilon)$ und $(q, v, C) \vdash_A^j (q, \varepsilon, \varepsilon)$. Wir können nun bei der ersten dieser beiden Berechnungen C unter den Keller von A legen, ohne die Berechnung zu verändern. Es gilt also $(q, u, BC) \vdash_A^i (q, \varepsilon, C)$. Somit gilt

$$(q, uv, A) \vdash_A (q, uv, BC) \vdash_A^i (q, v, C) \vdash_A^j (q, \varepsilon, \varepsilon).$$

Somit ist $(q, w, A) \vdash_A^{n+1} (q, \varepsilon, \varepsilon)$, da $uv = w$ und $n = i + j$. □(Behauptung 1)

Aufgrund von Behauptung 1 gilt $(q, w, S) \vdash_A^* (q, \varepsilon, \varepsilon)$ für alle $w \in \Sigma^*$ genau dann, wenn $S \Rightarrow_G^* w$. Also ist $\mathcal{L}_K(A) = \mathcal{L}(G)$. Somit kann jede kontextfreie Sprache von einem PDA erkannt werden. □

Die Konstruktion aus dem Beweis von Lemma 4.66 lässt sich nicht nur auf Grammatiken in CNF anwenden, sondern auch auf alle CFGs, bei denen jede Regel die Form

$$A \to \Gamma^* \qquad \text{oder} \qquad A \to \Sigma \cdot \Gamma^*$$

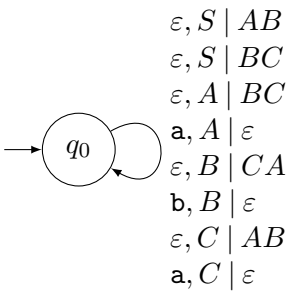
haben. Hierbei muss der Korrektheitsbeweis nur leicht angepasst werden. Mit etwas Zusatzaufwand lässt sich der Beweis sogar auf beliebige CFGs übertragen: Hierzu wählt man $\Gamma := (\Sigma \cup V)$ und passt die Übergänge entsprechend an. Die Details der Konstruktion und ihre Korrektheit seien Ihnen als Übung überlassen.

Da jeder Schritt des konstruierten PDA einem Linksableitungsschritt der Grammatik entspricht, können wir die Berechnungen von PDAs zur Syntaxanalyse (also der Konstruktion von Ableitungsbäumen) verwenden. Durch die nichtdeterministische Arbeitsweise der PDAs ist dies allerdings im Allgemeinen kein praktischer Ansatz für beliebige CFGs. Daher müssen die Grammatiken und PDAs noch beschränkt werden. Wir werden uns in Abschnitt 4.2.2 ein wenig ausführlicher mit dieser Thematik befassen.

Beispiel 4.67 Sei $\Sigma := \mathtt{a}, \mathtt{b}, \mathtt{c}$. Die Grammatik $G := (\Sigma, V, P, S)$ mit $V := \{S, A, B, C\}$ sei definiert wie in Beispiel 4.39. Die Regelmenge P enthält also die folgenden Regeln:

$$\begin{aligned} S &\to AB \mid BC, \\ A &\to BC \mid \mathtt{a}, \\ B &\to CA \mid \mathtt{b}, \\ C &\to AB \mid \mathtt{a}. \end{aligned}$$

Wir konstruieren nun einen PDA A mit Kelleralphabet $\Gamma = V$ und Startsymbol $K_0 = S$ wie folgt:



Es gilt $\mathcal{L}(G) = \mathcal{L}_K(A)$. Ja, das ist wirklich immer so einfach, wenn die Grammatik in CNF ist. ◊

Durch den Beweis von Lemma 4.66 können wir auch unmittelbar die Zahl der Zustände eines PDA begrenzen:

Korollar 4.68 *Sei L eine kontextfreie Sprache. Dann existieren PDAs A_K und A_Z mit $\mathcal{L}_K(A_K) = \mathcal{L}_Z(A_Z) = L$, außerdem hat A_K genau einen Zustand, und A_Z hat genau drei Zustände.*

Beweis: Die Existenz eines PDA A_K mit $\mathcal{L}_K(A_K) = L$ folgt direkt aus dem Beweis von Lemma 4.66. Wie im Beweis von Satz 4.64 beschrieben kann A_Z aus A_K konstruiert werden. Dafür werden ein neuer Startzustand und ein neuer akzeptierender Zustand benötigt. □

Zusammen mit Lemma 4.70 (mit dem wir uns gleich befassen werden) können wir also jeden PDA in einen äquivalenten PDA mit einem oder drei Zuständen (je nach gewünschtem Akzeptanzverhalten) umwandeln. Leider bedeutet dies nicht nur, dass sich PDAs in Bezug auf die Zahl ihrer Zustände minimieren lassen, sondern vor allem, dass die Zahl der Zustände kein nützliches Maß für die Größe eines PDAs ist. Sie können

diesen Ansatz aber verwenden, um sehr schwer zu verstehende PDAs mit sehr wenig Zuständen zu erzeugen.

Hinweis 4.69 Je nach Akzeptanzmodus genügt ein PDA mit drei Zuständen oder sogar einem einzigen Zustand, um jede kontextfreie Sprache definieren zu können. Allerdings kann man in den meisten Fällen durch zusätzliche Zustände deutlich lesbarere PDAs konstruieren und außerdem unnötige Fehlerquellen vermeiden.

Wir wenden uns nun der anderen Richtung des Beweises von Satz 4.65 zu. Dieser verwendet eine Technik, die oft auch als **Tripelkonstruktion** bezeichnet wird. Hinter dem Beweis finden Sie auf Seite 176 noch ein paar Erläuterungen zur Tripelkonstruktion, die hoffentlich zu einem besseren Verständnis beitragen.

Lemma 4.70 *Sei A ein PDA. Dann ist $\mathcal{L}(A)$ kontextfrei.*

Beweisidee: Wir zeigen, dass aus jedem PDA A eine CFG G mit $\mathcal{L}(G) = \mathcal{L}_K(A)$ konstruiert werden kann. Die Grundidee ist, dass jeder Linksableitungsschritt von G einem Schritt von A entspricht. Dazu benutzen wir eine ähnliche Konstruktion wie im Beweis vom Lemma 4.35 (dem Abschluss von CFL unter Schnitt mit regulären Sprachen). Die Variablen von G bestehen hier ebenfalls aus Tripeln aus $Q \times V \times Q$ (daher der Name *Tripelkonstruktion*). Aus einer Variable $[p, K, q]$ sollen dabei genau die Terminalwörter w abgeleitet werden können, für die $(p, w, K) \vdash_A^* (q, \varepsilon, \varepsilon)$. Falls der Keller mehr als K enthält (also $K \cdot \gamma$ mit $\gamma \in \Gamma^*$), beschreibt $[p, K, q]$ aller Wörter, bei denen der PDA vom Zustand p in den Zustand q wechselt und K vom Keller entfernt, ohne den Rest γ des Kellers auch nur zu betrachten. Der PDA darf zwar direkt K entfernen, dabei weitere Symbole auf den Keller legen und mit diesen beliebig weiterarbeiten, aber der Rest γ des Kellers darf keine Rolle spielen: Kein Symbol von γ, auch nicht das oberste, wird dabei auch nur gelesen. Zur Veranschaulichung ist dies auch in Abbildung 4.11 skizziert.

Um die Grammatik zu konstruieren, konvertieren wir jeden Übergang von A in eine Menge von Regeln. Ist $(p_1, K_1 \cdots K_n) \in \delta(p, a, K)$ (mit $a \in (\Sigma \cup \{\varepsilon\})$, $n \geq 0$, $K_1, \ldots, K_n \in V$), so verwenden wir für alle $q \in Q$ alle möglichen Regeln der Form

$$[p, K, q] \to a[p_1, K_1, p_2][p_2, K_2, p_3] \cdots [p_{n-1}, K_{n-1}, p_n][p_n, K_n, p_{n+1}]$$

mit $p_2, \ldots, p_{n+1} \in Q$ und $p_{n+1} = q$. Da wir zu diesem Zeitpunkt nicht wissen können, welche Zustände der PDA bei der Berechnung verwenden wird, erzeugen wir hier einfach alle möglichen Kombinationen. Für den Fall $n = 0$ (also Übergänge, in denen kein Symbol auf den Keller gelegt wird) haben die Regeln dann die Form

$$[p, K, q] \to a,$$

wenn $\delta(p, a, K) \ni (q, \varepsilon)$ gilt.

Für $n > 0$ entspricht das Anwenden einer Regel auf eine Variable $[p, K, q]$ einer Zerlegung der erzeugten Wörter: Beim Einlesen von a (aus der Regel oben) im Zustand p

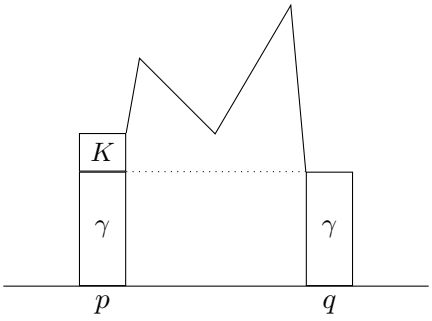


Abbildung 4.11: Eine Veranschaulichung der Bedeutung eines Tripels $[p, K, q]$ in der Tripelkonstruktion im Beweis von Lemma 4.70. Aus $[p, K, q]$ können alle Wörter w abgeleitet werden, die den PDA mit Kellerinhalt $K\gamma$ ($\gamma \in \Gamma^*$ beliebig) vom Zustand p in den Zustand q bringen. Solange γ bei der Berechnung nicht gelesen wird, darf der PDA beliebig mit dem Keller arbeiten. Der Automat darf also nur den Teil des Kellers verwenden, der oberhalb der gepunkteten Linie liegt. Die gezackte Linie symbolisier die Höhe des Kellers in der Berechnung zwischen den Zuständen p und q. Diese Berechnung kann in Teilschritte zerlegt werden, siehe dazu Abbildung 4.12.

mit oberstem Kellersymbol K legt der PDA $K_1 \cdots K_n$ auf den Keller und geht in den Zustand p_1. Um das Symbol K_1 abzuarbeiten und in den Zustand p_2 zu wechseln, liest der PDA ein Wort ein, das aus der Variable $[p_1, K_1, p_2]$ abgeleitet werden kann (diese Ableitung entspricht dem Anwenden einer Regel mit $[p_1, K_1, p_2]$ auf der linken Seite, die entsprechende Berechnung kann dabei weiter zerlegt werden). Danach wird schrittweise für alle weiteren $[p_i, K_i, p_{i+1}]$ abgeleitet. Eine graphische Darstellung dieser Berechnung finden Sie in Abbildung 4.12.

□

Beweis: Ohne Beeinträchtigung der Allgemeinheit betrachten wir einen PDA, der mit leerem Keller akzeptiert (gemäß Satz 4.64 ist dies möglich). Sei $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, \emptyset)$ ein PDA. Wir definieren nun eine CFG $G := (\Sigma, V, P, S)$ durch

- $V := \{S\} \cup \{[p, K, q] \mid p, q \in Q, K \in \Gamma\}$,
- die Menge P enthalte folgende Regeln:
  1. $S \to [q_0, K_0, q]$ für alle $q \in Q$.
  2. Für alle $p, q \in Q$, $a \in (\Sigma \cup \{\varepsilon\})$, $K \in \Gamma$ und jedes $(p_1, \gamma) \in \delta(p, a, K)$ erzeugen wir wie folgt eine Menge von Regeln: Ist $\gamma = K_1 \cdots K_n$ ($n \geq 0$), so enthalte P alle Regeln

$$[p, K, q] \to a[p_1, K_1, p_2][p_2, K_2, p_3] \cdots [p_n, K_n, p_{n+1}]$$

  mit $p_2, \ldots, p_{n+1} \in Q$ und $q = p_{n+1}$. Insbesondere gilt $[p, K, p_1] \to a$ wenn $n = 0$.

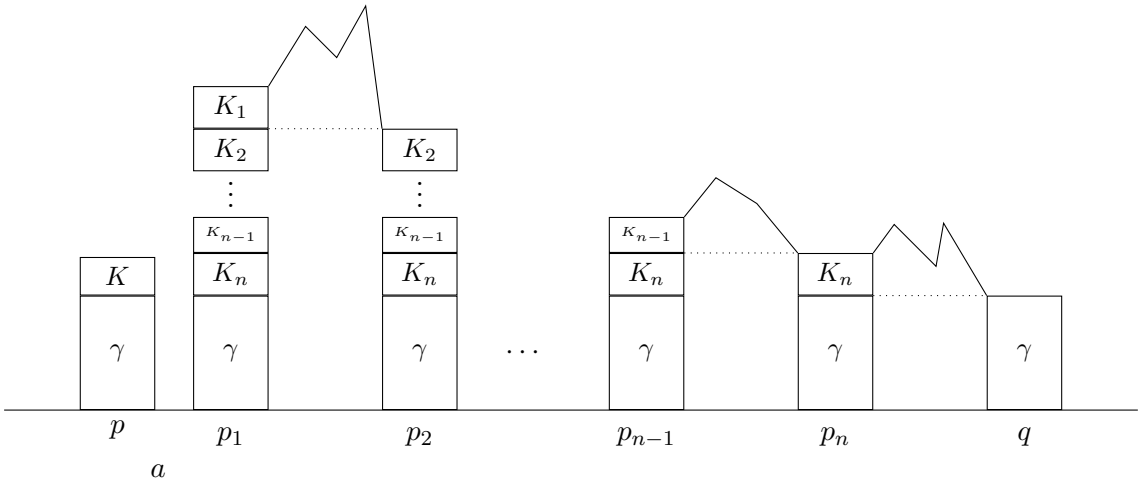


Abbildung 4.12: Graphische Darstellung der Anwendung einer Regel der Form $[p, K, q] \to a[p_1, K_1, p_2][p_2, K_2, p_3] \cdots [p_{n-1}, K_{n-1}, p_n][p_n, K_n, p_{n+1}]$. Jedes der Symbole $[p_i, K_i, p_{i+1}]$ entspricht dabei einer Berechnung (wie in Abbildung 4.11) und kann ebenfalls zerlegt werden.

Natürlich müssen wir nun noch beweisen, dass $\mathcal{L}(G) = \mathcal{L}_K(A)$. Dazu zeigen wir die folgende Behauptung:

Behauptung 1 *Für alle $w \in \Sigma^*$, alle $p, q \in Q$, alle $K \in \Gamma$ und alle $n \in \mathbb{N}_{>0}$ gilt:*

$$(p, w, K) \vdash_A^n (q, \varepsilon, \varepsilon) \text{ genau dann, wenn } [p, K, q] \Rightarrow_G^n w.$$

Beweis: Wir zeigen dies durch Induktion über n.

Induktionsanfang: Sei $n = 1$.

Behauptung: Für alle $w \in \Sigma^*$, alle $p, q \in Q$, alle $K \in \Gamma$ gilt:

$$(p, w, K) \vdash_A (q, \varepsilon, \varepsilon) \text{ genau dann, wenn } [p, K, q] \Rightarrow_G w.$$

Beweis: Wir beginnen mit der *Hin-Richtung:* Angenommen, es gilt $(p, w, K) \vdash_A (q, \varepsilon, \varepsilon)$ für $p, q \in Q$, $w \in \Sigma^*$ und $K \in \Gamma$. Dann muss $w = a \in (\Sigma \cup \{\varepsilon\})$ gelten, und $(q, \varepsilon) \in \delta(p, a, K)$. Also enthält P die Regel $[p, K, q] \to a$, und es gilt $[p, K, q] \Rightarrow_G a = w$.

Rück-Richtung: Angenommen, $[p, K, q] \Rightarrow_G w$ gilt für $p, q \in Q$, $w \in \Sigma^*$ und $K \in \Gamma$. Dann existiert in P eine Regel $[p, K, q] \to w$, und gemäß der Definition von P existiert ein $a \in (\Sigma \cup \{\varepsilon\})$ mit $w = a$. Außerdem gilt $(q, \varepsilon) \in \delta(p, a, K)$, und somit $(p, a, K) \vdash_A (q, \varepsilon, \varepsilon)$. Da $w = a$, ist $(p, w, K) \vdash_A (q, \varepsilon, \varepsilon)$.

Induktionsschritt: Sei $n \in \mathbb{N}_{>0}$.

Induktionsannahme: Für alle $i \leq n$ gelte die Behauptung.

*Behauptung:*Für alle $w \in \Sigma^*$, alle $p, q \in Q$, alle $K \in \Gamma$ gilt:

$$(p, w, K) \vdash_A^{n+1} (q, \varepsilon, \varepsilon) \text{ genau dann, wenn } [p, K, q] \Rightarrow_G^{n+1} w.$$

Beweis: Auch hier beginnen wir mit der *Hin-Richtung:* Seien $p, q \in Q$, $K \in \Gamma$, $w \in \Sigma^*$ mit $(p, w, K) \vdash_A^{n+1} (q, \varepsilon, \varepsilon)$. Dann existiert eine Konfiguration (p', w', γ) mit $(p, w, K) \vdash_A (p', w', \gamma) \vdash_A^n (q, \varepsilon, \varepsilon)$. Da $n \geq 1$ muss $\gamma \in \Gamma^+$ gelten, da kein PDA mit leerem Stack weiterrechnen kann. Sei $k = |\gamma|$, und seien $K_1, \ldots, K_k \in \Gamma$ mit $\gamma = K_1 \cdots K_k$. Dann existieren $p_1, \ldots, p_k \in Q$, ein $a \in (\Sigma \cup \{\varepsilon\})$ und $w_1, \ldots, w_k \in \Sigma^*$ sowie $j_1, \ldots, j_k \in \mathbb{N}_{>0}$ mit $w = aw' = aw_1 \cdots w_k$ und

$$\begin{aligned}(p, w, K) &\vdash_A (p_1, w_1 w_2 \cdots w_k, K_1 K_2 \cdots K_k)\\ &\vdash_A^{j_1} (p_2, w_2 \cdots w_k, K_2 \cdots K_k)\\ &\vdash_A^{j_2} \cdots\\ &\vdash_A^{j_{k-1}} (p_k, w_k, K_k)\\ &\vdash_A^{j_k} (q, \varepsilon, \varepsilon).\end{aligned}$$

Außerdem gilt $n = j_1 + j_2 + \cdots + j_k$. Wir wählen die j_i und p_i so, dass bei der Berechnung $(p_i, w_i \cdots w_k, K_i \cdots K_k) \vdash_A^{j_i} (p_{i+1}, w_{i+1} \cdots w_k, K_{i+1} \cdots K_k)$ der Kellerinhalt unterhalb von K_i (also $K_i \cdots K_k$) nicht betrachtet wird (selbst K_i darf dabei nicht einmal gelesen werden). Daher gilt für alle $1 \leq i < k$:

$$(p_i, w_i, K_i) \vdash_A^{j_i} (p_{i+1}, \varepsilon, \varepsilon).$$

Gemäß unserer Induktionsannahme gilt daher für alle $1 \leq i < k$:

$$[p_i, K_i, p_{i+1}] \Rightarrow_G^{j_i} w_i,$$

und außerdem

$$[p_k, K_k, q] \Rightarrow_G^{j_k} w_k.$$

Wie wir eingangs festgestellt haben, gilt $(p, w, K) \vdash_A (p', w', \gamma)$. Diese Aussage ist äquivalent zu $(p, aw_1 \cdots w_k, K) \vdash_A (p_1, w_1 \cdots w_k, K_1 \cdots K_k)$. Daher enthält P die Regel

$$[p, K, q] \to [p_1, K_1, p_2][p_2, K_2, p_3] \cdots [p_k, K_k, q].$$

Es gilt also:

$$\begin{aligned}[p, K, q] &\Rightarrow_G a[p_1, K_1, p_2][p_2, K_2, p_3] \cdots [p_k, K_k, q]\\ &\Rightarrow_G^{j_1} aw_1[p_2, K_2, p_3] \cdots [p_k, K_k, q]\\ &\Rightarrow_G^{j_2 + \cdots + j_k} aw_1 w_2 \cdots w_n\\ &= w.\end{aligned}$$

Somit ist $[p, K, q] \Rightarrow_G^{n+1} w$. Dies beendet den Beweis der Hin-Richtung.

Rück-Richtung: Seien $p, q \in Q$, $K \in \Gamma$, $w \in \Sigma^*$ mit $[p, K, q] \Rightarrow_G^{n+1} w$. Dann existieren ein $a \in (\Sigma \cup \{\varepsilon\})$, ein $k \in \mathbb{N}_{>0}$ und $B_1, \ldots, B_k \in V$ mit

$$[p, K, q] \Rightarrow_G aB_1 \cdots B_k \Rightarrow_G^n w.$$

In P ist also die Regel $[p, K, q] \to aB_1 \cdots B_k$ enthalten. Außerdem existieren $j_1, \ldots, j_k \in \mathbb{N}_{>0}$ und $w_1, \ldots, w_k \in \Sigma^*$ mit $B_i \Rightarrow_G^{j_i} w_i$ für $1 \leq i \leq k$, und es gilt $w = aw_1 \cdots w_k$ sowie $n = j_1 + \cdots + j_k$. Außerdem existieren $p_1, \ldots, p_{k+1} \in Q$ und $K_1 \ldots, K_k \in \Gamma$ mit $B_i = [p_i, K_i, p_{i+1}]$, wobei $p_{k+1} = q$.

Es gilt also $[p_i, K_i, p_{i+1}] \Rightarrow_G^{j_i} w_i$ für alle $1 \leq i \leq k$. Gemäß der Induktionsannahme ist also $(p_i, w_i, K_i) \vdash_A^{j_i} (p_{i+1}, \varepsilon, \varepsilon)$. Diese Berechnungen sind immer noch erlaubt, wenn wir sowohl an die Eingabe als auch an den Keller beliebige Wörter anhängen. Also gilt

$$(p_i, w_i \cdots w_k, K_i \cdots K_k) \vdash_A^{j_i} (p_{i+1}, w_{i+1} \cdots w_k, K_{i+1} \cdots K_k)$$

für alle $1 \leq i \leq k$. Durch Zusammensetzen dieser einzelnen Berechnungen erhalten wir

$$(p_1, w_1 \cdots w_k, K_1 \cdots K_k) \vdash_A^{j_1+\cdots+j_k} (p_{k+1}, \varepsilon, \varepsilon).$$

Wie wir eingangs festgestellt haben, enthält P die Regel $[p, K, q] \to aB_1 \cdots B_k$, also

$$[p, K, q] \to a[p_1, K_1, p_2] \cdots [p_k, K_k, q].$$

Wegen der Definition von P muss $(p_1, K_1 \cdots K_k) \in \delta(p, a, K)$ gelten, und somit $(p, w, K) \vdash (p, w_1 \cdots w_k, K_1 \cdots K_k)$, da $w = aw_1 \cdots w_k$.

Wir setzen diese einschrittige Berechnung mit der zuvor erstellten Berechnung mit $j_1 + \cdots + j_k$ Schritten zusammen, und erhalten:

$$\begin{aligned}(p, w, K) &\vdash_A (p_1, w_1 \cdots w_k, K_1 \cdots K_K) \\ &\vdash_A^{j_1+\cdots+j_k} (p_{k+1}, \varepsilon, \varepsilon).\end{aligned}$$

Wegen $p_{k+1} = q$ und $j_1 + \cdots + j_k$ gilt $(p, w, K) \vdash_A^{n+1} (q, \varepsilon, \varepsilon)$. Somit haben wir auch die Rück-Richtung bewiesen. □(Behauptung 1)

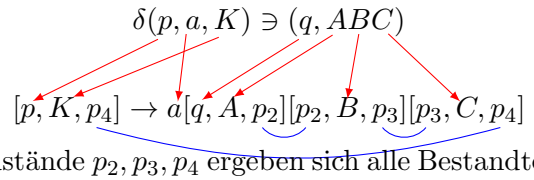
Für alle $w \in \Sigma^*$ gilt $w \in \mathcal{L}(G)$ genau dann, wenn ein $q \in Q$ existiert, so dass $[q_0, K_0, q] \Rightarrow_G^* w$. Gemäß Behauptung 1 ist dies genau dann der Fall, wenn $(q_0, w, K_0) \vdash_A^* (q, \varepsilon, \varepsilon)$; also genau dann, wenn $w \in \mathcal{L}_K(A)$. □

Erfahrungsgemäß bereitet das Ablesen der Regeln aus den Übergängen eines Kellerautomaten anfangs oft Schwierigkeiten. Am einfachsten ist es, wenn Sie einen Übergang nach dem anderen bearbeiten. Angenommen, Sie wollen bei einem Kellerautomaten $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, \emptyset)$ einen Übergang der folgenden Form zu einer Menge von Regeln konvertieren:

Dieser Übergang entspricht in der Übergangsrelation δ dem Paar $(q, ABC) \in \delta(p, a, K)$. Diesen Zusammenhang erkennen Sie etwas leichter, wenn Sie dies umgekehrt schreiben, nämlich

$$\delta(p, a, K) \ni (q, ABC).$$

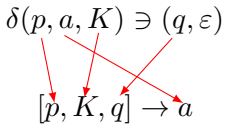
So entspricht die Reihenfolge der einzelnen Teile weitgehend der Reihenfolge in der graphischen Darstellung. Dieser Übergang wird nun wie folgt in eine Menge von Regeln konvertiert:



Mit Ausnahme der Zustände p_2, p_3, p_4 ergeben sich alle Bestandteile der Regeln aus dem untersuchten Übergang $\delta(p, a, K) \ni (q, ABC)$; die Zusammenhänge sind mit roten Pfeilen gekennzeichnet. Die Vorkommen von p_2, p_3, p_4 in den Tripeln sind blau miteinander verbunden: Hier müssen jeweils die verbundenen Positionen übereinstimmen.

Wir erzeugen nun aus jeder möglichen Wahl von $p_2, p_3, p_4 \in Q$ eine Regel, indem wir die Zustände passend für p_2, p_3, p_4 einsetzen – insgesamt also $|Q|^3$ Regeln. Allgemein werden also aus einem Übergang der Form $\delta(p, a, K) \ni (q, \gamma)$ (mit $\gamma \in \Gamma^*$) Zustände $p_2, \ldots, p_{|\gamma|+1}$ gewählt und somit stets $|Q|^{|\gamma|}$ Regeln erzeugt. Im Allgemeinen werden viele dieser Regeln nicht benötigt, da sie Tripel enthalten, die nicht abgeleitet werden können. Deswegen betrachten wir im Anschluss in Hinweis 4.71 einige Optimierungsmöglichkeiten.

Besondere Erwähnung verdienen Übergänge der Form $\delta(p, a, K) \ni (q, \gamma)$. Diese werden wie folgt konvertiert:



Diese Regeln sind ein einfacherer Sonderfall der verwendeten Definition (siehe dazu auch den Beweis von Lemma 4.70). Im Prinzip ist es nicht nötig, diesen Fall getrennt zu betrachten, allerdings ist dieser Fall so einfacher nachzuvollziehen.

Hinweis 4.71 Wenn Sie die Tripelkonstruktion ausführen, um aus einem Kellerautomaten eine kontextfreie Grammatik zu konstruieren, können Sie sich unnötige Arbeit sparen und überflüssige Regeln direkt entfernen. Dies gilt besonders für Regeln, die Variablen enthalten, die niemals abgeleitet werden können. In zwei Fällen sind solche Variablen leicht zu erkennen. Sei $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, \emptyset)$ ein PDA. Die beiden folgenden Bedingungen sind hinreichend, um eine Variable $[p, K, q]$ als überflüssig zu erkennen:

1. Der Zustand q ist vom Zustand p aus nicht erreichbar. Dann kann keine Berechnung von p aus in q enden, also kann die Variable $[p, K, q]$ niemals abgeleitet werden.

2. Für alle $a \in (\Sigma \cup \{\varepsilon\})$ ist $\delta(p, a, K) = \emptyset$. Dann kann der PDA im Zustand p keine Aktion ausführen, in der K als oberstes Kellersymbol gelesen wird. Also stürzt A in allen entsprechenden Konfigurationen ab, also kann $[p, K, q]$ nicht

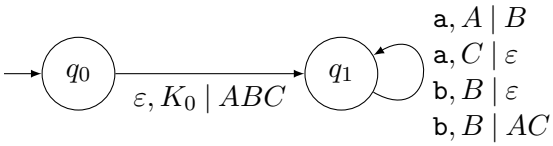
abgeleitet werden.

Wenn Sie also beim Bilden einer Regel für eine Wahl von $p_2, \ldots, p_{n+1}$ auf der rechten Seite auch nur ein einziges Tripel erhalten, das auch nur eine dieser beiden Bedingungen erfüllt, so können Sie diese Wahl von $p_2, \ldots, p_{n+1}$ ignorieren.

Im Prinzip wäre es zwar möglich, noch weitere Optimierungen vorzunehmen; allerdings erreichen Sie dabei schnell einen Punkt, an dem die Optimierungen mehr Arbeit verursachen, als sie einsparen. Bei den Aufgaben, mit denen Sie im Rahmen dieser Vorlesung zu tun haben, sollten keine weiteren Optimierungen notwendig sein.

Wir betrachten nun zwei Beispiele für die Anwendung der Tripelkonstruktion. In jedem dieser Beispiele konstruieren wir die Grammatik bereits mit einigen Optimierungen, die Hinweis 4.71 befolgen. An dieser Stelle stand in einer früheren Version noch ein anderes Beispiel für einen anderen Ansatz einer optimierten Tripelkonstruktion. Da sich dieses als unnötig verwirrend erwiesen hat, wurde es entfernt.

Beispiel 4.72 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir betrachten den PDA A mit Kelleralphabet $\Gamma := \{K_0, A, B, C\}$ und Startsymbol K_0, der wie folgt definiert ist:



Wir betrachten Akzeptanz durch leeren Keller (bei Akzeptanz durch Zustände können wir die Tripelkonstruktion nicht anwenden). Um eine Grammatik G mit $\mathcal{L}(G) = \mathcal{L}_K(A)$ zu konstruieren, betrachten wir nun nacheinander die einzelnen Übergänge von A. Dabei führen wir in den Schritten gegebenenfalls Optimierungen auf, indem wir nutzlose Regeln entfernen. Diese Optimierungen sind nicht notwendig, sie sparen uns lediglich Schreibarbeit. Bei dem in diesem Beispiel gewählten Ansatz hängen die weiteren Schritte nicht davon ab, ob diese Optimierungen durchgeführt werden, da die einzelnen Schritte sich direkt aus den Übergängen des PDA ergeben.

1. $\underline{\delta(q_0, \varepsilon, K_0) \ni (q_1, ABC)}$: Hierzu erzeugen wir alle möglichen Regeln der Form

$$[q_0, K_0, p_4] \to \varepsilon[q_1, A, p_2][p_2, B, p_3][p_3, C, p_4]$$

mit $p_2, p_3, p_4 \in Q$. Im Grunde müssten wir hier also acht verschiedene Regeln erzeugen (da wir für p_2 bis p_4 jeweils zwei Optionen wählen können). Allerdings können wir hier auf zwei Arten optimieren: Zum einen gilt $\delta(q_0, a, K) = \emptyset$ für alle $a \in (\Sigma \cup \{\varepsilon\})$ und alle $K \in \{A, B, C\}$. Das heißt: Ist der PDA im Zustand q_0 während oben auf dem Keller ein A, B oder C liegt, stürzt der PDA ab. Wählen wir also $p_2 = q_0$ oder $p_3 = q_0$, so entstehen die Symbole $[q_0, B, p_3]$ bzw. $[q_0, C, p_4]$, die niemals abgeleitet werden können. Also können wir davon ausgehen, dass $p_2 = p_3 = q_1$. Außerdem wissen wir, dass von q_1 kein anderer Zustand als q_1 erreicht werden

kann. Also ist $p_4 = q_1$ (und außerdem können wir so ebenfalls darauf schließen, dass $p_2 = p_3 = q_1$). Somit müssen wir nur die folgende Regel berücksichtigen:

$$[q_0, K_0, q_1] \to \varepsilon[q_1, A, q_1][q_1, B, q_1][q_1, C, q_1].$$

Das ε können wir an dieser Stelle auch einfach weglassen, wir verwenden also die folgende Regel:

$$[q_0, K_0, q_1] \to [q_1, A, q_1][q_1, B, q_1][q_1, C, q_1].$$

Durch die Optimierungen haben wir in diesem Schritt nur eine einzige Regel konstruiert, anstelle der acht Regeln, die sich aus allen möglichen Kombinationen für p_2, p_3, p_4 Wir können allerdings auch die sieben anderen möglichen Regeln aufnehmen; das führt zwar zu mehr Schreibarbeit, ist aber nicht falsch.

2. $\underline{\delta(q_1, \mathtt{a}, A) \ni (q_1, B)}$: Die aus diesem Übergang erzeugten Regeln haben die Form

   $$[q_1, A, p_2] \to \mathtt{a}[q_1, B, p_2]$$

   für alle $p_2 \in Q$. Wie im vorigen Fall können wir $p_2 = q_0$ wegen fehlender Erreichbarkeit von q_1 aus ausschließen. Wir erzeugen also lediglich die Regel

   $$[q_1, A, q_1] \to \mathtt{a}[q_1, B, q_1].$$

3. $\underline{\delta(q_1, \mathtt{a}, C) \ni (q_1, \varepsilon)}$: Da hier kein Symbol auf den Keller gelegt werden muss, wird aus diesem Übergang nur eine Regel erzeugt, nämlich

   $$[q_1, C, q_1] \to \mathtt{a}.$$

4. $\underline{\delta(q_1, \mathtt{b}, B) \ni (q_1, \varepsilon)}$: Analog zum in 3 betrachteten Fall erzeugen wir die Regel

   $$[q_1, B, q_1] \to \mathtt{b}.$$

5. $\underline{\delta(q_1, \mathtt{b}, B) \ni (q_1, AC)}$: Hierzu müssen wieder mehrere mögliche Regeln erzeugt werden, nämlich Regeln der Form

   $$[q_1, B, p_3] \to \mathtt{b}[q_1, A, p_2][p_2, C, p_3]$$

   mit $p_2, p_3 \in Q$. Auch hier können wir $p_2 = q_0$ und $p_3 = q_0$ ausschließen und so anstelle von vier Regeln nur die folgende Regel verwenden:

   $$[q_1, B, q_1] \to \mathtt{b}[q_1, A, q_1][q_1, C, q_1].$$

Da wir nun alle Übergänge des PDA betrachtet haben, sind wir mit der Konstruktion der Grammatik fast fertig. Wir benötigen nur noch Regeln, die das Startsymbol der Grammatik auf Tripel abbilden. Dazu kommen Regeln dieser Form in Frage:

$$S \to [q_0, K_0, p]$$

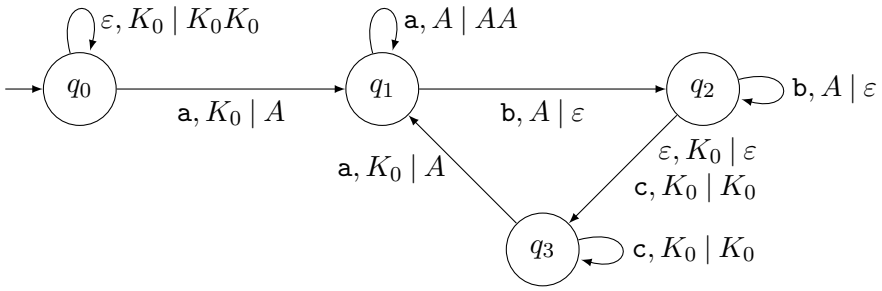
mit $p \in Q$. Hier können wir ebenfalls optimieren: Wir wissen, dass wir keine Regel angelegt haben, bei der $[q_0, K_0, q_0]$ auf der linken Seite steht. Wir können also den Fall $p = q_0$ ausschließen und uns auf $p = q_1$ beschränken.

Insgesamt nehmen wir also die folgenden Regeln in unsere Regelmenge auf:

$$\begin{aligned} S &\to [q_0, K_0, q_1], \\ [q_0, K_0, q_1] &\to [q_1, A, q_1][q_1, B, q_1][q_1, C, q_1], \\ [q_1, A, q_1] &\to \mathtt{a}[q_1, B, q_1], \\ [q_1, C, q_1] &\to \mathtt{a}, \\ [q_1, B, q_1] &\to \mathtt{b}, \\ [q_1, B, q_1] &\to \mathtt{b}[q_1, A, q_1][q_1, C, q_1]. \end{aligned}$$

Hätten wir auch die anderen Regeln erzeugt, so würden diese Nichtterminale enthalten, die niemals abgeleitet werden können (und somit in Bezug auf die erzeugte Sprache sinnlos sind). Wir können nun $V := \{S, [q_0, S, q_1], [q_1, A, q_1], [q_1, C, q_1], [q_1, B, q_1]\}$ wählen. Im Prinzip könnten wir nun noch die ersten beiden Regeln zusammenziehen und die Variablen kompakter benennen, aber diese Optimierungen sind eher kosmetischer Natur und Geschmacksfrage. ◊

Beispiel 4.73 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Wir betrachten den PDA A mit Kelleralphabet $\Gamma := \{K_0, A\}$ und Startsymbol K_0, der wie folgt definiert ist:



Wir konstruieren nun mittels der Tripelkonstruktion aus A eine äquivalente kontextfreie Grammatik. Dazu betrachten wir nacheinander jeden Übergang von A. Dabei nehmen wir wieder nahe liegende Optimierungen vor. Diese sind optional, sparen uns aber viel Schreibarbeit.

1. $\underline{\delta(q_0, \varepsilon, K_0) \ni (q_0, K_0K_0)}$: Aus diesem Übergang leiten wir Regeln der Form

   $$[q_0, K_0, p_3] \to \varepsilon[q_0, K_0, p_2][p_2, K_0, p_3]$$

   ab, und zwar für jede mögliche Wahl von $p_2, p_3 \in Q$. Es gilt:

   - Das Kellersymbol K_0 kann nicht im Zustand q_1 gelesen werden. Also muss $p_2 \neq q_1$ gelten.
   - Außerdem kann q_0 nicht von den anderen drei Zuständen aus erreicht werden. Für den Fall $p_3 = q_0$ folgt daher zwangsläufig $p_2 = q_0$.

Somit konstruieren wir nur die folgenden zehn Regeln:

$$
\begin{aligned}
&[q_0, K_0, q_0] \to [q_0, K_0, q_0][q_0, K_0, q_0],\\
&[q_0, K_0, q_1] \to [q_0, K_0, q_0][q_0, K_0, q_1],\\
&[q_0, K_0, q_1] \to [q_0, K_0, q_2][q_2, K_0, q_1],\\
&[q_0, K_0, q_1] \to [q_0, K_0, q_3][q_3, K_0, q_1],\\
&[q_0, K_0, q_2] \to [q_0, K_0, q_0][q_0, K_0, q_2],\\
&[q_0, K_0, q_2] \to [q_0, K_0, q_2][q_2, K_0, q_2],\\
&[q_0, K_0, q_2] \to [q_0, K_0, q_3][q_3, K_0, q_2],\\
&[q_0, K_0, q_3] \to [q_0, K_0, q_0][q_0, K_0, q_3],\\
&[q_0, K_0, q_3] \to [q_0, K_0, q_2][q_2, K_0, q_3],\\
&[q_0, K_0, q_3] \to [q_0, K_0, q_3][q_3, K_0, q_3].
\end{aligned}
$$

Für jede dieser Regeln haben wir auf das ε (aus $\varepsilon[q_0, K_0, p_2][p_2, K_0, p_3]$) verzichtet. Dies ist erlaubt, da sich die erzeugte Sprache dadurch nicht ändert.

