THEORETISCHE INFORMATIK INTUITIVES SKRIPT

FLORIAN ESSER

Inhaltsverzeichnis

1. Einführung

1.1. **Was ist dieses Dokument?** Die theoretische Informatik leiht leiht sich in ihrer Herangehensweise viele Konzepte aus der Mathematik. Es wird ein recht formaler Ansatz von rigiden Definitionen und Beweisen verwendet. Wer nicht vertraut mit dieser Herangehensweise ist, kann hiervon schnell eingeschüchtert sein. In dieser Datei möchte ich die formalen Aspekte der Vorlesung ausführlicher in Worten erklären und auf einige Stolperfallen hinweisen, in die man leicht fallen kann. Die Hoffnung ist, dass dies den Einstieg in die theoretische Informatik erleichtern kann.

Gleichzeitig bin ich selber aber auch ein Mathematiker, der gerne auf technische Details einzelner Definitionen achtet. Ich möchte es mir nicht nehmen lassen, auf einzelne Feinheiten einzugehen oder mal Fragen zu beantworten, die mir während des Schreibens durch den Kopf kommen. Dabei kann es „aus Versehen“ passieren, dass dieser Text an Stellen auf Details eingeht, die für diese Vorlesung garantiert nicht so tief verstanden werden müssen. Ich werde diese Stellen durch graue Hinterlegung kennzeichnen. Wer sich nur eine intuitive Erklärung der Begriffe wünscht, kann diese Kästen gerne überspringen. Umgekehrt können neugierige Studierende, die den Vorlesungsstoff bereits verstanden haben, vielleicht trotzdem Interesse an den grauen Kästen finden, da diese vielleicht auf Dinge eingehen, die in der Vorlesung nicht behandelt wurden.

Dieser Text ist als mein persönliches Projekt zu betrachten. Professor Damm hat diesen Text nicht verfasst und es ist auch nicht garantiert, dass er den Text bereits in seiner Gänze gelesen hat. Im Konfliktfall ist daher auf das zu hören, was Professor Damm sagt und nicht auf das, was in diesem Text steht. Ein weiterer Nachteil davon, dass dies ein rein persönliches Projekt ist, ist dass ich nicht garantieren kann, dass das Dokument am Ende die gesamte Vorlesung behandeln wird. Aktuell habe ich erst bis Kapitel 3.7 geschrieben. Ich habe vor, den Text im Laufe des Semesters so weit zu ergänzen, dass am Ende der gesamte Inhalt der Vorlesung abgedeckt ist. Doch es wird sich zeigen, ob ich die Zeit finde, die Abschnitte zu kommenden Themen zu schreiben. Ich glaube aber, dass eine gute Erklärung von nur der ersten Hälfte immerhin besser ist, als keine gute Erklärung. Deshalb (und auch, um mich zu motivieren, dieses Projekt auch endlich mal zu beenden) lade ich meinen Fortschritt jetzt schon einmal hoch.

1.2. **Einleitung.** Die Informatik beschäftigt sich mit Maschinen, die Inputs verarbeiten und einen Output liefern. Häufig besteht dieser Output aus einem simplen Ja/Nein. In dieser Vorlesung werden wir solche Maschinen betrachten - losgelöst von dem physischen Aufbau der Maschinen. Wir werden auf rein abstrakter Ebene Maschinen definieren, die bestimmte Inputs erhalten und auf bestimmte Weise verarbeiten können. Wir werden verschiedene Modelle von Maschinen definieren und ihnen weitere Möglichkeiten zur Verarbeitung von Inputs geben. Dabei werden wir sehen, dass die fortgeschritteneren Maschinen zwar erweiterte Möglichkeiten haben, aber wir werden auch die Grenzen der neu definierten Maschinen betrachten.

Zunächst werden wir uns endliche Automaten anschauen. Diese bekommen eine Abfolge von Inputs und verarbeiten diese hintereinander. Dabei wird eine bestimmte Art Input in einem bestimmten Zustand des Automaten immer auf die gleiche Weise verarbeitet. Ein endlicher Automat hat also keinen „Verlauf“, der einen Einfluss auf die aktuelle Verarbeitung hat.

Wir werden dieses Konzept mithilfe von Kellerautomaten erweitern. Diese haben eine Art Speicher. Dieser ist jedoch recht limitiert. Bei den Speichern der Kellerautomaten kann stets nur auf das zuletzt gespeicherte Zeichen zurückgegriffen werden und es ist recht aufwändig, wieder an die „untersten“ Elemente im Speicher heranzukommen.

Danach werden wir uns mit Turing-Maschinen beschäftigen. Turing-Maschinen haben einen Speicher, in dem die gesamte Eingabe auf einmal einsehbar ist und sie können beliebige Stellen des Speichers lesen und ändern. Hier unterscheiden wir noch zwischen solchen Turing-Maschinen mit einem begrenzten Speicherplatz und solchen mit einem theoretisch unendlichen Speicherplatz.