2. $\underline{\delta(q_0, \mathtt{a}, K_0) \ni (q_1, A)}$: Aus diesem Übergang leiten wir Regeln der Form

$$[q_0, K_0, p_2] \to \mathtt{a}[q_1, A, p_2]$$

ab, für alle Wahlmöglichkeiten von $p_2 \in Q$. Von q_1 aus sind alle Zustände außer q_0 erreichbar, also erzeugen wir die drei Regeln

$$
\begin{aligned}
&[q_0, K_0, q_1] \to \mathtt{a}[q_1, A, q_1],\\
&[q_0, K_0, q_2] \to \mathtt{a}[q_1, A, q_2],\\
&[q_0, K_0, q_3] \to \mathtt{a}[q_1, A, q_3].
\end{aligned}
$$

3. $\underline{\delta(q_1, \mathtt{a}, A) \ni (q_1, AA)}$: Dieser Übergang führt zu Regeln der Form

$$[q_1, A, p_3] \to \mathtt{a}[q_1, A, p_2][p_2, A, p_3],$$

wobei $p_2, p_3 \in Q$. Wieder können wir optimieren: Das Kellersymbol A kann nur in den Zuständen q_1 und q_2 gelesen werden, also ist $p_2 \in \{q_1, q_2\}$. Da q_0 von diesen Zuständen aus nicht erreichbar ist, gilt $p_3 \neq q_0$. Wir erhalten also die folgenden Regeln:

$$
\begin{aligned}
&[q_1, A, q_1] \to \mathtt{a}[q_1, A, q_1][q_1, A, q_1],\\
&[q_1, A, q_2] \to \mathtt{a}[q_1, A, q_1][q_1, A, q_2],\\
&[q_1, A, q_3] \to \mathtt{a}[q_1, A, q_1][q_1, A, q_3],\\
&[q_1, A, q_1] \to \mathtt{a}[q_1, A, q_2][q_2, A, q_1],\\
&[q_1, A, q_2] \to \mathtt{a}[q_1, A, q_2][q_2, A, q_2],\\
&[q_1, A, q_3] \to \mathtt{a}[q_1, A, q_2][q_2, A, q_3].
\end{aligned}
$$

4. $\underline{\delta(q_1, \mathtt{b}, A) \ni (q_2, \varepsilon)}$: Dieser Übergang führt zu einer einzigen Regel, nämlich

$$[q_1, A, q_2] \to \mathtt{b}.$$

5. $\underline{\delta(q_2, \mathtt{b}, A) \ni (q_2, \varepsilon)}$: Auch dieser Übergang führt nur zu einer Regel, und zwar

$$[q_2, A, q_2] \to \mathtt{b}.$$

6. $\underline{\delta(q_2, \varepsilon, K_0) \ni (q_3, \varepsilon)}$: Wie zu erwarten ist führt dieser Übergang zu einer einzigen Regel, und zwar

$$[q_2, K_0, q_3] \to \varepsilon.$$

7. $\underline{\delta(q_2, \mathtt{c}, K_0) \ni (q_3, K_0)}$: Die aus diesem Übergang abgeleiteten Regeln haben die Form

$$[q_2, K_0, p_2] \to \mathtt{c}[q_3, K_0, p_2]$$

mit $p_2 \in Q$. Da q_0 von q_2 aus nicht erreicht werden kann, dürfen wir davon ausgehen, dass $p_2 \in \{q_1, q_2, q_3\}$. Wir konstruieren also die folgenden drei Regeln:

$$\begin{aligned}
[q_2, K_0, q_1] &\to \mathtt{c}[q_3, K_0, q_1],\\
[q_2, K_0, q_2] &\to \mathtt{c}[q_3, K_0, q_2],\\
[q_2, K_0, q_3] &\to \mathtt{c}[q_3, K_0, q_3].
\end{aligned}$$

8. $\underline{\delta(q_3, \mathtt{c}, K_0) \ni (q_3, K_0)}$: Die Konstruktion dieser Regeln verläuft analog zu Fall 7. Wir konstruieren die Regeln

$$\begin{aligned}
[q_3, K_0, q_1] &\to \mathtt{c}[q_3, K_0, q_1],\\
[q_3, K_0, q_2] &\to \mathtt{c}[q_3, K_0, q_2],\\
[q_3, K_0, q_3] &\to \mathtt{c}[q_3, K_0, q_3].
\end{aligned}$$

9. $\underline{\delta(q_3, \mathtt{a}, K_0) \ni (q_1, A)}$: Analog zu Fall 2 erzeugen wir die drei folgenden Regeln:

$$\begin{aligned}
[q_3, K_0, q_1] &\to \mathtt{a}[q_1, A, q_1],\\
[q_3, K_0, q_2] &\to \mathtt{a}[q_1, A, q_2],\\
[q_3, K_0, q_3] &\to \mathtt{a}[q_1, A, q_3].
\end{aligned}$$

Abschließend benötigen wir noch Regeln für das Startsymbol. Dazu erzeugen Regeln der Form

$$S \to [q_0, K_0, p]$$

für alle $p \in Q$. Hier gibt es keine Optimierungsmöglichkeiten, also erhalten wir die Regeln

$$\begin{aligned}
S &\to [q_0, K_0, q_0],\\
S &\to [q_0, K_0, q_1],
\end{aligned}$$

$$S \to [q_0, K_0, q_2],$$
$$S \to [q_0, K_0, q_3].$$

Wir definieren nun $G := (\Sigma, V, P, S)$, wobei $V := \{S\} \cup \{[p, K, q] \mid p, q \in Q, K \in \Gamma\}$, und P enthalte genau die Regeln, die wir konstruiert haben. Mit Optimierung erhalten wir 39 Regeln, ohne Optimierungen 55 Regeln (in beiden Fällen sind die vier Regeln für das Startsymbol enthalten). Die Wahl von V ist nicht optimal, da nicht jede der darin enthaltenen Variablen verwendet wird. Aber diese Schreibweise ist deutlich kompakter und macht weniger Arbeit, als eine Liste der tatsächlich verwendeten Variablen zu erstellen. ◊

Gelegentlich ist es deutlich einfacher, mit Kellerautomaten anstelle von kontextfreien Grammatiken zu arbeiten. Ein Beispiel dafür ist der Beweis der folgenden Abschlusseigenschaft:

Lemma 4.74 *Die Klasse* CFL *ist abgeschlossen unter inversem Homomorphismus.*

Beweisidee: Analog zu unserem Beweis für den Abschluss von REG unter inversen Homomorphismen (Lemma 3.104) können wir aus einem PDA A und einem Homomorphismus h einen PDA A_I konstruieren, der beim Einlesen jedes Terminals a den PDA A beim Einlesen von $h(a)$ simuliert. Allerdings ist es möglich, dass A beim Einlesen von $h(a)$ mehrere Schritte lang auf dem Keller arbeitet, was A_I nicht in einem Schritt erledigen kann. Daher erweitern wir A_I um einen Zwischenspeicher, der anhand einer Produktautomatenkonstruktion realisiert wird. Die Zustände von A_I sind daher Paare der Form $[q, x]$, wobei q einen Zustand von A bezeichnet, und x ein Suffix eines $h(a)$ (nämlich den Teil von $h(a)$, der noch abgearbeitet werden muss). □

Beweis: Seien Σ_1 und Σ_2 Alphabete, sei $h\colon \Sigma_1^* \to \Sigma_2^*$ ein Homomorphismus und sei $L \subseteq \Sigma_2^*$ eine kontextfreie Sprache. Dann existiert ein PDA $A := (\Sigma_2, \Gamma, Q, \delta, q_0, K_0, F)$ mit $\mathcal{L}(A) = L$. Sei Wir definieren nun den PDA $A_I := (\Sigma_1, \Gamma, Q_I, \delta_I, q_{0,I}, K_0, F_I)$. Dabei sei

- $Q_I := \{[q, x] \mid q \in Q, x \in \text{suffix}(h(a)), a \in \Sigma_1\}$,
- $q_{0,I} := [q_0, \varepsilon]$,
- $F_I := \{[q, \varepsilon] \mid q \in F\}$.

Die Übergangsrelation δ_I definieren wir für alle $q \in Q$, $x \in \text{suffix}(h(\Sigma_1))$ und $K \in \Gamma$ wie folgt:

1. $\delta_I([q, x], \varepsilon, K)$ enthält alle $([p, x], \gamma)$ für die $(p, \gamma) \in \delta(q, \varepsilon, K)$ (dies simuliert die ε-Bewegungen von A ohne den Zwischenspeicher zu berücksichtigen),
2. $\delta_I([q, ax], \varepsilon, K)$ enthält alle $([p, x], \gamma)$ für die $(p, \gamma) \in \delta(q, a, K)$ (simuliert Abarbeiten eines Buchstaben aus dem Zwischenspeicher),
3. $\delta_I([q, \varepsilon], a, K) := \{([q, h(a)], K)\}$ für alle $a \in \Sigma_1$ (lädt $h(a)$ in den Zwischenspeicher, Zustand und Keller bleiben unverändert).

Wir zeigen nun durch doppelte Inklusion, dass $\mathcal{L}(A_I) = h^{-1}(L)$. Wir beginnen mit $\mathcal{L}(A_I) \supseteq h^{-1}(L)$. Dazu machen wir erst eine Beobachtung. Seien $p, q \in Q$, $a \in \Sigma_1$ und $\alpha, \beta \in \Gamma^*$ mit

$$(q, h(a), \alpha) \vdash_A^* (p, \varepsilon, \beta).$$

Dann können wir zuerst einen Übergang gemäß (3) verwenden und danach Übergänge gemäß (1) und (2), und erhalten somit

$$([q, \varepsilon], a, \alpha) \vdash_{A_I} ([q, h(a)], \varepsilon, \alpha) \vdash_{A_I}^* ([p, \varepsilon], \varepsilon, \beta).$$

Angenommen, $h(w) \in \mathcal{L}(A)$. Dann existieren ein $q \in Q$ und ein $\gamma \in \Gamma^*$ mit

$$(q_0, h(w), K_0) \vdash_A^* (p, \varepsilon, \gamma).$$

Nach unserer Beobachtung folgt hieraus

$$([q_0, \varepsilon], w, K_0) \vdash_{A_I}^* ([p, \varepsilon], \varepsilon, \gamma).$$

Da $[p, \varepsilon] \in F_I$ folgt hieraus $w \in \mathcal{L}(A_I)$. Somit gilt $\mathcal{L}(A_I) \supseteq h^{-1}(\mathcal{L}(A)) = h^{-1}(L)$.

Um zu zeigen, dass $\mathcal{L}(A_I) \subseteq h^{-1}(L)$, wählen wir ein $w \in \mathcal{L}(A_I)$. Sei $n := |w|$, und seien $a_1, \ldots, a_n \in \Sigma_1$ mit $w = a_1 \cdots a_n$. Da $w \in \mathcal{L}(A_I)$ gilt

$$([q_0, \varepsilon], w, K_0) \vdash_{A_I}^* ([p_{n+1}, \varepsilon], \varepsilon, \gamma_{n+1})$$

für ein $p_{n+1} \in F$ und ein $\gamma_{n+1} \in \Gamma^*$. Da A_I nur mit Übergängen gemäß (3) die Eingabe abarbeiten kann, lässt sich diese Berechnung wie folgt darstellen:

$$\begin{aligned}
([q_0, \varepsilon], a_1 \cdots a_n, K_0) &\vdash_{A_I}^* ([p_1, \varepsilon], a_1 \cdots a_n, \gamma_1) \\
&\vdash_{A_I} ([p_1, h(a_1)], a_2 \cdots a_n, \gamma_1) \\
&\vdash_{A_I}^* ([p_2, \varepsilon], a_2 \cdots a_n, \gamma_2) \\
&\vdash_{A_I} ([p_2, h(a_2)], a_3 \cdots a_n, \gamma_2) \\
&\vdots \\
&\vdash_{A_I}^* ([p_n, \varepsilon], a_n, \gamma_n) \\
&\vdash_{A_I} ([p_n, h(a_n)], \varepsilon, \gamma_n) \\
&\vdash_{A_I}^* ([p_{n+1}, \varepsilon], \varepsilon, \gamma_{n+1}).
\end{aligned}$$

Hierbei wechseln sich immer Übergänge gemäß (3) mit mehrfacher Anwendung von Übergängen gemäß (1) oder gemäß (2) ab. Also gilt

$$(q_0, \varepsilon, K_0) \vdash_A^* (p_1, \varepsilon, \gamma_1)$$

und, für alle $1 \leq i < n$,

$$(p_i, h(a_i), \gamma_i) \vdash_A^* (p_{i+1}, \varepsilon, \gamma_{i+1}).$$

Durch Zusammensetzen dieser Berechnungen erhalten wir

$$(q_0, h(a_1 \cdots a_n), K_0) \vdash_A^* (p_n, \varepsilon, \gamma_n).$$

Es gilt also $h(w) \in \mathcal{L}(A)$ und somit $h(w) \in L$. Daher ist $w \in h^{-1}(L)$. Da $w \in \mathcal{L}(A_I)$ frei gewählt wurde, gilt $\mathcal{L}(A_I) \subseteq h^{-1}(L)$. □

Um zu zeigen, dass Sprachen nicht kontextfrei sind, können wir nun die gleichen Beweistechniken verwenden wie bei den regulären Sprachen. Insbesondere können wir die Kombination aus inversem Homomorphismus, Schnitt mit regulären Sprachen und Homomorphismus verwenden, um gezielt Wörter umzuschreiben. Außerdem erhalten wir nun die folgenden Abschlusseigenschaften fast geschenkt:

Lemma 4.75 *Die Klasse der kontextfreien Sprachen ist abgeschlossen unter Rechts-Quotient mit regulären Sprachen. Das heißt: Ist* $L \in \text{CFL}$ *und* $R \in \text{REG}$*, dann ist die Sprache* L/R *kontextfrei.*

Die Klasse der kontextfreien Sprachen ist außerdem abgeschlossen unter den Operationen prefix *und* suffix*.*

Beweis: Übung (Aufgabe 4.2). □

4.2.2 Deterministische Kellerautomaten

Kellerautomaten haben zwar die gleiche Ausdrucksstärke wie kontextfreie Grammatiken, allerdings führt der Nichtdeterminismus dieses Modells beim Auswerten schnell zu Schwierigkeiten. Daher wird in der Praxis oft mit *deterministischen Kellerautomaten* gearbeitet. Diese sind wie folgt definiert:

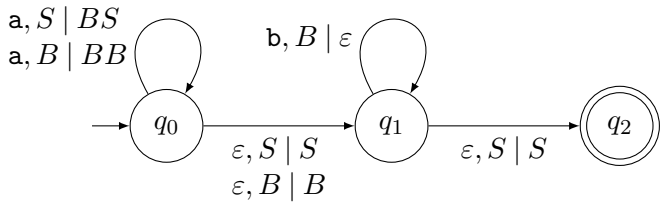
Definition 4.76 Sei $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, F)$ ein Kellerautomat. Wir bezeichnen A als **deterministischen Kellerautomaten (DPDA)**, wenn

$$|\delta(q, a, K)| + |\delta(q, \varepsilon, K)| \leq 1$$

für alle $q \in Q$, $a \in \Sigma$, $K \in \Gamma$.

Wie ein DFA kann ein DPDA in jedem Schritt höchstens eine Aktion ausführen. Im Gegensatz zu einem DFA darf ein DPDA in Zustand q mit oberstem Kellersymbol K also einen ε-Übergang benutzen, außer zu diesem q und K existiert noch ein Übergang der kein ε-Übergang ist.

Beispiel 4.77 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Wir betrachten zuerst den folgenden PDA A_N mit $\Gamma := \{S, B\}$ und Startsymbol S, der wie folgt definiert ist:



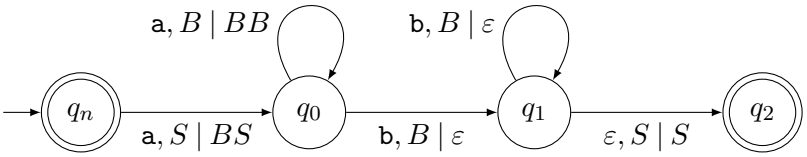
Wir haben A_N in Beispiel 4.62 kennengelernt und dort festgestellt, dass $\mathcal{L}(A_N) = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Wir stellen fest, dass A_N *kein* DPDA ist. Es gilt:

$$|\delta(q_0, \mathtt{a}, S)| + |\delta(q_0, \varepsilon, S)| = 2 > 1,$$

$$|\delta(q_0, \mathtt{a}, B)| + |\delta(q_0, \varepsilon, B)| = 2 > 1.$$

Der PDA hat also im Zustand q_0 mit oberstem Kellersymbol S (oder auch oberstem Kellersymbol B) die Möglichkeit, das Terminal $\mathtt{a}$ von der Eingabe zu lesen oder einen ε-Übergang auszuführen. Dadurch kann er sich nichtdeterministisch entscheiden; A_N ist also kein DPDA.

Allerdings lässt sich ein äquivalenter DPDA A_D konstruieren. Dazu wählen wir das gleiche Kelleralphabet Γ und das gleiche Startsymbol S und definieren A_D wie folgt:



Wie leicht zu erkennen ist, ist A_D ein DPDA. Der neue Startzustand q_n dient dazu, dass A_D auch ε akzeptieren kann. Liest A_D ein $\mathtt{a}$, so wechselt A_D in q_0 und beginnt eine Hochzähl-Phase, in der die Zahl der eingelesenen $\mathtt{a}$ in Form von B auf dem Keller gespeichert wird. Dabei markiert S den Boden des Kellers. Sobald A_D das erste $\mathtt{b}$ einliest, wechselt der DPDA in den Zustand q_1 und beginnt eine Herunterzähl-Phase, in der für jedes $\mathtt{b}$ ein B vom Keller genommen wird. Durch das Symbol S erkennt A_D, dass alle B entfernt wurden, und wechselt in den akzeptierenden Zustand q_2. Wenn nun die ganze Eingabe gelesen wurde, akzeptiert A_D das Eingabewort. Es gilt also $\mathcal{L}_Z(A_D) = \mathcal{L}_Z(A_N) = \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. ◊

Die deterministischen Kellerautomaten benutzen wir zur Definition einer weiteren Sprachklasse:

Definition 4.78 Eine kontextfreie Sprache L ist **deterministisch kontextfrei**, wenn ein DPDA A existiert, so dass $L = \mathcal{L}_Z(A)$.

Wir bezeichnen die **Klasse aller deterministisch kontextfreien Sprachen** über dem Alphabet Σ mit DCFL_Σ, und definieren die Klasse aller determinstisch kontextfreien Sprachen $\mathrm{DCFL} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{DCFL}_\Sigma$.

Wir stellen zuerst fest, dass der Akzeptanzmodus bei DPDAs eine entscheidende Rolle spielt:

Lemma 4.79 *Sei A ein DPDA und sei $u \in \mathcal{L}_K(A)$. Dann existiert kein $w \in \mathcal{L}_K(A)$, so dass u ein echtes Präfix von w ist.*

Beweis: Wir beweisen dies indirekt: Angenommen, es existieren ein solcher DPDA $A := (\Sigma, \Gamma, Q, \delta, q_0, K_0, F)$ und solche Wörter u und w. Da u ist ein echtes Präfix von w ist, existiert ein $v \in \Sigma^*$ mit $w = uv$. Wegen $u \in \mathcal{L}_K(A)$ existiert außerdem ein $q \in Q$ mit

$$(q_0, u, K_0) \vdash *[A](q, \varepsilon, \varepsilon).$$

Da A deterministisch ist, können von der Startkonfiguration (q_0, u, K_0) nur Konfigurationen erreicht werden, die zwischen der Startkonfiguration und der Endkonfiguration $(q, \varepsilon, \varepsilon)$ liegen (jeder Schritt ist eindeutig festgelegt, und da der Keller in der Endkonfiguration leer ist, kann von dort nicht weitergerechnet werden).

Wird A nun auf der Eingabe uv gestartet, so arbeitet der DPDA erst einmal wie auf Eingabe u. Er gerät also zwangsläufig in die Konfiguration (q, v, ε), da

$$(q_0, uv, K_0) \vdash *[A](q, v, \varepsilon).$$

Da der Keller leer ist, kann A nicht weiterarbeiten; da aber die Eingabe noch nicht verarbeitet wurde, kann A auch nicht akzeptieren. Also gilt $uv \notin \mathcal{L}_K(A)$, und somit auch $w \notin \mathcal{L}_L(A)$, da $w = uv$. □

Bei deterministisch kontextfreien Sprachen schränkt Akzeptanz durch leeren Keller die Ausdrucksstärke also deutlich ein[60]. Ein Beispiel für eine Sprache, die auf diese Art nicht mehr ausgedrückt werden kann, ist die folgende:

Beispiel 4.80 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{w \in \Sigma^* \mid |w|_{\mathtt{a}} = |w|_{\mathtt{b}}\}$. Dann gilt $L \in$ DCFL (Sie können als Übung einen entsprechenden DPDA konstruieren). Sei nun $u := \mathtt{ab}$ und $w := \mathtt{abab}$. Dann ist $u, w \in L$, und u ist ein echtes Präfix von w. Also existiert gemäß Lemma 4.79 kein DPDA A mit $\mathcal{L}_K(A) = L$. ♢

Anders als bei den endlichen Automaten haben deterministische und nichtdeterministische Kellerautomaten eine unterschiedliche Ausdrucksstärke:

Satz 4.81 REG $\subset$ DCFL $\subset$ CFL

Beweis: Die Aussage REG $\subseteq$ DCFL folgt direkt aus der Definition der Automatenmodelle, da jeder DFA auch als DPDA interpretiert werden kann (der Keller wird einfach nicht verwendet). Beispiel für nichtreguläre Sprachen aus DCFL haben wir in Beispiel 4.77 und 4.80 betrachtet.

Die Aussage DCFL $\subseteq$ CFL folgt ebenfalls direkt aus der Definition (jeder DPDA ist ein PDA). Ein Beispiel für eine Sprache aus CFL $-$ DCFL ist die Sprache

$$L := \{ww^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}.$$

Einen PDA für diese Sprache haben wir in Beispiel 4.63 betrachtet. Auf einen ausführlichen Beweis, dass $L \notin$ DCFL, verzichten wir an dieser Stelle[61].

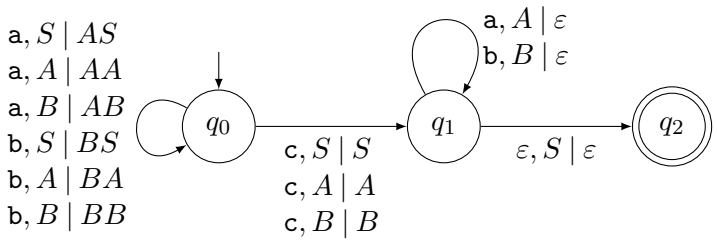
Stattdessen veranschaulichen wir sehr grob, warum kein DPDA für L existieren kann. Angenommen, es existiert ein DPDA A mit $\mathcal{L}(A) = L$. Beim Einlesen eines Wortes ww^R aus L muss A zuerst w abarbeiten und dabei w^R auf dem Keller speichern. Allerdings kann A nicht erkennen, wann w fertig eingelesen wurde. Der DPDA weiß also nicht, ab wann er überprüfen muss, ob die zweite Hälfte der Eingabe w^R entspricht. □

[60] Dies ist einer der Gründe, warum wir allgemein $\mathcal{L}(A) := \mathcal{L}_Z(A)$ für alle PDAs definiert haben.

[61] Am einfachsten zeigt man dies mit einem Resultat wie Theorem 4.7.4 aus Shallit [19]. Dieses besagt: Ist für eine Sprache L jede Äquivalenzklasse der Nerode-Relation $\equiv_L$ endlich, so ist $L \notin$ DCFL. Um zu zeigen, dass jede Äquivalenzklasse von $\equiv_L$ endlich ist, zeigt man dann noch $[x]_{\equiv_L} = \{x\}$ für alle $x \in \{\mathtt{a}, \mathtt{b}\}^*$.

Wir haben eben $\{ww^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$ betrachtet als Beispiel für eine kontextfreie Sprache, die Nichtdeterminismus benötigt. Durch eine leichte Abwandlung lässt sich allerdings eine sehr ähnliche Sprache konstruieren, die deterministisch kontextfrei ist:

Beispiel 4.82 Sei $\Sigma := \{\mathtt{a},\mathtt{b},\mathtt{c}\}$ und sei $L := \{w\mathtt{c}w^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$. Dann ist $L \in$ DCFL. Wir geben dazu den folgenden DPDA A mit Kelleralphabet $\Gamma := \{S, A, B\}$ und Startsymbol S an:



Dieser DPDA ist fast identisch mit dem PDA für die Sprache $\{ww^R \mid w \in \{\mathtt{a},\mathtt{b}\}^*\}$, den wir in Beispiel 4.63 betrachtet haben: Beide haben in q_0 eine „Speicher-Phase“, in der w eingelesen und w^R auf dem Keller gespeichert wird, und in q_1 ein „Lösch-Phase“, in der die Eingabe durch Leeren des Kellers mit w^R verglichen wird. Der einzige Unterschied ist, dass die Übergänge zwischen q_0 und q_1 keine ε-Übergänge sind, sondern den Buchstaben $\mathtt{c}$ einlesen. Das Symbol $\mathtt{c}$ teilt hierbei dem DPDA mit, wann von der ersten in die zweite Phase gewechselt werden soll. Dadurch ist hier kein Nichtdeterminismus mehr notwendig. ◊

Wir werden noch weitere Beispiel für kontextfreie Sprachen kennenlernen, die nicht deterministisch kontextfrei sind. Dabei ist das folgende Resultat hilfreich:

Satz 4.83 *Die Klasse* DCFL *ist abgeschlossen unter Komplementbildung.*

Auf den Beweis verzichten wir hier, da er recht technisch ist. Um aus einem DPDA A einen DPDA für das Komplement von $\mathcal{L}(A)$ zu konstruieren genügt es nicht, einfach durch Vertauschen von akzeptierenden und nicht-akzeptierende Zuständen einen Komplementautomaten A_K zu definieren (wie wir es für DFAs in Lemma 3.14 getan haben). Dabei gibt es zwei Probleme: Erstens kann A bei seiner Berechnung den Keller leeren. Wenn eine Eingabe u zu einer Konfiguration führt, kann A_K keine Eingabe der Form uv (mit $v \neq \varepsilon$) akzeptieren. Zweitens ist es möglich, dass A durch ε-Übergänge in Endlosschleifen gerät. Angenommen, A gerät nach kompletten Einlesen einer Eingabe w in eine solche Endlosschleife, die mindestens einen akzeptierenden und einen nicht-akzeptierenden Zustand enthält. Dann gilt $w \in \mathcal{L}(A)$, aber auch $w \in \mathcal{L}(A_K)$. Mit zuätzlichem Aufwand lassen sich diese Probleme allerdings lösen, so dass zu jedem DPDA A ein DPDA A_K mit $\mathcal{L}(A_K) = \overline{\mathcal{L}(A_K)}$ konstruiert werden kann.

Durch Satz 4.83 haben wir eine weitere Möglichkeit zu zeigen, dass eine Sprache nicht deterministisch kontextfrei ist:

Beispiel 4.84 Sei $\Sigma := \{\mathtt{a},\mathtt{b}\}$ und sei $\mathrm{COPY}_\Sigma := \{ww \mid w \in \Sigma^*\}$. Wir interessieren uns für das Komplement dieser Sprache, also $L := \overline{\mathrm{COPY}_\Sigma}$.

Mit etwas Aufwand können wir zeigen, dass L kontextfrei ist (siehe dazu Aufgabe 4.4). Angenommen, $L \in$ DCFL. Dann wäre nach Satz 4.83 auch $\overline{L} \in$ DCFL und somit $\overline{L} \in$ CFL. Widerspruch, da $\overline{L} = \text{COPY}_\Sigma$, und diese Sprache ist nicht kontextfrei (siehe Beispiel 4.36). Also gilt $L \in (\text{CFL} - \text{DCFL})$. ◊

Eine Sprache kann also nur dann deterministisch kontextfrei sein, wenn ihr Komplement kontextfrei ist (dieses muss dann natürlich auch deterministisch kontextfrei sein). Wir haben nun das Handwerkszeug, um ein paar weitere Abschlusseigenschaften zu betrachten und dadurch die Klasse DCFL ein wenig besser zu verstehen:

Lemma 4.85 *Die Klasse* DCFL *ist*

1. *nicht abgeschlossen unter Schnitt,*
2. *nicht abgeschlossen unter Mengendifferenz,*
3. *nicht abgeschlossen unter Vereinigung,*
4. *nicht abgeschlossen unter Homomorphismus,*
5. *nicht abgeschlossen unter regulärer Substitution,*
6. *abgeschlossen unter Schnitt mit regulären Sprachen,*
7. *abgeschlossen unter inversem Homomorphismus.*

Beweis: Wir skizzieren nur grob die Beweisideen.

Zu 1: Sei $L_1 := \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^j \mid i,j \in \mathbb{N}\}$ und $L_2 := \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^j \mid i,j \in \mathbb{N}\}$. Es gilt $L_1, L_2 \in$ DCFL, aber $L_1 \cap L_2$ ist nicht einmal kontextfrei.

Zu 2: Angenommen, DCFL ist abgeschlossen unter Mengendifferenz. Da DCFL unter Komplementbildung abgeschlossen ist folgt dann auch Abschluss unter Schnitt, denn für alle Sprachen L_1 und L_2 gilt

$$\begin{aligned} L_1 - \overline{L_2} &= L_1 \cap \overline{\overline{L_2}} \\ &= L_1 \cap L_2. \end{aligned}$$

Zu 6 (Idee): Sei $L \in$ CFL und $R \in$ REG. Verwende DPDA A_L mit $\mathcal{L}_Z(A_L) = L$ und DFA A_R mit $\mathcal{L}(A_R) = R$. Konstruiere einen DPDA als Produktautomat aus A_L und A_R.

Zu 3: Für alle Sprachen L_1 und L_2 gilt

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}.$$

Da DCFL unter Komplementbildung abgeschlossen ist, würde also aus Abschluss unter Vereinigung auch Abschluss unter Schnitt folgen.

Zu 4: Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $L := \{w\mathtt{c}w^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}$. Aus Beispiel 4.82 wissen wir, dass $L \in$ DCFL. Der Homomorphismus $h\colon \Sigma^* \to \Sigma^*$ sei definiert durch $h(\mathtt{a}) := \mathtt{a}$, $h(\mathtt{b}) := \mathtt{b}$, $h(\mathtt{c}) := \varepsilon$. Dann ist

$$h(L) = \{ww^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^*\}.$$

Wir wissen, dass $h(L) \notin$ DCFL, also ist DCFL nicht abgeschlossen unter Homomorphismen.

Zu 5: Da jeder Homomorphismus auch eine reguläre Substitution ist, folgt dies direkt aus Behauptung 4.

Zu 7: Dies lässt sich analog zu Lemma 4.74 beweisen. □

Auch sonst ist die Klasse DCFL nicht sonderlich robust: Mit etwas Aufwand kann man noch den Abschluss unter prefix und regulärem Rechts-Quotient beweisen; unter den Operationen Konkatenation, Shuffle-Produkt, suffix, Reversal, n-fache Konkatenation, Kleene-Plus und Kleene-Stern ist DCFL allerdings nicht abgeschlossen.

Die Arbeit mit deterministisch kontextfreien Sprachen hat aber einen entscheidenden Vorteil:

Satz 4.86 *Sei $L \in$ DCFL. Dann ist L nicht inhärent mehrdeutig.*

Auch hier verzichten wir auf den Beweis, Sie finden ihn zum Beispiel in Shallit [19]. Es gibt auch eine Klasse von Grammatiken, die sogenannten *$LR(k)$-Grammatiken*, deren Ausdrucksstärke eng mit der Klasse DCFL verbunden ist. Parser für diese Grammatiken sind im Prinzip DPDAs und finden in Compilern für viele Programmiersprachen Verwendung (siehe z. B. Aho et al. [1]).

Abschließend betrachten wir noch ein paar weitere Beispiele:

Beispiel 4.87 Wir betrachten die folgenden Sprachen:

$$
\begin{aligned}
L_1 &:= \{\mathtt{a}^i\mathtt{b}^j \mid i,j \in \mathbb{N}, i = j \text{ oder } 2i = j\},\\
L_2 &:= \{\mathtt{a}^i\mathtt{b}^j \mid i,j \in \mathbb{N}, i = j \text{ oder } i = 2j\},\\
L_3 &:= \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^j \mid i,j \in \mathbb{N}\} \cup \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^j \mid i,j \in \mathbb{N}\},\\
L_4 &:= \{\mathtt{aa}^i\mathtt{b}^i\mathtt{c}^j \mid i,j \in \mathbb{N}\} \cup \{\mathtt{ba}^i\mathtt{b}^j\mathtt{c}^j \mid i,j \in \mathbb{N}\}.
\end{aligned}
$$

Es gilt: $L_i \notin$ DCFL für $i \in \{1,2,3\}$, $L_4 \in$ DCFL. ◊

4.3 Entscheidungsprobleme

Wir haben uns bereits in Abschnitt 4.1.3 mit dem Wortproblem für CFGs befasst: Ist die Grammatik in CNF kann direkt der CYK-Algorithmus angewendet werden, andernfalls konvertiert man die Grammatik in CNF oder verwendet einen der Algorithmen, die beliebige CFGs als Eingabe akzeptieren (diese haben wir dort aber nur kurz erwähnt). Das Wortproblem für PDAs lässt sich auf das Wortproblem für CFGs reduzieren, indem man den PDA mittels der Tripelkonstruktion in eine CFG konvertiert.

Zwei weitere Probleme für kontextfreie Grammatiken (und dadurch auch indirekt für PDAs) sind ebenfalls vergleichsweise einfach zu bewältigen:

Leerheitsproblem für CFGs

Eingabe: Eine CFG G.
Frage: Ist $\mathcal{L}(G) = \emptyset$?

Unendlichkeitsproblem für CFGs

Eingabe: Eine CFG G.
Frage: Ist $|\mathcal{L}(G)| = \infty$?

Diese beiden Probleme sind effizient lösbar; die Suche nach entsprechenden Algorithmen sei Ihnen als Übung überlassen. Die meisten anderen interessanten Probleme für kontextfreie Grammatiken sind unentscheidbar. Damit wir dies beweisen können, müssen wir zuerst noch zu einem verwandten Thema abschweifen.

4.3.1 Das Postsche Korrespondenzproblem

Um die Unentscheidbarkeit verschiedener Entscheidungsprobleme zu kontextfreien Grammatik und (D)PDAs zu beweisen, verwenden wir das sogenannte *Postsche Korrespondenzproblem*, kurz *PCP*[62]. Dies lässt sich auf zwei verschiedene Arten definieren, von denen wir beide betrachten werden. In der ersten Variante wird das PCP über Paare von Homomorphismen definiert:

Definition 4.88 Seien Σ_1 und Σ_2 Alphabete und $g, h\colon \Sigma_1^* \to \Sigma_2^*$ Homomorphismen. Eine **Lösung** für das **Postsche Korrespondenzproblem (PCP)** für g und h ist ein Wort $x \in \Sigma_1^+$ mit $g(x) = h(x)$. Wir bezeichnen $(\Sigma_1, \Sigma_2, g, h)$ als **Instanz des PCP (in der Homomorphismenvariante)**.

Oft wird anstelle dieser Definition auch die folgende verwendet:

Definition 4.89 Sei Σ ein Alphabet und sei $P := ((x_1, y_1), \ldots, (x_n, y_n))$ mit $n \geq 1$ eine Folge von Paaren von Wörtern über Σ (d. h. $x_i, y_i \in \Sigma^*$). Eine **Lösung** für das **Postsche Korrespondenzproblem (PCP)** für P ist eine Folge $i_1, \ldots, i_k \in \{1, \ldots, n\}$ für $k \geq 1$ mit

$$x_{i_1} \cdot x_{i_2} \cdots x_{i_k} = y_{i_1} \cdot y_{i_2} \cdots y_{i_k}.$$

Wir bezeichnen (Σ, P) als **Instanz des PCP (in der Dominovariante)**.

Die Bezeichnung *Dominovariante* kommt daher, dass Paare aus P oft als Dominosteine bezeichnet werden. Jedes Paar (x_i, y_i) entspricht dabei einem Dominostein, der in der oberen Zeile mit x_i und in der unteren mit y_i beschriftet ist:

$$\left[\frac{x_1}{y_1}\right], \left[\frac{x_2}{y_2}\right], \ldots, \left[\frac{x_n}{y_n}\right]$$

[62] Das Postsche Korrespondenzproblem ist benannt nach Emil Leon Post; die Abkürzung PCP steht für *Post correspondence problem* (gelegentlich auch *Post's* statt *Post*). In der Komplexitätstheorie wird die Abkürzung PCP auch für ein anderes Modell verwendet (probabilistically checkable proof).

Eine Lösung des PCP in der Dominovariante entspricht eine Folge nebeinandergelegter Dominosteine, bei der in jede der beiden Zeilen das gleiche Wort entsteht.

Dabei ist leicht zu sehen, dass die beiden Definitionen äquivalent sind: Jeder Stein der Dominovariante entspricht einem Buchstaben aus Σ_1 der Homomorphismenvariante, die obere Zeile eines Steins a enthält $g(a)$, und die untere $h(a)$. Demzufolge entspricht das Alphabet Σ_2 (aus der Homomorphismenvariante) dem Alphabet Σ aus der Dominovariante; und die Buchstaben aus Σ_1 kann man als Bezeichner der Dominosteine interpretieren.

Wir definieren nun ein entsprechendes Entscheidungsproblem. Dazu verwenden wir die Homomorphismenvariante; genausogut könnten wir aber die Dominovariante benutzen.

Beispiel 4.90 Sei $\Sigma_1 := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und $\Sigma_2 := \{0, 1\}$. Wir definieren die Homomorphismen $g, h\colon \Sigma_1^* \to \Sigma_2^*$ durch

$$\begin{aligned} g(\mathtt{a}) &:= 0, & g(\mathtt{b}) &:= 01, & g(\mathtt{c}) &:= 11, \\ h(\mathtt{a}) &:= 01, & h(\mathtt{b}) &:= 1, & h(\mathtt{c}) &:= 10. \end{aligned}$$

Die Instanz $(\Sigma_1, \Sigma_2, g, h)$ des PCP in der Homomorphismenvariante lässt sich anhand der Folge

$$P := ((0, 01), (01, 1), (11, 10))$$

als Instanz (Σ_2, P) des PCP in der Dominovariante ausdrücken. Die entsprechenden Dominosteine sind dabei

$$\left[\frac{0}{01}\right], \left[\frac{01}{1}\right], \left[\frac{11}{10}\right].$$

Die Steine entsprechen (von links nach rechts) den Buchstaben $\mathtt{a}$, $\mathtt{b}$ und $\mathtt{c}$. Es ist leicht zu sehen, dass eine Lösung für diese Instanz des PCP mit $\mathtt{a}$ (dem ersten Stein) beginnen muss, da die beiden anderen Buchstaben Steine sofort zu einem Widerspruch führen. Danach ergibt sich $\mathtt{c}$ zwangsläufig als nächster Buchstabe, und wir erhalten

$$\begin{aligned} g(\mathtt{ac}) &= 011, \\ h(\mathtt{ac}) &= 0110. \end{aligned}$$

Nun können wir noch $\mathtt{b}$ anhängen, und erhalten

$$\begin{aligned} g(\mathtt{acb}) &= 01101, \\ h(\mathtt{acb}) &= 01101, \end{aligned}$$

und da $g(\mathtt{acb}) = 01101 = h(\mathtt{acb})$ haben ist $\mathtt{acb}$ eine Lösung dieser Instanz des PCP. $\Diamond$

Beispiel 4.91 Sei $\Sigma_1 := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ und $\Sigma_2 := \{0, 1\}$. Wir definieren die Homomorphismen $g, h\colon \Sigma_1^* \to \Sigma_2^*$ durch

$$\begin{aligned} g(\mathtt{a}) &:= 1, & g(\mathtt{b}) &:= 1, & g(\mathtt{c}) &:= 0, & g(\mathtt{d}) &:= 01, \\ h(\mathtt{a}) &:= 11, & h(\mathtt{b}) &:= 00, & h(\mathtt{c}) &:= 01, & h(\mathtt{d}) &:= 100. \end{aligned}$$

Die entsprechenden Dominosteine sind

$$\left[\frac{1}{11}\right], \left[\frac{1}{00}\right], \left[\frac{0}{01}\right], \left[\frac{01}{100}\right].$$

Diese Instanz des PCP hat keine Lösung. In diesem Fall ist dies leicht zu sehen: Für alle $a \in \Sigma_1$ gilt $|h(a)| > |g(a)|$, also ist $g(x) \neq h(x)$ für alle $x \in \Sigma_1^+$. ◊

Allerdings können die Lösungen selbst einfacher Instanzen des PCP schnell sehr groß werden, wie das folgende Beispiel zeigt:

Beispiel 4.92 Das folgende Beispiel stammt von Heiko Stamer[63]. Sei $\Sigma_1 := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und $\Sigma_2 := \{0, 1\}$. Wir definieren die Homomorphismen $g, h\colon \Sigma_1^* \to \Sigma_2^*$ durch

$$\begin{aligned} g(\mathtt{a}) &:= 0, & g(\mathtt{b}) &:= 01, & g(\mathtt{c}) &:= 1, \\ h(\mathtt{a}) &:= 1, & h(\mathtt{b}) &:= 0, & h(\mathtt{c}) &:= 101. \end{aligned}$$

Wir verzichten hier auf die Darstellung der Dominosteine, da sie diese direkt aus den Homomorphismen ablesen können. Die kürzeste Lösung für dieses Beispiel ist das Wort

$$s := \mathtt{babbababbabbcbbccbaacbbabcbbaabcbbbacbcbccac}.$$

Dieses hat die Länge 44, und es gilt

$$g(s) = h(s) = 01001010010010100101110101110100101010011010100011010101010110111 01.$$

Dies von Hand zu überprüfen ist natürlich lästig und aufwändig; und diese Lösung zu finden ist natürlich noch lästiger und noch aufwändiger. Aus diesem Grund erledigt man dies normalerweise mit einem Programm[64]. ◊

Die Frage, ob eine Instanz des PCP eine Lösung besitzt, ist also nicht unbedingt trivial. Um die Schwierigkeit genau bestimmen zu können, formulieren wir ein entsprechendes Entscheidungsproblem:

PCP

Eingabe: Eine Instanz $(\Sigma_1, \Sigma_2, g, h)$ des PCP.