Dies bringt uns dann zu wichtigen Erkentnissen über die Computer, die wir täglich nutzen. Eine wichtige Erkenntnis ist diejenige, dass auch moderne Computer zu nichts in der Lage sind, was nicht auch durch eine Turing-Maschine getan werden könnte. Lässt sich also zeigen, dass bestimmte Dinge von keiner Turing-Maschine getan werden können, zeigt uns dies Grenzen der modernsten Computer auf. Ein solches Beispiel ist das Halteproblem. Es gibt keine Turing-Maschine, die als Input den Code einer anderen Turing-Maschine erhält und als Output bestimmt, ob die Maschine in einer Endlosschleife endet. Für moderne Computer bedeutet das: Niemand kann ein Programm (in egal welcher Programmiersprache) schreiben, dass zu jedem als Input gegebenen Programm korrekt entscheidet, ob das Programm in eine Endlosschleife gerät. Dieser Art von Erkenntnissen werden wir in dieser Vorlesung begegnen.

1.3. **Symbole, Wörter und Sprachen.**

Symbol, Alphabet, Wort. Wir führen zunächst einige Begriffe ein. Ziel dieser Definitionen wird es sein, Eingaben in eine Maschine zu simulieren. Eine Maschine wird zwischen verschiedenen Eingaben unterscheiden können. Dafür nennen wir die Menge aller möglichen Eingaben das **Alphabet**. Eine einzelne Eingabe wird auch als **Zeichen** bezeichnet. Wir gehen davon aus, dass wir stets mindestens eine, aber nie unendlich viele mögliche Eingaben haben. Ein Alphabet muss daher eine nicht-leere, endliche Menge von Zeichen sein.

Wir können Zeichen als Strings der Länge 1 wie a oder 0 verstehen. Ein Alphabet ist dann zum Beispiel $\{0, 1\}$. Ein solches Alphabet aus zwei Zeichen wird auch **binäres Alphabet** genannt.

Wir wollen unseren Maschinen auch mehrere Eingaben hintereinander geben können. Auf diese Weise können wir längere Strings interpretieren. Wir werden einen String von Zeichen so interpretieren, dass zunächst das linkeste Zeichen eingegeben wird, danach das zweitlinkeste usw... Wir bezeichnen Strings aus Zeichen auch als **Wörter**

Besonders hevorgehoben sei das leere Wort ε. Wir wollen auch die Möglichkeit haben, der Maschine keine Eingabe zu geben. Dafür nutzen wir den String der Länge 0.

Diese Interpretation von Strings gibt der Konkatenation eine besondere Bedeutung. Wenn wir zwei Strings - also zwei Folgen von Eingaben - aneinanderhängen, können wir dies interpretieren, als bekäme die Maschine zunächst den ersten String und dann den zweiten String als Eingabe. Eine Konkatenation kann also als eine Art Hintereinanderausführung (von links nach rechts) betrachtet werden.

Konkatenieren wir 01101101 und 00, erhalten wir 0110110100. Konkatenation mit dem leeren Wort ändert ein Wort nicht:

$$\varepsilon \cdot a = a$$

$$bc \cdot \varepsilon = bc$$

Diejenigen, die sich mit Gruppentheorie auskennen, erkennen das leere Wort als neutrales Element der Konkatenation.

Konkatenationen eines Wortes mit sich selbst schreiben wir als Potenzen. Die Konkatenation vv schreiben wir also auch als v^2. Allgemein nutzen wir v^n, wenn wir das Wort v genau n mal hintereinanderschreiben wollen. Es ist also zum Beispiel:

$$(abc)^3 = abcabcabc$$

Häufig interessiert es uns, ob ein bestimmtes Wort in einem anderen Wort vorkommt. Ist dies der Fall, sprechen wir von einem **Teilwort**. Insbesondere solche Teilwörter, die am Anfang des Wortes vorkommen (**Präfixe**), und Teilwörter, die am Ende eines Wortes vorkommen (**Suffixe**), sind für uns besonders relevant.

Das soeben konstruierte Wort *abcabcabc* hat also zum Beispiel *ab* als Präfix und *cabc* als Suffix. Es hat zum Beispiel *cabca* als Teilwort:

$$ab\mathbf{cabca}bc$$

Definition endlicher Sprachen. Im Folgenden werden wir jede Menge von Wörtern als **Sprache** bezeichnen. Die Sprache, die alle Wörter enthält, die sich aus einem bestimmten Alphabet Σ bilden lassen, bezeichnen wir als die **Kleensche Hülle** von Σ und schreiben sie als Σ^*. Die Sprache, die alle nichtleeren Wörter enthält, die sich aus einem bestimmten Alphabet Σ bilden lassen, bezeichnen wir als die **positive Hülle** von Σ und schreiben sie als Σ^+. Offenbar erhalten wir Σ^+, indem wir aus Σ^* das leere Wort entfernen.

Sprachen sind zum Beispiel die leere Sprache $\emptyset$, die Sprache, die nur das leere Wort enthält $\{\varepsilon\}$ (man beachte, dass diese Sprache nicht die leere Sprache ist) oder Sprachen wie $\{\varepsilon, aa, abc\}$ oder $\{01, 0101, 010101, 01010101, 0101010101, \dots\}$. Es ist:

$$\{1\}^* = \{\varepsilon, 1, 11, 111, 1111, 11111 \dots\}$$
$$\{1\}^+ = \{1, 11, 111, 1111, 11111 \dots\}$$

Längenlexikographische Ordnung. Haben wir eine Sprache definiert, wollen wir auch wissen, welche Elemente sie enthält. Die Elemente einer endlichen Sprache lassen sich leicht in eine Liste schreiben, doch auch bei unendlichen Sprachen hätten wir gerne eine „unendlich lange Liste“, in der alle Elemente der Sprache vorkommen. Genauer: Zu einer Sprache L suchen wir eine Funktion $f : \mathbb{N} \to L$ mit $L = \{f(1), f(2), \dots\}$. Eine solche Funktion nennen wir **Aufzählung** der Sprache L. Auch wenn wir im Folgenden eine Aufzählung ohne Mehrfachnennung finden werden, sind Mehrfachnennungen bei Aufzählungen grundsätzlich erlaubt.

Eine Idee, zum finden einer Aufzählung, wäre es, eine Ordnung auf der Menge der Wörter zu definieren. Dies könne zum Beispiel die alphabetische Ordnung sein. Wir könnten mit dem alphabetisch ersten Wort starten und danach das alphabetisch nächste Wort aufzählen. Hierbei stoßen wir aber auf ein Problem: Wir zählen das Element a auf. Als nächstes folgt aa. An n-ter Stelle zählen wir das Wort a^n auf. Wir kommen also nie zum Wort b. Deshalb werden so nicht alle Worte aufgezählt, sodass die alphabetische Ordnung (auch lexikographische Ordnung genannt) sich nicht zur Definition einer Aufzählung eignet.

Dieses Problem können wir mit der längen-lexikographischen Ordnung umgehen. Dafür sortieren wir die Wörter zunächst nach ihrer Länge. Dann werden alle Wörter gleicher Länge „alphabetisch“ sortiert. Wir können so die Sprache $\{a, b\}^*$ sortieren und erhalten:

$$\{\underbrace{\epsilon}_{\text{Länge 0}}, \underbrace{a, b}_{\text{Länge 1}}, \underbrace{aa, ab, ba, bb}_{\text{Länge 2}}, \underbrace{aaa, aab, aba, abb, baa, bab, bba, bbb}_{\text{Länge 3}}, \underbrace{\dots}_{\text{Länge } \geq 4}\}$$

Hierbei sollte ein Detail nicht verloren gehen: Um eine längenlexikographische Ordnung erhalten zu können, benötigen wir also erst einmal ein Verständnis davon, was es heißt, dass eine Liste „alphabetisch“ geordnet ist. Genauer: Wir benötigen eine zunächst vorgegebene Ordnung auf der Menge der Zeichen bevor wir die Wörter längenlexikographisch Ordnen können. Eine längenlexikographische Ordnung ist daher auch nicht eindeutig, sondern ist immer auch abhängig von der zugrundeliegenden Ordnung der Zeichen. Eine andere Ordnung der Zeichen kann auch eine andere längenlexikographische Ordnung mit sich bringen.

Ist in der zugrundeliegenden Ordnung $a < b$, so ist in der längenlexikographischen Ordnung der Wörter

$$a < b < aa < ab$$

Ist in der zugrundeliegenden Ordnung $b < a$, so ist in der längenlexikographischen Ordnung der Wörter

$$b < a < ab < aa$$

1.4. **Operationen auf Sprachen.**

Produkt von Sprachen. Wir haben die Operation der Konkatenation bisher nur auf der Ebene der Wörter definiert. Wir können nun aber auch eine Art Konkatenation von Sprachen definieren. Für diese wollen wir beliebig ein Wort aus der ersten Sprache mit einem Wort aus der zweiten Sprache konkatenieren können. Die Menge aller möglichen Ergebnisse liefert wieder eine Sprache.

Offenbar kommt es bei diesem Produkt auf die Reihenfolge an. Ist zum Beispiel $A = \{a, aa, ab\}, B = \{b, ba, bb\}$, liefert das Produkt AB ein anderes Ergebnis als BA:

$$AB = \{ab, aab, aba, abb, aaba, aabb, abba, abbb\}$$
$$BA = \{ba, baa, bab, bba, baaa, baab, bbaa, bbab\}$$

Die Mengen enthalten nur 8 statt 9 Wörtern, da $a(bb) = (ab)b$ und $b(aa) = (ba)a$ ist.

Nullement und Einselement. Die leere Menge und die Menge $\{\varepsilon\}$ haben jeweils eine besondere Rolle für das Produkt von Mengen. Ein Produkt mit der leeren Menge ergibt immer die leere Menge. Multiplikation mit $\{\varepsilon\}$ verändert eine Menge nicht.

$$A \cdot \emptyset = \emptyset \cdot A = \emptyset$$
$$A \cdot \{\varepsilon\} = \{\varepsilon\} \cdot A = A$$

Diejenigen, die sich mit Gruppentheorie auskennen, erkennen, dass sich $\emptyset$ und $\{\varepsilon\}$ wie eine 0 und eine 1 in einem Ring verhalten.

Potenzierung von Sprachen. Besonderes Augenmerk legen wir auf das Produkt einer Sprache mit sich selbst. Dies bringt uns zum Begriff der **Potenz**. Das n-fache Produkt einer Sprache mit sich selbst schreiben wir als n-te Potenz einer Sprache.