Frage: Existiert eine Lösung für g und h?

Wie sich herausstellt, ist dieses Problem unentscheidbar:

[63] Die einzige Quelle, die ich für dieses Beispiel auftreiben konnte, ist eine archivierte Kopie einer inzwischen toten Webseite, nämlich `http://archive.today/JBK7M`. Dort finden sich auch weitere Beispiele.

[64] Ein solches Programm ist recht schnell geschrieben. Wenn Sie sich die Mühe sparen wollen: Um diese Lösung hier zu berechnen habe ich selbst ein kleines Programm erstellt, dass Sie unter `http://www.tks.informatik.uni-frankfurt.de/ddf/downloads#pcp` herunterladen können.

Satz 4.93 *Das PCP ist unentscheidbar.*

Beweisidee: Die Grundidee ist, das Halteproblem für Turingmaschinen auf das PCP zu reduzieren. Dazu definiert man zuerst Konfigurationen von Turingmaschinen, ähnlich zu unseren Konfigurationen von PDAs. Diese Konfigurationen fast man als Wörter auf; eine Berechnung von M entspricht dann eine Folge dieser Wörter. Diese Folge kann man anhand von Trennsymbolen als ein Wort codieren. Aus einer gegebenen Turingmaschine M konstruiert man nun durch geschickte Wahl der Dominosteine (oder Homomorphismen) eine Instanz des PCP, so dass eine Lösung dieser Instanz genau einer Codierung einer akzeptierenden Berechnung von M entspricht. □

Wir werden auf den Beweis dieses Resultats nicht weiter eingehen; eine ausführliche Darstellung finden Sie zum Beispiel in Sipser [20] oder Hopcroft und Ullman [8]. Stattdessen wenden wir dieses Resultat auf verschiedene Entscheidungsprobleme für kontextfreie Grammatiken und Sprachen an.

4.3.2 Unentscheidbare Probleme und kontextfreie Sprachen

Dank Satz 4.93 können wir nun auch Aussagen über die Entscheidbarkeit verschiedener Problem zu Modellen für kontextfreie Sprachen machen. Wir beginnen mit einem weiteren Entscheidungsproblem, das uns von großem Nutzen sein wird:

Schnitt-Problem für CFGs

Eingabe: Zwei CFGs G_1 und G_2.
Frage: Ist $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$?

Analog dazu definieren wir das Schnitt-Problem für DPDAs. Anhand von Satz 4.93 können wir folgendes zeigen:

Lemma 4.94 *Es gilt:*

1. *Das Schnitt-Problem für DPDAs ist unentscheidbar.*
2. *Das Schnitt-Problem für CFGs ist unentscheidbar.*

Beweis: Es genügt, die Unentscheidbarkeit für DPDAs zu zeigen. Daraus folgt dann unmittelbar die Unentscheidbarkeit für CFGs: Da wir jeden PDA (und somit auch jeder DPDA) anhand der Tripelkonstruktion berechenbar in eine äquivalente CFG umwandeln können, lässt sich das Schnitt-Problem für DPDAs sofort auf das Schnitt-Problem für CFGs reduzieren.

Wir zeigen die Unentscheidbarkeit des Schnitt-Problems für DPDAs, indem wir das PCP auf das Schnitt-Problem reduzieren. Sei $(\Sigma_1, \Sigma_2, g, h)$ eines Instanz des PCP (in der Homomorphismenvariante). Unser Ziel ist es, DPDAs A_g und A_h zu konstruieren, so dass $\mathcal{L}(A_g) = \mathcal{L}(A_h)$ genau dann gilt, wenn die PCP-Instanz $(\Sigma_1, \Sigma_2, g, h)$ eine Lösung hat.

Ohne Beeinträchtigung der Allgemeinheit nehmen wir an, dass $\Sigma_1 \cap \Sigma_2 = \emptyset$. Sei $\#$ ein neuer Buchstabe mit $\# \notin (\Sigma_1 \cup \Sigma_2)$. Sei $\Sigma := (\Sigma_1 \cup \Sigma_2 \cup \{\#\})\}$. Unser Ziel ist es nun, für beiden folgenden Sprachen $L_g, L_h \subseteq \Sigma^*$ zu finden:

$$L_g := \{a_n \cdots a_2 a_1 \# g(a_1) g(a_2) \cdots g(a_n) \mid n \geq 1, a_1, \ldots, a_n \in \Sigma_1\},$$
$$L_h := \{a_n \cdots a_2 a_1 \# h(a_1) h(a_2) \cdots h(a_n) \mid n \geq 1, a_1, \ldots, a_n \in \Sigma_1\}.$$

Behauptung 1 *Es gilt $(L_g \cap L_h) \neq \emptyset$ genau dann, wenn die Instanz $(\Sigma_1, \Sigma_2, g, h)$ des PCP eine Lösung hat.*

Beweis: Es gilt $(L_g \cap L_h) \neq \emptyset$ genau dann, wenn ein $x \in \Sigma_1^+$ existiert, so dass $x \in (L_g \cap L_h)$. Das ist genau dann der Fall, wenn ein $n \geq 1$ und $a_1, \ldots, a_n \Sigma_1$ existieren, so dass

$$a_n \cdots a_2 a_1 \# g(a_1 \cdot a_2 \cdot \cdots \cdot a_n) = a_n \cdots a_2 a_1 \# h(a_1 \cdot a_2 \cdot \cdots \cdot a_n),$$

und dies gilt genau dann, wenn $a_1, \ldots, a_n \Sigma_1$ existieren, für die

$$g(a_1 \cdot a_2 \cdot \cdots \cdot a_n) = h(a_1 \cdot a_2 \cdot \cdots \cdot a_n).$$

Dies wiederum gilt genau dann, wenn die PCP-Instanz $(\Sigma_1, \Sigma_2, g, h)$ eine Lösung hat. Die Lösungen der PCP-Instanz lassen sich also direkt aus den Wörtern von $L_g \cap L_h$ ablesen. □(Behauptung 1)

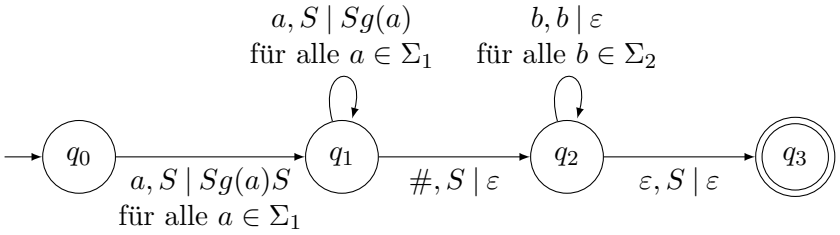
Wir müssen nun noch angeben, wie aus g und h DPDAs für diese beiden Sprachen konstruiert werden können. Dazu definieren wir zuerst den DPDA $A_g := (\Sigma, \Gamma, Q, \delta, q_0, S, q_3)$ mit

- $\Gamma := \Sigma_2 \cup \{S\}$ (wobei S ein neues Symbol ist),
- $Q = \{q_0, q_1, q_2, q_3\}$,

wobei δ wie folgt definiert sei:

$$\begin{aligned}
\delta(q_0, a, S) &:= (q_1, Sg(a)S) && \text{für alle } a \in \Sigma_1,\\
\delta(q_1, a, S) &:= (q_1, Sg(a)) && \text{für alle } a \in \Sigma_1,\\
\delta(q_1, \#, S) &:= (q_2, \varepsilon), &&\\
\delta(q_2, b, b) &:= (q_2, \varepsilon) && \text{für alle } b \in \Sigma_2,\\
\delta(q_2, \varepsilon, S) &:= (q_3, \varepsilon). &&
\end{aligned}$$

Graphisch lässt sich der DPDA A_g wie folgt skizzieren:



Nun ist leicht zu sehen, dass $\mathcal{L}_Z(A_g) = L_g$ gilt. Der DPDA A_g arbeitet dabei in zwei Phasen: In q_0 und q_1 liest A_g ein Folge von Buchstaben $a_n, \ldots, a_1 \in \Sigma_1$ (mit $n \geq 1$) ein und legt dabei das Wort $Sg(a_1) \cdots g(a_n)S = Sg(a_1 \cdots a_n)S$ auf den Keller. Sobald der DPDA das Terminal # sieht, entfernt er das obere S vom Keller und wechselt in den Zustand q_2. Dort beginnt die zweite Phase, in der A_g die Eingabe mit dem Kellerinhalt $g(a_1 \cdots a_n)$ abgleicht. Sobald nur noch das Zeichen S auf dem Keller ist, wechselt A_g in den Zustand q_3 und akzeptiert (falls die Eingabe vollständig eingelesen wurde). Analog zu A_g definieren wird den DPDA A_h für die Sprache L_h.

Angenommen, das Schnitt-Problem für DPDAs ist entscheidbar. Dann können wir zu jeder Instanz $I := (\Sigma_1, \Sigma_2, g, h)$ des PCP zwei DPDAs A_g und A_h konstruieren, so dass $(\mathcal{L}(A_g) \cap \mathcal{L}(A_h)) \neq \emptyset$ genau dann gilt, wenn I eine Lösung hat. Somit wäre das PCP entscheidbar, Widerspruch zu Satz 4.93. □

Lemma 4.94 hat mehrere unmittelbare Konsequenzen. Analog zu den Entscheidungsproblemen für DFAs definieren wir das Universalitäts-, das Inklusions- und das Äquivalenzproblem für CFGs:

Universalitätsproblem für CFGs

Eingabe: Eine CFL G über einem Alphabet Σ.
Frage: Ist $\mathcal{L}(G) = \Sigma^*$?

Inklusionsproblem für CFGs

Eingabe: Zwei CFLs G_1 und G_2.
Frage: Ist $\mathcal{L}(G_1) \subseteq \mathcal{L}(G_2)$?

Äquivalenzproblem für CFGs

Eingabe: Zwei CFGs G_1 und G_2.
Frage: Ist $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

Durch eine Anpassung des Beweises von Lemma 4.94 können wir zeigen, dass auch diese Probleme unentscheidbar sind:

Satz 4.95 *Die folgenden Probleme für kontextfreie Grammatiken sind unentscheidbar:*

- *Das Universalitätsproblem,*
- *das Inklusionsproblem,*
- *das Äquivalenzproblem.*

Beweis: Wir zeigen die Unentscheidbarkeit des Universalitätsproblem. Da sich eine CFG für die Sprache Σ^* angeben lässt, folgt daraus auch die Unentscheidbarkeit der anderen beiden Probleme.

Um zu zeigen, dass das Universalitätsproblem unentscheidbar ist, reduzieren wir das PCP darauf. Sei $(\Sigma_1, \Sigma_2, g, h)$ eine Instanz des PCP. Wir konstruieren nun DPDAs A_g und A_h wie im Beweis von Lemma 4.94. Zu jedem DPDA A lässt sich ein DPDA für das Komplement der Sprache $\mathcal{L}(A)$ konstruieren. Also können wir DPAs A_g^K und A_h^K konstruieren mit $\mathcal{L}(A_g^K) = \overline{\mathcal{L}(A_g)}$ und $\mathcal{L}(A_h^K) = \overline{\mathcal{L}(A_h)}$. Anhand der Tripelkonstruktion konvertieren wir diese in kontextfreie Grammatiken G_g^k und G_h^k mit $\mathcal{L}(G_g^K) = \overline{\mathcal{L}(A_g)}$ und $\mathcal{L}(G_h^K) = \overline{\mathcal{L}(A_h)}$. Daraus konstruieren wir nun eine kontextfreie Grammatik G mit $\mathcal{L}(G) = (\mathcal{L}(G_g^k) \cup \mathcal{L}(G_g^k))$. Nun gilt:

$$\begin{aligned}
\mathcal{L}(G) &= \mathcal{L}(G_g^k) \cup \mathcal{L}(G_h^k) \\
&= \mathcal{L}(A_g^k) \cup \mathcal{L}(A_h^k) \\
&= \overline{\mathcal{L}(A_g)} \cup \overline{\mathcal{L}(A_h)} \\
&= \overline{L_g} \cup \overline{L_h} \\
&= \overline{L_g \cap L_h}.
\end{aligned}$$

Somit ist $\mathcal{L}(G) = \Sigma^*$ genau dann, wenn $(L_g \cap L_h) = \emptyset$. Ein Algorithmus, der das Universalitätsproblem für CFGs entscheidet, kann also auch das PCP (und das Schnittproblem für DPDAs) entscheiden. Widerspruch. □

Da das Universalitätsproblem für CFGs unentscheidbar ist, wissen wir, dass kein Minimierungsalgorithmus für CFGs oder PDAs existieren kann. Für DPDAs ist die Situation übrigens anders: Das Inklusionsproblem ist zwar unentscheidbar, allerdings ist das Universalitätsproblem für DPDAs entscheidbar. Die Frage, ob auch das Äquivalenzproblem entscheidbar ist, war lange Zeit offen. Im Jahr 1997 hat dann Géraud Sénizergues gezeigt, dass das Problem entscheidbar ist, und hat dafür im Jahr 2002 den Gödel-Preis erhalten. Der Artikel mit dem vollständige Beweis für dieses Resultat (Sénizergues [17]) ist 166 Seiten[65] lang, die vereinfachte Version (Sénizergues [18]) umfasst immer noch 54 Seiten.

Ein weiteres Problem, dass im Zusammenhang mit kontextfreien Grammatiken interessant ist, ist das folgende:

Mehrdeutigkeitsproblem für CFGs

Eingabe: Eine CFG G.

Frage: Ist G mehrdeutig?

Auch dieses Problem ist unentscheidbar:

[65] Da Zeitschriftenartikel ein überaus kompaktes Format haben, können Sie davon ausgehen, dass der Artikel länger ist als dieses Skript.

Satz 4.96 *Das Mehrdeutigkeitsproblem für CFGs ist unentscheidbar.*

Beweis: Wir verwenden hierfür wieder die Sprachen L_g und L_h aus dem Beweis von Lemma 4.94. Zu jeder der beiden Sprachen lässt sich eine kontextfreie Grammatik angeben. Sei $(\Sigma_1, \Sigma_2, g, h)$ eine Instanz des PCP, und sei $\Sigma := (\Sigma_1 \cup \Sigma_2 \cup \{\#\})$. Wir definieren kontextfreie Grammatiken $G_g := (\Sigma, V_g, P_g, S_g)$ und $G_h := (\Sigma, V_h, P_h, S_h)$ mit $V_g = \{S_g, A_g\}$, $V_h = \{S_h, A_h\}$ sowie

$$P_g = \{S_g \to aA_g g(a) \mid a \in \Sigma_1\} \cup \{A_g \to aA_g g(a) \mid a \in \Sigma_1\} \cup \{A_g \to \#\},$$
$$P_h = \{S_h \to aA_h h(a) \mid a \in \Sigma_1\} \cup \{A_h \to aA_h h(a) \mid a \in \Sigma_1\} \cup \{A_h \to \#\}.$$

Wie leicht zu erkennen ist, gilt $\mathcal{L}(G_g) = L_g$ und $\mathcal{L}(G_h) = L_h$. Außerdem sind beide Grammatiken eindeutig: In jeder Satzform der beiden Grammatiken kommt immer nur höchstens eine Variable vor, und bei jeder Linksableitung ist durch die Terminale eindeutig bestimmt, welche Regel verwendet werden muss. Also hat jedes Wort aus $\mathcal{L}(G_g)$ bzw. $\mathcal{L}(G_h)$ genau eine Linksableitung und somit (gemäß Lemma 4.49) genau einen Ableitungsbaum.

Wir definieren nun eine kontextfreie Grammatik $G := (\Sigma, V, P, S)$ mit

$$V := V_g \cup V_h \cup \{S\},$$
$$P := V_g \cup V_h \cup \{S \to S_g, S \to S_h\}.$$

Da G_g und G_h eindeutig sind, ist G genau dann mehrdeutig, wenn $(\mathcal{L}(G_g) \cap \mathcal{L}(G_h)) \neq \emptyset$. Wie im Beweis von Lemma 4.94 dargelegt, ist dies genau dann der Fall, wenn die PCP-Instanz $(\Sigma_1, \Sigma_2, g, h)$ eine Lösung hat. Ein Algorithmus zum Entscheiden des Mehrdeutigkeitsproblems für CFGs kann also zum Entscheiden des PCP verwendet werden. Widerspruch. □

Die Frage, ob eine gegeben kontextfreie Grammatik mehrdeutig ist oder nicht haben Sie schon in einigen Übungsaufgaben bearbeitet. Durch Satz 4.96 wissen wir nun, dass es keinen Algorithmus gibt, der diese Fragestellung immer zuverlässig löst. Außerdem gibt Satz 4.96 auch Grund zur Vermutung, dass es keinen Algorithmus geben kann, der Mehrdeutigkeit aus Grammatiken eliminiert (zumindest aus Grammatiken, deren Sprachen nicht inhährent mehrdeutig sind). Mit zusätzlichem Aufwand kann man beweisen, dass so ein Algorithmus nicht existieren kann, selbst wenn man garantiert bekommt, dass eine eindeutige Grammatik existiert.

Eine weitere natürliche Frage zu kontextfreien Grammatiken ist, ob diese reguläre Sprachen erzeugen.

Regularitätsproblem für CFGs

Eingabe: Eine CFG G.

Frage: Ist $\mathcal{L}(G)$ regulär?

Satz 4.97 *Das Regularitätsproblem für CFGs ist nicht entscheidbar.*

Beweis: Wir reduzieren das Universalitätsproblem für CFGs auf das Regularitätsproblem für CFGs. Sei $G := (\Sigma, V, P, S)$ eine kontextfreie Grammatik mit $|\Sigma| = 2$. Wir definieren $L := \mathcal{L}(G)$. O. B. d. A. gelte $\mathtt{a}, \mathtt{b} \in \Sigma$. Sei $L_0 := \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Wir wählen nun einen neuen Buchstaben $\# \notin \Sigma$ und definieren

$$L_1 := (L_0 \cdot \{\#\} \cdot \Sigma^*) \cup (\Sigma^* \cdot \{\#\} \cdot L).$$

Da L_1 und $L(G)$ kontextfrei sind, ist auch L_1 kontextfrei. Darüber hinaus können wir aus G und einer CFG für L_0 auch eine CFG G_1 mit $L_1 = \mathcal{L}(G_1)$ konstruieren. Nun gilt folgendes:

Behauptung 1 *Es gilt $L_1 \in$ REG genau dann, wenn $L = \Sigma^*$.*

Beweis: Wir beginnen mit der Rück-Richtung: Sei $L = \Sigma^*$. Dann ist $L_1 = (\Sigma^* \cdot \{\#\} \cdot \Sigma^*)$. Diese Sprache ist offensichtlich regulär.

Für die Hin-Richtung zeigen wir, dass aus $L \neq \Sigma^*$ stets $L_1 \notin$ REG folgt. Angenommen, $L \neq \Sigma^*$ und $L_1 \in$ REG. Da $L \neq \Sigma^*$ existiert ein Wort $w \in (\Sigma^* - L)$. Da REG abgeschlossen ist unter regulärem Rechts-Quotient (Lemma 3.112) ist auch $L_1/\{\#w\}$ regulär. Es gilt aber

$$\begin{aligned} L_1/\{\#w\} &= \{u \in \Sigma^* \mid u\#w \in L_1\} \\ &= \{u \in \Sigma^* \mid u \in L_0\} && (\text{da } w \notin L) \\ &= L_0. \end{aligned}$$

Somit folgt hieraus $L_0 \in$ REG, Widerspruch. □(Behauptung 1)

Also ist $L_1 = \mathcal{L}(G_1) \in$ REG genau dann, wenn $L(G) = \Sigma^*$. Da das Universalitätsproblem für CFGs nicht entscheidbar ist, kann das Regularitätsproblem für CFGs auch nicht entscheidbar sein. □

Der Beweis von Satz 4.97 lässt sich zu einem Resultat verallgemeinern, das als *Satz von Greibach*[66] bekannt ist. Durch diesen lässt sich unter anderem auch beweisen, dass ebenfalls nicht entscheidbar ist, ob die von einer CFG G erzeugte Sprache deterministisch kontextfrei oder inhärent mehrdeutig ist.

Satz 4.97 zeigt unter anderem auch nahe, dass es keine garantiert zuverlässige Methode geben kann, Aufgaben der Art "Beweisen oder widerlegen Sie, dass die folgende Sprache regulär ist" zu lösen. Der tatsächliche Beweis erfordert zusätzlichen Aufwand.

4.4 Aufgaben

Aufgabe 4.1 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$ und seien

$$L_1 := \left\{ w \in \Sigma^* \mid |w|_\mathtt{a} = |w|_\mathtt{b} = \sqrt[|w|_\mathtt{c}]{|w|_\mathtt{d}} \right\},$$
$$L_2 := \left\{ w \in \Sigma^* \mid |w|_\mathtt{a} = \sqrt[|w|_\mathtt{c}]{|w|_\mathtt{b}} \right\},$$

[66] Benannt nach Sheila Adele Greibach. Mehr Details zum Satz von Greibach erfahren Sie zum Beispiel in Abschnitt 8.7 von Hopcroft und Ullman [8]

$$L_3 := \left\{ w \in \Sigma^* \mid |w|_{\mathtt{a}} = (|w|_{\mathtt{b}})^{1+|w|_{\mathtt{c}}} \right\}.$$

Zeigen Sie: L_1, L_2 und L_3 sind nicht kontextfrei.

Aufgabe 4.2 Beweisen Sie Lemma 4.75.

- Beweisen Sie, dass CFL unter Rechts-Quotient mit regulären Sprachen abgeschlossen ist. Zeigen Sie dazu: Ist $L \in \mathrm{CFL}_\Sigma$ und $R \in \mathrm{REG}_\Sigma$, dann ist die Sprache

  $$L/R = \{x \in \Sigma^* \mid xy \in L, y \in R\}$$

  kontextfrei.

- Beweisen Sie, dass CFL unter prefix und suffix abgeschlossen ist.

Aufgabe 4.3 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $L := \{w \in \Sigma^* \mid |w|_{\mathtt{a}} = |w|_{\mathtt{b}}\}$. Zeigen Sie: $L \in \mathrm{DCFL}$. Konstruieren Sie dazu einen DPDA A mit $\mathcal{L}_Z(A) = L$.

Aufgabe 4.4 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $\mathrm{COPY}_\Sigma := \{ww \mid w \in \Sigma^*\}$. Zeigen Sie: $\overline{\mathrm{COPY}_\Sigma}$ ist kontextfrei.

Aufgabe 4.5 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$. Für $n \in \mathbb{N}$ sei

$$L_n := \{w^n \mid w \in \Sigma^*\}.$$

1. Zeigen Sie: Für $n \geq 2$ ist L_n nicht kontextfrei.
2. Für welche n ist $\overline{L_n}$ kontextfrei? Beweisen Sie Ihre Antwort. Es lohnt sich, Aufgabe 4.4 gelöst zu haben.

4.5 Bibliographische Anmerkungen

Dieser Abschnitt ist momentan nur ein Platzhalter. In Kürze werden hier einige Kommentare zu den verwendeten Quellen und weiterführendem Lesematerial zu finden sein.

5 Jenseits der kontextfreien Sprachen

Dieses Kapitel befasst sich mit einer Reihe von Modelle, die über die Klasse der kontextfreien Sprachen hinausgehen.

5.1 Kontextsensitive Sprachen

Wir haben in Kapitel 4 nicht nur die Klasse CFL der kontextfreien Sprachen kennen gelernt, sondern auch einige Sprachen außerhalb dieser Klasse. In diesem Abschnitt befassen wir uns mit Mechanismen, die nicht-kontextfreie Sprachen beschrieben können. Dabei beginnen wir mit einer weiteren Verallgemeinerung, den *kontextsensitiven Sprachen*.

5.1.1 Kontextsensitive und monotone Grammatiken

Definition 5.1 Eine **kontextsensitive Grammatik (CSG)** G über einem Alphabet Σ wird definiert durch

1. eine Alphabet V von **Variablen**, wobei $(\Sigma \cap V) = \emptyset$,
2. eine endliche Menge $P \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ von **Regeln** der Form

$$\alpha A \beta \to \alpha \gamma \beta$$

   mit $\alpha, \beta \in (\Sigma \cup V)^*$, $\gamma \in (\Sigma \cup V)^+$ und $A \in V$,

3. eine Variable $S \in V$ (dem **Startsymbol**).

Wir schreiben dies als $G := (\Sigma, V, P, S)$. Wenn das Startsymbol S auf keiner rechten Seite einer Regel aus P erscheint, darf eine kontextsensitive Grammatik auch die Regel $S \to \varepsilon$ enthalten.

Im Gegensatz zu kontextfreien Grammatiken können kontextsensitive Grammatiken auf der linken Seite einer Regel noch zusätzliche Zeichen enthalten, den sogenannten **Kontext**). Wie wir in der Definition der Semantik der kontextsensitiven Grammatiken (Definition 5.3) sehen werden, bestimmt der Kontext ob eine Regel angewendet werden kann, die Regel beeinflusst den Kontext aber nicht. Anschaulich bedeutet eine Regel $\alpha A \beta \to \alpha \gamma \beta$, dass A durch γ ersetzt werden kann, aber nur im Kontext α, β.

Bevor wir uns der Semantik zuwenden, betrachten wir noch eine nützliche Verallgemeinerung der kontextsensitiven Grammatiken:

Definition 5.2 Eine **monotone Grammatik**[67] G über einem Alphabet Σ wird definiert durch

1. eine Alphabet V von **Variablen** (auch **Nichtterminal** oder **Nichtterminalsymbol** genannt), wobei $(\Sigma \cap V) = \emptyset$,
2. eine endliche Menge $P \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ von **Regeln** (oder **Produktionen**) der Form
$$\alpha \to \beta$$
mit $\alpha, \beta \in (\Sigma \cup V)^*$ und $|\alpha| \leq |\beta|$,
3. eine Variable $S \in V$ (dem **Startsymbol**).

Wir schreiben dies als $G := (\Sigma, V, P, S)$. Wenn das Startsymbol S auf keiner rechten Seite einer Regel aus P erscheint, darf eine monotone Grammatik auch die Regel $S \to \varepsilon$ enthalten.

An der Definition ist leicht zu erkennen, dass jede kontextsensitive Grammatik auch eine monotone Grammatik ist. Wir können uns daher darauf beschränken, den Begriff einer Ableitung nur für monotone Grammatiken zu definieren und müssen CSGs nicht getrennt behandeln.

Definition 5.3 Sei $G := (\Sigma, V, P, S)$ eine monotone Grammatik. Ein Wort $\alpha \in (\Sigma \cup V)^*$ bezeichnen wir auch als **Satzform** von G. Auf der Menge aller Satzformen von G definieren wir die Relation $\Rightarrow_G$ wie folgt:

Für alle $\gamma_1, \gamma_2 \in (\Sigma \cup V)^*$ gilt $\alpha \Rightarrow_G \beta$ genau dann, wenn Satzformen $\alpha_1, \alpha_2 \in (\Sigma \cup V)^*$ und eine Regel $(\gamma_1, \gamma_2) \in P$ existieren, so dass

$$\alpha = \alpha_1 \cdot \gamma_1 \cdot \alpha_2, \qquad \text{und} \qquad \beta = \alpha_1 \cdot \gamma_2 \cdot \alpha_2.$$

Wir erweitern die Relation $\Rightarrow_G$ analog zu Definition 3.115 sowohl zu Relationen $\Rightarrow_G^n$ für alle $n \in \mathbb{N}$, als auch zu einer Relation $\Rightarrow_G^*$.

Die von G **erzeugte Sprache** $\mathcal{L}(G)$ definieren wir als

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

Da jede kontextsensitive Grammatik auch monoton ist, gilt diese Definition auch für kontextsensitive Grammatiken. Eine Sprache $L \subseteq \Sigma^*$ heißt **kontextsensitiv**, wenn eine kontextsensitive Grammatik G existiert, für die $\mathcal{L}(G) = L$. Wir bezeichnen[68] die **Klasse aller kontextsensitive Sprachen** über dem Alphabet Σ mit CSL_Σ, und definieren die Klasse aller kontextfreien Sprachen $\mathrm{CSL} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{CSL}_\Sigma$.

[67] In der englischsprachigen Literatur werden monotone Grammatiken oft auch als *length increasing grammar* bezeichnet.

Manche Autoren verwenden den Begriff „kontextsensitive Grammatik“ für monotone Grammatiken. Dies wird dadurch gerechtfertigt, dass beide Modelle die gleiche Ausdrucksstärke haben:

Satz 5.4 *Sei G eine monotone Grammatik. Dann ist $\mathcal{L}(G)$ kontextsensitiv.*

Beweisidee: Dieser Satz lässt sich beweisen, indem man zeigt, dass aus jeder monotonen Grammatik $G = (\Sigma, V, P, S)$ eine kontextsensitive Grammatik G_S konstruiert werden kann, für die $\mathcal{L}(G_S) = \mathcal{L}(G)$ gilt. Wir skizzieren hier nur die Konstruktionsweise, einen ausführlichen Beweis finden Sie zum Beispiel in Abschnitt 7.1 von Shallit [19]. Die Konstruktion verläuft dabei in zwei Schritten.

1. Schritt: Zuerst wird sichergestellt, dass alle Regeln eine der beiden folgenden Formen haben:

1. $\alpha \to \beta$, mit $\alpha, \beta \in V^+$, oder
2. $A \to a$, mit $A \in V$, $a \in \Sigma$.

Dies ist durch einführen zusätzlicher Variablen der Form V_a und entsprechender Regeln $V_a \to a$ problemlos möglich.

2. Schritt: Nach dem ersten Schritt betrachten wir alle Regeln, die Variablen auf der rechten Seite haben. Wenn die linke Seite aus genau einer Variable bestehet können wir die Regel ersatzlos übernehmen. In allen anderen Fällen hat die Regel die Form

$$A_1 \cdots A_m \to B_1 \cdots B_n$$

mit $m, n \in \mathbb{N}_{>0}$, $m \geq 2$, $n \geq m$ (den Fall $m = 1$ haben wir gerade ausgeschlossen, und da G monoton ist, muss $m \leq n$ gelten) und $A_1, \ldots, A_m, B_1, \ldots, B_n \in V$. Wir ersetzen diese Regel nun durch folgende Regeln:

$$\begin{aligned}
A_1 A_2 \cdots A_m &\to X_1 A_2 \cdots A_m, \\
X_1 A_2 \cdots A_m &\to X_1 X_2 A_3 \cdots A_m, \\
&\vdots \\
X_1 X_2 \cdots X_{m-1} A_m &\to X_1 X_2 \cdots X_m B_{m+1} \cdots B_n, \\
X_1 X_2 \cdots X_m B_{m+1} \cdots B_n &\to B_1 X_2 \cdots X_m B_{m+1} \cdots B_n, \\
&\vdots \\
B_1 \cdots B_{m-1} X_m B_{m+1} \cdots B_n &\to B_1 \cdots B_n,
\end{aligned}$$

wobei $X_1, \ldots, X_m$ neue Variablen sind. Anschaulich gesprochen ersetzen wir also zuerst schrittweise die Variablen $A_1, \ldots, A_{m-1}$ durch neue Variablen $X_1, \ldots, X_{m-1}$. Dann wird A_m durch $X_m B_{m+1} \cdots B_n$ ersetzt. Anschließend ersetzen wir $X_1, \ldots, X_{m-1}$ schrittweise durch $B_1, \ldots, B_{m-1}$.

Jede dieser Regeln ist kontextsensitiv, und da keine der Variablen X_i in einer anderen Regel vorkommen darf wird so die erzeugte Sprache nicht verändert (der eigentlich Beweis dafür ist aufwändig und wird hier ausgelassen). □

[68] Die Abkürzung CSL stammt von *context-sensitive language(s)*.

Dank Satz 5.4 können wir kontextsensitive Grammatiken anhand von monotonen Grammatiken definieren. In vielen Fällen ist dies deutlich einfacher, allerdings sind auch monotone Grammatiken oft recht schwer zu lesen. Wir betrachten hierzu drei Beispiele:

Beispiel 5.5 Seit $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $G := (\Sigma, V, P, S)$ eine kontextsensitive Grammatik mit $V := \{S, A, B, C\}$ und

$$\begin{aligned} P := &\{S \to SABC, S \to ABC\} \\ &\cup \{XY \to YX \mid X, Y \in \{A, B, C\}\} \\ &\cup \{A \to \mathtt{a}, B \to \mathtt{b}, C \to \mathtt{c}\}. \end{aligned}$$

Sei außerdem $L := \{w \in \Sigma^+ \mid |w|_\mathtt{a} = |w|_\mathtt{b} = |w|_\mathtt{c}\}$. Es gilt $L = \mathcal{L}(G)$. Dies lässt sich auch vergleichsweise leicht zeigen. Um zu beweisen, dass $L \subseteq \mathcal{L}(G)$ wählen wir ein $w \in L$. Dann existiert ein $n \in \mathbb{N}_{>0}$ mit $n = |w|_\mathtt{a} = |w|_\mathtt{b} = |w|_\mathtt{c}$. Durch $(n-1)$-faches Anwenden der Regel $S \to SABC$ und anschließendes Anwenden von $S \to ABC$ erhalten wir

$$S \Rightarrow_G^n (ABC)^n.$$

Durch anwenden der Regeln der Form $XY \to YX$ können wir nun die Variablen in dieser Satzform beliebig untereinander vertauschen. Anschließend leiten wir sie entsprechend zu Terminalen ab und erhalten so w.

Um zu zeigen, dass $\mathcal{L}(G) \subseteq L$ stellen wir zuerst fest, dass alle aus S ableitbaren Satzformen γ (also alle $\gamma \in (\Sigma \cup V)^*$ mit $S \Rightarrow_G^* \gamma$) die Gleichung

$$|\gamma|_A + |\gamma|_\mathtt{a} = |\gamma|_B + |\gamma|_\mathtt{b} = |\gamma|_C + |\gamma|_\mathtt{c}$$

erfüllen[69]. Insbesondere gilt

$$|w|_\mathtt{a} = |w|_\mathtt{b} = |w|_\mathtt{c}$$

für alle $w \in \Sigma^+$ mit $S \Rightarrow_G^* w$, und somit für alle $w \in \mathcal{L}(G)$. Also ist $\mathcal{L}(G) \subseteq L$. ◊

Beispiel 5.6 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $G := (\Sigma, V, P, S)$ eine kontextsensitive Grammatik mit $V := \{S, B\}$, und

$$P := \{S \to \mathtt{a}SB\mathtt{c}, \qquad S \to \mathtt{abc}, \qquad \mathtt{c}B \to B\mathtt{c}, \qquad \mathtt{b}B \to \mathtt{bb}.\}$$

Sei außerdem $L := \{\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i \mid i \in \mathbb{N}_{>0}\}$. Wir behaupten nun, dass $\mathcal{L}(G) = L$. Dabei ist leicht zu sehen, dass $L \subseteq \mathcal{L}(G)$ gilt: Für $\mathtt{abc} \in L$ können wir einfach die Regel $S \to \mathtt{abc}$ anwenden. Für alle $n \in \mathbb{N}$ und die entsprechenden Wörter $\mathtt{a}^{n+1}\mathtt{b}^{n+1}\mathtt{c}^{n+1}$ wenden wir zuerst die Regel $S \to \mathtt{a}SB\mathtt{c}$ insgesamt n-mal an und erhalten so

$$S \Rightarrow_G^n \mathtt{a}^n S(B\mathtt{c})^n,$$

aus der wir mittels der Regel $S \to \mathtt{abc}$ die Satzform $\mathtt{a}^{n+1}\mathtt{bc}(B\mathtt{c})^n$ ableiten. Durch die Regel $\mathtt{c}B \to B\mathtt{c}$ schieben wir nun alle B nach links. Es gilt also

$$S \Rightarrow_G^* \mathtt{a}^{n+1}\mathtt{bc}(B\mathtt{c})^n \Rightarrow_G^* \mathtt{a}^{n+1}\mathtt{b}B^n\mathtt{c}^{n+1}.$$

[69] Dies sieht man direkt, oder man beweist es durch eine kleine Induktion über die anwendbaren Regeln.

Nun wenden wir n-mal die Regel $\mathtt{b}B \to \mathtt{bb}$ an und erhalten

$$S \Rightarrow_G^* \mathtt{a}^{n+1}\mathtt{b}B^n\mathtt{c}^{n+1} \Rightarrow_G^* \mathtt{a}^{n+1}\mathtt{b}^{n+1}\mathtt{c}^{n+1}.$$

Es gilt also $L \subseteq \mathcal{L}(G)$.