Offenbar gilt hier auch das Potenzgesetz $L^n L^m = L^{n+m}$. Wollen wir L^0 definieren, so wünschen wir uns, dass auch $L^0 L^n = L^{0+n} = L^n$ gilt. Also muss L^0 wie eine Einheit der Multiplikation von Sprachen wirken. Wie wir oben gesehen haben, erfüllt $\{\varepsilon\}$ diese Eigenschaft. Dies motiviert $L^0 := \{\varepsilon\}$. Wir erhalten zum Beispiel:

$$\emptyset^0 = \{\varepsilon\}$$

$$\emptyset^{17} = \emptyset$$
$$\{a, aa, ab\}^2 = \{aa, aaa, aab, aba, aaaa, aaab, abaa, abab\}$$

Kleene-Star und positive Hülle. Die Kleensche Hülle haben wir über einem Alphabet bereits definiert als die Menge von Strings bestehend aus Zeichen des Alphabets. In Analogie wollen wir nun auch die Kleensche Hülle einer Sprache definieren. Diese ist also die Menge aller Strings, die wir erhalten, wenn wir beliebig Wörter aus dieser Sprache aneinanderhängen können.

Genauer: Wir können auch kein Wort aus der Sprache nehmen. Die Kleensche Hülle von L muss also ε enthalten. Wir können genau ein Wort nehmen. Die Kleensche Hülle muss also alle Wörter aus L enthalten. Wir benötigen auch alle Möglichkeiten, zwei Wörter aus L zu kombinieren. Es müssen also alle Wörter aus L^2 enthalten sein. Für jedes n muss die Kleensche Hülle alle Möglichkeiten, n Wörter aus L zu kombinieren, enthalten - es muss also L^n in der Kleenschen Hülle enthalten sein.

Insgesamt können wir die Kleensche Hülle also als die Vereinigung $L^* \coloneqq \bigcup_{n=0}^{\infty} L^n$ definieren.

Es ist zum Beispiel:

$$\{a, aa, ab\}^* = \{\varepsilon, a, aa, ab, aaa, aab, aba, aaaa, aaab, aaba, abaa, abab, \dots\}$$

Hier sind nur die Wörter mit bis zu vier Zeichen aufgeführt.

Für Alphabete haben wir die positive Hülle definiert als die Menge aller nichtleeren Wörter. Wir bilden über Sprachen also die Menge aller Produkte von Wörtern aus der Sprache mit mindestens einem Faktor. In der Definition der Vereinigung können wir dabei den Laufindex einfach bei 1 beginnen lassen: $L^+ \coloneqq \bigcup_{n=1}^{\infty} L^n$

ACHTUNG! Da wir die positive Hülle über Alphabeten als „Menge aller nichtleeren Wörter“ bezeichnet haben, ist es verführerisch, davon auszugehen, dass auch positive Hüllen von Sprachen das leere Wort auch nicht enthalten können. Dies ist jedoch nicht der Fall. Enthält die Sprache selbst bereits das leere Wort, so ist nach gerade aufgestellter Definition das leere Wort auch Element der positiven Hülle.

$$\emptyset^+ = \emptyset$$
$$\{\varepsilon\}^+ = \{\varepsilon\}$$
$$\{a, aa, ab\}^+ = \{a, aa, ab, aaa, aab, aba, aaaa, aaab, aaba, abaa, abab, \dots\}$$

Mengenoperationen. Folgende Operationen sind auch häufig nützlich. Zum Beispiel haben wir die Vereinigung oben bereits einmal genutzt:

Vereinigung: Die Sprache $L_1 \cup L_2$ enthält alle Wörter, die in L_1 oder in L_2 sind.

Schnitt: Die Sprache $L_1 \cap L_2$ enthält alle Wörter, die in L_1 und in L_2 sind.

Differenz: Die Sprache $L_1 \setminus L_2$ enthält alle Wörter, die in L_1, aber nicht in L_2 sind.

Symmetrische Differenz: Die Sprache $L_1 \Delta L_2$ enthält alle Wörter, die in L_1 oder in L_2, aber nicht in beiden Sprachen sind.

Komplement: Die Sprache $\overline{L}$ enthält alle Wörter, die nicht in L sind.

Ein Komplement lässt sich nur bilden, wenn klar ist, über welchem Alphabet wir arbeiten. Ist unser Alphabet $\{a\}$, so ist

$$\overline{\{a\}} = \{\varepsilon, aa, aaa, \dots\}$$

Ist unser Alphabet $\{a, b\}$, so ist

$$\overline{\{a\}} = \{\varepsilon, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\}$$

Um diese Operationen in Aktion zu sehen, betrachten wir zum Beispiel die Sprachen $L_1 = \{a, b\}^2, L_2 = \{a, b\}^4, L_3 = \{aaaa, abba, baab, bbbb\}$. Dann erhalten wir:

$$L_1 \cup L_3 = \{aa, ab, ba, bb, aaaa, abba, baab, bbbb\}$$
$$L_2 \cap L_3 = \{aaaa, abba, baab, bbbb\}$$
$$L_2 \setminus L_3 = \{aaab, aaba, aabb, abaa, abab, abbb, baaa, baba, babb, bbaa, bbab, bbba\}$$
$$L_3 \setminus L_2 = \emptyset$$
$$L_2 \Delta L_3 = \{aaab, aaba, aabb, abaa, abab, abbb, baaa, baba, babb, bbaa, bbab, bbba\}$$
$$\overline{L_1} = \{\varepsilon, a, b, aaa, aab, aba, abb, baa, bab, bba, bbb, \dots\}$$

Gruppentheorie. Wie bereits an einigen Stellen angesprochen, können wir die Konkatenation von Wörtern gruppentheoretisch auffassen. Dies liegt daran, dass die Verknüpfung assoziativ ist. Für Wörter u, v, w gilt $u(vw) = (uv)w$. Außerdem liegt mit ε ein neutrales Element vor. Es gilt also $w\varepsilon = \varepsilon w = w$ für alle Wörter w. Mangels inverser Elemente können wir Σ^* jedoch nicht als Gruppe, sondern nur als **Monoid** auffassen.