Für die andere Richtung der Inklusion ist Vorsicht geboten. Da die Grammatik sich die Reihenfolge der Ableitungen aussuchen kann, müssen wir besonders aufpassen, dass hier keine unerwarteten Nebeneffekte auftreten. Die Regeln $\mathtt{c}B \to B\mathtt{c}$ und $\mathtt{b}B \to \mathtt{bb}$ können erst angewendet werden, wenn die Regel $S \to \mathtt{abc}$ angewendet wurde (und ab diesem Zeitpunkt können nur noch diese beiden Regeln angewendet werden, da die Satzform kein S mehr enthält). Ist also $w \in \mathcal{L}(G)$, so existiert ein $n \in \mathbb{N}$ mit

$$S \Rightarrow_G^* \mathtt{a}^{n+1}\mathtt{bc}(B\mathtt{c})^n \Rightarrow_G^* w.$$

Die Variablen B können nur entfernt werden, indem sie mittels der Regel $\mathtt{b}B \to \mathtt{bb}$ ersetzt werden. Dazu müssen sie durch die Regel $\mathtt{c}B \to B\mathtt{c}$ nach links geschoben werden. Die genaue Reihenfolge, in der diese beiden Regeln verwendet werden, ist dabei größtenteils unerheblich, da die beiden Regeln sich nicht in die Quere kommen können. Also gilt

$$S \Rightarrow_G^* \mathtt{a}^{n+1}\mathtt{b}^{n+1}\mathtt{c}^{n+1} = w$$

und somit $w \in L$. Also ist $\mathcal{L}(G) \subseteq L$. ◊

Beispiel 5.7 Sei Σ ein Alphabet und sei $\mathrm{COPY}_\Sigma := \{xx \mid x \in \Sigma^*\}$. Wir betrachten nun eine monotone Grammatik G mit $\mathcal{L}(G) = (\mathrm{COPY}_\Sigma - \{\varepsilon\})$. Dazu definieren wir $G := (\Sigma, V, P, S)$ mit

$$V := \{S\} \cup \{L_a, R_a, X_a \mid a \in \Sigma\}$$

und

$$\begin{aligned} P := &\{S \to aSX_a \mid a \in \Sigma\} \\ &\cup \{S \to L_aR_a \mid a \in \Sigma\} \\ &\cup \{R_aX_b \to X_bR_a \mid a, b \in \Sigma\} \\ &\cup \{R_a \to a \mid a \in \Sigma\} \\ &\cup \{L_aX_b \to L_aR_b \mid a, b \in \Sigma\} \\ &\cup \{L_a \to a \mid a \in \Sigma\}. \end{aligned}$$

Wir betrachten zuerst $\mathcal{L}(G) \subseteq (\mathrm{COPY}_\Sigma - \{\varepsilon\})$. Wie in Beispiel 5.6 wird auch hier den Regeln durch die vorhandenen Variablen eine Reihenfolge vorgeben: Jede Ableitung eines Wortes $w \in \mathcal{L}(G)$ beginnt mit mehrfacher Anwendung von Regeln aus der ersten Menge der Definition von P, gefolgt von einer Anwendung einer Regel aus der zweiten Menge. Dann gilt:

$$\begin{aligned} S &\Rightarrow_G^* a_1 \cdots a_n S X_{a_n} \cdots X_{a_1} \\ &\Rightarrow_G a_1 \cdots a_n L_{a_{n+1}} R_{a_{n+1}} X_{a_n} \cdots X_{a_1} \Rightarrow_G^* w \end{aligned}$$

mit $n \in \mathbb{N}$ und $a_1, \ldots, a_{n+1} \in \Sigma$. An dieser Stelle können zwar mehrere Regeln angewendet werden, wie zum Beispiel $L_{a_{n+1}} \to a_{n+1}$, allerdings führen die meisten dieser Regeln in Sackgassen, also zu Satzformen die nicht zu einem Terminalwort abgeleitet werden können. Die einzige Regel, bei der dies nicht der Fall ist, ist die Regel $R_{a_{n+1}} X_{a_n} \to X_{a_n} R_{a_{n+1}}$. Anhand von Regeln dieser Form können wir nun $R_{a_{n+1}}$ ganz nach rechts schieben:

$$\begin{aligned} S &\Rightarrow_G^* a_1 \cdots a_n L_{a_{n+1}} R_{a_{n+1}} X_{a_n} \cdots X_{a_1} \\ &\Rightarrow_G^* a_1 \cdots a_n L_{a_{n+1}} X_{a_n} \cdots X_{a_1} R_{a_{n+1}} \Rightarrow_G^* w. \end{aligned}$$

Wir könnten zwar vorher die Regel $R_{a_{n+1}} \to a_{n+1}$ anwenden, dann könnten wir aber rechts davon mindestens ein X_{a_i} nicht mehr ableiten. Wir können also annehmen, dass die Ableitung $S \Rightarrow_G^* w$ zuerst $R_{a_{n+1}}$ nach rechts schiebt und danach zu a_{n+1} ableitet. Es gilt also

$$S \Rightarrow_G^* a_1 \cdots a_n L_{a_{n+1}} X_{a_n} \cdots X_{a_1} a_{n+1} \Rightarrow_G^* w.$$

Nun kann wieder nur eine Regel angewendet werden, nämlich $L_{a_{n+1}} X_{a_n} \to L_{a_{n+1}} R_{a_n}$ (prinzipiell wäre noch $L_{a_{n+1}}$ möglich, aber dann befänden wir uns in einer Sackgasse). Die so entstandene Variable R_{a_n} wird nun so weit wie möglich nach rechts geschoben und dann nach a_n abgeleitet. Dieser Schritt wird für alle verbliebenen X_{a_i} wiederholt. Auf diese Art erhalten wir

$$S \Rightarrow_G^* a_1 \cdots a_n L_{a_{n+1}} a_1 \cdots a_n a_{n+1} \Rightarrow_G^* w.$$

Da keine andere Regel als $L_{a_{n+1}} \to a_{n+1}$ anwendbar ist, muss

$$S \Rightarrow_G^* a_1 \cdots a_n a_{n+1} a_1 \cdots a_n a_{n+1} = w$$

gelten. Dass $(\mathrm{COPY}_\Sigma - \{\varepsilon\}) \subseteq \mathcal{L}(G)$ gilt, wird anhand dieser Überlegungen ebenfalls schnell klar. ◊

Hinweis 5.8 Eigentlich erlauben monotone Grammatiken keine Kontrolle über die Reihenfolge der angewendeten Regeln. Durch ein paar Tricks lässt sich die Reihenfolge allerdings oft dennoch beeinflussen. Wir haben dies zum Beispiel in Beispiel 5.7 gesehen. Wenn hier eine Regel $L_a \to a$ verwendet wird, bevor alle X_b ersetzt wurden, können die X_b nicht mehr abgeleitet werden. Eine solche Ableitung kann also zu keinem Terminalwort führen. Dadurch wird eine gewisse Reihenfolge erzwungen.

In Hopcroft und Ullman [8] (Example 9.5 in Abschnitt 9.3) ist außerdem eine monotone Grammatik für die Sprache $\{\mathtt{a}^{2^i} \mid i \in \mathbb{N}_{>0}\}$ angegeben. Allerdings ist diese Grammatik aufwändiger als unserer Beispiel hier, so dass wir diese nicht genauer betrachten[70].

Durch jedes dieser Beispiele können wir die Klassen CFL und CSL voneinander trennen:

[70] Falls Sie das Beispiel in Hopcroft und Ullman [8] nachlesen, sollten Sie beachten, dass dort „unsere“ monotonen Grammatiken als kontextsensitive Grammatiken bezeichnet werden.

Satz 5.9 CFL $\subset$ CSL

Beweis: Die Inklusion CFL $\subseteq$ CSL gilt nach Definition: Jede kontextfreie Grammatik ist auch eine kontextsensitive Grammatik (da der Kontext leer sein darf). In Beispiel 5.5, Beispiel 5.6 und Beispiel 5.7 haben wir drei Sprachen betrachtet, die jeweils CFL $\neq$ CSL belegen. □

Analog zu den bisher betrachteten Wortproblemen können wir das Wortproblem für monotone Grammatiken definieren:

WORTPROBLEM FÜR MONOTONE GRAMMATIKEN

Eingabe: Eine monotone Grammatik $G = (\Sigma, V, P, S)$ und ein Wort $w \in \Sigma^*$.
Frage: Ist $w \in \mathcal{L}(G)$?

Satz 5.10 *Das Wortproblem für monotone Grammatiken ist entscheidbar.*

Beweis: Sei $G = (\Sigma, V, P, S)$ eine monotone Grammatik und sei $w \in \Sigma^*$. Da G monoton ist wissen wir, dass für alle Satzformen γ_1, γ_2 mit $\gamma_1 \Rightarrow_G \gamma_2$ stets $|\gamma_1| \leq |\gamma_2|$ gelten muss. Also gilt $|\gamma| \leq w$ für alle Satzformen γ mit $S \Rightarrow_G^* \gamma \Rightarrow_G^* w$.

Um zu entscheiden, ob $w \in \mathcal{L}(G)$ ist, können wir also einen *brute force*-Ansatz wählen. Dazu zählen wir alle Satzformen γ auf, für die $S \Rightarrow_G^* \gamma$ und $|\gamma| \leq |w|$ gilt. Wenn w in dieser Aufzählung vorkommt, so ist $w \in \mathcal{L}(G)$. Wenn w in dieser Aufzählung nicht vorkommt, so kann $S \Rightarrow_G^* w$ nicht gelten, da eine längenverkürzende Ableitung[71] benutzt werden müsste, was in einer monotonen Grammatik nicht möglich ist. □

Leider ist diese Nachricht nur von begrenztem praktischen Nutzen: Mit zusätzlichem Aufwand lässt sich zeigen, dass das Wortproblem für monotone Grammatiken PSPACE-vollständig ist. Die höhere Ausdrucksstärke der monotonen Grammatiken (im Vergleich zu den kontextfreien Grammatiken) wir also teuer erkauft. Zusätzlich dazu sind monotone und kontextsensitive Grammatiken oft umständlich zu konstruieren und nur schwer zu lesen. Um zu zeigen, dass eine Sprache kontextsensitiv ist, verwendet man daher gewöhnlich ein anderes Modell, das wir im Folgenden kurz betrachten werden.

5.1.2 Linear beschränkte Automaten

Wir werden uns in diesem Abschnitt kurz mit einem Maschinenmodell für kontextsensitive Sprachen befassen, den sogenannten **linear beschränkten Turingmaschinen**, kurz **LBA** (von *linear bounded automaton*). Diese bestehen aus folgenden Komponenten (illustriert in Abbildung 5.1):

- einem Eingabeband, auf dem nicht geschrieben werden kann, und auf dem sich der Lesekopf nach links und rechts bewegen kann,
- einer nichtdeterministischen endliche Kontrolle,

[71] Also eine Ableitung $\gamma_1 \Rightarrow_G^* \gamma_2$ mit $|\gamma_1| > |\gamma_2|$.

- einem Arbeitsband, auf dem gelesen und geschrieben werden kann, das aber in seiner Länge auf die Länge der Eingabe beschränkt ist.

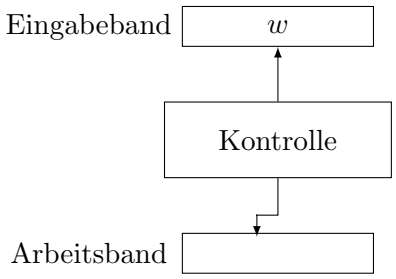


Abbildung 5.1: Eine schematische Darstellung eines LBA.

Ingesamt ist ein LBA also eine nichtdeterministische Turingmaschine mit einem Arbeitsband, dessen Länge auf die Länge der Eingabe beschränkt ist. Wir werden die formalen Details nicht weiter betrachten; stattdessen stellen wir nur fest, dass LBAs und kontextsensitive Grammatiken die gleiche Ausdrucksstärke haben:

Satz 5.11 *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$. Dann gilt:*

L ist genau dann eine kontextsensitive Sprache, wenn ein LBA existiert, der L akzeptiert.

Beweisidee: Einen ausführlichen Beweis finden Sie beispielsweise in Shallit [19] (oder, über einen Umweg, in Hopcroft und Ullman [8]). Wir skizzieren hier nur kurz die Beweisidee.

Um zu zeigen, dass die Sprache $\mathcal{L}(G)$ einer kontextsensitiven Grammatik G von einem LBA akzeptiert wird, konvertiert man G in einen LBA der die Ableitungen von G nichtdeterministisch simuliert. Da für jede Eingabe w nur Satzformen γ mit $|\gamma| \leq |w|$ betrachtet werden müssen, kann dies komplett auf dem Arbeitsband geschehen.

Für die andere Richtung konstruiert man aus einem LBA A eine monotone Grammatik G, die A simuliert. Dazu modifiziert man A zuerst so, dass A zuerst die Eingabe auf das Arbeitsband kopiert und dann alle Berechnungen auf diesem vornimmt (ohne die Eingabe zu beachten). Die Grammatik G simuliert nun die Arbeitsweise von A auf dem Arbeitsband; dabei entsprechen die Satzformen von G dem Inhalt des Arbeitsbandes; zusätzlich werden in jeweils einer Variablen der aktuelle Zustand der endlichen Kontrolle sowie die Position des Kopfes auf dem Arbeitsband kodiert. Die Übergänge des LBA können nun anhand von monotonen Regeln umgesetzt werden. □

In der Komplexitätstheorie wird die Klasse CSL auch als NSPACE($O(n)$) bezeichnet (da ein LBA eine nichtdeterministische Turingmaschine mit linear beschränktem Platz ist).

Analog zu Kellerautomaten und Turingmaschinen können wir deterministische LBAs konstruieren. Die Frage, ob diese alle kontextsensitiven Sprachen erzeugen können (also

ob deterministische und nichtdeterministische LBAs die gleiche Ausdrucksstärke haben) ist ein schweres offenes Problem, das auch oft als **das LBA-Problem** oder **das erste LBA-Problem** bezeichnet wird.

Das **zweite LBA-Problem** ist die Frage, ob die Klasse CSL unter Komplementbildung abgeschlossen ist. Diese Frage war zwar rund 20 Jahre lang offen, wurde aber 1987 positiv beantwortet. Der Abschluss der Klasse CSL unter Komplementbildung ist eine direkte Konsequenz des Satzes von Immerman und Szelepcsényi[72]. Wir werden hier nicht weiter auf diesen Satz eingehen, den Beweis dazu finden Sie zum Beispiel in Shallit [19] (für den Spezialfall CSL) oder Mateescu und Salomaa [11] (für die allgemeine Version). Im Gegensatz zum Beweis für den Abschluss von DCFL unter Komplement ist dieser Beweis recht einfach nachzuvollziehen.

5.2 Die Chomsky-Hierarchie

Um die kontextsensitiven Grammatiken in einen größeren Kontext zu setzen, definieren wir noch eine weitere Sprachklasse:

Definition 5.12 Eine **Sprache** $L \subseteq \Sigma^*$ heißt **rekursiv aufzählbar**, wenn eine Turingmaschine M existiert, für die $\mathcal{L}(M) = L$. Wir bezeichnen[73]die **Klasse aller rekursiv aufzählbaren Sprachen** über dem Alphabet Σ mit RE_Σ, und definieren die Klasse aller rekursiv aufzählbaren Sprachen $\mathrm{RE} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{RE}\,\Sigma$.

Die rekursiv aufzählbaren Sprachen sind also genau die Sprachen, die von Turingmaschinen erkannt werden. Wir stellen fest:

Satz 5.13 $\mathrm{CSL} \subset \mathrm{RE}$

Beweis: Sei $L \in \mathrm{CSL}$. Dann existiert eine kontextsensitive Grammatik G mit $\mathcal{L}(G) = L$. Aus Satz 5.10 wissen wir, dass das Wortproblem für monotone Grammatiken entscheidbar ist. Somit existiert eine Turingmaschine M mit $\mathcal{L}(M) = \mathcal{L}(G) = L$. Also ist $L \in \mathrm{RE}$ und somit $\mathrm{CSL} \subseteq \mathrm{RE}$.

Die Beobachtung $\mathrm{CSL} \neq \mathrm{RE}$ folgt unmittelbar aus der Erkenntnis, dass es rekursiv aufzählbare Sprachen mit nicht-entscheidbarem Wortproblem gibt (wie zum Beispiel die Sprache aller Codierungen von immer haltenden Turingmaschinen). □

Durch Diagonalisierungsargumente lässt sich zeigen, dass es auch entscheidbare Sprachen gibt, die nicht kontextsensitiv sind. Einen Beweis dafür finden Sie zum Beispiel in Shallit [19] (Theorem 7.1.7).

Der Vollständigkeit halber betrachten wir auch ein Grammatikmodell für rekursiv aufzählbare Sprachen:

[72] Benannt nach Neil Immerman und Róbert Szelepcsényi. Kurioserweise haben beide den gleichen Satz zur gleichen Zeit unabhängig voneinander entdeckt.

[73] Die Abkürzung RE stammt von *recursively enumerable*.

Definition 5.14 Eine **Phrasenstrukturgrammatik (PSG)** (oder auch nur **Grammatik**) G über einem Alphabet Σ wird definiert durch

1. eine Alphabet V von **Variablen**, wobei $(\Sigma \cap V) = \emptyset$,
2. eine endliche Menge $P \subseteq (\Sigma \cup V)^* \times (\Sigma \cup V)^*$ von **Regeln**,
3. eine Variable $S \in V$ (dem **Startsymbol**).

Wir schreiben dies als $G := (\Sigma, V, P, S)$. Die Relationen $\Rightarrow_G$, $\Rightarrow^n$ $[G]$ und $\Rightarrow_G^*$ definieren wir analog zu Definition 5.3.

Die von G **erzeugte Sprache** $\mathcal{L}(G)$ definieren wir als

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

Im Gegensatz zu allen Grammatiken, die wir bisher betrachten haben, sind die Regeln bei Phrasenstrukturgrammatiken überhaupt nicht eingeschränkt. Im Vergleich zu monotonen Grammatiken ist der wichtigste Unterschied, dass bei PSGs auch solche Regeln der Form $\alpha \to \beta$ erlaubt sind, bei denen $|\alpha| > |\beta|$ (außerdem darf das Startsymbol auf der rechten Seite einer Regel erscheinen). Insbesondere ist es bei PSGs auch erlaubt, beliebige Variablen durch ε zu ersetzen. Dadurch erhöht sich die Ausdrucksstärke der Grammatiken immens:

Satz 5.15 *Sei Σ ein Alphabet und $L \subseteq \Sigma^*$ eine beliebige Sprache. Die folgenden Aussagen sind äquivalent:*

1. *Es existiert eine Phrasenstrukturgrammatik G mit $\mathcal{L}(G) = L$.*
2. *Es existiert eine Turingmaschine M mit $\mathcal{L}(M) = L$.*
3. *Die Sprache L ist rekursiv aufzählbar, also $L \in \mathrm{RE}$.*

Beweisidee: Die Äquivalenz von 2 und 3 folgt direkt aus Definition 5.12.

Die Äquivalenz von 1 und 2 lässt sich analog zu Satz 5.11: Wir können eine Turingmaschine konstruieren, die eine PSG simuliert; und ebenso können wir eine Turingmaschine in einer PSG simulieren. Details zu diesem Beweis finden sich beispielsweise in Hopcroft und Ullman [8]. □

Nun haben wir alle Grammatiken kennen gelernt, die wir benötigen, um die sogenannte **Chomsky-Hierarchie** zu definieren, die auch als **Chomsky-Schützenberger-Hierarchie**[74] bekannt ist. Noam Chomsky hat 1956 die folgenden vier Typen von Grammatiken klassifiziert:

- Typ 0, die wir als Phrasenstrukturgrammatiken bezeichnen,
- Typ 1, die kontextsensitiven Grammatiken,
- Typ 2, die kontextfreien Grammatiken,

[74] Benannt nach Noam Chomsky (nach dem auch die Chomsky-Normalform benannt wurde) und Marcel-Paul Schützenberger.

- Typ 3, die regulären Grammatiken.

Die entsprechenden Sprachklassen haben wir im Lauf dieser Vorlesung ausführlich untersucht. Wir fassen hier noch einmal die Klassen und die entsprechenden Modelle zusammen:

Grammatik	**Automatenmodell**	**Sprachklasse**
Typ 0 PSG	Turingmaschine	RE
Typ 1 CSG	LBA	CSL
Typ 2 CFG	PDA	CFL
Typ 3 reg. Gramm.	endlicher Automat	REG

Wie wir an verschiedenen Stellen gesehen haben, bilden die Klassen eine Hierarchie:

Satz 5.16 REG $\subset$ CFL $\subset$ CSL $\subset$ RE

Diese Hierarchie wird of auch als die Chomsky-Hierarchie[75] bezeichnet. Zwei andere Sprachklassen, die wir in dieser Vorlesung betrachtet haben, nämlich FIN und DCFL, lassen sich gut in diese Hierarchie einsortieren. Es gilt:

Satz 5.17 FIN $\subset$ REG $\subset$ DCFL $\subset$ CFL $\subset$ CSL $\subset$ RE

In den Forschungsgebieten Automaten und formale Sprachen wurde eine Vielzahl weiterer Modelle untersucht. Oft lassen diese sich innerhalb dieser Hierarchie einsortieren. Zum Beispiel wurden in der Computerlinguistik mehrere Klassen von sogenannten *schwach kontextsensitiven Sprachen*[76] untersucht, die zwischen CFL und CSL liegen (aber näher an CFL). Ein häufiger Standpunkt ist dabei, dass Klassen, die zu nahe an CSL herankommen, im Allgemeinen nicht mehr handhabbar sind.

Allerdings dient die Chomsky-Hierarchie eher als grobe Orientierung denn als verbindlicher Maßstab. Wir werden uns im Folgenden zwei Klassen von Sprachen ansehen, die sich nicht brav in diese Hierarchie einsortieren lassen.

5.3 Modelle mit Wiederholungsoperatoren

Die meisten Sprachklassen der Chomsky-Hierarchie sind nicht besonders gut dafür geeignet, Wiederholungen beliebiger Wörter auszudrücken (im Sinne von „das gleiche Wort noch einmal“). Beispielsweise ist die Copy-Sprache, also die Sprache aller Wörter der Form ww, nicht kontextfrei; und eine monotone Grammatik für diese Sprache ist schnell unübersichtlich (siehe Beispiel 5.7).

[75] Oder, wie gesagt, als Chomsky-Schützenberger-Hierarchie. Schützenberger hat diese Hierarchie als einer der ersten ausführlich untersucht und dadurch die Theorie der formalen Sprachen geprägt.
[76] Engl. *mildly context sensitive languages*.

Für viele Anwendungen ist es allerdings wünschenswert, solche Wiederholungen in einer allgemeinen Form spezifizieren zu können. In diesem Abschnitt werden wir zwei Modelle betrachten, die mit solchen Wiederholungen arbeiten.

5.3.1 Patternsprachen

Das erste dieser Modelle sind die sogenannten *Patternsprachen*:

Definition 5.18 Seien Σ und X disjunkte Alphabete. Wir bezeichnen Σ als **Terminalalphabet** und X als **Variablenalphabet**.

Ein **Pattern** ist ein nicht-leeres Wort $\alpha \in (\Sigma \cup X)^+$. Eine **$\Sigma^+$-Ersetzung** ist ein Homomorphismus $\sigma\colon (\Sigma \cup X)^+ \to \Sigma^+$ mit:

- $\sigma(a) = a$ für alle $a \in \Sigma$,
- $\sigma(x) \neq \varepsilon$ für alle $x \in X$.

Das Pattern α erzeugt die **Sprache**

$$\mathcal{L}_\Sigma(\alpha) := \{\sigma(\alpha) \mid \sigma \text{ ist eine } \Sigma^+\text{-Ersetzung}\}.$$

Eine Sprache $L \subseteq \Sigma^*$ ist eine **Patternsprache**, wenn ein Variablenalphabet X und ein Pattern $\alpha \in (\Sigma \cup X)^+$ existieren, so dass $\mathcal{L}_\Sigma(\alpha) = L$. Wir bezeichnen die **Klasse aller Patternsprachen** über dem Alphabet Σ mit PAT_Σ, und definieren die Klasse aller Patternsprachen $\mathrm{PAT} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{PAT}_\Sigma$.

Die Sprache eines Pattern α enthält also genau die Wörter, die erzeugt werden können, indem man alle Variablen in α durch Terminalwörter ersetzt (und die Terminale in α nicht verändert). Da jede Σ^+-Ersetzung ein Homomorphismus ist, müssen gleiche Variablen durch eine Σ^+-Ersetzung auch gleich ersetzt werden.

Beispiel 5.19 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $X := \{x, y, z\}$. Wir betrachten die folgenden Pattern:

$$\begin{aligned}\alpha_1 &:= xx,\\ \alpha_2 &:= xyz,\\ \alpha_3 &:= \mathtt{a}xyz\mathtt{b},\\ \alpha_4 &:= xx\mathtt{c}yy.\end{aligned}$$

Diese Pattern erzeugen die folgenden Sprachen:

- $\mathcal{L}_\Sigma(\alpha_1) = \{xx \mid x \in \Sigma^+\}$. Es gilt also $\mathcal{L}_\Sigma(\alpha_1) = \mathrm{COPY}_\Sigma - \{\varepsilon\}$. Da Σ^+-Ersetzungen niemals eine Variable durch ε ersetzen dürfen, gilt $\varepsilon \notin \mathcal{L}_\Sigma(\alpha_1)$ (dies gilt auch für jedes andere Pattern und jedes andere Terminalalphabet).
- $\mathcal{L}_\Sigma(\alpha_2) = \{w \in \Sigma^+ \mid |w| \geq 3\}$.

- $\mathcal{L}_\Sigma(\alpha_3)$ ist die Sprache aller $w \in \Sigma^+$, die mindestens die Länge 5 haben, mit `a` beginnen und auf `b` enden.

- $\mathcal{L}_\Sigma(\alpha_4) = \{xx\mathtt{c}yy \mid x, y \in \Sigma^+\} = (\mathrm{COPY}_\Sigma - \{\varepsilon\})\{\mathtt{c}\}(\mathrm{COPY}_\Sigma - \{\varepsilon\})$.

In jedem Fall können Sie aus einem Pattern α direkt die Sprache $\mathcal{L}_\Sigma(\alpha)$ ablesen. ◊

Aus dem Blickwinkel der Modelle, die wir bisher betrachtet haben, haben Patternsprachen einige ungewöhnliche Eigenschaften. Eine dieser ungewöhnlichen Eigenschaften ermöglicht es uns, die Klasse PAT von den unteren Ebenen der Chomsky-Hierarchie zu unterscheiden:

Lemma 5.20 *Sei $L \in$ PAT. Dann gilt entweder $|L| = 1$ oder $|L| = \infty$.*

Beweis: Dies folgt fast direkt aus der Definition. Angenommen, $L \in$ PAT. Dann existieren Alphabete Σ und X und ein Pattern $\alpha \in (\Sigma \cup X)^+$ mit $\mathcal{L}_\Sigma(\alpha) = L$.

Angenommen, α enthält keine Variablen. Dann ist $\alpha \in \Sigma^+$, und für jede Σ^+-Ersetzung σ gilt $\sigma(\alpha) = \alpha$. Somit ist $L = \mathcal{L}_\Sigma(\alpha) = \{\alpha\}$, also $|L| = 1$.

Angenommen, α enthält mindestens eine Variable. Für jedes $w \in \Sigma^+$ definieren wir die Σ^+-Ersetzung σ_w durch:

$$\sigma_w(x) := \begin{cases} x & \text{falls } x \in \Sigma, \\ w & \text{falls } x \in X. \end{cases}$$

Wie leicht zu sehen ist, gilt $\sigma_u(\alpha) \neq \sigma_v(\alpha)$ für alle $u, v \in \Sigma^+$ mit $u \neq v$. Da $L = \mathcal{L}_\Sigma(\alpha)$ gilt $|L| = \infty$. □

Als unmittelbare Konsequenz von Lemma 5.20 können wir die Klassen FIN und PAT voneinander trennen:

Korollar 5.21 FIN *und* PAT *sind unvergleichbar.*

Beweis: Konkrete Beispiele, die diese Klassen trennen, sind unter anderem die unären Sprachen $\{\mathtt{a}, \mathtt{aa}\}$ (endlich, keine Patternsprache) und $\{\mathtt{a}\}^+$ (nicht endlich, aber eine Patternsprache). □

Dadurch folgt direkt, dass alle Klasse der Chomsky-Hierarchie nicht in PAT enthalten sind. Für die zwei unteren Klassen gilt dies auch umgekehrt:

Lemma 5.22 CFL *und* PAT *sind unvergleichbar.*

Beweis: Die Sprache $L := \{xx \mid x \in \Sigma^+\}$ wird vom Pattern xx erzeugt. Für $|\Sigma| \geq 2$ gilt $L \notin$ CFL. Also ist PAT $\not\subseteq$ CFL.

Umgekehrt haben wir bereits in Korollar 5.21 festgestellt, dass FIN $\not\subseteq$ PAT. Also gilt auch CFL $\not\subseteq$ PAT. Ein konkretes Beispiel hierfür ist die Sprache $\{a, aa\}$ für ein beliebiges $a \in \Sigma$. □

Umgekehrt kann man zeigen, dass jede Patternsprache kontextsensitiv ist:

Satz 5.23 PAT $\subset$ CSL

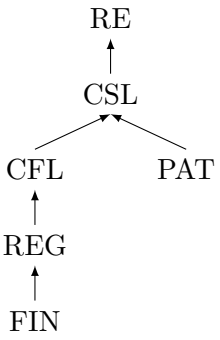


Abbildung 5.2: Die Zusammenhänge zwischen PAT und den Klassen der Chomsky-Hierarchie (und FIN). Ein Pfeil von einer Klasse A zu einer Klasse B bedeutet, dass A in B echt enthalten ist (also $A \subset B$).

Beweisidee: Am einfachsten zeigt man dies, indem man aus einem Pattern α einen LBA A konstruiert. Dieser rät eine Σ^+-Ersetzung σ und testet dann, ob $\sigma(\alpha)$ mit der Eingabe identisch ist. Mit einigermaßen vertretbarem Aufwand lässt sich aber auch zeigen, dass jedes Pattern in eine äquivalente monotone Grammatik konvertiert werden kann. □

Solche Zusammenhänge zwischen Klassen werden oft in Inklusionsdiagrammen dargestellt. Das entsprechende Diagramm finden Sie in Abbildung 5.2. Im Gegensatz zu den Sprachklassen, die wir bisher kennen gelernt haben, ist die Auswahl des Terminalalphabets bei Patternsprachen besonders wichtig:

Lemma 5.24 *Seien* Σ_1 *und* Σ_2 *Alphabete mit* $\Sigma_1 \subset \Sigma_2$. *Sei* $L \in \mathrm{PAT}_{\Sigma_1}$. *Dann gilt* $L \in \mathrm{PAT}_{\Sigma_2}$ *genau dann, wenn* $|L| = 1$.

Beweis: Da $L \in \mathrm{PAT}_{\Sigma_1}$ existieren ein Variablenalphabet X (das zu Σ_1 und Σ_2 disjunkt ist) und ein Pattern $\alpha \in (\Sigma \cup X)^+$ mit $\mathcal{L}_{\Sigma_1}(\alpha) = L$.

Wir betrachten nun zuerst die Rück-Richtung. Angenommen, $|L| = 1$. Dann muss $\alpha \in \Sigma_1^+$ und $\mathcal{L}_{\Sigma_1}(\alpha) = \{\alpha\}$ gelten. Da $\Sigma_1 \subset \Sigma_2$ gilt $\alpha \in \Sigma_2^+$, und $\mathcal{L}_{\Sigma_2}(\alpha) = \{\alpha\} = L$.

Nun zeigen wir die Hin-Richtung durch ihre Kontraposition. Wir nehmen also an, dass $|L| = \infty$, und $L \in \mathrm{PAT}_{\Sigma_2}$. Dann existiert ein Pattern $\beta \in (\Sigma_2 \cup X)^+$ mit $\mathcal{L}_{\Sigma_2}(\beta) = L$. Da $|L| = \infty$ muss β mindestens eine Variable enthalten. Da $\Sigma_1 \subset \Sigma_2$ existiert ein Buchstabe $b \in (\Sigma_2 - \Sigma_1)$. Wir definieren nun die Σ_2^+-Ersetzung σ_b durch $\sigma_b(a) := a$ für alle $a \in \Sigma_1$, und $\sigma_b(x) := b$ für alle $x \in X$. Sei $w := \sigma_b(\beta)$. Da β mindestens eine Variable enthält, ist $|w|_b \geq 1$. Also gilt $w \notin L$, und somit $L \neq \mathcal{L}_{\Sigma_2}(\beta)$. Widerspruch. □

Insbesondere folgt aus Lemma 5.24, dass für Alphabete $\Sigma_1 \subset \Sigma_2$ stets $\mathrm{PAT}_{\Sigma_1} \not\subseteq \mathrm{PAT}_{\Sigma_2}$ gilt (bei den Sprachen der Chomsky-Hierarchie ist dies nicht der Fall, es gilt beispielsweise $\mathrm{REG}_{\Sigma_1} \subseteq \mathrm{REG}_{\Sigma_2}$).

Beispiel 5.25 Sei $\Sigma_1 := \{\mathtt{a}, \mathtt{b}\}$, $\Sigma_2 := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und $X := \{x\}$. Wir betrachten das Pattern x. Es gilt $\mathcal{L}_{\Sigma_1}(x) = \Sigma_1^+$ und $\mathcal{L}_{\Sigma_2}(x) = \Sigma_2^+$. Da Patternsprachen über einem Alphabet Σ jede mögliche Σ^+-Ersetzung akzeptieren müssen, können sie keine Buchstaben

ausschließen. Die Sprache $\mathcal{L}_{\Sigma_1}(x)$ ist also eine Patternsprache über dem Alphabet Σ_1, aber nicht über dem Alphabet Σ_2. ◊

Wir wenden uns nun einigen Entscheidungsfragen zu Patternsprachen zu. Dazu führen wir zuerst eine weitere Definition ein:

Definition 5.26 Sei Σ ein Terminal- und X ein Variablenalphabet. Für zwei Pattern $\alpha, \beta \in (\Sigma \cup X)^+$ heißt α **aus** β **erzeugbar**, wenn ein Homomorphismus $h\colon (\Sigma \cup X)^+ \to (\Sigma \cup X)^+$ existiert, der die folgenden Bedingungen erfüllt:

1. $h(a) = a$ für alle $a \in \Sigma$,
2. $h(x) \neq \varepsilon$ für alle $x \in X$,
3. $h(\beta) = \alpha$.

Wir schreiben dies als $\alpha \preceq \beta$. Gilt $\alpha \preceq \beta$ und $\beta \preceq \alpha$, so schreiben wir $\alpha \approx \beta$, und sagen α **und** β **sind identisch modulo Umbenennung**.

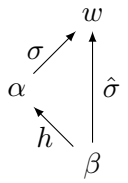
Anschaulich gesprochen bedeutet $\alpha \preceq \beta$, dass man das Pattern α durch geeignetes Umschreiben aus β erhalten kann. Der Homomorphismus h kann dabei als Verallgemeinerung einer Σ^+-Ersetzung verstanden werden (im Gegensatz zu dieser darf h auch Variablen erzeugen). Es ist leicht zu sehen, dass die Relation $\preceq$ eine Ordnungsrelation ist, und dass $\approx$ die entsprechende Äquivalenzrelation ist. Die Bezeichnung *identisch modulo Umbenennung* rührt daher, dass $\alpha \approx \beta$ nur gelten kann, wenn α und β bis auf eine eventuell notwendige Umbenennung der Variablen identisch sind (da die für die Definition von $\preceq$ verwendeten Homomorphismen weder Symbole löschen, noch Terminal umschreiben dürfen). Die Relation $\preceq$ gibt uns ein hinreichendes Kriterium für die Inklusion von Patternsprachen:

Lemma 5.27 *Sei Σ ein Terminal- und X ein Variablenalphabet. Für alle Pattern $\alpha, \beta \in (\Sigma \cup X)^+$ gilt: Aus $\alpha \preceq \beta$ folgt $\mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$.*

Beweis: Sei $w \in \mathcal{L}_\Sigma(\alpha)$. Dann existiert eine Σ^+-Ersetzung σ mit $\sigma(\alpha) = w$. Da $\alpha \preceq \beta$ existiert ein Homomorphismus $h\colon (\Sigma \cup X)^+ \to (\Sigma \cup X)^+$ existiert, der die folgenden Bedingungen erfüllt:

1. $h(a) = a$ für alle $a \in \Sigma$,
2. $h(x) \neq \varepsilon$ für alle $x \in X$,
3. $h(\beta) = \alpha$.

Wir definieren nun den Homomorphismus $\hat{\sigma}\colon (\Sigma \cup X)^+ \to (\Sigma \cup X)^+$ durch $\hat{\sigma} := (\sigma \circ h)$, also durch $\hat{\sigma}(x) = \sigma(h(x))$ für alle $x \in (\Sigma \cup X)$. Dann gilt $\hat{\sigma}(\alpha) = \sigma(h(\beta)) = \sigma(\alpha) = w$.



Wir müssen also nur noch zeugen, dass $\hat{\sigma}$ eine Σ^+-Ersetzung ist. Da $\hat{\sigma}$ als Komposition von zwei Homomorphismen definiert ist, ist $\hat{\sigma}$ ebenfalls ein Homomorphismus $\hat{\sigma}\colon (\Sigma \cup X)^+ \to \Sigma^+$. Außerdem gilt:

1. $\hat{\sigma}(a) = \sigma(h(a)) = \sigma(a) = a$ für alle $a \in \Sigma$,
2. $\hat{\sigma}(x) = \sigma(h(x)) \neq \varepsilon$ für alle $x \in X$ (da σ und h niemals auf ε abbilden).

Somit ist $\hat{\sigma}$ eine Σ^+-Ersetzung, und da $\hat{\sigma}(\beta) = w$ gilt $w \in \mathcal{L}_\Sigma(\beta)$. Da w frei aus $\mathcal{L}_\Sigma(\alpha)$ gewählt wurde, gilt $\mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$. □

Leider ist dieses Kriterium nicht charakteristisch für die Inklusion von Patternsprachen, wie das folgende Beispiel zeigt.

Beispiel 5.28 Sei Σ ein Terminalalphabet mit $|\Sigma| = n$ für ein $n \in \mathbb{N}_{>0}$ mit $n \geq 2$ und sei $X := \{x_1, \ldots, x_{2n+3}, y, z\}$ ein Variablenalphabet. Wir definieren die Pattern $\alpha, \beta \in (\Sigma \cup X)^+$ durch

$$\alpha := x_1\, x_2\, x_3 \cdots x_{2n+3},$$
$$\beta := x_1\, y\, z\, y\, x_2.$$

Nun gilt $\alpha \not\preceq \beta$, denn da in α jede Variable nur ein einziges Mal vorkommt, folgt für jeden Homomorphismus h aus $h(\beta) = \alpha$ zwangsläufig $h(y) = \varepsilon$ (und dies widerspricht Definition 5.26).

Allerdings ist $\mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$. Dies ist nicht unbedingt offensichtlich, lässt sich aber leicht beweisen: Angenommen, $w \in \mathcal{L}_\Sigma(\alpha)$. Dann gilt $|w| \geq 2n + 3$ (denn $|w| \geq |\alpha| = 2n + 3$). Wir zerlegen nun w in Buchstaben $u_1, u_2 \in \Sigma$ und ein Wort $v \in \Sigma^+$ mit $w = u_1 v u_2$. Nun gilt $|v| \geq 2n + 1$. Da $n = |\Sigma|$ ist, existiert ein Buchstabe $a \in \Sigma$ mit $|v|_a \geq 3$. Also existieren Wörter $v_1, v_2, v_3, v_4 \in \Sigma^*$ mit

$$v = v_1\, a\, v_2\, a\, v_3\, a\, v_4.$$

Wir definieren nun den Homomorphismus $\sigma\colon (\Sigma \cup X)^+ \to \Sigma^+$ durch $\sigma(b) := b$ für alle $b \in \Sigma$, sowie

$$\sigma(x_1) := u_1\, v_1, \qquad \sigma(y) := a,$$
$$\sigma(x_2) := v_4\, u_2, \qquad \sigma(z) := v_2\, a\, v_3.$$

Offensichtlich ist σ eine Σ^+-Ersetzung. Außerdem gilt:

$$\sigma(\beta) = \sigma(x_1\, y\, z\, y\, x_2)$$

$$= u_1\, v_1\, a\, v_2\, a\, v_3\, a\, v_4\, u_2$$
$$= u_1\, v\, u_2 = w.$$

Somit ist $w \in \mathcal{L}_\Sigma(\beta)$, und da $w \in \mathcal{L}_\Sigma(\alpha)$ frei gewählt wurde gilt $\mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$. Wir können sogar feststellen, dass $\mathcal{L}_\Sigma(\alpha) \subset \mathcal{L}_\Sigma(\beta)$, denn da $|\beta| < |\alpha|$ muss $\mathcal{L}_\Sigma(\alpha) \neq \mathcal{L}_\Sigma(\beta)$ gelten. ◊

Genau genommen haben wir in Beispiel 5.28 nicht nur gezeigt, dass $\mathcal{L}_\Sigma(\alpha) \subset \mathcal{L}_\Sigma(\beta)$ gilt, sondern wir haben auch bewiesen, dass die Sprache $\mathcal{L}_\Sigma(\beta)$ kofinit ist (und somit regulär), denn sie enthält alle Wörter aus Σ^+, die mindestens die Länge $2n+3$ haben[77].

Mit zusätzlichem Aufwand lässt sich zeigen, dass das Inklusionsproblem für Pattern nicht entscheidbar ist; selbst, wenn man Σ und X in ihrer Größe beschränkt. In beiden Fällen ist der entsprechende Beweis aber vergleichsweise lang, daher gehen wir nicht weiter darauf ein[78].

Wenn wir allerdings wissen, dass zwei Pattern die gleiche Länge haben, lässt wird die Inklusion durch $\preceq$ charakterisiert:

Lemma 5.29 *Sei Σ ein Terminalalphabet mit $|\Sigma| \geq 2$ und sei X ein Variablenalphabet. Für alle Pattern $\alpha, \beta \in (\Sigma \cup X)^+$ mit $|\alpha| = |\beta|$ gilt:*

$$\alpha \preceq \beta \text{ genau dann, wenn } \mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta).$$

Beweis: Die Hin-Richtung ist ein Sonderfall von Lemma 5.27. Für die Rückrichtungen nehmen wir an, dass $\mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$. Wir definieren nun

$$M(\alpha) := \{w \in \mathcal{L}_\Sigma(\alpha) \mid |w| = |\alpha|\}.$$

Offensichtlich gilt $M(\alpha) \subseteq \mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$. Darüber hinaus ist $M(\alpha)$ aber reichhaltig genug, um daraus α zu rekonstruieren:

Behauptung 1 *Ist $\mathcal{L}_\Sigma(\beta) \supseteq M(\alpha)$, so gilt $\alpha \preceq \beta$.*

Der Beweis dieser Behauptung sei Ihnen als Übung überlassen. Da $\mathcal{L}_\Sigma(\beta) \supseteq \mathcal{L}_\Sigma(\alpha)$ und $\mathcal{L}_\Sigma(\alpha) \supseteq M(\alpha)$, gilt $\mathcal{L}_\Sigma(\beta) \supseteq M(\alpha)$. Gemäß Behauptung 1 folgt hieraus $\alpha \preceq \beta$. □

Im Gegensatz zur Inklusion ist die Äquivalenz von Patternsprachen leicht zu überprüfen:

Satz 5.30 *Sei Σ ein Terminalalphabet mit $|\Sigma| \geq 2$ und sei X ein Variablenalphabet. Für alle Pattern $\alpha, \beta \in (\Sigma \cup X)^+$ gilt $\alpha \approx \beta$ genau dann, wenn $\mathcal{L}_\Sigma(\alpha) = \mathcal{L}_\Sigma(\beta)$.*

Beweis: Die Hin-Richtung folgt unmittelbar aus Lemma 5.27 (da $\alpha \approx \beta$ per Definition genau dann gilt, wenn $\alpha \preceq \beta$ und $\beta \preceq \alpha$).

Für die Rück-Richtung stellen wir Folgendes fest: Gilt $\mathcal{L}_\Sigma(\alpha) = \mathcal{L}_\Sigma(\beta)$, muss zwangsläufig $|\alpha| = |\beta|$ gelten (sonst wäre das kürzeste Wort der einen Sprache kürzer als das

[77] Sprachen wie $\mathcal{L}_\Sigma(\beta)$ eignen sich daher gut für hinterhältige Übungsaufgaben der Form „beweisen oder widerlegen Sie die Regularität dieser Sprache“. Ein weiteres (und anspruchsvolleres) Beispiel finden Sie in Aufgabe 3.15

[78] Sie finden den Beweis für unbeschränkte Alphabete in Jiang et al. [9], den Beweis für die beschränkten Alphabete in [3].

kürzeste Wort der anderen Sprache). Also können wir mittels Lemma 5.29 aus $\mathcal{L}_\Sigma(\alpha) \subseteq \mathcal{L}_\Sigma(\beta)$ und $\mathcal{L}_\Sigma(\alpha) \supseteq \mathcal{L}_\Sigma(\beta)$ auf $\alpha \preceq \beta$ und $\beta \preceq \alpha$ schließen. Somit gilt $\alpha \approx \beta$. □

Im Gegensatz zum Inklusionsproblem ist das Äquivalenzproblem für Pattern also trivial entscheidbar: Zwei Pattern erzeugen genau dann die gleiche Sprache, wenn sie identisch sind (abgesehen von einer gegebenenfalls notwendigen Umbenennung der Variablen)[79].

Wir befassen uns nun mit dem Wortproblem für Pattern. Dieses definieren wir wie gewohnt:

Wortproblem für Pattern

Eingabe: Ein Terminalalphabet Σ, ein Variablenalphabet X, ein Pattern $\alpha \in (\Sigma \cup X)^+$ und ein Wort $w \in \Sigma^+$.

Frage: Ist $w \in \mathcal{L}_\Sigma(\alpha)$?

Wie wir sehen werden, ist das Wortproblem für Pattern NP-vollständig. Um dies zu beweisen, betrachten wir das folgende NP-vollständige Entscheidungsproblem:

Drei-Färbbarkeitsproblem

Eingabe: Ein ungerichteter Graph $G = (V, E)$.

Frage: Existiert eine Funktion $f\colon V \to \{\mathtt{r}, \mathtt{g}, \mathtt{b}\}$ mit $f(i) \neq f(j)$ für alle $\{i, j\} \in E$?

Wenn ein Graph G die im Drei-Färbbarkeitsproblem genannten Bedingungen erfüllt, bezeichnen wir G als drei-färbbar, die entsprechende Funktion f bezeichnen wir als Drei-Färbung. Das Drei-Färbbarkeitsproblem ist NP-vollständig (siehe z. B. Garey und Johnson [7][80]). Dadurch können wir nun Folgendes beweisen:

Satz 5.31 *Das Wortproblem für Pattern ist NP-vollständig.*

Beweis: Es ist leicht zu sehen, dass das Wortproblem für Pattern in NP ist: Um nachzuweisen, dass $w \in \mathcal{L}_\Sigma(\alpha)$, genügt es, eine Σ^+-Ersetzung σ zu raten und zu verifizieren, dass $\sigma(\alpha) = w$. Dabei können alle σ mit $|\sigma(\alpha)| > |w|$ ausgeschlossen werden.

Um zu zeigen, dass das Wortproblem NP-hart ist, reduzieren wir das Drei-Färbbarkeitsproblem darauf. Sei $G := (V, E)$ ein ungerichteter Graph. Wir können davon ausgehen, dass $V = \{1, \ldots, n\}$ für ein $n \in \mathbb{N}_{>0}$. Wir wählen nun $\Sigma := \{\mathtt{r}, \mathtt{g}, \mathtt{b}, \#, 0\}$. Das Variablenalphabet X werden wir am Ende der Konstruktion definieren. Unser Ziel ist es nun, ein Pattern α und ein Terminalwort w zu definieren, so dass $w \in \mathcal{L}_\Sigma(\alpha)$ genau

[79] Wie bei den DPDAs haben wir es also mit einer Sprachklasse zu tun, bei der die Inklusion unentscheidbar ist, während die Äquivalenz entscheidbar ist. Allerdings ist die Äquivalenz hier deutlich leichter zu entscheiden als bei den DPDAs.

[80] Dort ist das Problem unter dem Titel Graph K-Colorability aufgeführt, wir verwenden den Fall $K = 3$. Auch in diesem Spezialfall ist das Problem NP-vollständig.

dann gilt, wenn G drei-färbbar ist. Dazu definieren wir erst eine Reihe von Hilfspattern und Hilfswörtern. Für alle $i \in V = \{1, \ldots, n\}$ sei

$$u_i := \#0\,\mathtt{rr}\,\mathtt{gg}\,\mathtt{bb}\,0\#,$$
$$\beta_i := \#y_i\,x_i x_i\,\hat{y}_i\#.$$

Für alle $1 \leq i < j \leq n$ mit $\{i, j\} \in E$ sei

$$v_{i,j} := \#0\,\mathtt{rgbrbgr}\,0\#,$$
$$\gamma_{i,j} := \#z_{i,j}\,x_i x_j\,\hat{z}_{i,j}\#.$$

Außerdem gelte $v_{i,j} := \gamma_{i,j} := \varepsilon$ für alle $1 \leq i < j \leq n$ mit $\{i, j\} \notin E$. Abschließend definieren wir

$$w := u_1 \cdot u_2 \cdots u_n \cdot v_{1,2} \cdot v_{1,3} \cdots v_{n-1,n},$$
$$\alpha := \beta_1 \cdot \beta_2 \cdots \beta_n \cdot \gamma_{1,2} \cdot \gamma_{1,3} \cdots \gamma_{n-1,n}.$$

Wir können wählen X nun so, dass alle in α vorkommenden Variablen darin enthalten sind. Sowohl $|\alpha|$ als auch $|w|$ sind polynomiell in Bezug auf die Größe von G. Wir müssen also nur noch die Korrektheit der Reduktion beweisen:

Behauptung 1 *Wenn G drei-färbbar ist, gilt $w \in \mathcal{L}_\Sigma(\alpha)$.*

Beweis: Angenommen G ist drei-färbbar. Dann existiert eine Funktion $f\colon V \to \{\mathtt{r}, \mathtt{g}, \mathtt{b}\}$ mit $f(i) \neq f(j)$ für alle $\{i, j\} \in E$. Wir verwenden nun f, um eine Σ^+-Ersetzung σ mit $\sigma(\alpha) = w$ zu konstruieren.

Dazu definieren wir $\sigma(\#) := \#$, sowie $\sigma(x_i) := f(i)$ für alle $i \in V$. Unser Ziel ist nun, σ so zu wählen, dass

$$\sigma(\beta_i) = u_i \qquad \text{für alle } i \in V,$$
$$\sigma(\gamma_{i,j}) = v_{i,j} \qquad \text{für alle } \{i, j\} \in E \text{ mit } i < j.$$

Wir beginnen mit den β_i. Hier ist jeweils aus $y_i\,x_i x_i\,\hat{y}_i$ das Wort $0\,\mathtt{rr}\,\mathtt{gg}\,\mathtt{bb}\,0$ zu erzeugen (um die # an den beiden Enden müssen wir uns hier nicht mehr kümmern). Dabei gilt (gemäß unserer Wahl von $\sigma(x_i)$), dass $\sigma(x_i x_i) \in \{\mathtt{rr}, \mathtt{gg}, \mathtt{bb}\}$. Wir können also $\sigma(y_i)$ und $\sigma(\hat{y}_i)$ so wählen, dass diese den Rest des Wortes erzeugen. Durch die beiden Vorkommen von 0 ist jeweils sicher gestellt, dass keine der Variablen durch ε ersetzt werden muss.

Für die $\gamma_{i,j}$ können wir ähnlich vorgehen: Da das Wort $\mathtt{rgbrbgr}$ für jede mögliche Wahl von $a, b \in \{\mathtt{r}, \mathtt{g}, \mathtt{b}\}$ mit $a \neq b$ das Wort ab als Teilwort enthält, können wir $\sigma(x_i x_j)$ in $\mathtt{rgbrbgr}$ „unterbringen“. Der Rest von $\mathtt{rgbrbgr}$ wird nun von $\sigma(z_{i,j})$ oder $\sigma(\hat{z}_{i,j})$ erzeugt. Dabei ist wiederum durch die beiden Vorkommen von 0 sicher gestellt, dass keine der Variablen durch ε ersetzt werden muss. □(Behauptung 1)

Nun müssen wir nur noch beweisen, dass aus $w \in \mathcal{L}_\Sigma(\alpha)$ auch Drei-Färbbarkeit folgt:

Behauptung 2 *Ist $w \in \mathcal{L}_\Sigma(\alpha)$, so ist G drei-färbbar.*

Beweis: Angenommen, $w \in \mathcal{L}_\Sigma(\alpha)$. Dann existiert eine Σ^+-Ersetzung σ mit $\sigma(\alpha) = w$. Gemäß Definition von Σ^+-Ersetzungen muss $\sigma(\#) = \#$ gelten. Da α und w gleich viele Vorkommen von # enthalten, muss $|\sigma(x)|_\# = 0$ für alle $x \in X$ gelten. Somit können wir die Gleichung $\sigma(\alpha) = w$ an den # in Teilgleichungen zerlegen, und zwar

$$\begin{aligned} \sigma(\beta_i) &= u_i && \text{für alle } i \in V, \\ \sigma(\gamma_{i,j}) &= v_{i,j} && \text{für alle } \{i,j\} \in E \text{ mit } i < j. \end{aligned}$$

Aus jeder Gleichung $\sigma(\beta_i) = u_i$ folgt $\sigma(y_i x_i x_i \hat{y}_i) = 0\texttt{rrggbb}0$ (durch „Abknapsen“ der # an den beiden Enden). Da keine der Variablen auf ε abgebildet werden darf, werden die beiden Vorkommen von 0 aus y_i bzw. $\hat{y}_i$ erzeugt. Also muss $\sigma(x_i x_i)$ in `rrggbb` liegen. Daraus folgt unmittelbar $\sigma(x_i) \in \{\texttt{r}, \texttt{g}, \texttt{b}\}$, denn $|\sigma(x_i)| > 1$ würde sofort zu einem Widerspruch führen. Wir definieren nun $f(i) := \sigma(x_i)$ für alle $i \in V$.

Wir zeigen nun, dass f eine Drei-Färbung von G ist. Angenommen, $\{i,j\} \in E$ mit $i < j$. Dann ist

$$\begin{aligned} \sigma(\gamma_{i,j}) &= \#\sigma(z_{i,j}\, x_i x_j\, \hat{z}_{i,j})\# \\ &= \#\sigma(z_{i,j})\, \sigma(x_i x_j)\, \sigma(\hat{z}_{i,j})\# \\ &= \#0\, \texttt{rgbrbgr}\, 0\#. \end{aligned}$$

Da $\sigma(z_{i,j}) \neq \varepsilon$ und $\sigma(\hat{z}_{i,j}) \neq \varepsilon$ ist, werden die beiden Vorkommen von 0 von diesen Variablen erzeugt. Also muss $\sigma(x_i x_j)$ in `rgbrbgr` liegen. Da $\sigma(x_i), \sigma(x_j) \in \{\texttt{r}, \texttt{g}, \texttt{b}\}$, muss $\sigma(x_i) \neq \sigma(x_j)$ gelten, denn `rgbrbgr` enthält keine direkt aufeinander folgenden Vorkommen des selben Buchstaben. Somit ist $f(x_i) \neq f(x_j)$, und f ist eine Drei-Färbung von G. □(Behauptung 2)

Durch die Konstruktion von α und w aus G haben wir also eine Polynomialzeitreduktion von Drei-Färbbarkeit auf das Wortproblem für Pattern angeben. Das Wortproblem für Pattern ist also NP-hart und somit auch NP-vollständig. □

Durch etwas zusätzlichen Aufwand ist es möglich, α ohne Terminale zu konstruieren (also $\alpha \in X^+$), und außerdem kann Σ auf ein zweibuchstabiges Alphabet beschränkt werden.

Je nach Standpunkt kann Satz 5.31 als schlechte Nachricht verstanden werden, nämlich als Zeichen, dass Patternsprachen nicht nur eine vergleichsweise ungewöhnliche oder gar zu stark eingeschränkte Ausdrucksstärke haben, sondern dass sie auch in Bezug auf die Handhabbarkeit eher unpraktische Eigenschaften haben.

Allerdings ist es auch möglich, dies als einen Vorteil zu betrachten: In vielen praxisrelevanteren Modellen ist es wünschenswert, einen Mechanismus zur Spezifikation von Wiederholungen zu haben (ein solches Modell werden wir gleich im Anschluss betrachten). Sobald dieser Mechanismus es erlaubt, Patternsprachen auszudrücken, sind deren Negativresultate (wie zum Beispiel unentscheidbare Inklusion und NP-Härte des Wortproblems) auch beim mächtigeren Mechanismus vorhanden. Statt die Beweise für jedes einzelne Negativresultat neu führen zu müssen, genügt es dann, die Ausdrückbarkeit von Patternsprachen zu beweisen.

5.3.2 Erweiterte Reguläre Ausdrücke

In Abschnitt 3.3 haben wir die regulären Ausdrücke kennen gelernt. Reguläre Ausdrücke wurden zwar ursprünglich als Modell der theoretischen Informatik erfunden und sind dort weiterhin im Gebrauch, parallel dazu haben Sie sich aber auch zu einem häufig genutzten Werkzeug in der praktischen Informatik entwickelt. Dort werden sie oft verwendet, um Muster in Texten zu beschreiben, wie zum Beispiel bei Suchen[81], oder zum Definieren von Filterregeln.

Dabei haben sich schnell mehrere Dialekte entwickelt, die im Vergleich zu den klassischen regulären Ausdrücken einige Erweiterungen besitzen. Viele dieser Erweiterungen ändern die Ausdrucksstärke der Sprachen nicht. Wir betrachten kurz einige Beispiele, die in den meisten verbreiteten Dialekten anzutreffen sind:

Eckige Klammern Die meisten Dialekte von regulären Ausdrücken erlauben es, Abkürzungen wie `[a-z]` zu verwenden. Dies steht für die Buchstaben `a` bis `z`, und ist äquivalent zu dem Ausdruck $(\texttt{a} \mid \cdots \mid \texttt{z})$.

Zeichenklassen Außerdem haben häufige Mengen von Buchstaben eigene Abkürzungen, wie zum Beispiel `[:alpha:]` für die Buchstaben von `a` bis `z` (in Groß- und Kleinschreibung), oder `[:digit:]` für die Ziffern `0` bis `9`.

Fragezeichen Das Zeichen `?` steht als Abkürzung für „oder ε“. Für beliebige Ausdrücke α ist also $\mathcal{L}(\alpha?) = \mathcal{L}(\alpha) \cup \{\varepsilon\}$.

Punkt Das Zeichen `.` steht für einen beliebigen Buchstaben.

Quantoren Ist α ein regulärer Ausdruck, so kann die Schreibweise $(\alpha)\{m, n\}$ verwendet werden (mit $m, n \in \mathbb{N}$, $m \leq n$). Es gilt $\mathcal{L}((\alpha)\{m, n\}) = \bigcup_{m \leq i \leq n} \mathcal{L}(\alpha)^i$.

Es ist leicht zu sehen, dass keine dieser Erweiterungen zu nicht-regulären Sprachen führt, da die entsprechenden klassischen Ausdrücke schnell konstruiert werden können. Allerdings verfügen viele modernen Implementierungen von regulären Ausdrücken über einen weiteren Operator, der die Ausdrucksstärke der regulären Ausdrücke deutlich erweitert. Wir betrachten dazu das folgende Beispiel:

Beispiel 5.32 In der Programmiersprache PERL (und jeder Programmiersprache, die PERL-kompatible reguläre Ausdrücke verwendet), kann der folgende „reguläre“ Ausdruck verwendet werden:

```
((a|b)*)\1
```

Der Teilausdruck `((a|b)*)` beschreibt dabei auf die gewohnte Art ein beliebiges Wort w über den Buchstaben `a` und `b`. Der Rückreferenzoperator `\1` beschreibt dabei exakt das gleiche Wort wie der Teilausdruck, der aus dem ersten Klammerpaar gebildet wird. Wir zählen dabei die Klammernpaare von links, ausgehend von der öffnenden Klammer –

[81] Wenn Sie in dem Texteditor Ihrer Wahl nicht anhand von regulären Ausdrücken suchen können, dann kennen Sie den Texteditor Ihrer Wahl nicht besonders gut, oder Sie haben ihn schlecht ausgewählt.

der entsprechende Teilausdruck ist in diesem Beispiel `((a|b)*)`, und der (hier nicht vorkommende) Operator `\2` würde sich auf `(a|b)` beziehen.

Der oben angegebene „reguläre“ Ausdruck beschreibt also die Sprache aller Wörter der Form ww, mit $w \in \{\mathtt{a}, \mathtt{b}\}^*$ (also die uns bereits vertraute Copy-Sprache). Somit können in „regulären“ Ausdrücken, wie sie in PERL verwendet werden, nicht-reguläre Sprachen definiert werden.

Seit einigen Jahren ist es in vielen Sprachen auch möglich, diese Wiederholungen durch direktes benennen der zu wiederholenden Stellen zu ermöglichen. Dadurch entfällt das eher umständliche Zählen von Klammernpaaren. Beispielsweise kann der oben angegebene Ausdruck in der Sprache Python wie folgt umgesetzt werden:

```
(?P<X>(a|b)*)(?P=X)
```

Dabei kann `(?P<X>(a|b)*)` verstanden werden als „verarbeite den Ausdruck `(a|b)*` und speichere das entsprechende Wort in der Variable `X`, während `(?P=X)` dem Inhalt der Variablen `X` entspricht. Der Teilausdruck `(?P=X)` übernimmt also die Rolle von `\1` im Ausdruck weiter oben. ♢

Wir werden uns im Folgenden die Eigenschaften von regulären Ausdrücken mit solchen Wiederholungsoperatoren genauer ansehen. Um nicht von den (oft nicht oder nur unzureichend dokumentierten) Eigenschaften echter Systeme abhängig zu sein, definieren wir daher eine Variante der regulären Ausdrücke. Es gibt eine Vielzahl von verschiedenen Ansätzen, um die Syntax und Semantik von solchen erweiterten regulären Ausdrücken zu definieren. Viele unterscheiden sich nur in einigen Spezialfällen, und oft wird die Definition der Semantik überraschend aufwändig. Wir verwenden die folgende Definition:

Definition 5.33 Sei Σ ein Terminalalphabet und X ein Variablenalphabet. Zu Beginn haben alle Variablen den Wert $\emptyset$.

Die Menge der **erweiterten regulären Ausdrücke (über Σ, mit Variablen aus X)** und der durch sie definierten Sprachen ist wie folgt rekursiv definiert:

1. $\emptyset$ ist ein erweiterter regulärer Ausdruck und passt auf kein Wort..
2. ε ist ein erweiterter regulärer Ausdruck und passt auf ε.
3. Jedes Terminal $a \in \Sigma$ ist ein erweiterter regulärer Ausdruck und passt auf das Wort a.
4. Jede Variable $x \in X$ ist ein erweiterter regulärer Ausdruck und passt auf den aktuellen Wert der Variable x.
5. **Konkatenation:** Sind α und β erweiterte reguläre Ausdrücke, dann ist auch $(\alpha \cdot \beta)$ ein erweiterter regulärer Ausdruck passt auf alle Wörter uv, für die α auf u und β auf v passt.

6. **Vereinigung:** Sind α und β erweiterte reguläre Ausdrücke, dann ist auch $(\alpha \mid \beta)$ ein erweiterter regulärer Ausdruck, der auf alle Wörter passt auf die α oder β passen.

7. **Kleene-Stern:** Ist α ein erweiterter regulärer Ausdruck, dann ist auch α^* erweiterter ein regulärer Ausdruck. Dieser passt auf alle Wörter der Form $w_1 \cdots w_n$ ($n \in \mathbb{N}$), für die α auf alle w_i passt.

8. **Variablen-Bindung:** Ist α ein erweiterter regulärer Ausdruck und ist $x \in X$, dann ist auch $\langle x \colon \alpha \rangle$ ein erweiterter regulärer Ausdruck. Wenn α auf ein Wort w passt, dann passt auch $\langle x \colon \alpha \rangle$ auf w, und gleichzeitig wird x auf den Wert w gesetzt.

Ist α ein erweiterter regulärer Ausdruck, so definieren wir die Sprache $\mathcal{L}(\alpha)$ als die Menge aller Wörter $w \in \Sigma^*$, auf die α passt.

Wir gestatten dabei die gleichen Vereinfachungen und Kurzschreibweisen wie bei den klassischen regulären Ausdrücken. Um Verwirrungen zu vermeiden vereinbaren wir, dass in jedem erweiterten regulären Ausdruck jede Variable x höchstens ein einziges Mal in einem Teilausdruck der Form $\langle x \colon \alpha \rangle$ vorkommen darf. Um den Mechanismus ein wenig besser zu verstehen, betrachten wir ein paar Beispiele.

Beispiel 5.34 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $X := \{x\}$. Wir betrachten den erweiterten regulären Ausdruck $\alpha := \langle x \colon (\mathtt{a} \mid \mathtt{b})^* \rangle x$. Dieser erzeugt die Sprache $\mathcal{L}(\alpha) = \{ww \mid w \in \Sigma^*\}$. ◊

Beispiel 5.35 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei $X := \{x, y, z\}$. Wir betrachten den erweiterten regulären Ausdruck

$$\alpha := \langle x \colon (\mathtt{a} \mid \mathtt{b})^+ \rangle\, x\, \langle y \colon (\mathtt{a} \mid \mathtt{b})^+ \rangle\, y\, \langle z \colon (\mathtt{a} \mid \mathtt{b})^+ \rangle\, z\, xx.$$

Dieser Ausdruck erzeugt die gleiche Sprache wie das Pattern $\beta := xx\ yy\ zz\ xx$; es gilt also $\mathcal{L}(\alpha) = \mathcal{L}_\Sigma(\beta)$. ◊

Beispiel 5.36 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$, $n \in \mathbb{N}_{>0}$ $X = \{x_1, \ldots, x_n\}$. Sei

$$\alpha_n := \langle x_1 \colon (\mathtt{a} \mid \mathtt{b}) \rangle \langle x_2 \colon (\mathtt{a} \mid \mathtt{b}) \rangle \cdots \langle x_n \colon (\mathtt{a} \mid \mathtt{b}) \rangle\, x_n\, x_{n-1} \cdots x_1.$$

Dann ist $\mathcal{L}(\alpha_n) = \{ww^R \mid w \in \{\mathtt{a}, \mathtt{b}\}^n\}$. Der erweiterte reguläre Ausdruck α_n erzeugt also alle Palindrome der Länge $2n$ über dem Alphabet Σ. ◊

Beispiel 5.37 Sei $\Sigma := \{\mathtt{a}\}$ und $X := \{x, y, z\}$. Wir betrachten den erweiterten regulären Ausdruck

$$\alpha := \langle x : \mathtt{aa} \rangle \langle y : xx \rangle \langle z : yy \rangle zz.$$

Dieser erzeugt die Sprache $\mathcal{L}(\alpha) = \{\mathtt{a}^3 2\}$. Dabei wird in der Variablen x das Wort $\mathtt{a}^2$ gespeichert, in y das Wort $\mathtt{a}^4$ und in z das Wort $\mathtt{a}^8$. ◊

Beispiel 5.38 Sei $\Sigma := \{\mathtt{a}\}$ und $X := \{x, y\}$. Wir betrachten den erweiterten regulären Ausdruck $\alpha := \langle x \colon \mathtt{a}(\mathtt{a}^+) \rangle (x)^+$. Diese erzeugt die folgende Sprache:

$$\mathcal{L}(\alpha) = \{w \cdot w^n \mid w = \mathtt{aa}^m, n \in \mathbb{N}_{>0}, m \in \mathbb{N}_{>0}\}$$

$$
\begin{aligned}
&= \{w^{i+2} \mid w = \mathtt{a}^{j+2}, i \in \mathbb{N}, j \in \mathbb{N}\} \\
&= \{(\mathtt{a}^{j+2})^{i+2} \mid i \in \mathbb{N}, j \in \mathbb{N}\} \\
&= \{\mathtt{a}^{(j+2)(i+2)} \mid i \in \mathbb{N}, j \in \mathbb{N}\} \\
&= \{\mathtt{a}^z \mid z \in \mathbb{N}_{>0}, z \text{ ist eine zusammengesetzte Zahl}\}.
\end{aligned}
$$

Zur Erinnerung: Eine Zahl $z \in \mathbb{N}_{>0}$ ist zusammengesetzt, wenn Primzahlen p, q existieren, so dass $z = pq$. Dabei müssen die Zahlen $(i+2)$ und $(j+2)$ selbst keine Primzahlen sein; es genügt zu beobachten, dass das Produkt zweier Zahlen größer gleich 2 stets auch das Produkt von (mindestens) zwei Primzahlen ist.

Definieren wir nun $\beta := (\alpha) \mid \mathtt{a}$, also $\beta := (\langle x \colon \mathtt{a}(\mathtt{a}^+)\rangle(x)^+) \mid \mathtt{a}$, so erhalten wir einen erweiterten regulären Ausdrücke für die Sprache

$$\mathcal{L}(\beta) = \{\mathtt{a}^n \mid n \in \mathbb{N}_{>0},\ n \text{ ist keine Primzahl}\}. \qquad \Diamond$$

Hinweis 5.39 Unsere Definition der Semantik der erweiterten regulären Ausdrücke in Definition 5.33 hat einige Schwächen. Beispielsweise ist es nicht klar, wie Ausdrücke wie $\alpha := x\ \langle x \colon \mathtt{a}\rangle$ interpretiert werden sollen. Beschreibt dieser Ausdruck die Sprache $\{\mathtt{aa}\}$, oder die Sprache $\emptyset$? In den meisten praktischen Anwendungen wird davon ausgegangen, dass die Ausdrücke von links nach rechts ausgewertet werden, so dass von diesem Standpunkt aus $\mathcal{L}(\alpha) = \emptyset$ gelten müsste. Außerdem werden unbelegte Variablen in der Praxis oft mit ε belegt, so dass $\mathcal{L}(\alpha) = \{\varepsilon\}$ gelten müsste.

Ähnliche Problem treten bei Ausdrücken wie $\beta := \langle x \colon \mathtt{a}\rangle\ x\ \langle x \colon \mathtt{a}\rangle$ und $\gamma := (\mathtt{a} \mid \langle x \colon \mathtt{b}\rangle)x$ auf. Die Resultate, die wir im Folgenden betrachten, gelten unabhängig von diesen definitorischen Feinheiten – wir werden diese Unterschiede ignorieren und uns mit diesen Randfällen der Definition nicht weiter befassen. Allerdings zeigen diese Schwierigkeiten, dass es bei der Erstellung eines eigentlich einfachen Modells schnell zu Definitionsschwierigkeiten kommen kann.

Um die Eigenschaften der Sprachen untersuchen zu können, die von erweiterten regulären Ausdrücken erzeugt werden, definieren wir eine entsprechende Sprachklasse:

Definition 5.40 Sei Σ ein Alphabet. Eine Sprache $L \subseteq \Sigma^*$ heißt **erweitert regulär** wenn ein erweiterter regulärer Ausdruck α existiert, so dass $L = \mathcal{L}(\alpha)$. Wir bezeichnen die **Klasse aller erweitert regulären Sprachen** über dem Alphabet Σ mit XREG_Σ, und definieren die Klasse aller erweitert regulären Sprachen $\mathrm{XREG} := \bigcup_{\Sigma \text{ ist ein Alphabet}} \mathrm{XREG}_\Sigma$.

Der nahe liegende nächste Schritt ist der Vergleich der Klasse XREG mit den Sprachklassen, die wir bereits kennen gelernt haben. Um dieses Thema an einem Stück erledigen zu können, erstellen wir zuallererst das nötige Handwerkszeug, zu beweisen, dass eine Sprache nicht erweitert regulär ist:

Lemma 5.41 (Pumping-Lemma für erweitert reguläre Sprachen) *Sei Σ ein Alphabet. Für jede erweitert reguläre Sprache $L \subseteq \Sigma^*$ existiert eine* ***Pumpkonstante*** *$n_L \in \mathbb{N}_{>0}$, so dass für jedes Wort $z \in L$ mit $|z| \geq n_L$ folgende Bedingung erfüllt ist: Es existieren ein $m \in \mathbb{N}_{>0}$ und Wörter $x_0, x_1, \ldots, x_m, y \in \Sigma^*$ mit*

1. *$x_0 \cdot y \cdot x_1 \cdot y \cdot x_2 \cdots x_{m-1} \cdot y \cdot x_m = z$,*
2. *$|x_0 y| \leq n_L$,*
3. *$|y| \geq 1$,*
4. *für alle $i \in \mathbb{N}$ ist $x_0 \cdot y^i \cdot x_1 \cdot y^i \cdot x_2 \cdots x_{m-1} \cdot y^i \cdot x_m \in L$.*

Beweis: Da $L \in \mathrm{XREG}$ existiert ein erweiterter regulärer Ausdruck α mit $\mathcal{L}(\alpha) = L$. Wir definieren nun $n_L := |\alpha| 2^k + 1$, wobei k die Anzahl von Vorkommen von Variablen in α ist. Sei $z \in \mathcal{L}(\alpha)$ mit $|z| \geq n_L$.

Um z aus α zu erzeugen, muss ein Teilausdruck mit einem Kleene-Stern verwendet werden, denn ohne Benutzung eines Kleene-Stern können maximal Wörter der Länge $|\alpha| 2^k$ erzeugt werden (jede Verwendung einer Variablen kann die Wortlänge verdoppeln, bei k Vorkommen von Variablen kann maximal eine Ver-2^k-fachung eintreten).

Wir zerlegen z nun in Wörter $x_0 y w = z$, wobei y das erste nicht-leere Teilwort von z ist, das durch einen Teilausdruck mit Kleene-Stern erzeugt wird. Das heißt, es existiert ein Teilausdruck $(\beta)^*$, und y wird von β erzeugt. Dadurch ist sichergestellt, dass $|x_0 y| \leq |\alpha| 2^k < n_L$.

Wir unterscheiden nun die beiden folgenden Fälle:

1. Der Teilausdruck β^* wird nicht in einer Variable gespeichert, die später verwendet wird.
2. Der Teilausdruck β^* wird in einer Variable gespeichert, die später verwendet wird.

Dabei bezieht sich Fall 2 auch auf solche Fälle, in denen ein Teilausdruck gespeichert wird, der β^* umgibt – also alle Fälle, in denen y an einer anderen Stelle wiederholt wird.

Fall 1: Da β^* nicht in einer Variable gespeichert wird, die später verwendet wird, können den Kleene-Stern beliebig oft wiederholen (oder auslassen). Wir wählen $m := 1$ und $x_1 := w$. Es gilt also $z = x_0 y x_1$, und für alle $i \in \mathbb{N}$ ist $x_0 y^i x_1 \in L$.

Fall 2: Da β^* in einer Variable gespeichert wird, wird y an weiteren Stellen wiederholt. Wir zerlegen w so, dass $w = x_1 y x_2 y \cdots x_{m-1} y x_m$, wobei m und die x_i so gewählt sind, dass die y in dieser Zerlegung *allen* Stellen entsprechen, in denen die Variable, in der β^* gespeichert ist, referenziert wird (dies gilt auch für eventuell existierende andere Variablen, in denen diese Variable mit gespeichert wird, wie in Beispiel 5.37). Nun gilt

$$\begin{aligned} z &= x_0 y w \\ &= x_0 y x_1 y x_2 y \cdots x_{m-1} y x_m. \end{aligned}$$

Durch Wiederholen des Kleene-Stern in β^* können wir nun das erste y pumpen, und durch die Variablen wird dieses Pumpen an allen weiteren Vorkommen von y nachvollzogen. Es gilt also

$$x_0 y^i x_1 y^i x_2 y^i \cdots x_{m-1} y^i x_m \in L$$

für alle $i \in \mathbb{N}$. □

Anhand des Pumping-Lemmas für erweitert reguläre Sprachen können wir nun einige Sprachen aus der Klasse XREG ausschließen:

Beispiel 5.42 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $L := \{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Angenommen, $L \in$ XREG. Dann existiert eine Pumpkonstante n_L. Wir wählen nun das Wort $z := \mathtt{a}^{n_L}\mathtt{b}^{n_L}$. Dann existieren ein $m \geq 1$ und eine Zerlegung von z in Wörter $x_0, \ldots, x_m, y \in \Sigma^*$, so dass diese das Pumping-Lemma für XREG erfüllen. Es gilt also $x_0 \cdot y \cdot x_1 \cdot y \cdot x_2 \cdots x_{m-1} \cdot y \cdot x_m = z$ und $|x_0 y| \leq n_L$. Somit ist $y = \mathtt{a}^m$ mit $1 \leq m \leq n_L$. Nun führt Aufpumpen mit $i = 2$ zum gewünschten Widerspruch, denn für das Wort

$$z_2 := x_0 \cdot y^2 \cdot x_1 \cdot y^2 \cdot x_2 \cdots x_{m-1} \cdot y^2 \cdot x_m,$$

gilt $|z_2|_\mathtt{a} > |z_2|_\mathtt{b}$ und somit $z_2 \notin L$. ◊

Beispiel 5.43 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und

$$\begin{aligned} L &:= \{ucv \mid u, v \in \{\mathtt{a}, \mathtt{b}\}^*, |u| = |v|\} \\ &= \{\{\mathtt{a}, \mathtt{b}\}^i\mathtt{c}\{\mathtt{a}, \mathtt{b}\}^i \mid i \in \mathbb{N}\}. \end{aligned}$$

Angenommen, $L \in$ XREG. Dann existiert eine Pumpkonstante n_L. Wir wählen nun das Wort $z := \mathtt{a}^{n_L}\mathtt{c}\mathtt{b}^{n_L}$. Es gilt $z \in L$. Also existieren ein $m \geq 1$ und Wörter $x_0, \ldots, x_m, y \in \Sigma^*$, so dass diese das Pumping-Lemma für XREG erfüllen. Da $|x_0 y| \leq n_L$, kann y nur aus as bestehen. Also kann rechts von c kein y vorkommen, somit führt Aufpumpen zu einem Wort, bei dem zur Linken des c mehr Buchstaben stehen als zu seiner Rechten. Das aufgepumpte Wort kann also nicht in L liegen; somit muss $L \in$ XREG gelten. ◊

Beispiel 5.44 Sei $\Sigma := \{\mathtt{a}\}$ und $L := \{\mathtt{a}^p \mid p \text{ ist eine Primzahl}\}$. Angenommen, $L \in$ XREG. Dann existiert eine Pumpkonstante n_L. Wir wählen nun das Wort $z := \mathtt{a}^p$, wobei $p \geq n_L$ eine Primzahl sei (da unendlich viele Primzahlen existieren ist dies möglich, und offensichtlich gilt $z \in L$). Nun existieren ein $m \geq 1$ und eine Zerlegung von z in Wörter $x_0, \ldots, x_m, y \in \Sigma^*$, so dass diese das Pumping-Lemma für XREG erfüllen. Sei $k := |y|$. Da wir Aufpumpen dürfen wissen wir, dass für alle $j \in \mathbb{N}_{>0}$ $\mathtt{a}^{p+jmk} \in L$ gelten muss (y kommt insgesamt m mal vor und hat die Länge k). Also ist $p + jmk$ für alle $j \in \mathbb{N}$ eine Primzahl. Wir wählen nun $j = p$ und erhalten so

$$\begin{aligned} p' &:= p + jmk \\ &= p + pmk \\ &= p(1 + mk). \end{aligned}$$

Da $|m| \geq 1$ und $k = |y| \geq 1$, ist $mk \geq 1$ und somit $(1 + mk) \geq 2$. Also kann p' keine Primzahl sein, Widerspruch. ◊

Nun haben wir das nötige Werkzeug, um die Klasse XREG mit den anderen bisher behandelten Klassen zu vergleichen. In Abbildung 5.3 werden diese Verhältnisse auch graphisch dargestellt.