Das Alphabet Σ erzeugt das Monoid Σ^*. Wir können sogar genauer sagen, dass Σ^* **frei** von Σ erzeugt wird. Das bedeutet, dass es zu einem Wort w *genau* eine Folge von Zeichen $w_1, w_2, \dots, w_n$ mit $w_1 w_2 \dots w_n = w$ gibt.

Homomorphismen. Homomorphismen sind bestimmte Abbildungen zwischen Sprachen. Am besten verstehen wir einen Homomorphismus, indem wir die Bilder der einzelnen Zeichen betrachten. Wir können das Bild eines Wortes ermitteln, indem wir die Bilder der Zeichen in der richtigen Reihenfolge aneinanderhängen. Die Bilder der Zeichen können dabei beliebige Worte der Zielsprache sein. Hierbei ist es erlaubt, dass zwei Zeichen das gleiche Bild haben („Homomorphismen müssen nicht injektiv sein.“), ein oder mehrere Zeichen können auch auf das leere Wort abgebildet werden und es ist auch erlaubt, dass ein Zeichen auf ein Wort abgebildet wird, das aus mehr als einem Zeichen besteht.

Injektive Homomorphismen (also solche, bei denen keine zwei Worte auf das gleiche Wort abgebildet werden) können als eine Einbettung verstanden werden. Wir finden quasi unsere Ausgangssprache in der Zielsprache wieder, wenn wir alle Wörter betrachten, die sich im Bild des Homomorphismusses befinden.

Nicht-injektive Homomorphismen können als ein absichtliches Vergessen von Informationen verstanden werden. Bilden wir zum Beispiel zwei Zeichen auf ein einzelnes Zeichen ab, können wir das so verstehen, dass wir nicht mehr dazwischen unterscheiden wollen, welches dieser Zeichen ursprünglich vorlag.

Über dem Alphabet $\{a, b\}$ reicht es aus, die Bilder von a und b zu kennen, um das Bild eines Wortes zu bestimmen:

$$\phi_1(a) = c, \phi_1(b) = d \implies \phi_1(abba) = cddc$$
$$\phi_2(a) = c, \phi_2(b) = c \implies \phi_1(abba) = cccc$$
$$\phi_3(a) = cdd, \phi_3(b) = dcccd \implies \phi_1(abba) = cdddcccddcccdcdd$$
$$\phi_4(a) = c, \phi_4(b) = \varepsilon \implies \phi_1(abba) = cc$$
$$\phi_5(a) = a, \phi_5(b) = aa \implies \phi_5(aa) = \phi_5(b) = aa$$

Reflexion. Die Operation, die ein Wort entgegen nimmt und die Reihenfolge der Zeichen in dem Wort umdreht, nennen wir **Reflexion**. Die Reflexion operiert also durch $(w_1 w_2 \dots w_n)^R = w_n \dots w_2 w_1$.

Die Reflexion kann verstanden werden als eine Abbildung $\varphi : \Sigma^* \to \Sigma^*$ mit $\varphi(a) = a$ für alle Zeichen a und $\varphi(vw) = \varphi(w)\varphi(v)$ für alle Wörter v, w. Es ist insbesondere auch die einzige Abbildung mit diesen beiden Eigenschaften. Das Fordern dieser Eigenschaften kann also auch als Definition der Reflexion verstanden werden.

Es ist zum Beispiel:

$$(abc)^R = cba$$
$$(HelloWorld!)^R = !dlorWolleH$$

Die Reflexion ist selbstinvers. Zweifache Anwendung liefert die ursprüngliche Eingabe.

$$((HelloWorld!)^R)^R = HelloWorld!$$

Wir können die Reflexion einer Sprache bilden, indem wir jedes Wort in der Sprache reflektieren.

$$\{abc, HelloWorld!\}^R = \{cba, !dlorWolleH\}$$

Quotient. Die Operationen zu Quotient und Shuffle werden auf den Folien zum Skript zwar erst im kommenden Abschnitt definiert, doch wir nehmen sie hier jetzt bereits auf.