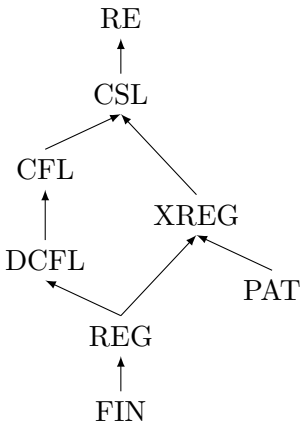


Abbildung 5.3: Die Zusammenhänge zwischen allen bisher von uns betrachteten Klassen. Ein Pfeil von einer Klasse A zu einer Klasse B bedeutet, dass A in B echt enthalten ist (also $A \subset B$).

Satz 5.45 *Die Klasse* XREG *verhält sich wie folgt zu den anderen Klassen:*

1. REG $\subset$ XREG*,*
2. PAT $\subset$ XREG*,*
3. CFL *und* XREG *sind unvergleichbar,*
4. XREG $\subset$ CSL*.*

Beweis: *Zu 1:* Die Inklusion REG $\subseteq$ XREG gilt per Definition, denn jeder reguläre Ausdruck ist auch ein erweitert regulärer Ausdruck. Gemäß Beispiel 5.34 ist die Copy-Sprache eine erweitert reguläre Sprache, also gilt REG $\neq$ XREG.

Zu 2: Es gilt PAT $\subseteq$ XREG, denn jedes Pattern lässt sich unmittelbar in einen äquivalenten erweiterten regulären Ausdruck konvertieren. Sei Σ ein Terminal- und X ein Variablenalphabet, und sei $\alpha \in (\Sigma \cup X)^+$ ein Pattern. Wir konvertieren nun von links nach rechts das Pattern α in einen erweiterten regulären Ausdruck α_R, indem wir jede Stelle von α umschreiben. Ist die aktuelle Stelle ein Terminal, so bleibt dies unverändert. Ist es das erste Vorkommen einer Variable $x \in X$, so ersetzen wir dieses durch $\langle x\colon (\Sigma^+)\rangle$ (wobei der Teilausdruck Σ^+ für die Veroderung der Buchstaben aus Σ stehe). Jedes weitere Vorkommen einer Variable $x \in X$ bleibt unverändert. Für den so entstehenden erweiterten regulären Ausdruck α_R gilt offensichtlich $\mathcal{L}_\Sigma(\alpha) = \mathcal{L}(\alpha_R)$. In Beispiel 5.35 finden Sie ein Beispiel für diese Konvertierung. Die Ungleichung PAT $\neq$ XREG folgt aus REG $\subset$ XREG und der Unvergleichbarkeit von PAT und REG.

Zu 3: Dass XREG $\not\subseteq$ CFL gilt, folgt aus Beispiel 5.34, denn die Copy-Sprache ist erweitert regulär, aber nicht kontextfrei. Um zu zeigen, dass CFL $\not\subseteq$ XREG gilt, betrachten wir die Sprache $\{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$. Diese ist kontextfrei, aber gemäß Beispiel 5.42 nicht erweitert regulär.

Zu 4: Der wahrscheinlich leichteste Beweis für die Inklusion XREG $\subseteq$ CSL ist die Angabe eines Verfahrens, das aus einem erweiterten regulären Ausdruck α einen LBA für die Sprache $\mathcal{L}(\alpha)$ konstruiert. Da wir dies nicht genauer betrachtet haben, lassen wir diesen Teil des Beweises aus. Die Echtheit der Inklusion folgt aus CFL $\not\subseteq$ XREG. □

Anhand der Sprache $\{\mathtt{a}^i\mathtt{b}^i \mid i \in \mathbb{N}\}$ lässt sich auch zeigen, dass XREG und DCFL unvergleichbar sind.

Analog zu den bisher betrachteten Modellen definieren wir das Wortproblem für erweiterte reguläre Ausdrücke folgendermaßen:

Wortproblem für erweiterte reguläre Ausdrücke

Eingabe: Ein erweiterter regulärer Ausdruck α über einem Alphabet Σ und ein Wort $w \in \Sigma^*$.

Frage: Ist $w \in \mathcal{L}(\alpha)$?

Dies ist einer der Fälle, in denen wir von den Erkenntnissen zu Patternsprachen profitieren können:

Satz 5.46 *Das Wortproblem für erweiterte reguläre Ausdrücke ist NP-vollständig.*

Beweis: NP-Härte folgt direkt aus der NP-Vollständigkeit des Wortproblems für Pattern, da sich jedes Pattern direkt in einen erweiterten regulären Ausdruck umschreiben lässt.

Zugehörigkeit zu NP lässt sich auf die übliche Art beweisen: Eine Zuordnung von Teilausdrücken von α zu Teilwörtern von w lässt sich nichtdeterministisch raten und kann dann in Polynomialzeit überprüft werden. □

Da die in der Praxis verwendeten „regulären“ Ausdrücke in den meisten Dialekten mindestens die gleiche Ausdrucksstärke haben wie unsere erweiterten regulären Ausdrücke, ist das Wortproblem für diese also NP-hart. Auf den ersten Blick mag es widersinnig erscheinen, dass in der Praxis ein Algorithmen mit einer (nach momentanem Kenntnisstand) exponentiellen Laufzeit verwendet werden. Allerdings bedeutet dies nur, dass beim Auswerten von diesen „regulären“ Ausdrücken keine polynomiellen Laufzeitgarantien gegeben werden können. Beim erstellen von solchen regulären Ausdrücken muss also zusätzlich zur Korrektheit noch auf die Effizienz geachtet werden. Dazu wird von den Programmierenden erwartet, dass sie nicht nur die Bedeutung der Ausdrücke beherrschen, sondern zusätzlich mit der Arbeitsweise des Auswertungsalgorithmus gut genug vertraut sind, um die Laufzeit abschätzen zu können[82].

Durch Patternsprachen wissen wir ebenfalls, dass das Inklusionsproblem für erweiterte reguläre Ausdrücke unentscheidbar ist. Wenn man nun zusätzlichen Aufwand betreibt und die Konstruktion aus dem Beweis für Patternsprachen verfeinert und dabei die zusätzliche Ausdrucksstärke von erweiterten regulären Ausdrücken verwendet, kann man sogar zeigen, dass für diese auch Äquivalenz, Universalität und Regularität unent-

[82] Oder man frickelt einfach schnell einen Ausdruck hin und hofft, dass er schon effizient genug ist. Das kann häufig und lange genug funktionieren.

scheidbar sind, selbst X nur eine einzige Variable enthält und diese nur in der Form $\langle x\colon \mathtt{a}^+\rangle$ verwendet wird[83].

5.4 Aufgaben

Aufgabe 5.1 Sei $\Sigma := \{\mathtt{a}\}$. Die monotone Grammatik $G := (\Sigma, V, P, S)$ sei definiert durch $V := \{S, A, C, D, E\}$ und eine Menge P, die genau die folgenden Regeln enthalte:

$$\begin{array}{lll} S \to CAD, & A\mathtt{a} \to \mathtt{a}\mathtt{a}A, & AD \to B\mathtt{a}D, \\ \mathtt{a}B \to B\mathtt{a}, & CB \to CA, & CA \to AA, \\ AD \to \mathtt{a}E, & AE \to \mathtt{aa}, & S \to \mathtt{aa}. \end{array}$$

Geben Sie die Sprache $\mathcal{L}(G)$ an.

Aufgabe 5.2 Beweisen Sie Behauptung 1 im Beweis von Lemma 5.29.

Aufgabe 5.3 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und sei

$$L := \left\{ \mathtt{a}^m \mathtt{b}^n \mathtt{a}^{\frac{m}{2}} \mid m, n \in \mathbb{N}_{>0}, m \text{ gerade} \right\}.$$

Geben Sie einen erweiterten regulären Ausdruck α an mit $\mathcal{L}(\alpha) = L$.

5.5 Bibliographische Anmerkungen

Dieser Abschnitt ist momentan nur ein Platzhalter. In Kürze werden hier einige Kommentare zu den verwendeten Quellen und weiterführendem Lesematerial zu finden sein.

[83] Details können Sie in der Arbeit [6] finden.

6 Anwendungen

Dieses Kapitel geht auf einige mögliche Anwendungsfelder des Stoffes ein.

6.1 XML und deterministische reguläre Ausdrücke

In diesem Abschnitt befassen wir uns kurz mit einigen formalen Aspekten der Auszeichungssprache XML (die Abkürzung XML steht für *Extensible Markup Language*). Diese ist eine textbasierte Sprache, die zur hierarchischen Darstellung von Daten in Textform verwendet wird. Wegen der Komplexität von XML und der Vielzahl von theoretisch interessanten Fragestellungen können wir dabei nur einen Bruchteil der Fragestellungen betrachten. Wir konzentrieren uns dabei auf die Beschreibung der Struktur von XML-Dateien durch sogenannte *DTDs* (von *Document Type Definition*). Selbst diese betrachten wir nur aus einem vergleichsweise eingeschränkten Blickwinkel; allerdings können viele der dadurch gewonnenen Erkenntnisse auf allgemeinere Fälle übertragen werden.

Wichtigster Baustein von XML sind die sogenannten *Elemente*. Diese haben einen Namen, wie zum Beispiel `hmtl`, und werden durch einen durch *Auszeichner* (engl. *Tags*) begrenzt. Dabei wird bei einem öffnenden Auszeichner der Name des Elements mit spitzen Klammern umgeben, zum Beispiel `<html>`, bei schließenden Auszeichner wird nach der vorderen spitzen Klammer noch ein Schrägstrich hinzugefügt, zum Beispiel `</html>`. Zwischen den beiden Auszeichnern wird der eigentliche Inhalt des Elements aufgeführt. Dieser kann aus weiteren Elementen bestehen, oder aus Text.

Dabei muss darauf geachtet werden, dass Auszeichner nicht miteinander verschränkt werden; es ist als zum Beispiel verboten, Elemente mit Namen `a` und `b` in der Form `<a><b></a></b>` zu verschachteln, während `<a><b></b></a>` gestattet ist. Daher können XML-Dateien auch als Bäume interpretiert werden. Dabei stehen jeweils die Namen der Elemente in den Knoten; wenn ein Element weitere Elemente enthält, so sind diese dessen Kinder (oder Kindeskinder, abhängig von der Verschachtelung). Diese Kinder haben eine wohldefinierte Reihenfolge, die der Reihenfolge in der Datei entspricht. Neben diesen Kindern kann ein Element auch Text enthalten. Jeder zusammenhängende Textblock wird in einem *Textknoten* gespeichert; diese Textknoten werden wie die Elemente als Kinder behandelt.

Jede XML-Datei besteht dabei aus einem Element, dessen Auszeichner die komplette Datei umschließen. Da dieses Element die Wurzel des entsprechenden Baumes bildet, wird es auch als *Wurzelement* bezeichnet. Wir betrachten dies an einem kurzen Beispiel:

Beispiel 6.1 Wir betrachten eine kurze XML-Datei, die eine Sammlung von Cocktail-Rezepten enthält (wir haben diese in Beispiel 4.2 schon einmal erwähnt):

```
<cocktails>
 <cocktail>
  <name>Whisky Sour</name>
  <zutaten>
   <zutat><name>Bourbon</name><menge>4 cl</menge></zutat>
   <zutat><name>Zitronensaft</name><menge>2 cl</menge></zutat>
   <zutat><name>Gomme-Sirup</name><menge>1 cl</menge></zutat>
  </zutaten>
 </cocktail>
 <cocktail>
  <name>Brandy Alexander</name>
  <zutaten>
   <zutat><name>Weinbrand</name><menge>3 cl</menge></zutat>
   <zutat><name>Creme de Cacao</name><menge>3 cl</menge></zutat>
   <zutat><name>Sahne</name><menge>3 cl</menge></zutat>
  </zutaten>
 </cocktail>
 <cocktail>
  <name>Golden Nugget</name>
  <zutaten>
   <zutat><name>Maracujanektar</name><menge>12 cl</menge></zutat>
   <zutat><name>Zitronensaft</name><menge>2 cl</menge></zutat>
   <zutat><name>Limettensirup</name><menge>2 cl</menge></zutat>
  </zutaten>
 </cocktail>
 <cocktail>
  <name>Martins wilde Party</name>
  <zutaten>
   <zutat><name>Mineralwasser (still)</name><menge>10 cl</menge></zutat>
   <zutat><name>Leitungswasser</name><menge>10 cl</menge></zutat>
  </zutaten>
 </cocktail>
</cocktails>
```

Diese Datei hat ein Wurzelelement mit dem Namen `cocktails`, die Kinder dieses Elements sind vier Elemente mit dem Namen `cocktail`. Jedes dieser Kinder hat wiederum ein Kind mit Namen `name`, und ein Kind mit dem Namen `zutaten`. Unter den `name`-Knoten befindet sich jeweils ein Textknoten, der als Text den Namen des Cocktails enthält. Die `zutaten`-Knoten enthalten jeweils mehrere Kinder mit dem Namen `zutat`, die wiederum genau ein Kind mit dem Namen `name` und ein Kind mit dem Namen `menge` enthalten, unter denen sich Textknoten mit den entsprechenden Inhalten befinden. Einen Ausschnitt der Baumdarstellung finden Sie in Abbildung 6.1. Dabei sind die Textknoten

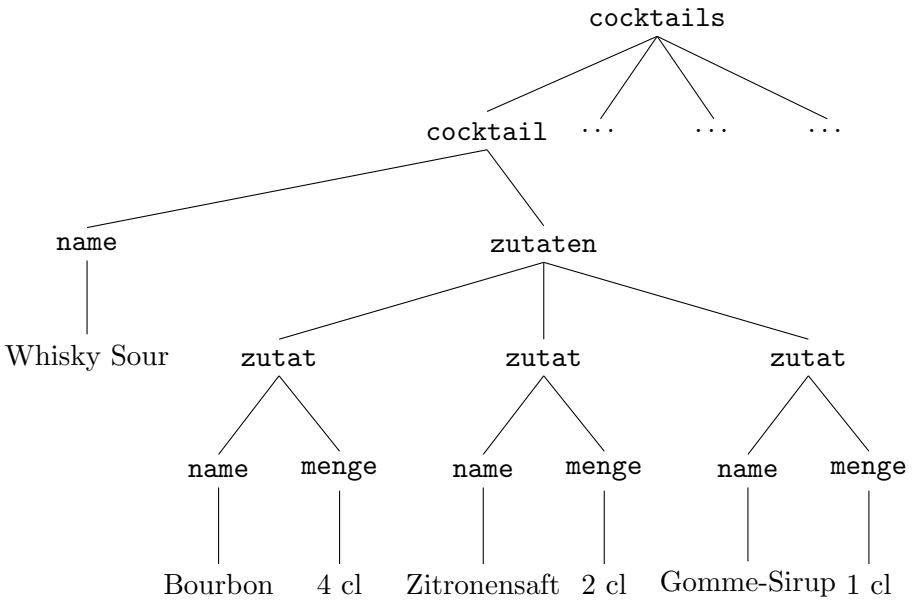


Abbildung 6.1: Ein Ausschnitt aus der Baumdarstellung der in Beispiel 6.1 angegebenen XML-Datei.

mit dem in ihnen enthaltenen Text beschriftet, bei den anderen Knoten ist jeweils der Name des Elements angegeben. ◊

Unser Ziel in diesem Abschnitt ist es vor allem, die Struktur von XML-Dateien zu beschreiben. Dabei wollen wir die eigentlichen Inhalte der Textknoten außer acht lassen und uns nur mit den Elementen betrachten. Daher repräsentieren wir von nun an Textknoten nicht mehr durch ihren Inhalt, sondern durch den Platzhalter DATA, der stellvertretend für den eigentlich Inhalt ist. Im folgenden Beispiel gehen wir noch ein wenig mehr auf Textknoten und auf die Verwendung von DATA ein.

Beispiel 6.2 Wir betrachten die folgende XML-Datei:

```
<text>
<absatz>Es war kein einfacher Regen mehr; vielmehr schien es so, als
    weinte der Himmel Sturzbäche von Tränen, die sich in der Grube, die
    an Stelle des AfE-Turms klaffte, zu einem tristen Tümpel vereinten,
    der finstersten Wolken als Spiegel diente. Auch die Herzen der
    Studierenden waren schwer, ihre Minen von Kummer getrübt. <zitat>
    Warum nur,</zitat> klagten sie, <zitat>warum trifft uns dieses
    schwere Schicksal?</zitat></absatz>
<absatz>Wahrlich, ihr Klagen war berechtigt; denn ein steter Quell der
    Freude sollte bald für lange Zeit, vielleicht für immer, versiegen:
    Mit dem Ende des Semester nahte auch, so düster wie unaufhaltsam, das
```

```
    Ende der Vorlesung <fett>Theoretische Informatik 2</fett>. Sehr
   bedauerlich.</absatz>
</text>
```

Der Baum, der dieser Datei entspricht, ist in Abbildung 6.2 aufgeführt. Dabei werden

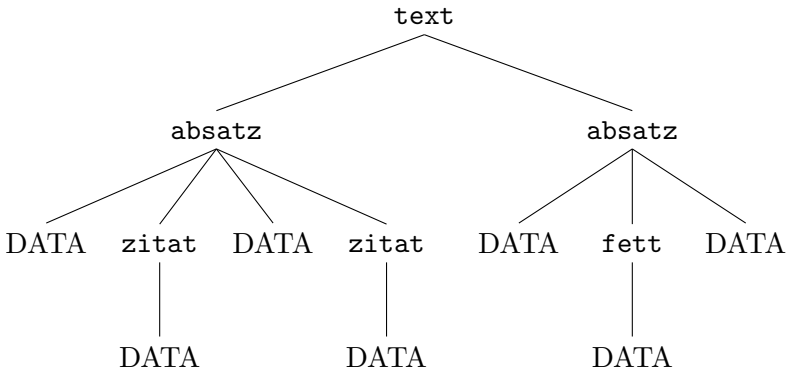


Abbildung 6.2: Der Baum, der der XML-Datei aus Beispiel 6.2 entspricht. Die Textknoten wurden hier bereits durch DATA-Knoten ersetzt. Eine Erläuterung, welcher DATA-Knoten welchem Text entspricht, befindet sich in Beispiel 6.2.

die Textknoten als DATA-Knoten dargestellt. Wir betrachten nun den linken der beiden `absatz`-Knoten. Dieser hat vier Kinder, die den folgenden Text enthalten:

- Das am weitesten links stehende Kind ist mit DATA beschriftet. Es steht für einen Textknoten, der den Text von „`Es war kein`“ bis „`getrübt.`“ enthält.
- Das nächste Kind ist mit `zitat` beschriftet. Dieses enthält direkt keinen Text, hat aber als einziges Kind einen Textknoten, der „`Warum nur,`“ enthält.
- Das dritte Kind ist wieder ein Textknoten und enthält den Text „`klagten sie,`“.
- Das vierte Kind ist mit `zitat` beschriftet und hat als ein Kind einen Textknoten mit dem Inhalt „`warum trifft uns dieses schwere Schicksal?`“

Für den rechten `absatz`-Knoten lassen sich analoge Betrachtungen anstellen. ◊

In vielen Anwendungsgebieten von XML ist es vorteilhaft, den verwendeten Dateien eine Struktur vorzuschreiben. Für eine Cocktail-Rezeptliste wie in Beispiel 6.1 kann es zum Beispiel sinnvoll sein zu fordern, dass jeder Cocktail genau einen Namen hat und genau eine Zutatenliste enthält (und so weiter). Diese Funktion übernimmt gewöhnlich ein sogenanntes *Schema*, das in einer *Schema-Sprache* verfasst wird. Im Laufe der Zeit wurden verschiedene dieser Schema-Sprachen entwickelt; die älteste und wahrscheinlich einfachste Schema-Sprache sind die *Document Type Definitions* (DTDs). Diese wurden durch *XML Schema (XSD)* verallgemeinert, eine weitere prominente Schema-Sprache ist *Relax NG*. Wir werden hier nur auf DTDs eingehen.

Beispiel 6.3 Eine DTD für die XML-Datei aus Beispiel 6.1 könnte beispielsweise wie folgt aussehen:

```
<!DOCTYPE [cocktails
  <!ELEMENT cocktails (cocktail+)>
  <!ELEMENT cocktail (name,zutaten)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT zutaten (zutat+)>
  <!ELEMENT zutat (name,menge)>
  <!ELEMENT menge (#PCDATA)>
]>
```

Dabei gibt `<!DOCTYPE [cocktails` (die *Dokumenttypdeklaration*) an, dass das Wurzelelement den Namen `cocktails` haben muss. Der Rest der DTD besteht aus *Elementtypdeklarationen*, wie zum Beispiel `<!ELEMENT cocktails (cocktail+)>`. Diese geben jeweils zu einem Element (in diesem Fall `cocktails`) an, welche Namen die Kinder des Elements haben dürfen, und in welcher Reihenfolge diese vorkommen können. Dies geschieht durch einen regulären Ausdruck, in diesem Fall `cocktail+`. Diese DTD schreibt also vor, dass Elemente mit dem Namen `cocktails` nur Kinder mit dem Namen `cocktail` haben dürfen, und zwar mindestens eins, aber beliebig viele. Diese wiederum haben jeweils genau ein Kind mit dem Namen `name`, und ein Kind mit dem Namen `zutaten`. Dabei muss `name` immer ganz links stehen.

Das Schlüsselwort `#PCDATA` dient hierbei zur Spezifikation von Textknoten. Diese DTD legt also fest, dass nur unterhalb von Elementen mit dem Namen `name` oder `menge` ein Textknoten vorkommen darf (und muss).

Außerdem betrachten wir noch mögliche DTD für die Datei aus Beispiel 6.2:

```
<!DOCTYPE [text
  <!ELEMENT text (absatz|#PCDATA)+)>
  <!ELEMENT absatz (zitat|fett|#PCDATA)+>
  <!ELEMENT zitat (#PCDATA)>
  <!ELEMENT fett (#PCDATA)>
]>
```

Da hier `absatz` sowohl Textknoten als auch andere Elemente enthält, sind die regulären Ausdrücke ein wenig komplexer. Allerdings gestattet es diese DTD zum Beispiel nicht, in einem `zitat`-Element noch `fett`-Elemente zu verwenden. Dies könnte man ändern, indem man die Elementtypdeklaration für `zitat` durch die folgende ersetzt:

```
<!ELEMENT zitat (emph|#PCDATA)+>
```

Natürlich sind auch weitere Änderungen denkbar; da keine konkrete Anwendung für diese DTD existiert sind weitere Überlegungen dazu allerdings müßig. ◊

In der Praxis kann in DTDs noch deutlich mehr definiert werden (zum Beispiel können die Elemente noch mit Attributen versehen werden); allerdings würde dies den Rahmen der Vorlesung sprengen und vom eigentlichen Kern dieses Abschnitts ablenken.

6.1.1 XML, DTDs und deterministische reguläre Ausdrücke

In diesem Abschnitt werden wir diese Sichtweise auf XML-Dateien und DTDs formalisieren. Wir beginnen mit einer Definition von XML-Bäumen:

Definition 6.4 Sei Σ ein Alphabet (das Alphabet der **Elementnamen**). Sei DATA ein Buchstabe mit DATA $\notin \Sigma$, und sei $\Sigma_D := \Sigma \cup \{\text{DATA}\}$.

Ein **XML-Baum (über Σ)** ist ein gerichteter endlicher Baum, dessen Knoten mit Buchstaben aus Σ_D beschriftet sind. Dabei sind innere Knoten mit Buchstaben aus Σ beschriftet; an Blättern sind alle Buchstaben aus Σ_D gestattet.

Der in Abbildung 6.2 dargestellte Baum zu der Datei aus Beispiel 6.2 ist ein XML-Baum über dem Alphabet $\Sigma := \{\texttt{text}, \texttt{absatz}, \texttt{zitat}, \texttt{fett}\}$. Wenn wir in Abbildung 6.1 in dem Baum zu Beispiel 6.1 alle Textknoten durch DATA ersetzen, erhalten wir einen XML-Baum über dem Alphabet

$$\Sigma := \{\texttt{cocktails}, \texttt{cocktail}, \texttt{name}, \texttt{zutaten}, \texttt{zutat}, \texttt{menge}\}.$$

Wir sind nun bereit, DTDs zu definieren. Dabei beginnen wir mit einer Definition, die allgemeiner ist als die in der Praxis gebräuchliche. Wir werden diese daher später noch geeignet einschränken.

Definition 6.5 Sei Σ ein Alphabet (das Alphabet der Elementnamen). Sei DATA ein Buchstabe mit DATA $\notin \Sigma$, und sei $\Sigma_D := (\Sigma \cup \text{DATA})$.

Eine **verallgemeinerte DTD (über Σ)** wird definiert durch

- ein **Wurzelelement** $\rho \in \Sigma$, und
- eine Funktion τ, die jedem Elementnamen $a \in \Sigma$ einen regulären Ausdruck $\tau(a)$ über Σ_D zuordnet.

Wir schreiben dies als (Σ, ρ, τ) und bezeichnen τ als **Elementtypfunktion**.

Eine verallgemeinerte DTD legt also nichts weiter fest als ein Alphabet Σ von möglichen Elementnamen, den Namen des Wurzelelements, und zu jedem Elementnamen $a \in \Sigma$ einen Elementyp $\tau(a)$.

Definition 6.6 Sei Σ ein Alphabet, sei t ein XML-Baum über Σ und sei $D = (\Sigma, \rho, \tau)$ eine verallgemeinerte DTD. Der Baum T ist **gültig (unter der DTD D)**, wenn er folgende Bedingungen erfüllt:

- Die Wurzel von T ist mit ρ beschriftet.
- Für jeden Knoten v von T seien $a_1, a_2 \ldots, a_n \in \Sigma$ mit $n \in \mathbb{N}$ die Namen der Wurzeln der Kinder von v, wobei a_1 für das am weitesten links stehende Kind

steht, a_2 für das nächste Kind, und so weiter, bis zum am weitesten rechts stehenden Kind a_n. Dann gilt: $a_1 a_2 \cdots a_n \in \mathcal{L}(\tau(b))$, wobei b der Name der Beschriftung von v ist[84].

Wir schreiben dies als $T \models D$. Jede DTD $D = (\Sigma, \rho, \tau)$ definiert eine Baumsprache

$$\mathcal{L}(D) := \{T \text{ ist ein XML-Baum über } \Sigma \mid T \models D\}.$$

Während wir bisher immer mit Sprachen zu tun hatten, die Mengen aus Wörtern sind, definieren wir durch DTDs Mengen von Bäumen. Ähnlich wie bei „Wortsprachen“ wurde zu Baumsprachen eine umfangreiche Theorie entwickelt, auf die wir hier leider nur am Rande eingehen können.

Notation 6.7 Bei der Definition einer DTD (Σ, ρ, τ) verwenden wir bei der Definition von τ eine Schreibweise mit Pfeilen: Anstelle von $\tau(a) := \alpha$ (wobei $a \in \Sigma$ gilt, und α ein regulärer Ausdruck über Σ_D ist), schreiben wir $a \to \alpha$.

Beispiel 6.8 Sei $\Sigma := \{\texttt{text}, \texttt{absatz}, \texttt{zitat}, \texttt{fett}\}$, und sei $D_1 := (\Sigma, \texttt{text}, \tau_1)$ eine verallgemeinerte DTD, wobei τ_1 wie folgt definiert sei:

$$\begin{aligned} \texttt{text} &\to \texttt{absatz}^+, \\ \texttt{absatz} &\to (\texttt{zitat} \mid \texttt{fett} \mid \text{DATA})^*, \\ \texttt{zitat} &\to \text{DATA} \\ \texttt{fett} &\to \text{DATA}\,. \end{aligned}$$

Sei T der in Abbildung 6.2 dargestellte Baum zu der Datei aus Beispiel 6.2. Um zu überprüfen, ob $T \models D_1$, werfen wir zuerst einen Blick auf die Wurzel von T. Diese ist mit `text` beschriftet, wie es von D_1 gefordert wird. Nun müssen wir jeden Knoten von T betrachten. Wir beginnen mit der Wurzel. Diese hat zwei Kinder, die jeweils mit `absatz` beschriftet sind. Es gilt:

$$\begin{aligned} \texttt{absatz} \cdot \texttt{absatz} \in &\mathcal{L}(\tau_1(\texttt{text})) \\ &= \mathcal{L}(\texttt{absatz}^+). \end{aligned}$$

Wir werden nun nicht jeden der Knoten in diesem Beispiel explizit überprüfen; stattdessen betrachten wir nur noch den linken der beiden `absatz`-Knoten. Dessen Kinder bilden das Wort

$$\begin{aligned} \text{DATA} \cdot \texttt{zitat} \cdot \text{DATA} \cdot \texttt{zitat} \in &\mathcal{L}((\texttt{zitat} \mid \texttt{fett} \mid \text{DATA})^*) \\ &= \mathcal{L}(\tau_1(\texttt{absatz})). \end{aligned}$$

Durch Betrachtung aller Knoten von T stellen wir fest, dass jedes der Wörter in der Sprache des entsprechenden regulären Ausdrucks liegt. Es gilt also $T \models D_1$.

[84] Hat v keine Kinder, ist also $n = 0$, so ist $a_1 \cdots a_n = \varepsilon$. In diesem Fall muss also $\varepsilon \in \mathcal{L}(\tau(b))$ gelten.

Eine andere mögliche verallgemeinerte DTD wäre zum Beispiel $D_2 := (\Sigma, \rho, \tau_2)$, wobei τ_2 wie folgt definiert ist:

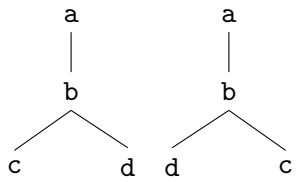
$$\begin{aligned} \texttt{text} &\to \texttt{absatz}^+ \mid \texttt{zitat}, \\ \texttt{absatz} &\to (\texttt{zitat} \mid \texttt{fett} \mid \text{DATA})^*, \\ \texttt{zitat} &\to (\texttt{fett} \mid \text{DATA})^+ \\ \texttt{fett} &\to \text{DATA}\,. \end{aligned}$$

Bei dieser DTD darf ein Text auch genau ein Element mit dem Namen `zitat` enthalten, statt wie bei D_1 eine Folge von `absatz`-Elementen enthalten zu müssen. Zusätzlich ist es erlaubt, in einem `zitat`-Element auch `fett`-Elemente zu verwenden. Wir beobachten, dass $\mathcal{L}(\tau_1(a)) \subseteq \mathcal{L}(\tau_2(a))$ für alle $a \in \Sigma$ gilt. Aus $T \models D_1$ folgt also $T \models D_2$, und dies gilt nicht nur für unseren Beispielbaum T, sondern für alle XML-Bäume über Σ. Daher ist $\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$. Sei nun T' der folgende XML-Baum über Σ:

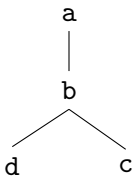
Wie Sie leicht sehen können, gilt $T' \models D_2$, während $T' \not\models D_1$. Also demonstriert T', dass $\mathcal{L}(D_1) \neq \mathcal{L}(D_2)$ und somit $\mathcal{L}(D_1) \subset \mathcal{L}(D_2)$. ◊

Verallgemeinerte DTDs haben einige Nachteile. Dabei fällt besonders ins Gewicht, dass sie **lokal** sind, denn sie können an den Knoten eines XML-Baumes immer nur auf dessen Kinder zugreifen. Die Entscheidung, ob das aus den Namen der Kindern eines Knoten gebildete Wort zulässig ist, kann nicht von anderen Knoten des Baumes beeinflusst werden. Dadurch können selbst einfache endliche Baumsprachen nicht anhand einer verallgemeinerten DTD ausgedrückt werden, wie die beiden folgenden Beispiel zeigen:

Beispiel 6.9 Sei $\Sigma := \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$. Die Baumsprache L bestehe aus genau den beiden folgenden XML-Bäumen über Σ:



Angenommen, es existiert eine verallgemeinerte DTD $D = (\Sigma, \rho, \tau)$ mit $L = \mathcal{L}(D)$. Wegen des linken Baums muss $\texttt{cd} \in \mathcal{L}(\tau_1(\texttt{b}))$ gelten, und wegen des rechten Baums $\texttt{dc} \in \mathcal{L}(\tau_1(\texttt{b}))$. Also enthält $\mathcal{L}(D)$ auch den folgenden Baum:

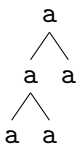


Somit ist $L \neq \mathcal{L}(D)$, Widerspruch. ◊

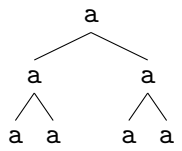
Beispiel 6.10 Sei $\Sigma := \mathtt{a}\}$. Wir betrachten die Baumsprache

$$L := \{T \text{ ist ein XML-Baum über } \Sigma \mid T \text{ hat genau 5 Knoten}\}.$$

Dann enthält L unter anderem den folgenden Baum T:



Offensichtlich ist T ein XML-Baum über Σ, der genau 5 Knoten hat. Es gilt also $T \in L$. Auch hier können wir ein Vertauschungsargument anwenden: Angenommen, es existiert eine verallgemeinerte DTD $D = (\Sigma, \rho, \tau)$ mit $\mathcal{L}(\tau) = L$. Wir betrachten nun das linke Kind der Wurzel: Dessen Kinder bilden das Wort $\mathtt{aa}$; es gilt also $\mathtt{aa} \in \mathcal{L}(\tau(\mathtt{a}))$. Somit muss auch der folgende Baum T' gültig sein:



Da T' insgesamt sieben Knoten hat, gilt $T' \notin L$, und somit $\mathcal{L}(D) \neq L$. Widerspruch. ◊

Aus Sicht der häufigsten Anwendungsfälle von XML kann Beispiel 6.10 als übertrieben künstlich betrachtet werden: Gewöhnlich hat Σ weit mehr als ein Element, und selbst in Anwendungen, die ihre Schemata durch mächtigere Sprachen als DTDs definieren, sind die Bäume strukturierter.

Wir betrachten nun einige Entscheidungsprobleme für verallgemeinerte DTDs, die aus theoretischer Sicht natürlich und außerdem leicht durch die Praxis motivierbar sind:

Wortproblem für verallgemeinerte DTDs

Eingabe: Eine verallgemeinerte DTD D und ein XML-Baum T über Σ.
Frage: Ist $T \in \mathcal{L}(D)$?

Universalitätsproblem für verallgemeinerte DTDs

Eingabe: Eine verallgemeinerte DTD $D = (\Sigma, \rho, \tau)$.
Frage: Ist $\mathcal{L}(D) = \{T \mid T \text{ ist ein XML-Baum über } \Sigma \text{ mit Wurzel } \rho\}$?

Inklusionsproblem für verallgemeinerte DTDs

Eingabe: Verallgemeinerte DTDs D_1, D_2 über einem Alphabet Σ.
Frage: Ist $\mathcal{L}(D_1) \subseteq \mathcal{L}(D_2)$?

Da $T \in \mathcal{L}(D)$ genau dann gilt, wenn $T \models D$, kann das Wortproblem auch als **Gültigkeitsproblem** interpretiert werden. Dieses Problem wird oft auch als **Validierungsproblem** bezeichnet. Validierung und ist einer der Hauptgründe, warum XML-Schemata überhaupt verwendet werden, nämlich um zu entscheiden, ob XML-Dokumente gültig sind.

Das Universalitätsproblem kann verwendet werden, um eine erstellte DTD zu überprüfen: Im Prinzip ist es ja erlaubt, eine DTD zu erstellen, unter der jeder XML-Baum über einem bestimmten Alphabet gültig ist. Für die meisten Anwendungen dürft dies allerdings nicht sonderlich hilfreich sein. Außerdem sind untere Schranken für das Universalitätsproblem sofort auch untere Schranken für das Inklusionsproblem.

Das Inklusionsproblem wiederum kann verwendet werden, um DTDs zu vergleichen. Stellen Sie sich vor, Sie haben eine existierende DTD umgeschrieben und an einigen Stellen optimiert. Nun müssen Sie jemand anders überzeugen, dass Ihre neue DTD immer noch die gleiche Sprache definiert und dass Sie nicht versehentlich einen Fehler eingeführt haben. Außerdem kann das Inklusionsproblem beim Lernen vom DTDs verwendet werden.

Da verallgemeinerte DTDs über reguläre Ausdrücke definiert sind, können wir die unteren Schranken für das Universalitätsproblem, die wir in Abschnitt 3.4 diskutiert haben, hier gleich anwenden. Das Universalitätsproblem für verallgemeinerte DTDs ist also PSPACE-hart; und diese untere Schranke gilt ebenso für das Inklusionsproblem.

Zusätzlich zu diesen Schwierigkeiten gibt es noch einen weiteren Grund, anstelle der vollen Klasse der verallgemeinerten DTDs eine eingeschränkte Unterklasse zu betrachten: Selbst wenn das Wortproblem für verallgemeinerte DTDs in Polynomialzeit ausgewertet werden kann, da es sich durch vielfaches Anwenden des Wortproblems für reguläre Ausdrücke lösen lässt, ist dies für manche praktischen Anwendungen nicht schnell genug.

Die in der Praxis verwendeten (und im XML-Standard definierten) DTDs werden daher eingeschränkt, und zwar indem nicht mehr beliebige reguläre Ausdrücke zur Verfügung stehen. Stattdessen werden sogenannte *deterministische reguläre Ausdrücke* verwendet. Um diese zu definieren, benötigen wir noch ein zusätzliches Werkzeug:

Definition 6.11 Sei Σ ein Alphabet. Für jedes $n \in \mathbb{N}_{>0}$ sei die **n-Indizierung von** Σ definiert als $\Sigma_{(n)} := \{a_i \mid 1 \leq i \leq n\}$. Sei α ein regulärer Ausdruck über Σ. Die **Indizierung von** α, geschrieben $\widetilde{\alpha}$, definieren wir als den regulären Ausdruck der entsteht, wenn in α für jedes $a \in \Sigma$ und jedes $1 \leq i \leq |\alpha|_a$ das von links aus i-te Vorkommen von a durch a_i ersetzt wird.

Beispiel 6.12 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $\alpha := (\mathtt{aba})^* \mid ((\mathtt{bc})^*\mathtt{a})$. Für die Indizierung von α gilt $\widetilde{\alpha} = (\mathtt{a}_1\mathtt{b}_2\mathtt{a}_2)^* \mid ((\mathtt{b}_2\mathtt{c}_1)^*\mathtt{a}_3)$, und $\widetilde{\alpha}$ ist ein regulärer Ausdruck über dem Alphabet[85] $\Sigma_{(3)} = \{\mathtt{a}_1, \mathtt{a}_2, \mathtt{a}_3, \mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \mathtt{c}_1, \mathtt{c}_2, \mathtt{c}_3\}$. ◊

Es ist leicht zu sehen, dass für den Homomorphismus h, der für alle $a \in \Sigma$ und alle $i \in \mathbb{N}_{>0}$ durch $h(a_i) := a$ definiert ist, $h(\widetilde{\alpha}) = \alpha$ gilt. Da $\widetilde{\alpha}$ ein regulärer Ausdruck ist, kann daraus wie gewohnt eine Sprache $\mathcal{L}(\widetilde{\alpha})$ erzeugt werden. Diese ist eine Verfeinerung von $\mathcal{L}(\alpha)$, die zusätzlich noch die Information enthält, welcher Buchstabe aus welcher Stelle des regulären Ausdrucks erzeugt wurde. Diese Beobachtung wird bei der Definition von deterministischen regulären Ausdrücken ausgenutzt:

Definition 6.13 Sei Σ ein Alphabet. Ein regulärer Ausdruck α über Σ ist ein **deterministischer regulärer Ausdruck**, wenn für alle Wörter $x, y, z \in \Sigma_{(n)}^*$ und alle $a_i, a_j \in \Sigma_{(n)}$ gilt:

$$\text{Aus } x \cdot a_i \cdot y \in \mathcal{L}(\widetilde{\alpha}) \text{ und } x \cdot a_j \cdot z \in \mathcal{L}(\widetilde{\alpha}) \text{ folgt } i = j.$$

Hierbei sei $n := \max\{|\alpha|_a \mid a \in \Sigma\}$.