Die Quotientenoperation kann als ein Versuch gesehen werden, die Produktoperation umzukehren. Für gewöhnlich ändern wir nichts, wenn wir „mal x durch x“ rechnen. Dies gibt uns einen Hinweis darauf, wie wir es definieren wollen, wie wir eine Sprache durch ein Wort „teilen“. Nehmen wir zu einer Sprache A das Produkt $A \cdot \{x\}$, dann sollte der Quotient $(A \cdot \{x\})/x$ wieder A ergeben. Bei der Bildung von $A \cdot \{x\}$ hängen wir an jedes Wort aus A das Wort x als Suffix an. Das Teilen durch das Wort x muss dann also von jedem Wort den Suffix x entfernen. Dies kann leider nicht ganz als Definition für allgemeine Sprachen verändert werden. Wollen wir im allgemeinen ein Wort x aus einer Sprache B herausteilen (insbesondere muss B nicht aus einer $A \cdot \{x\}$ Operation entstanden sein), so ist nicht immer klar, dass auch jedes Wort der Sprache x als Suffix hat. In diesem Fall lassen wir die Wörter, die x nicht als Suffix haben, einfach verfallen. Wir konstruieren B/x also, indem wir für jedes Wort in B das Wort verwerfen, wenn es x nicht als Suffix hat und sonst den Suffix x entfernen und das Ergebnis in B/x aufnehmen.

Daher gilt zwar immer $(A \cdot \{x\})/x = A$, aber nicht notwendigerweise auch $(A/x) \cdot \{x\} = A$. Es gilt aber tatsächlich $(A/x) \cdot \{x\} \subseteq A$.

Den Quotienten A/B von zwei Sprachen definieren wir analog. Für jedes Wort a in A gehen wir alle möglichen Wörter b in B durch. Ist b ein Suffix von a, entfernen wir diesen Suffix und fügen das Ergebnis A/B hinzu. Gibt es zwei Wörter b_1, b_2 in B, die beide Suffix von $a \in A$ sind, wird natürlich sowohl a ohne den Suffix b_1 als auch a ohne den Suffix b_2 zu A/B hinzugefügt.

Es gilt immernoch $A \subseteq (A \cdot B)/B$. Die Richtung $(A \cdot B)/B \subseteq A$ muss aber nicht mehr gelten. Ist zum Beispiel $A = \{a\}$ und $B = \{\varepsilon, b\}$, dann ist $A \cdot B = \{a, ab\}$. Dann ist

$(A \cdot B)/B = \{a, ab\}$. Wir erhalten ab, indem wir von ab den Suffix ε entfernen. Somit enthält $(A \cdot B)/B$ ein Wort, das nicht in A enthalten war.

Wie beim Quotienten mit einem einzigen Wort muss $A \subseteq (A/B) \cdot B$ nicht gelten. Beim Quotienten mit einer Sprache kann aber auch die Richtung $(A/B) \cdot B \subseteq A$ schief gehen. Ist zum Beispiel $A = \{ac, bd\}$ und $B = \{c, d\}$, dann ist $(A/B) = \{a, b\}$ und $(A/B) \cdot B = \{ac, ad, bc, bd\}$, wobei offenbar einige Worte dazugekommen sind. Der Quotient von Sprachen kehrt das Produkt von Sprachen also nicht mehr so gut um. Es gibt aber auch keine andere Operation, die das Produkt von Sprachen wirklich umkehrt.

Shuffle. Das Shuffleprodukt ist eine weitere Operation, die wir betrachten wollen. Wollen wir das Shuffleprodukt von zwei Wörtern bilden, ist das Ergebnis eine Menge von Wörtern. Den Namen hat das Produkt vom Mischen von Karten. Wir stellen uns vor, wir haben wir zwei Kartenstapel in einer festen Reihenfolge. Wenn wir diese zusammenmischen, erhalten wir einen Stapel, der die Karten beider Karten enthält. Bei der Mischmethode, die als Shuffle bekannt ist, wird dabei die interne Reihenfolge eines der Stapel nicht verändert. Lag Karte a im ursprünglichen Stapel über Karte b, so ist dies auch im zusammengemischten Stapel der Fall.

Dies tut auch das Shuffleprodukt von Wörtern. Gegeben sind zwei Wörter (zum Beispiel abc und de). Wir generieren nun Wörter, die genau die Zeichen beider Wörter zusammen enthalten (also Wörter, aus a, b, c, d, e, in denen jedes dieser Zeichen genau einmal vorkommt), die die interne Reihenfolge der ursprünglichen Wörter beibehalten. In unserem Beispiel kommt a vor b, b vor c und d vor e. Das Shuffleprodukt ist die Menge aller Wörter, die so entstehen können. Wir können so

$$abcde, abdce, adbce, dabce, abdec, adbec, dabec, adebc, daebc, deabc$$

erhalten. Nicht möglich sind zum Beispiel $bdace$ (b kommt vor a) oder $abecd$ (e kommt vor d). Wir schreiben das Shuffleprodukt der Worte u und v als $u\#v$

Das Shuffleprodukt zweier Sprachen A und B besteht aus allen Wörtern w, zu denen es ein Wort u aus A und ein Wort v aus B gibt, sodass w ein mögliches Shuffle von u und v ist. (Also, bei denen $w \in u\#v$ ist. Wir schreiben das Shuffleprodukt von A und B als $A\#B$.

Die formale Definition des Shuffleprodukts wird angegeben als:

$$L\#K \coloneqq \{x_1 z_1 x_2 z_2 \dots x_n z_n : x_i, z_i \in \Sigma^* \text{ für } 1 \leq i \leq n, x_1 \dots x_n \in L, z_1 \dots z_n \in K\}$$

Auf den ersten Blick kann es so aussehen, als würde diese Definition nicht mit der oben genannten Erklärung übereinstimmen. Hier scheinen sich Zeichen aus L und Zeichen aus K viel häufiger abzuwechseln. Es sei aber darauf hingewiesen, dass die x_i und z_i nur Elemente von Σ^* und nicht aus Σ sein müssen. Sie können also mehr als ein Zeichen enthalten und sie können auch das leere Wort sein. Wollen wir zum Beispiel $dabce$ aus abc und de erhalten, können wir

$$x_1 = \varepsilon, z_1 = d, x_2 = abc, z_2 = e$$

wählen und erhalten das gewünschte Wort. Dies ist nicht einmal die einzige Möglichkeit. Wir könnten auch

$$x_1 = \varepsilon, z_1 = d, x_2 = a, z_2 = \varepsilon, x_3 = b, z_3 = \varepsilon, x_4 = c, z_4 = e$$

wählen. Egal, wie wir es wählen, sehen wir, dass die Definition tatsächlich die oben beschriebenen Möglichkeiten zulässt. Es können mehrere Zeichen aus einem Wort aufeinander folgen und das Wort muss auch nicht mit einem Zeichen aus L anfangen oder mit einem Zeichen aus K enden.

1.5. **Sprachfamilien.** Wir wollen im Folgenden Eigenschaften von Sprachen betrachten. Konkreter wollen wir zu verschiedenen Eigenschaften, die wir untersuchen, bestimmen, welche Sprachen diese Eigenschaften erfüllen. Uns interessiert also die Menge aller Sprachen mit einer bestimmten Eigenschaft. (Zu dieser Formulierung folgt ein Exkurs am Ende dieses Abschnitts.)

Bestimmte „einfache“ Eigenschaften interessieren uns besonders. Wir wollen, dass die Menge der Sprachen mit dieser Eigenschaft noch übersichtlich bleibt. Dafür definieren wir den Begriff der **Sprachklasse**. Eine Menge L von Sprachen ist eine Sprachklasse, wenn:

- In jeder Sprache in L nur endlich viele verschiedene Zeichen verwendet werden.
- Die Menge aller in L verwendeten Zeichen abzählbar ist.
- Es mindestens eine nichtleere Sprache in L gibt.

Eine Eigenschaft, die uns im Folgenden noch länger interessieren wird, ist die Eigenschaft der **regulären Mengen**. Intuitiv wollen wir eine Menge als „regulär“ bezeichnen, wenn sie sich durch einen recht einfachen Bauplan zusammensetzen lässt. Hierfür müssen wir klären, was wir als „einfach“ ansehen.

Zunächst bezeichnen wir eine Menge als regulär, wenn sie nur endlich viele Wörter enthält. Dann bezeichnen wir die Anwendung der Kleenschen Hülle als einen „einfachen“ Bauschritt. Die Kleensche Hülle einer regulären Menge ist also wieder regulär. Ebenso sind Vereiniung und Konkatenation einfache Bauschritte. Die Vereiniung oder Konkatenation zweier regulärer Mengen soll also ebenfalls wieder regulär sein. Jede Menge, die wir auf diese Weise konstruieren können, bezeichnen wir als „regulär“. Jede Menge, die so nicht konstruiert werden kann, bezeichnen wir nicht als regulär.

Bei der Menge der regulären Mengen, genannt *Reg*, handelt es sich also um die Sprachfamilie aller Sprachen, die sich aus regulären Sprachen durch einen endlichen „Bauplan“ aus Vereinigung, Konkatenation und Bildung der Kleenschen Hülle zusammensetzen lassen.

Per Definition ist *Reg* abgeschlossen unter Bildung von Vereinigung, Konkatenation und Kleenscher Hülle. Wir werden aber sehen, dass *Reg* auch noch unter vielen weiteren Operationen, wie der Bildung von Quotienten oder der Bildung des Shuffle-Produkts abgeschlossen ist.

Die Formulierung „Die Menge aller Sprachen mit einer bestimmten Eigenschaft“ ist ein wenig unpräzise gewählt. Bei Sprachen handelt es sich um Mengen. Eine Menge von Sprachen ist also eine Menge, deren Elemente selber Mengen sind. Solche Formulierungen führen gelegentlich zu Paradoxa. Bekannt ist vielleicht das Paradoxon der „Menge aller Mengen, die sich nicht selbst enthalten“. Eine solche Menge kann es nicht geben. (Enthält diese Menge sich selbst?)

Wir können dieses Problem lösen, indem wir Mengen eine in Stufen eingeteilte Hierarchie geben. Mengen, deren Elemente selbst keine Mengen sind, nennen wir **Mengen erster Stufe**. Mengen, deren Elemente Mengen erster Stufe sind, nennen wir **Mengen zweiter Stufe** oder auch **Klassen**. Präziser wäre es also gewesen, von der „*Klasse* aller Sprachen mit einer bestimmten Eigenschaft“ zu sprechen. Ebenso sollte in der Definition einer Sprachfamilie von einer Klasse gesprochen werden und wir sollten auch von der Klasse der regulären Mengen anstatt von der Menge der regulären Mengen sprechen.