Mit anderen Worten: Ist α ein deterministischer regulärer Ausdruck, so können wir jedes Wort $w \in \mathcal{L}(\alpha)$ buchstabenweise von links nach rechts abarbeiten, und dabei in jedem Schritt eindeutig feststellen, welches Terminal in α den aktuellen Buchstaben von w erzeugt. Die Wörter aus $\mathcal{L}(\alpha)$ werden also nicht nur eindeutig aus α erzeugt, sondern sie können auch direkt beim Lesen von links nach rechts zugeordnet werden.

Beispiel 6.14 Wir betrachten zuerst den regulären Ausdruck $\alpha := (\mathtt{a} \mid \mathtt{b})^*\mathtt{b}$ und seine Indizierung $\widetilde{\alpha} = (\mathtt{a}_1 \mid \mathtt{b}_1)^*\mathtt{b}_2$. Es gilt: $\mathtt{b}_1\mathtt{b}_2 \in \mathcal{L}(\widetilde{\alpha})$ und $\mathtt{b}_2 \in \mathcal{L}(\widetilde{\alpha})$, also ist α kein deterministischer regulärer Ausdruck (wir wählen hierfür $x := z := \varepsilon$ und $y := \mathtt{b}_2$). Allerdings ist $\mathcal{L}(\alpha)$ durch einen deterministischen regulären Ausdruck definierbar: Sei $\alpha' := (\mathtt{a}^*\mathtt{b})^+$. Dann ist $\mathcal{L}(\alpha') = \mathcal{L}(\alpha)$ (dies können Sie als Übung überprüfen). Dass α' deterministisch ist, folgt direkt aus der Tatsache, dass kein Terminal in α' öfter als einmal vorkommt. Wir verallgemeinern diese Beobachtung in Lemma 6.15 weiter unten.

Sei $\beta := (\mathtt{ab})^+(\mathtt{ab})$. Auch dieser reguläre Ausdruck ist nicht deterministisch. Betrachten wir dazu die Indizierung $\widetilde{\beta} = (\mathtt{a}_1\mathtt{b}_1)^+(\mathtt{a}_2\mathtt{b}_2)$. Es gilt:

$$\mathtt{a}_1\mathtt{b}_1\mathtt{a}_2\mathtt{b}_2 \in \mathcal{L}(\widetilde{\beta}),$$

[85] Wir verwenden nicht alle Buchstaben aus $\Sigma_{(3)}$ in $\widetilde{\alpha}$, aber dieses größere Alphabet ist leichter zu definieren.

$$\mathtt{a}_1\mathtt{b}_1\mathtt{a}_1\mathtt{b}_1\mathtt{a}_2\mathtt{b}_2 \in \mathcal{L}(\widetilde{\beta}).$$

Wir wählen $x := \mathtt{a}_1\mathtt{b}_1$, $y := \mathtt{b}_2$ und $z = \mathtt{b}_1\mathtt{a}_2\mathtt{b}_2$, und erhalten so durch

$$x\mathtt{a}_2 y \in \mathcal{L}(\widetilde{\beta}), \qquad\qquad x\mathtt{a}_1 z \in \mathcal{L}(\widetilde{\beta})$$

einen Verstoß gegen die Kriterien für deterministische reguläre Ausdrücke. Aber auch hier existiert ein äquivalenter deterministischer regulärer Ausdruck, nämlich $\beta' := (\mathtt{ab})(\mathtt{ab})^+$. Auch hier kann man leicht zeigen, dass β' deterministisch ist, denn abgesehen von den ersten beiden Zeichen bestehen alle Wörter aus $\mathcal{L}(\widetilde{\beta'})$ nur aus den Buchstaben $\mathtt{a}_2$ und $\mathtt{b}_2$. ◊

Es ist vergleichsweise einfach, zu entscheiden, ob ein regulärer Ausdruck deterministisch ist oder nicht (wir werden dazu später geeignete Werkzeug entwickeln). Das folgende hinreichende Kriterium kann allerdings schon jetzt bewiesen werden:

Lemma 6.15 *Sei α ein regulärer Ausdruck über einem Alphabet Σ. Angenommen, jeder Buchstabe $a \in \Sigma$ kommt höchstens einmal in α vor. Dann ist α ein deterministischer regulärer Ausdruck.*

Beweis: Für jeden Buchstaben $a \in \Sigma$, der in α vorkommt, enthält $\widetilde{\alpha}$ nur den Buchstaben a_1 (da es kein zweites Vorkommen von a gibt, kann kein a_2 existieren). Also folgt aus $xa_iy \in \mathcal{L}(\widetilde{\alpha})$ und $xa_jz \in \mathcal{L}(\widetilde{\alpha})$ stets $i = 1$ und $j = 1$ und somit $i = j$ für alle Wörter x, y, z und alle indizierten Buchstaben a_i, a_j. □

Auf den ersten Blick mag diese Einschränkung zu stark wirken. Allerdings hat sich herausgestellt, dass viele DTDs in der Praxis solche *single occurrence regular expressions* verwenden (die regulären Ausdrücke in den bisher betrachteten Beispielen erfüllen auch dieses Kriterium). Nicht jede reguläre Sprache lässt sich durch einen deterministischen regulären Ausdruck definieren:

Beispiel 6.16 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}\}$ und $\alpha := (\mathtt{a} \mid \mathtt{b})^*\mathtt{a}(\mathtt{a} \mid \mathtt{b})$. Dann existiert kein deterministischer regulärer Ausdruck β mit $\mathcal{L}(\beta) = \mathcal{L}(\alpha)$. Leider ist dies weder einfach zu beweisen, noch einfach zu veranschaulichen[86]. ◊

Die Frage, ob eine durch einen beliebigen regulären Ausdruck α definierte Sprache $\mathcal{L}(\alpha)$ auch durch einen deterministischen regulären Ausdruck definiert werden kann, ist leider PSPACE-vollständig. Wir haben nun alles, was wir benötigen, um DTDs zu definieren:

Definition 6.17 Sei Σ ein Alphabet. Eine verallgemeinerte DTD $D = (\Sigma, \rho, \tau)$ ist eine **deterministische DTD** (oder auch nur **DTD**), wenn $\tau(a)$ für jedes $a \in \Sigma$ ein deterministischer regulärer Ausdruck ist.

Ein wichtiger Vorteil von deterministischen regulären Ausdrücken ist, dass diese direkt in DFAs umgewandelt werden können. Wir werden uns die entsprechende Konstruktion im nächsten Abschnitt genauer ansehen.

[86] Hier muss ich bei Gelegenheit noch einen Literaturverweis nachtragen.

6.1.2 Glushkov-Automaten

Zuerst betrachten wir eine alternative Verfahrensweise, um reguläre Ausdrücke in NFAs umzuwandeln. Im Gegensatz zu dem Ansatz, den wir in Abschnitt 3.3.1 betrachtet haben, konstruieren wir hier direkt NFAs ohne ε-Übergänge. Die so konstruierten Automaten werden auch als *Glushkov-Automaten*[87] bezeichnet. Wir beginnen mit der formalen Definition; im Anschluss daran wird das Verfahren noch genauer erläutert.

Definition 6.18 Sei Σ ein Alphabet und α ein regulärer Ausdruck über Σ. Dann ist der **Glushkov-Automat von** α ein NFA, der auf folgende Art rekursiv definiert wird:

1. $A_{\mathcal{G}}(\emptyset) := (\Sigma, \{q_0\}, \delta, q_0, \emptyset)$, und $\delta(q_0, a) := \emptyset$ für alle $a \in \Sigma$,
2. $A_{\mathcal{G}}(\varepsilon) := (\Sigma, \{q_0\}, \delta, q_0, \{q_0\})$, und $\delta(q_0, a) := \emptyset$ für alle $a \in \Sigma$,
3. Für jeden Buchstabe $a \in \Sigma$ ist $A_{\mathcal{G}}(a) := (\Sigma, \{q_0, q_1\}, \delta, q_0, \{q_1\})$, wobei für alle $q \in \{q_0, q_1\}$ und alle $b \in \Sigma$

$$\delta(q, b) := \begin{cases} \{q_1\} & \text{falls } q = q_0 \text{ und } b = a, \\ \emptyset & \text{sonst.} \end{cases}$$

4. Seien, β_1 und β_2 reguläre Ausdrücke mit

$$A_{\mathcal{G}}(\beta_1) = (\Sigma, Q_1, \delta_1, q_{0,1}, F_1),$$
$$A_{\mathcal{G}}(\beta_1) = (\Sigma, Q_2, \delta_2, q_{0,2}, F_2).$$

   Dann sei durch Umbenennung der Zustände sicher gestellt, dass $(Q_1 \cap Q_2) = \emptyset$. Dann definieren wir:

   a) $A_{\mathcal{G}}((\beta_1 \cdot \beta_2)) := (\Sigma, Q, \delta, q_{0,1}, F)$, wobei

$$Q := (Q_1 \cup Q_2) - \{q_{0,2}\},$$
$$F := \begin{cases} F_2 & \text{falls } q_{0,2} \notin F_2, \\ F_1 \cup (F_2 - \{q_{0,2}\}) & \text{falls } q_{0,2} \in F_2, \end{cases}$$

[87] Benannt nach Victor Mikhailovich Glushkov. Das folgende Gerücht über die Lomonossow-Universität hat nicht wirklich mit Glushkov zu tun, aber die Tatsache, dass jener dort promoviert wurde, soll mir als Anlass dafür genügen. Ein russischer Informatiker, der dort studiert hat, hat mir Folgendes berichtet: Das Hauptgebäude der Lomonossow-Universität hat einen besonders großen Turm, der von 1953 bis zum Bau des Frankfurter Messeturms 1990 das höchste Gebäude von Europa war (dieser Teil des Gerüchts ist leicht zu überprüfen). Allerdings ist der Boden unter diesem Turm so weich, dass er durch riesige Kühlanlagen permanent tiefgekühlt werden muss, weil der Turm sonst versinken würde.

außerdem sei für alle $q \in Q$ und alle $a \in \Sigma$

$$\delta(q,a) := \begin{cases} \delta_1(q,a) & \text{falls } q \in (Q_1 - \{F_1\}), \\ \delta_1(q,a) \cup \delta_2(q_{0,2},a) & \text{falls } q \in F_1, \\ \delta_2(q,a) & \text{falls } q \in Q_2. \end{cases}$$

b) $A_{\mathcal{G}}((\beta_1 \mid \beta_2)) := (\Sigma, Q, \delta, q_{0,1}, F)$, wobei

$$Q := (Q_1 \cup Q_2) - \{q_{0,2}\},$$

$$F := \begin{cases} F_1 \cup F_2 & \text{falls } q_{0,2} \notin F_2, \\ F_1 \cup (F_2 - \{q_{0,2}\}) \cup \{q_{0,1}\} & \text{falls } q_{0,2} \in F_2, \end{cases}$$

außerdem sei für alle $q \in Q$ und alle $a \in \Sigma$

$$\delta(q,a) := \begin{cases} \delta_1(q,a) \cup \delta_2(q,a) & \text{falls } q = \{q_{0,1}\}, \\ \delta_1(q,a) & \text{falls } q \in Q_1 - \{q_{0,1}\}, \\ \delta_2(q,a) & \text{falls } q \in Q_2. \end{cases}$$

c) $A_{\mathcal{G}}(\beta_1^*) := (\Sigma, Q, \delta, q_{0,1}, F)$, wobei

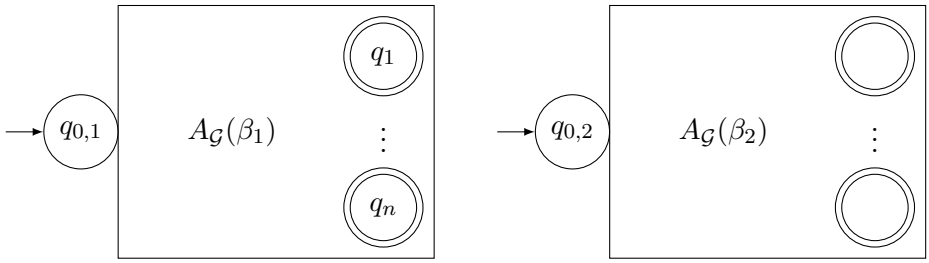
$$Q := Q_1,$$
$$F := F_1 \cup \{q_{0,1}\},$$

außerdem sei für alle $q \in Q$ und alle $a \in \Sigma$

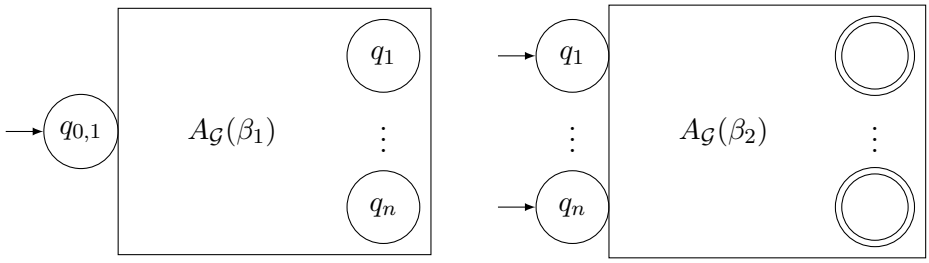
$$\delta(q,a) := \begin{cases} \delta_1(q,a) & \text{falls } q \in (Q - F_1), \\ \delta_1(q,a) \cup \delta_1(q_{0,1},a) & \text{falls } q \in F_1. \end{cases}$$

Die Automaten $A_{\mathcal{G}}(\emptyset)$, $A_{\mathcal{G}}(\varepsilon)$ und $A_{\mathcal{G}}(a)$ sind selbsterklärend. Die anderen Fälle betrachten wir nun Schritt für Schritt.

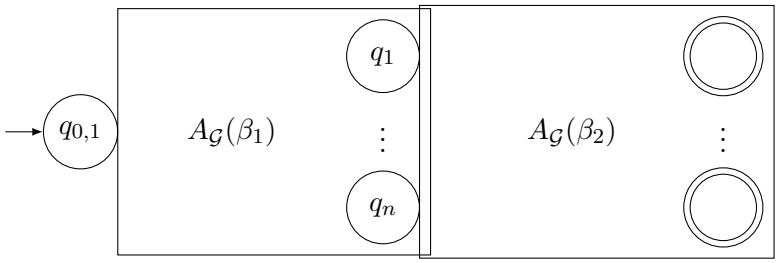
Konkatenation Anschaulich gesprochen wird $q_{0,2}$, der Startzustand von $A_{\mathcal{G}}(\beta_2)$, mit jedem akzeptierenden Zustand von $A_{\mathcal{G}}(\beta_1)$ verschmolzen. Die akzeptierenden Zustände von $A_{\mathcal{G}}((\beta_1 \cdot \beta_2))$ sind die akzeptierenden Zustände von $A_{\mathcal{G}}(\beta_2)$. Graphisch lässt sich dies folgendermaßen skizzieren. Angenommen, $A_{\mathcal{G}}(\beta_1)$ und $A_{\mathcal{G}}(\beta_2)$ sehen wie folgt aus:



Zuerst machen wir alle Zustände von F_1 zu nicht-akzeptierenden Zuständen. Außerdem vervielfachen wir $q_{0,2}$, bis wir für jeden Zustand von F_1 eine Kopie von $q_{0,2}$ erstellt haben:

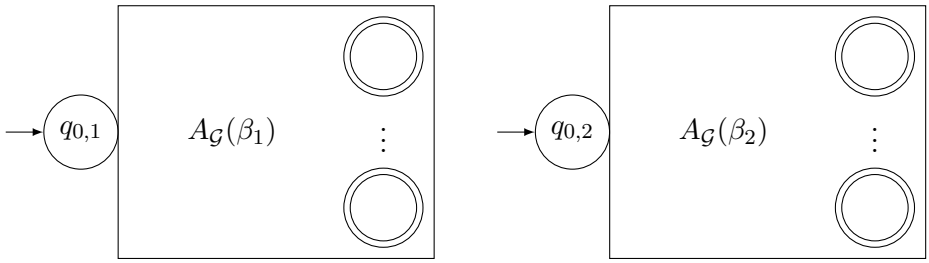


Im letzten Schritt verschmelzen wir jeden Zustande aus F_1 mit einer der Kopien von $q_{0,2}$:

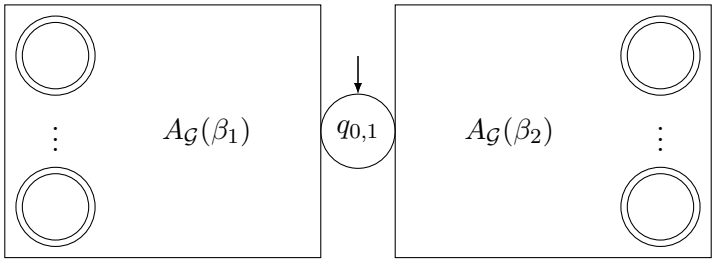


Von den Zuständen aus F_1 kann nun also zusätzlich weitergerechnet werden, als wäre der NFA in $q_{0,2}$. Abschließend muss noch der Spezialfall betrachtet werden, dass $q_{0,2}$ ein akzeptierenden Zustand ist (d. h., es gilt $\varepsilon \in \mathcal{L}(\beta_1)$). In diesem Fall werden die Zustände aus F_1 zu den akzeptierenden Zuständen hinzugenommen (dies gilt insbesondere auch, falls $q_{0,1} \in F$).

Vereinigung Anschaulich gesprochen werden hier die beiden Startzustände $q_{0,1}$ und $q_{0,2}$ miteinander verschmolzen. Wir betrachten dies wieder anhand von graphischen Darstellungen. Angenommen, $A_{\mathcal{G}}(\beta_1)$ und $A_{\mathcal{G}}(\beta_2)$ lassen sich wie folgt darstellen:

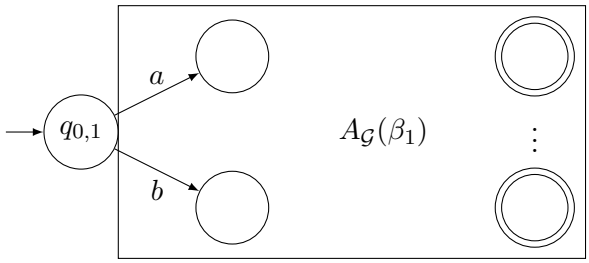


Dann erhalten wir durch Verschmelzen von $q_{0,1}$ und $q_{0,2}$ den folgenden NFA:

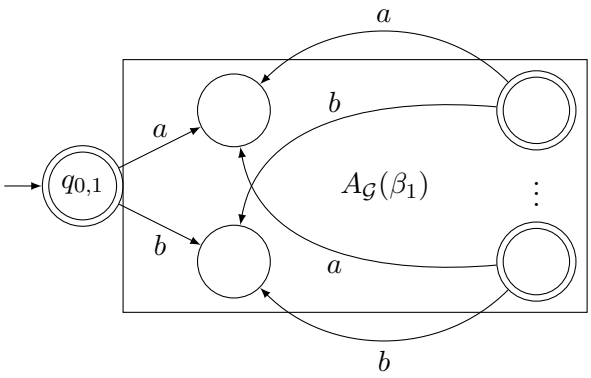


Auch hier muss wieder der Spezialfall $q_{0,2} \in F_2$ behandelt werden (indem der verschmolzene gemeinsame Startzustand gegebenenfalls zu einem akzeptierenden Zustand gemacht wird).

Kleene-Stern In diesem Fall wird der Glushkov-Automat $A_{\mathcal{G}}(\beta_1^*)$ konstruiert, indem die akzeptierenden Zustände von $A_{\mathcal{G}}(\beta_1)$ zusätzlich ihren bereits vorhandenen Kanten noch die gleichen Kanten wie $q_{0,1}$ erhalten. Angenommen, $A_{\mathcal{G}}(\beta_1)$ hat die folgende graphische Darstellung:



Nun übernimmt jeder akzeptierende Zustand alle aus $q_{0,1}$ ausgehenden Kanten:



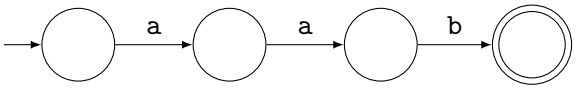
Zusätzlich dazu wird $q_{0,1}$ zu einem akzeptierenden Zustand gemacht (wenn wir $q_{0,1}$ nicht zu einem akzeptierenden Zustand machen, können wir so den Glushkov-Automaten für β_1^+ erzeugen[88]). Wie leicht zu sehen ist, erzeugt $A_{\mathcal{G}}(\alpha)$ die gleiche Sprache wie α:

Satz 6.19 *Für jeden regulären Ausdruck α ist $\mathcal{L}(\alpha) = \mathcal{L}(A_{\mathcal{G}}(\alpha))$.*

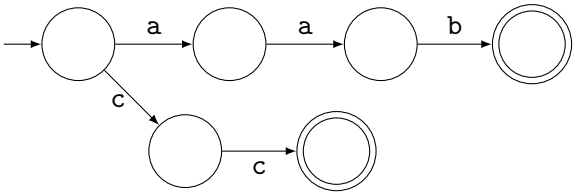
Beweisidee: Dies lässt sich leicht durch eine Induktion über den Aufbau von α zeigen. □

Beispiel 6.20 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Wir betrachten zuerst den regulären Ausdruck $\alpha_1 := \mathtt{aab}$. Dieser hat den folgenden Glushkov-Automaten $A_{\mathcal{G}}(\alpha_1)$:

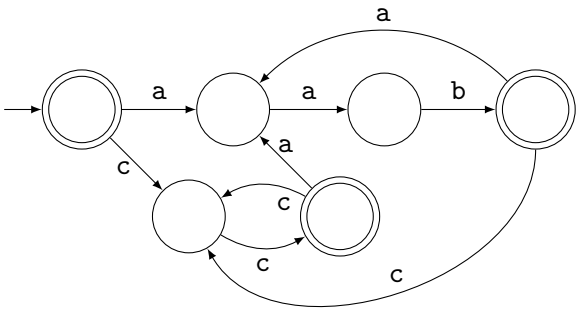
[88] Streng führt diese Abkürzung dazu, dass der Glushkov-Automat eines regulären Ausdrucks nicht mehr eindeutig definiert ist, denn eigentlich ist α^+ nur eine Kurzschreibweise für $\alpha\alpha^*$. Wenn wir diese Konstruktion für $A_{\mathcal{G}}(\alpha^+)$ einführen, müssen wir eigentlich die Definition der regulären Ausdrücke um die Verwendung von Kleene-Plus erweitern, oder wir sprechen von „*einem* Glushkov-Automaten für α“, anstelle von „*dem* Glushkov-Automaten für α“.



Als nächstes wenden wir uns dem Ausdruck $\alpha_2 := (\texttt{aab} \mid \texttt{cc})$ zu, und betrachten $A_{\mathcal{G}}(\alpha_2)$:



Nun sei $\alpha_2 := (\texttt{aab} \mid \texttt{cc})^*$. Der NFA $A_{\mathcal{G}}(\alpha_3)$ hat die folgende graphische Darstellung:



$\Diamond$

Hinweis 6.21 Wenn Sie einen regulären Ausdruck α in einen ε-NFA für $\mathcal{L}(\alpha)$ umwandeln sollen, können Sie anstelle der Konstruktion aus Abschnitt 3.3.1 auch $A_{\mathcal{G}}(\alpha)$ berechnen. Da jeder NFA auch ein ε-NFA ist, ist $A_{\mathcal{G}}(\alpha)$ eine korrekte Lösung.

Glushkov-Automaten haben einige erwähnenswerte Eigenschaften:

Lemma 6.22 *Sei α ein regulärer Ausdruck. Dann gilt:*

1. *Der Startzustand von $A_{\mathcal{G}}(\alpha)$ hat keine eingehenden Kanten.*
2. *Wenn α (für $n \in \mathbb{N}$) n Vorkommen von Terminalsymbolen enthält, dann hat $A_{\mathcal{G}}(\alpha)$ insgesamt $n+1$ Zustände.*
3. *Für jeden Zustand in $A_{\mathcal{G}}(\alpha)$ sind alle eingehenden Kanten mit dem gleichen Terminal beschriftet.*
4. *Kein Zustand von $A_{\mathcal{G}}(\alpha)$ ist eine Sackgasse.*

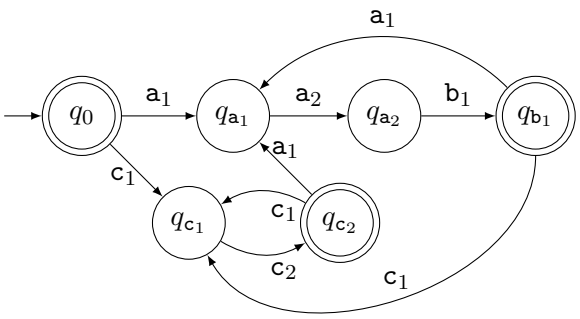
Beweis: Jede der vier Behauptungen folgt sofort aus den Konstruktionsvorschriften (zusätzlich lässt sich dies jeweils durch Induktion über den Aufbau von α zeigen). $\square$

Zusätzlich zu diesen Beobachtungen können wir außerdem beobachten, dass jedem Terminal von α genau ein Zustand von $A_{\mathcal{G}}(\alpha)$ entspricht (und abgesehen vom Startzustand gilt diese Beobachtung auch in umgekehrter Richtung). Wir übertragen diese Beobachtung nun auf Beispiel 6.20:

Beispiel 6.23 Wir betrachten den regulären Ausdruck $\alpha := (\mathtt{aab} \mid \mathtt{cc})^*$, den wir in Beispiel 6.20 als α_3 bezeichnet haben. Zum besseren Verständnis betrachten wir die Indizierung von α; also den Ausdruck

$$\widetilde{\alpha} = (\mathtt{a}_1\mathtt{a}_2\mathtt{b}_1 \mid \mathtt{c}_1\mathtt{c}_2)^*.$$

Wenn wir nun $\widetilde{\alpha}$ mittels der Glushkov-Konstruktion in einen NFA umwandeln, erhalten wir den folgenden Glushkov-Automaten $A_{\mathcal{G}}(\widetilde{\alpha})$:



Mit Ausnahme des Startzustands entspricht jeder Zustand eines Glushkov-Automaten genau einem Vorkommen eines Terminals im regulären Ausdrucks. ◊

Wir können nun feststellen, dass die Glushkov-Automatenkonstruktion verwendet werden, um deterministische reguläre Ausdrücke in DFAs umzuwandeln:

Lemma 6.24 *Ein regulärer Ausdruck α ist genau dann ein deterministischer regulärer Ausdruck, wenn $A_{\mathcal{G}}(\alpha)$ als DFA interpretiert werden kann*[89]*.*

Beweis: Sei α ein regulärer Ausdruck über einem Alphabet Σ, und sei $n \in \mathbb{N}_{>0}$ groß genug gewählt, dass $\mathcal{L}(\widetilde{\alpha}) \in \Sigma_{(n)}^*$. Der Homomorphismus $h\colon \Sigma^* \to \Sigma_{(n)}^*$ sei definiert durch $h(a_i) := a$ für alle $a \in \Sigma$ und alle $1 \leq i \leq n$. Sei $A_{\mathcal{G}}(\alpha) = (\Sigma, Q, \delta, q_0, F)$. Gemäß unserer Vorüberlegungen aus Lemma 6.22 können wir davon ausgehen, dass $A_{\mathcal{G}}(\alpha)$ und $A_{\mathcal{G}}(\widetilde{\alpha})$ abgesehen von dem zugrundeliegenden Alphabet und der Übergangsfunktion identisch sind. Zusätzlich muss gelten, dass $A_{\mathcal{G}}(\alpha)$ eine mit $a \in \Sigma$ beschriftete Kante zwischen zwei Zuständen hat, wenn $A_{\mathcal{G}}(\widetilde{\alpha})$ für ein geeignetes i eine mit a_i beschriftete Kante zwischen denselben Kanten besitzt.

Angenommen, α ist ein deterministischer regulärer Ausdruck, und $A_{\mathcal{G}}(\alpha)$ kann nicht als DFA interpretiert werden. Dann existieren ein Zustand $p \in Q$ und ein Buchstabe

[89] Das heißt, dass jeder Zustand von $A_{\mathcal{G}}(\alpha)$ zu jedem Terminal a höchstens eine ausgehende Kante haben darf, die mit a beschriftet ist.

$a \in \Sigma$, sowie unterschiedliche Zustände $q_1, q_2 \in Q$, so dass $\delta(p, a) \supseteq \{q_1, q_2\}$. Da jeder Zustand eines Glushkov-Automaten erreichbar ist, existiert ein $w \in \Sigma^*$ mit $p \in \delta(q_0, w)$. Da kein Zustand eine Sackgasse ist, existieren Wörter $u_1, u_2 \in \Sigma^*$ mit $wau_1, wau_2 \in \mathcal{L}(\alpha)$.

Wir betrachten nun dieselben Zustände in $A_{\mathcal{G}}(\widetilde{\alpha})$. Dann existiert ein indiziertes Wort $\widetilde{w} \in \Sigma^*$, das von q_0 zu p führt. Außerdem existieren a_i, a_j mit $1 \leq i, j \leq n$, so dass eine mit a_i beschriftete Kante von p zu q_1 führt, und eine mit a_j beschriftete Kante von p zu q_2. Schließlich existieren indizierte Wörter $\widetilde{u}_1$ und $\widetilde{u}_2$, die die gleiche Funktion wie u_1 und u_2 haben. Es gilt also: $\widetilde{w}a_iu_1 \in \mathcal{L}(\widetilde{\alpha})$ und $\widetilde{w}a_ju_2 \in \mathcal{L}(\widetilde{\alpha})$. Da jeder Zustand von $A_{\mathcal{G}}(\widetilde{\alpha})$ eindeutig einem Terminal aus $\widetilde{\alpha}$ entspricht, folgt aus $q_1 \neq q_2$, dass $i \neq j$ gelten muss. Also ist α kein deterministischer regulärer Ausdruck, Widerspruch.

Für die andere Richtung des Beweises nehmen wir nun an, dass α kein deterministischer regulärer Ausdruck ist, während $A_{\mathcal{G}}(\alpha)$ als DFA interpretiert werden kann. Dann existieren Wörter $x, y, z \in \Sigma_{(n)}^*$ und $a_i, a_j \in \Sigma_{(n)}$ mit $1 \leq i, j \leq n$ und $i \neq j$, so dass $xa_iy \in \mathcal{L}(\widetilde{\alpha})$ und $xa_jz \in \mathcal{L}(\widetilde{\alpha})$. Daher alle Kanten eines Glushkov-Automaten mit dem selben Terminal beschriftet sind, müssen die Wörter xa_i und xa_j in $A_{\mathcal{G}}(\widetilde{\alpha})$ zu unterschiedlichen Zuständen q_i und q_j führen. Sei $p := \delta(q_0, h(x))$. Dann existiert in $A_{\mathcal{G}}(\alpha)$ eine mit a beschriftete Kante sowohl von p zu q_i, als auch von p zu q_j. Also kann $A_{\mathcal{G}}(\alpha)$ nicht als DFA interpretiert werden, Widerspruch. □

6.2 Pattern-Matching

Ein häufiges Problem im Umgang mit Wörtern (im Sinne dieser Vorlesung) ist das sogenannte *Pattern-Matching*. Hier soll in einem längeren Text t das erste (oder jedes) Vorkommen eines Musters p gefunden werden[90]. Eine offensichtliche Anwendungsmöglichkeit ist dabei die Suchfunktion eines Text-Editors. Wir betrachten nun zuerst das Pattern-Matching mit Hilfe von Automaten, und wenden uns danach einem Algorithmus zu, der auf verwandten Betrachtungen basiert.

6.2.1 Pattern-Matching mit Automaten

Pattern-Matching lässt sich aus einer automatischen Sicht wie folgt auffassen: Gegeben sind ein Alphabet Σ und ein Suchmuster $p \in \Sigma^+$. Nun konstruiert man einen Automaten A_p für die Pattern-Match-Sprache $L_p := \Sigma^*\{p\}$. Wenn wir einen Text $t \in \Sigma^+$ nach Vorkommen von p durchsuchen wollen, genügt es, t buchstabenweise als Eingabe von A zu verwenden. Wann immer A in einen akzeptierenden Zustand gerät, wissen wir, dass die aktuelle Stelle von t das (rechte) Ende eines Vorkommens von p ist. Um das entsprechende Anfangsstück von p zu finden, genügt es, $|p| - 1$ Schritte nach links zu gehen. Wir betrachten zuerst die Konstruktion eines NFA:

[90] Der Begriff „Pattern-Matching“ hat nicht direkt mit den Pattern zu tun, die wir bei den Patternsprachen betrachtet haben. Es handelt zwar in beiden Fällen um Muster, dieser Begriff wird aber sehr unterschiedlich interpretiert.

Definition 6.25 Sei Σ ein Alphabet, $p = p_1 \cdots p_m$ mit $m \in \mathbb{N}_{>0}$ und $p_1, \ldots, p_m \in \Sigma$. Die **Pattern-Match-Sprache für p,** L_p, sei definiert als $L_p := \Sigma^*\{p\}$.

Der **Pattern-Match-NFA für p,** $A_{N,p}$, ist definiert als $A_{N,p} := (\Sigma, Q, \delta, q_0, F)$, wobei

$$Q := \{q_0, q_1, \ldots, q_m\}, \qquad F := \{q_m\},$$

$$\delta(q_0, a) := \begin{cases} \{q_0, q_1\} & \text{falls } a = p_1, \\ \{q_0\} & \text{falls } a \neq p_1, \end{cases}$$

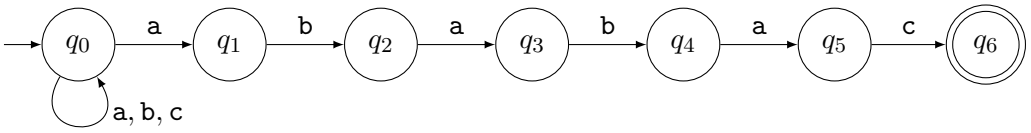
$$\delta(q_m, a) := \emptyset,$$

$$\delta(q_i, a) := \begin{cases} \{q_{i+1}\} & \text{falls } a = p_{i+1}, \\ \emptyset & \text{falls } a \neq p_{i+1} \end{cases}$$

für alle a und alle $1 \leq i < m$.

Offensichtlich gilt $L_p = \mathcal{L}(A_{N,p})$. Wir betrachten ein Beispiel für einen Pattern-Match-NFA:

Beispiel 6.26 Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $p := \mathtt{ababac}$. Dann lässt sich $A_{N,p}$ wie folgt graphisch darstellen:



Der Pattern-Match-NFA für ein Suchmuster p ist also nichts anderes als der kanonische NFA für die Pattern-Match-Sprache $L_p = \Sigma^*\{p\}$. ◊

Für viele praktische Anwendungen ist allerdings ein nichtdeterministischer Automat nicht akzeptabel. Unser Ziel ist es nun, einen entsprechenden DFA zu konstruieren.

Definition 6.27 Sei Σ ein Alphabet, $p \in \Sigma^+$. Der **Pattern-Match-DFA für p,** $A_{D,p}$, ist definiert als $A_{D,p} := \mathtt{pot}(A_{N,p})$, wobei $\mathtt{pot}$ die Potenzmengenkonstruktion bezeichnet.

Wir erhalten also $A_{D,p}$, indem wir einfach die Potenzmengenkonstruktion auf $A_{N,p}$ anwenden. Diese Entscheidung bedarf einer weiteren Rechtfertigung, denn wie wir bereits gesehen haben (Beispiel 3.56), garantiert nicht, dass ihr Resultat der minimale DFA ist. Bei Pattern-Match-DFAs droht hier allerdings keine Gefahr:

Lemma 6.28 *Sei Σ ein Alphabet, $p \in \Sigma^+$. Dann ist $A_{D,p}$ minimal.*

Beweis: Sei $p = p_1 \cdots p_m$ mit $m \in \mathbb{N}_{>0}$ und $p_1, \ldots, p_m \in \Sigma$. Wir betrachten nun zuerst den minimalen DFA für die Sprache $(L_p)^R = \{p^R\}\Sigma^*$. Dazu definieren wir $A_R := (\Sigma, Q_R, \delta_R, q_m, F_R)$ durch

$$Q_R := \{q_0, \ldots, q_m, q_{trap}\}, \qquad F := \{q_0\},$$
$$\delta(q_i, a) := \begin{cases} q_{i-1} & \text{falls } a = p_i, \\ q_{trap} & \text{falls } a \neq p_i \end{cases}$$

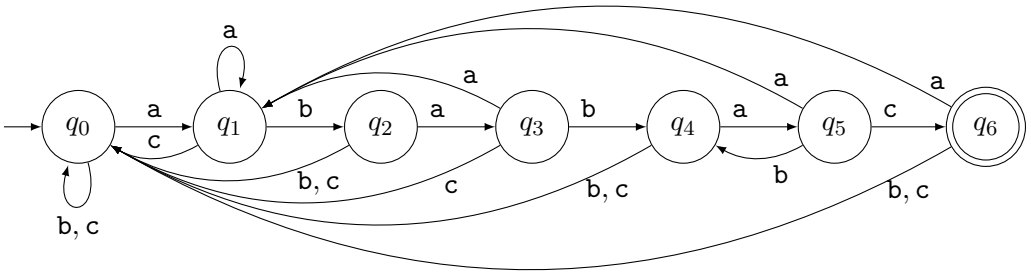
für alle $a \in \Sigma$ und $m \geq i \geq 1$, sowie $\delta(q_0, a) := q_0$ und $\delta(q_{trap}, a) := q_{trap}$ für alle $a \in \Sigma$.

Offensichtlich gilt $\mathcal{L}(A_R) = \{p^R\}\Sigma^* = \mathcal{L}(A_{D,p})^R$. Außerdem ist A_R ein vollständiger DFA, bei dem jeder Zustand erreichbar ist. Sei nun $A_N := \texttt{rev}(A_R)$; also ist A_N der NNFA, der aus A_R durch Umdrehen aller Kanten sowie durch Vertauschen der akzeptierenden und der Startzustände entsteht. Da A_R nur einen akzeptierenden Zustand hat, nämlich q_0, hat A_N nur einen Startzustand und ist daher nicht nur ein NNFA, sondern sogar ein NFA.

Außerdem können wir leicht sehen, dass A_N und $A_{N,p}$ fast identisch sind. Der einzige Unterschied ist, dass A_N noch zusätzlich den Zustand q_{trap} besitzt. Da dieser Zustand in A_R nur eingehende Kanten hat (mit Ausnahme der Schleifen zu sich selbst), hat q_{trap} in A_N nur eingehende Kanten und ist daher vom Startzustand aus nicht erreichbar. Da nicht erreichbare Zustände auf die Potenzmengenkonstruktion keinen Einfluss haben[91], muss $\texttt{pot}(A_N) = \texttt{pot}(A_{N,p})$ gelten (abgesehen von einer eventuell notwendigen Umbenennung der Zustände). Also ist $A_{D,p} = \texttt{pot}(\texttt{rev}(A_R))$. Da A_R ein vollständiger DFA ist, in dem jeder Zustand erreichbar ist, muss $A_{D,p}$ gemäß Lemma 3.67 minimal sein. □

Auch wenn nun sichergestellt ist, dass $A_{D,p}$ stets minimal ist, könnte es prinzipiell immer noch passieren, dass $A_{D,p}$ im Vergleich zu $A_{N,p}$ exponentiell so viele Zustände hat. Bevor wir diese Frage untersuchen, greifen wir Beispiel 6.26 wieder auf, indem wir den entsprechenden Pattern-Match-DFA konstruieren:

Beispiel 6.29 Sei $\Sigma := \{\texttt{a}, \texttt{b}, \texttt{c}\}$ und sei $p := \texttt{ababac}$. Der Pattern-Match-DFA $A_{D,p}$ hat die folgende graphische Darstellung:



Zerlegt man nun p in die einzelnen Buchstaben $p_1, \ldots, p_6 \in \Sigma$ (es gelte also $p_1 \cdots p_6 = p$), so fällt auf, dass jeder Zustand q_i dem Präfix $p_1 \cdots p_i$ von p entspricht. Ist der Automat

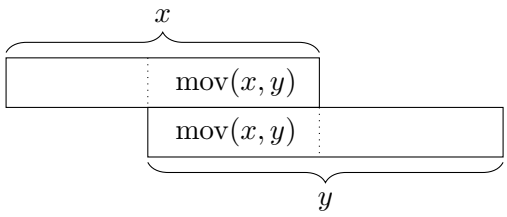
[91] Wir haben pot ausdrücklich so definiert.

beispielsqeise im Zustand q_5, so wurde ein Wort aus $\Sigma^*\{\texttt{ababa}\}$ eingelesen. Wird in diesem Zustand ein `c` eingelesen, so wird das Eingabewort akzeptiert, da es auf `ababac` endet. Interessanter ist der Fall, dass ein anderer Buchstabe eingelesen wird: In diesem Fall versucht der DFA so viel wie möglich von der bisher eingelesen Eingabe weiter zu verwenden. Wir zum Beispiel ein `b` eingelesen, so weiß der Automat, dass die bisherige Eingabe zur Sprache $\Sigma^*\{\texttt{ababab}\}$ gehört. Davon sind die letzten vier Buchstaben für das Zielwort relevant: Da `abab` ein Präfix von p ist, wechselt der Automat in den Zustand q_4. Analog dazu wird beim Einlesen eines `a` in den Zustand `a` gewechselt, da von `ababaa` nur das Suffix der Länge 1, alost `a`, ein Präfix von p ist. ◊

Die Beobachtungen am Ende von Beispiel 6.29 lassen sich verallgemeinern und auf jeden Pattern-Match-DFA anwenden. Wir werden daraus einen Ansatz entwickeln, mit dem wir Pattern-Match-DFAs direkt ohne den Umweg über den Pattern-Match-NFA konstruieren können. Dazu benötigen wir noch ein weiteres formales Werkzeug:

Definition 6.30 Sei Σ ein Alphabet. Für Wörter $x, y \in \Sigma^*$ sei die **maximale Überlappung von x und y,** $\text{mov}(x, y)$, definiert als das längste Wort aus der Menge $\text{suffix}(x) \cap \text{prefix}(y)$.

Die maximale Überlappung von x und y ist also das längste Wort, das sowohl ein Suffix („Anfangsstück“) von x, als auch ein Präfix („Endstück“) von y ist. Wir können $\text{mov}(x, y)$ auf die folgende Art graphisch darstellen:



Beispiel 6.31 Wir betrachten eine Reihe von Beispielen:

$$\begin{aligned}
\text{mov}(\texttt{abc}, \texttt{bca}) &= \texttt{bc}, & \text{mov}(\texttt{bca}, \texttt{abc}) &= \texttt{a},\\
\text{mov}(\texttt{acab}, \texttt{aba}) &= \texttt{ab}, & \text{mov}(\texttt{aba}, \texttt{acab}) &= \texttt{a},\\
\text{mov}(\texttt{aca}, \texttt{aba}) &= \texttt{a}, & \text{mov}(\texttt{aba}, \texttt{aca}) &= \texttt{a},\\
\text{mov}(\texttt{cbc}, \texttt{abc}) &= \varepsilon, & \text{mov}(\texttt{abc}, \texttt{cbc}) &= \texttt{bc},\\
\text{mov}(\texttt{baaa}, \texttt{aa}) &= \texttt{aa}, & \text{mov}(\texttt{aa}, \texttt{baaa}) &= \varepsilon.
\end{aligned}$$

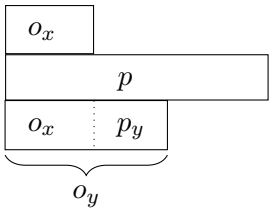
Wie leicht zu sehen ist, gilt im Allgemeinen nicht $\text{mov}(x, y) = \text{mov}(y, x)$, wobei es Fälle gibt, in denen diese Gleichheit gelten kann. Insbesondere gilt $\text{mov}(w, w) = w$ für jedes Wort w. ◊

Da ε sowohl Präfix als auch Suffix jedes Wortes ist, ist $\text{mov}(x, y)$ für alle Wörter x und y definiert. Für unsere Zwecke ist besonders nützlich, dass für alle Wörter p die Funktion mov verwendet werden kann, um die Nerode-Relation von L_p zu bestimmen:

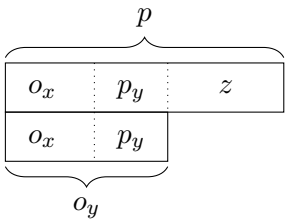
Lemma 6.32 *Sei Σ ein Alphabet und $p \in \Sigma^+$. Dann gilt für alle $x, y \in \Sigma^*$:*

$$x \equiv_{L_p} y \text{ genau dann, wenn } \mathrm{mov}(x,p) = \mathrm{mov}(y,p).$$

Beweis: Wir beginnen mit der Hin-Richtung. Angenommen, $x \equiv_{L_p} y$. Gemäß Definition der Nerode-Relation (Definition 3.28) gilt $D_x L_p = D_y L_p$. Das heißt, für alle $z \in \Sigma^*$ ist $xz \in L_p$ genau dann, wenn $yz \in L_p$. Wir definieren nun $o_x := \mathrm{mov}(x,p)$ und $o_y := \mathrm{mov}(y,p)$. Angenommen, es gilt $\mathrm{mov}(x,p) \neq \mathrm{mov}(y,p)$, also $o_x \neq o_y$. Da sowohl o_x als auch o_y ein Präfix von p ist, muss $|o_x| \neq |o_y|$ gelten. Ohne Beeinträchtigung der Allgemeinheit nehmen wir an, dass $|o_x| < |o_y|$. Da $o_x, o_y \in \mathrm{prefix}(p)$ und da o_x kürzer ist als o_y, muss o_x ein echtes Präfix von o_y sein. Es existiert also ein Wort $p_y \in \Sigma^+$ mit $o_y = o_x p_y$. Graphisch lässt sich dies wie folgt skizzieren:



Außerdem können wir nun auch p zerlegen. Da o_y ein Präfix von p ist, existiert ein $z \in \Sigma^*$ mit $p = o_y z$, und somit $p = o_x p_y z$. Zur Illustration betrachten wir auch hier eine Skizze:



Wir betrachten nun das Wort yz. Da y auf o_y endet, endet yz auf $o_y z = o_x p_y z = p$. Somit ist $yz \in L_p$ und, da $x \equiv_{L_p} y$, auch $xz \in L_p$. Da $|p| = |o_x| + |p_y| + |z|$, ragt p um Länge $|o_x| + |p_y| \geq 1$ in x hinein. Es existiert also ein Wort $w \in (\mathrm{suffix}(x) \cap \mathrm{prefix}(p))$ mit $|w| = |o_x| + |p_y| > |o_x|$. Also kann o_x nicht das längste Wort aus $(\mathrm{suffix}(x) \cap \mathrm{prefix}(p))$ sein. Widerspruch zu $o_x = \mathrm{mov}(x,p)$. Dies beendet den Beweis der Hin-Richtung.

Für die Rück-Richtung sei $\mathrm{mov}(x,p) = \mathrm{mov}(y,p)$. Wir zeigen nun, dass für alle $z \in \Sigma^*$ aus $xz \in L_p$ stets $yz \in L_p$ folgt. Daraus ergibt sich dann $D_x L_p \subseteq D_y L_p$; der Beweis für die umgekehrte Inklusion folgt dann direkt durch Vertauschen aller x und y. Angenommen, $xz \in L_p$. Ist $|z| \geq |p|$, so muss ein $z' \in \Sigma^*$ existieren, für das $z = z'p$; in diesem Fall ist auch $yz \in L_p$. Nehmen wir also an, dass $|z| < |p|$. Dann existieren ein $x_1 \in \Sigma^*$ und ein $x_2 \in \Sigma^+$ mit $x_1 x_2 = x$ und $x_2 z = p$. Also ist x_2 ein Suffix von $\mathrm{mov}(x,p)$, und somit auch ein Suffix von $\mathrm{mov}(y,p)$. Da $\mathrm{mov}(y,p)$ ein Suffix von y ist, ist x_2 ebenfalls ein Suffix von y. Somit existiert ein $y_1 \in \Sigma^*$ mit $y = y_1 x_2$. Es gilt also:

$$yz = y_1 x_2 z = y_1 p.$$

Also ist $yz \in L_p$. Somit folgt aus $xz \in L_p$ stets $yz \in L_p$; daher ist $D_x L_p \subseteq D_y L_p$. Durch Vertauschen von x und y erhalten wir auf die gleiche Art $D_y L_p \subseteq D_x L_p$, und somit $D_x L_p = D_y L_p$. Diese Beobachtung ist äquivalent zu $x \equiv_{L_p} y$. □

Dank Lemma 6.32 wissen wir, dass für jedes $p \in \Sigma^+$ die Äquivalenzklassen der Nerode-Relation $\equiv_{L_p}$ exakt den Wörtern aus prefix(p) entsprechen. Da es, gemäß Satz 3.41, eine Eins-zu-eins-Beziehung zwischen diesen Äquivalenzklassen und den Zuständen des minimalen DFA für L_p gibt, können wir nun schließen, dass der Pattern-Match-DFA $A_{D,p}$ (der gemäß Lemma 6.28 minimal ist) für jedes Wort p eine ähnliche Struktur wie in Beispiel 6.29 haben muss: Eine Folge von $|p| + 1$ Zuständen, die den Wörtern aus prefix(p) entsprechen, und eine Reihe von passenden Kanten.

Wir können also feststellen, dass die folgende Definition von $A_{D,p}$ äquivalent ist zu Definition 6.27:

Definition 6.33 Sei Σ ein Alphabet, $p = p_1 \cdots p_m$ mit $m \in \mathbb{N}_{>0}$ und $p_1, \ldots, p_m \in \Sigma$. Der **Pattern-Match-DFA für** p**,** $A_{D,p}$**,** ist definiert als $A_{D,p} := (\Sigma, Q, \delta, 0, F)$, wobei

$$\begin{aligned} Q &:= \{i \in \mathbb{N} \mid 0 \leq i \leq m\}, \\ F &:= \{m\}, \\ \delta(i, a) &:= |\operatorname{mov}(p_1 \cdots p_i \cdot a, p)| \end{aligned}$$

für alle $i \in Q$, $a \in \Sigma$.

Dabei gilt wie gewohnt $p_1 \cdots p_0 = \varepsilon$, also $\delta(0, a) = |\operatorname{mov}(a, p)|$ für alle $a \in \Sigma$. Bei der Definition nutzen wir aus, dass wir jeden Zustand anstelle von $p_1 \cdots p_i$ auch direkt durch i identifizieren können (da p ja fest ist).

Neben der in Definition 6.33 vorgestellten Sichtweise gibt es noch eine weitere Möglichkeit, die Übergangsfunktion von $A_{D,p}$ zu definieren. Da wir diese anschließend verwenden werden, werfen wir auch hierauf einen Blick:

Definition 6.34 Sei Σ ein Alphabet, $p = p_1 \cdots p_m$ mit $m \in \mathbb{N}_{>0}$ und $p_1, \ldots, p_m \in \Sigma$. Sei $Q := \{i \in \mathbb{N} \mid 0 \leq i \leq m\}$. Die **Fehlerfunktion** $\operatorname{fail}\colon (Q - \{0\}) \to Q$ sei für alle $i \in (Q - \{0\})$ definiert als

$$\operatorname{fail}(i) := |\operatorname{mov}(p_2 \cdots p_i, p_1 \cdots p_{i-1})|.$$

Hierbei gilt $\operatorname{fail}(1) = 0$. Außerdem definieren wir die Übergangsfunktion $\delta_F\colon Q \times \Sigma \to Q$ als

$$\delta_F(i, a) := \begin{cases} i + 1 & \text{falls } a = p_{i+1}, \\ 0 & \text{falls } a \neq p_{i+1} \text{ und } i = 0, \\ \delta_F(\operatorname{fail}(i), a) & \text{falls } a \neq p_{i+1} \text{ und } i \neq 0 \end{cases}$$

für alle $i \in Q$ und alle $a \in \Sigma$.

Gemäß Definition ist $\text{fail}(i) \leq (i-1)$ für alle $i \in (Q - \{0\})$. Bevor wir zeigen, dass δ_F und δ (aus Definition 6.33) stets die gleiche Funktion beschreiben, betrachten wir ein Beispiel.

Beispiel 6.35 Wir greifen noch einmal Beispiel 6.29 auf. Sei $\Sigma := \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ und sei $p := \mathtt{ababac}$. Es gilt $Q = \{0, 1, \ldots, 6\}$. Im folgenden sind die Werte von $\text{fail}(i)$ und das jeweils zugrundeliegende Wort $\text{mov}(p_2 \cdots p_i, p_1 \cdots p_{i-1})$ angegeben:

$$\begin{aligned}
&\text{fail}(1) = 0, && \text{gilt immer,}\\
&\text{fail}(2) = 0, && \text{wegen } \text{mov}(\mathtt{b}, \mathtt{a}) = \varepsilon,\\
&\text{fail}(3) = 1, && \text{wegen } \text{mov}(\mathtt{ba}, \mathtt{ab}) = \mathtt{a},\\
&\text{fail}(4) = 2, && \text{wegen } \text{mov}(\mathtt{bab}, \mathtt{aba}) = \mathtt{ab},\\
&\text{fail}(5) = 3, && \text{wegen } \text{mov}(\mathtt{baba}, \mathtt{abab}) = \mathtt{aba},\\
&\text{fail}(6) = 0, && \text{wegen } \text{mov}(\mathtt{babac}, \mathtt{ababa}) = \varepsilon.
\end{aligned}$$

Hieraus ergibt sich die folgende Übergangstablle für δ_F:

	a	b	c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	1	0	0

Sei nun $A := (\Sigma, Q, \delta_F, 0, \{6\})$. Mit ein wenig Aufwand können Sie leicht überprüfen, dass A und der DFA $A_{D,p}$ aus Beispiel 6.29 identisch sind (abgesehen von einer notwendigen Umbenennung der Zustände). ◊

Um mit der alternativen Übergangsfunktion δ_F aus Definition 6.34 arbeiten zu können, müssen wir zuerst zeigen, dass diese tatsächlich die gleiche Funktion beschreibt wie δ aus Definition 6.33:

Lemma 6.36 *Sei Σ ein Alphabet, $p = p_1 \cdots p_m$ mit $m \in \mathbb{N}_{>0}$ und $p_1, \ldots, p_m \in \Sigma$. Sei $Q := \{i \in \mathbb{N} \mid 0 \leq i \leq m\}$. Außerdem sei δ definiert wie in Definition 6.33, sowie* fail *und δ_F wie in Definition 6.34. Dann gilt $\delta(i, a) = \delta_F(i, a)$ für alle $i \in Q$ und alle $a \in \Sigma$.*

Beweis: Wir zeigen dies durch eine Induktion über i.

Induktionsanfang: $i = 0$.

Behauptung: Für alle $a \in \Sigma$ ist $\delta(0, a) = \delta_F(0, a)$.

Beweis: Angenommen, $a = p_1$. Dann ist $\delta(0, a) = p_1 = \delta_F(0, a)$. Angenommen, $a \neq p_1$. Dann ist $\delta(0, a) = 0 = \delta_F(0, a)$.

INDUKTIONSSCHRITT: Sei $1 \leq i \leq m$.
Induktionsannahme: Für alle j mit $0 \leq j < i$ und alle $a \in \Sigma$ gilt $\delta(j, a) = \delta_F(j, a)$.
Behauptung: Für alle $a \in \Sigma$ ist $\delta(i, a) = \delta_F(i, a)$.
Beweis: Angenommen, $a = p_{i+1}$. Dann gilt $\delta(i, a) = i + 1 = \delta(i + 1, a)$ nach Definition. Wir können also davon ausgehen, dass $a \neq p_{i+1}$ (insbesondere gilt dies wenn $i = m$, da hier p_{i+1} gar nicht definiert ist). Es gilt:

$$\begin{aligned}
\delta_F(i, a) &= \delta_F(\text{fail}(i), a) && (\text{da } i > 0 \text{ und } a \neq p_{i+1}) \\
&= \delta(\text{fail}(i), a) && (\text{da fail}(i) < i, \text{ und gemäß Ind.ann.}) \\
&= \delta(|\operatorname{mov}(p_2 \cdots p_i, p_1 \cdots p_{i-1})|, a) && (\text{Definition fail}).
\end{aligned}$$

Außerdem gilt $\operatorname{mov}(p_2 \cdots p_i, p_1 \cdots p_{i-1}) = \operatorname{mov}(p_2 \cdots p_i, p)$, da das hintere Teilstück $p_{i+1} \cdots p_m$ von p keinen Einfluss auf $\operatorname{mov}(p_2 \cdots p_i, p)$ haben kann; denn wegen $|p_2 \cdots p_i| = i - 1$ kann p nicht weiter in $p_2 \cdots p_i$ hineingeschoben werden. Zusammengefasst ergibt dies

$$\begin{aligned}
\delta_F(i, a) &= \delta(|\operatorname{mov}(p_2 \cdots p_i, p)|, a) \\
&= |\operatorname{mov}(\operatorname{mov}(p_2 \cdots p_i, p) \cdot a, p)| && (\text{Definition } \delta).
\end{aligned}$$

Gemäß unserer ursprünglichen Annahme ist $p_{i+1} \neq a$. Also muss $\operatorname{mov}(p_2 \cdots p_i, p)$ ein Präfix von $\operatorname{mov}(p_2 \cdots p_i \cdot a, p)$ sein. Wir können dies also zusammenfassen zu

$$\delta_F(i, a) = |(\operatorname{mov}(p_2 \cdots p_i \cdot a, p)|.$$

Wir vergleichen dies nun mit $\delta(i, a)$. Gemäß Definition ist

$$\delta(i, a) = |\operatorname{mov}(p_1 \cdots p_i \cdot a, p)|.$$

Da $p_{i+1} \neq a$, muss $\delta(i, a) \leq i$ gelten. Somit kann p_1 für die Berechnung von $\delta(i, a)$ keine Rolle spielen. Es gilt also

$$\begin{aligned}
\delta(i, a) &= |\operatorname{mov}(p_2 \cdots p_i \cdot a, p)| \\
&= \delta_F(i, a) && (\text{wie oben festgestellt}).
\end{aligned}$$

Somit ist $\delta(i, a) = \delta_F(i, a)$ für alle $i \in Q$ und alle $a \in \Sigma$. □

Lemma 6.36 demonstriert, dass wir $A_{D,p}$ auch über anhand der fail-Funktion konstruieren können.

Hinweis 6.37 Wir haben die folgenden vier Möglichkeiten kennen gelernt, um den Pattern-Match-DFA $A_{D,p}$ zu konstruieren:

1. Durch Konstruktion von $A_{N,p}$ (Definition 6.25) und Anwenden der Potenzmengenkonstruktion.
2. Durch Erstellen der Übergangsrelation δ wie in Definition 6.33.
3. Durch Erstellen die Übergangsrelation δ_F mittels fail wie in Definition 6.34. Dabei kann fail wie dort angegeben berechnet werden, oder wie mittels Algo-

rithmus 7 weiter unten.

4. Durch „Draufstarren“: Bei vergleichsweise kurzen Pattern p und relativ kleinen Alphabeten kann $A_{D,p}$ oft auch direkt konstruiert werden.

Welche Methode für Sie am besten geeignet ist, hängt von Ihren persönlichen Vorlieben und Stärken ab. Der vierte Ansatz kann dabei auch von einem tieferen Verständnis der beiden direkten Konstruktionen des Pattern-Match-DFA profitieren. In jedem Fall ist die einzige Schwierigkeit, die „Rückwärtskanten“, also die Übergänge $\delta(i, a)$ mit $a \neq p_{i+1}$ korrekt zu setzen. Dabei ist es hilfreich, sich vor Augen zu halten, dass im gesuchten Folgezustand genau der Teil von $p_1 \cdots p_i \cdot a$ gespeichert werden soll, der für das Erkennen von p relevant ist.

Der Ansatz, $A_{D,p}$ mittels der fail-Funktion zu konstruieren, kann auch für einen Pattern-Match-Algorithmus verwendet werden, der Automaten nicht explizit verwendet. Wir werden darauf im folgenden Abschnitt eingehen.

6.2.2 Der Knuth-Morris-Pratt-Algorithmus

Wir können das Problem, alle Vorkommen eines Suchmusters $p \in \Sigma^+$ in einem Text $t \in \Sigma^+$ zu finden, auch lösen, ohne explizit Automaten zu verwenden. Für dieses Problem existieren verschiedene Algorithmen; die Frage, welcher Pattern-Match-Algorithmus für eine Anwendung am besten geeignet ist hängt dabei stark von der Größe von Σ und von den Eigenschaften von p und t ab.

Ein nahe liegender, wenn auch naiver, Lösungsansatz ist dabei, p durch t zu schieben und dabei p und t buchstabenweise zu vergleichen. Wann immer ein Unterschied zwischen p und der aktuellen Stelle von t entdeckt wird, wird p um eine Stelle weitergeschoben und der Vergleich beginnt von vorne. Es lässt sich zeigen, dass dieser Ansatz im worst case eine Laufzeit von $O(mn)$ hat, wobei $m := |p|$ und $n := |t|$. Dies mag für einige Anwendungen akzeptabel sein, wie zum Beispiel vielen Einsatzbereichen von Text-Editoren. Für andere, wie zum Beispiel häufige Probleme der Bio-Informatik, ist dies allerdings noch nicht effizient genug.

Im vorigen Absatz haben wir bereits eine Methode kennen gelernt, die schneller ist als dieser Ansatz: Wenn wir den Pattern-Match-DFA $A_{D,p}$ konstruieren, können wir anschließend alle Vorkommen von p in einem Text t mit einer Laufzeit von $O(n)$ lösen. Zusätzlich zu dieser Zeit müssen wir natürlich noch den Aufwand für die Konstruktion von $A_{D,p}$ veranschlagen. Wenn wir dazu die Übergangsfunktion δ vollständig konstruieren, müssen wir $|\Sigma|m$ Felder füllen. Da prinzipiell $|\Sigma| = m$ kann, ist dies auch kein besonders effizienter Ansatz.

Stattdesssen betrachten wir einen Algorithmus, der das Pattern-Matching-Problem löst, indem er $A_{D,p}$ effizient simuliert, den **Knuth-Morris-Pratt-Algorithmus**, auch bekannt als **KMP-Algorithmus**[92]. Sie finden diesen unter dem Namen `KMP` als Algo-

[92] Benannt nach Donald Knuth, James Hiram Morris und Vaughan Pratt. Dabei haben Knuth und Pratt den Algorithmus durch komplexitätstheoretische Überlegungen entdeckt, während Morris ihn unabhängig davon für ein Programm entwickelt hatte. Allerdings fanden seine Kollegen den Algorith-

rithmus 6. Für beliebige Wörter $p = p_1 \cdots p_m$ und $t = t_1 \cdots t_n$ gibt `KMP`(p, t) alle Stellen j zurück, für die $t_{j-m+1} \cdots t_j \in L_p$, das heißt $t_{j-m+1} \cdots t_j = p$. Dabei simuliert `KMP` die Funktionsweise von $A_{D,p}$ unter Verwendung der Übergangsfunktion nach der Definition von δ_F (Definition 6.34) auf der Eingabe t. Die Variable i entspricht dem aktuellen Zustand des Automaten, und j markiert die aktuell betrachtete Stelle der Eingabe.

Algorithmus 6 : `KMP`

Eingabe : Wörter $p = p_1 \cdots p_m$ und $t = t_1 \cdots t_n$, $0 < m \leq n$
Ausgabe : Alle Stellen j mit $p = t_{j-m+1} \cdots t_j$

```
1  fail ← KMPfail(p);
2  i ← 0;
3  j ← 1;
4  while j ≤ n do
5      while (i > 0 and p_{i+1} ≠ t_j) do i ← fail(i);
6      if p_{i+1} = t_j then i ← i + 1;
7      if i > m then
8          output j;
9          i ← fail(i);
10     j ← j + 1;
```

Algorithmus 7 : `KMPfail`

Eingabe : Ein Wort $p = p_1 \cdots p_m$, $m \geq 1$
Ausgabe : Die Funktion fail für p

```
1  i ← 0;
2  j ← 2;
3  while j ≤ m do
4      while (i > 0 and p_{i+1} ≠ p_j) do i ← fail(i);
5      if p_{i+1} = p_j then
6          i ← i + 1;
7          fail(j) ← i
8      else fail(j) ← fail(i);
9      j ← j + 1;
10 return fail;
```

Die Fehlerfunktion fail wird dabei in einem Preprocessing-Schritt zu Beginn des Algorithmus berechnet, dazu wird die Unterfunktion `KMPfail` verwendet (Algorithmus 7).

mus zu kompliziert, so dass der Code nicht verwendet wurde. Knuth selbst erzählt diese Anekdote in einem Video, das Sie unter der Adresse `http://www.webofstories.com/play/donald.knuth/92` finden können.

Satz 6.38 *Sei Σ ein Alphabet, $p = p_1 \cdots p_m$ mit $m \in \mathbb{N}_{>0}$ und $p_1, \ldots, p_m \in \Sigma$ und $t = t_1 \cdots t_n$ mit $n \in \mathbb{N}_{>0}$ und $t_1, \ldots, t_n \in \Sigma$. Dann terminiert `KMP`$(p, t)$ in $O(m + n)$ Schritten*

Beweis: Wir betrachten zuerst die Laufzeit von `KMP` ohne Berücksichtigung der Laufzeit von `KMPfail`. Dabei stellen wir fest, dass die äußere while-Schleife exakt m mal durchlaufen wird, und zwar für alle möglichen Werte von j mit $1 \leq j \leq n$. Außerdem wird i höchstens so oft um 1 erhöht wie j. Bei jedem Durchlauf der inneren while-Schleife wird i durch fail(i) ersetzt. Gemäß Definition 6.34 ist fail$(i) < i$ für alle $i < 0$. Also verringert die innere while-Schleife in jedem Durchlauf i um mindestens 1, sie kann also höchstens n mal durchlaufen werden. Abgesehen von `KMPfail` hat `KMP` also eine Laufzeit von $O(n)$.

Durch eine analoge Argumentation lässt sich feststellen, dass `KMPfail` eine Laufzeit von $O(m)$ hat. Also terminiert `KMP`(p, t) nach $O(m + n)$ Schritten. □

In der Literatur werden oft Varianten des KMP-Algorithmus betrachtet, die die fail-Funktion ein wenig anders definieren; die prinzipielle Funktionsweise ist aber die gleiche. Die hier präsentierte Variante wurde gewählt, weil sie den Bezug zum Pattern-Match-DFA besonders verdeutlicht.

6.3 Aufgaben

Noch keine.

6.4 Bibliographische Anmerkungen

Dieser Abschnitt ist momentan nur ein Platzhalter. In Kürze werden hier einige Kommentare zu den verwendeten Quellen und weiterführendem Lesematerial zu finden sein.

Literaturverzeichnis

[1] A. V. Aho, M. S. Lam, R. Sethi, und J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley, 2006.

[2] J. Berstel, L. Boasson, O. Carton, und I. Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010. `http://arxiv.org/abs/1010.5318`.

[3] J. Bremer und D. D. Freydenberger. Inclusion problems for patterns with a bounded number of variables. *Information and Computation*, 220–221:15–43, 2012.

[4] V. Diekert, M. Kufleitner, und G. Rosenberger. *Diskrete algebraische Methoden*. De Gruyter Studium, 2013.

[5] P. Dömösi und M. Ito. *Context-Free Languages and Primitive Words*. World Scientific Publishing Company, 2014. Noch nicht erschienen.

[6] D. D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory of Computing Systems*, 53:159–193, 2013.

[7] M. R. Garey und D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[8] J. E. Hopcroft und J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[9] T. Jiang, A. Salomaa, K. Salomaa, und S. Yu. Decision problems for patterns. *Journal of Computer and Systems Sciences*, 50:53–63, 1995.

[10] M. Kudlek. Context-free languages. In C. Martin-Vide, V. Mitrana, und G. Păun, Hrsg., *Formal Languages and Applications*, Band 148 von *Studies in Fuzziness and Soft Computing*, Kapitel 5, Seiten 97–116. Springer, 2004.

[11] A. Mateescu und A. Salomaa. Aspects of classical language theory. In G. Rozenberg und A. Salomaa, Hrsg., *Handbook of Formal Languages*, Band 1, Kapitel 4, Seiten 175–251. Springer, 1997.

[12] G. Rozenberg und A. Salomaa. *Handbook of Formal Languages*, Band 1. Springer, 1997.

[13] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.

[14] G. Schnitger. *Skript zur Vorlesung Theoretische Informatik 2*. Goethe-Universität Frankfurt am Main, 2013. `http://www.thi.informatik.uni-frankfurt.de/lehre/gl2/sose13/gl2_sose13_skript.pdf`.

[15] U. Schöning. *Theoretische Informatik – kurzgefasst*. Springer, 2001.

[16] N. Schweikardt. *Skript zur Vorlesung Diskrete Modellierung*. Goethe-Universität Frankfurt am Main, 2013. `http://www.tks.informatik.uni-frankfurt.de/data/teaching/regularly/dismod/MOD-Skript.pdf`.

[17] G. Sénizergues. $L(A) = L(B)$? Decidability results from complete formal systems. *Theoretical Computer Science*, 251(1–2):1–166, 2001.

[18] Géraud Sénizergues. $L(A) = L(B)$? A simplified decidability proof. *Theoretical Computer Science*, 281(1–2):555–608, 2002.

[19] J. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008.

[20] M. Sipser. *Introduction to the theory of computation*. PWS Publishing, 2006.

[21] I. Wegener. *Kompendium Theoretische Informatik – eine Ideensammlung*. Teubner, 1996.

[22] I. Wegener. *Theoretische Informatik*. Teubner, 2005.

[23] S. Yu. Regular languages. In G. Rozenberg und A. Salomaa, Hrsg., *Handbook of Formal Languages*, Band 1, Kapitel 2, Seiten 41–110. Springer, 1997.

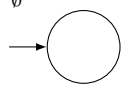
A Kurzreferenz

A.1 Übersicht über Abschlusseigenschaften

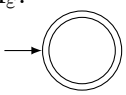
Operation	FIN	REG	CFL	DCFL
Vereinigung $L_1 \cup L_2$	abgeschlossen (Lemma 2.9)	abgeschlossen (Korollar 3.20)	abgeschlossen (Satz 4.16)	**nicht** abgeschl. (Lemma 4.85)
Schnitt $L_1 \cap L_2$	abgeschlossen (Lemma 2.9)	abgeschlossen (Lemma 3.18)	**nicht** abgeschl. (Lemma 4.22)	**nicht** abgeschl. (Lemma 4.85)
Schn. m. REG $L \cap R$, R reg.	abgeschlossen	abgeschlossen (Lemma 3.18)	abgeschlossen (Lemma 4.35)	abgeschlossen (Lemma 4.85)
Differenz $L_1 - L_2$	abgeschlossen (Lemma 2.9)	abgeschlossen (Korollar 3.20)	**nicht** abgeschl. (Lemma 3.14)	**nicht** abgeschl. (Lemma 4.85)
Komplement $\overline{L}$	**nicht** abgeschl. (Lemma 2.9)	abgeschlossen (Lemma 3.14)	**nicht** abgeschl. (Lemma 4.22)	abgeschlossen (Satz 4.83)
Konkatenation $L_1 \cdot L_2$	abgeschlossen (Lemma 2.9)	abgeschlossen (Lemma 3.77)	abgeschlossen (Satz 4.16)	**nicht** abgeschl.
Shuffle-Prod. shuffle(L_1, L_2)	abgeschlossen (Lemma 2.9)	abgeschlossen (Lemma 3.60)	**nicht** abgeschl. (Übung)	**nicht** abgeschl.
Präfix-Op. prefix(L)	abgeschlossen (Lemma 2.9)	abgeschlossen (Korollar 3.17)	abgeschlossen (Lemma 4.75)	abgeschlossen
Suffix-Op. suffix(L)	abgeschlossen (Lemma 2.9)	abgeschlossen (Lemma 3.69)	abgeschlossen (Lemma 4.75)	**nicht** abgeschl.
Reg. R.-Quot. L/R, R reg.	abgeschlossen	abgeschlossen (Lemma 3.112)	abgeschlossen (Lemma 4.75)	abgeschlossen
Reversal-Op. L^R	abgeschlossen (Lemma 2.9)	abgeschlossen (Lemma 3.64)	abgeschlossen (Satz 4.16)	**nicht** abgeschl.
n-fache Konk. L^n $(n \in \mathbb{N})$	abgeschlossen (Lemma 2.9)	abgeschlossen (Lemma 3.78)	abgeschlossen (Satz 4.16)	**nicht** abgeschl.
Kleene-Plus L^+	**nicht** abgeschl. (Lemma 2.9)	abgeschlossen (Korollar 3.81)	abgeschlossen (Satz 4.16)	**nicht** abgeschl.
Kleene-Stern L^*	**nicht** abgeschl. (Lemma 2.9)	abgeschlossen (Lemma 3.79)	abgeschlossen (Satz 4.16)	**nicht** abgeschl.
reg. Subst. $s(L)$	**nicht** abgeschl.	abgeschlossen (Lemma 3.97)	abgeschlossen (Satz 4.16)	**nicht** abgeschl. (Lemma 4.85)
Homom. $h(L)$	abgeschlossen	abgeschlossen (Lemma 3.97)	abgeschlossen (Satz 4.16)	**nicht** abgeschl. (Lemma 4.85)
inv. Homom. $h^{-1}(L)$	**nicht** abgeschl. (Beispiel 3.101)	abgeschlossen (Lemma 3.104)	abgeschlossen (Lemma 4.74)	abgeschlossen (Lemma 4.85)

A.2 Umwandlung von regulären Ausdrücken in ε-NFAs

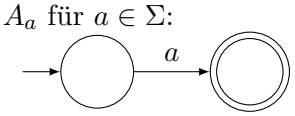
$A_\emptyset$:



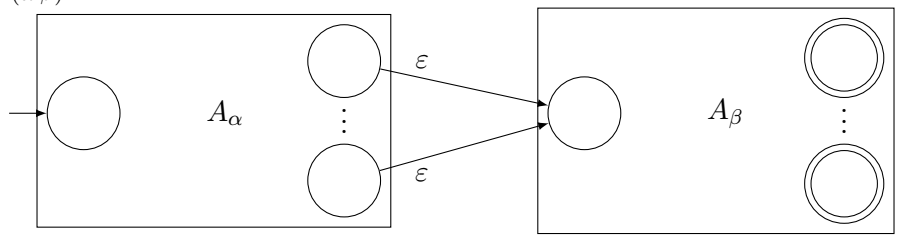
A_ε:



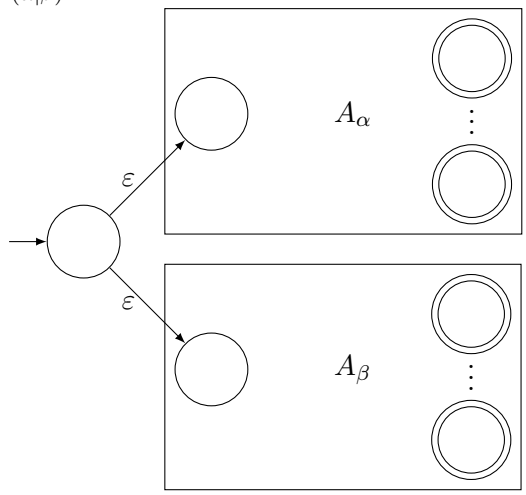
A_a für $a \in \Sigma$:



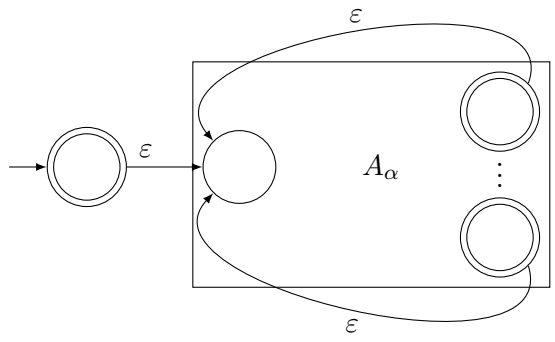
$A_{(\alpha\cdot\beta)}$, siehe Lemma 3.77:



$A_{(\alpha|\beta)}$, siehe Lemma 3.76:



A_{α^*}, siehe Lemma 3.79:



A.3 Rechenregeln für reguläre Ausdrücke

Seien α, β, γ reguläre Ausdrücke. Folgende Aussagen lassen sich zu den Sprachen dieser Ausdrücke treffen:

$$
\begin{aligned}
\mathcal{L}(\varepsilon \cdot \alpha) &= \mathcal{L}(\alpha \cdot \varepsilon) = \mathcal{L}(\alpha),\\
\mathcal{L}(\emptyset \cdot \alpha) &= \mathcal{L}(\alpha \cdot \emptyset) = \emptyset,\\
\mathcal{L}(\emptyset \mid \alpha) &= \mathcal{L}(\alpha \mid \emptyset) = \mathcal{L}(\alpha),\\
\mathcal{L}(\emptyset^*) &= \{\varepsilon\},\\
\mathcal{L}(\emptyset^+) &= \emptyset,\\
\mathcal{L}(\varepsilon^+) &= \mathcal{L}(\varepsilon^*) = \{\varepsilon\},\\
\mathcal{L}(\alpha \mid \alpha) &= \mathcal{L}(\alpha),\\
\mathcal{L}(\alpha \mid \beta) &= \mathcal{L}(\beta \mid \alpha),\\
\mathcal{L}((\alpha^*)^*) &= \mathcal{L}(\alpha^*),\\
\mathcal{L}((\alpha^+)^+) &= \mathcal{L}(\alpha^+),\\
\mathcal{L}((\alpha^+)^*) &= (\mathcal{L}(\alpha^*)^+) = \mathcal{L}(\alpha^*),\\
\mathcal{L}((\alpha^* \cdot \beta^*)^*) &= \mathcal{L}((\alpha^* \mid \beta)^*) = \mathcal{L}((\alpha \mid \beta)^*),\\
\mathcal{L}(\alpha^* \cdot \alpha^*) &= \mathcal{L}(\alpha^*),\\
\mathcal{L}(\alpha^* \cdot \alpha^+) &= \mathcal{L}(\alpha^+ \cdot \alpha^*) = \mathcal{L}(\alpha^+),\\
\mathcal{L}((\alpha \mid \beta) \cdot \gamma) &= \mathcal{L}((\alpha \cdot \gamma) \mid (\beta \cdot \gamma)),\\
\mathcal{L}(\alpha \cdot (\beta \mid \gamma)) &= \mathcal{L}((\alpha \cdot \beta) \mid (\alpha \cdot \gamma)),\\
\mathcal{L}((\alpha \cdot \beta)^* \cdot \alpha) &= \mathcal{L}(\alpha \cdot (\beta \cdot \alpha)^*),\\
\mathcal{L}(\alpha^+ \mid \beta^*) &= \mathcal{L}(\alpha^* \mid \beta^+) = \mathcal{L}(\alpha^* \mid \beta^*),\\
\mathcal{L}((\alpha \mid \varepsilon)^+) &= \mathcal{L}(\alpha^+ \mid \varepsilon) = \mathcal{L}((\alpha^+ \mid \varepsilon)^+) = \mathcal{L}(\alpha^*)\\
\mathcal{L}(\alpha \mid (\alpha \cdot \beta)) &= \mathcal{L}(\alpha \cdot (\varepsilon \mid \beta)),\\
\mathcal{L}((\alpha \cdot \beta) \mid \beta) &= \mathcal{L}((\alpha \mid \varepsilon) \cdot \beta).
\end{aligned}
$$

Der Beweis der Korrektheit bleibt Ihnen als Übung überlassen (Aufgabe 3.16).

Index