Technisch gesehen können wir die Klasse *Reg* der regulären Mengen so, wie wir sie definiert haben, nicht als Sprachklasse bezeichnen. Eine Sprachklasse hat die Voraussetzung, dass die Menge aller vorkommenden Zeichen abzählbar ist. Offenbar können wir uns überabzählbare Mengen von Zeichen vorstellen. Sei S eine solche Menge. Zu jedem Zeichen

$s \in S$ gibt es die endliche Sprache $\{s\}$. Da diese Sprache endlich ist, muss sie in *Reg* enthalten sein. Es kommt also jedes der überabzählbar vielen Zeichen aus S in *Reg* vor. Dies widerspricht der Definition einer Sprachfamilie.

Wir passen unsere Definition daher ein wenig an: Sei $\Gamma = \{a_1, a_2, a_3, \dots\}$ eine abzählbar unendliche Menge von Zeichen. Wir definieren *Reg* als die kleinste Sprachklasse, die alle endlichen Sprachen mit Zeichen aus Γ enthält und abgeschlossen ist unter endlich vielen Anwendungen von Vereinigung, Konkatenation und Kleenscher Hülle.

Dies führt aber nun dazu, dass wir viele Sprachen nicht mehr als regulär bezeichnen, die wir eigentlich als regulär bezeichnen wollen. Die Sprache $\{b\}$ ist zum Beispiel endlich und würde nach unserer ursprünglichen Definition als regulär bezeichnet werden. Da das Zeichen b in Γ aber gar nicht vorkommt, ist die Sprache nach dieser Definition nicht regulär. Wir definieren also den Begriff „regulär“ ebenfalls neu:

Zu einer Sprache L über dem Alphabet Σ betrachte eine injektive Abbildung $\varphi : \Sigma \to \Gamma$. Wir können diese zu einem Homomorphismus $\hat{\varphi} : \Sigma^* \to \Gamma^*$ fortsetzen. Ist das Bild $\hat{\varphi}(L)$ in *Reg* enthalten, bezeichnen wir L als regulär. (Es ist recht ersichtlich, dass die Frage, ob $\hat{\varphi}(L) \in Reg$ ist, unabhängig von der Wahl von φ ist, solange φ injektiv ist.)

Diese Definition wirkt sehr kompliziert. Intuitiv können wir uns aber einfach vorstellen, dass wir nichts an der ursprünglichen Definition geändert haben. Mengen werden immer noch genau dann als regulär bezeichnet, wenn sie nach unserer ursprünglichen Definition als regulär bezeichnet wurden.

1.6. **Reguläre Ausdrücke.** Wir haben reguläre Sprachen als solche Sprachen definiert, die einen simplen Bauplan haben. Wir können diesen auch explizit angeben. Einen solchen Bauplan bezeichnen wir als **regulären Ausdruck**.

In unserer Definition war zunächst jede endliche Menge eine reguläre Menge. Für einen regulären Ausdruck, der diese Menge beschreibt, schreiben wir in Klammern alle Wörter der Menge hintereinander - getrennt durch ein | Zeichen. Die Menge $\{ab, bab, aaba\}$ wird also durch den regulären Ausdruck $(ab|bab|aaba)$ beschrieben.

Die Klasse der regulären Mengen ist ebenfalls abgeschlossen unter Bildung von Vereinigung, Konkatenation und Kleenscher Hülle. Wir benötigen also ebenfalls Notation für diese Bauschritte. Wird die Menge R durch den regulären Ausdruck r und die Menge S durch den regulären Ausdruck s beschrieben, so wird $R \cup S$ durch den regulären Ausdruck $(r|s)$, RS durch den regulären Ausruck (rs) und R^* durch den regulären Ausdruck $(r*)$ beschrieben.

Die leere Menge erhält als eigenes Symbol den regulären Ausdruck $\emptyset$.

Beispiel: Die Menge $\{\varepsilon\} \cup (\{ab\}^*\{a, bc\})$ wird bescrieben durch den regulären Ausdruck $((\varepsilon)|(((ab)*)(a|bc)))$. Zur besseren Übersicht lassen wir auch einige der Klammern weg, solange eindeutig bleibt, welche Menge beschrieben wird:

$$\varepsilon|((ab) * (a|bc))$$

Homomorphismen und reguläre Mengen. Wir können reguläre Ausdrücke nutzen, um Eigenschaften von *Reg* herzuleiten. Seien Σ, Σ' Alphabete und L eine Sprache über Σ. Sei $\varphi : \Sigma^* \to \Sigma'^*$ ein Homomorphismus. Dann ist $\varphi(L)$ eine reguläre Menge. Dies lässt sich folgendermaßen erkennen. Man betrachte einen regulären Ausdruck l, der L beschreibt. Ersetzen wir jedes Wort a, das in l vorkommt, durch das Bild $\varphi(a)$, erhalten wir einen regulären Ausdruck, der $\varphi(L)$ beschreibt. Da es also einen regulären Ausdruck gibt, der $\varphi(L)$ beschreibt, muss $\varphi(L)$ selbst regulär sein.

2. Endliche Automaten und reguläre Sprachen

2.1. **Definition endlicher Automaten.**

Intuition. Nun können wir endlich unser erstes Modell für einen Computer definieren. Unser Modell soll simpel sein. Wir erhalten eine Folge von Eingaben und geben als Ausgabe ein einziges Bit 0/1 aus. Es ist sinnvoll anzunehmen, dass unser Computer nur endlich viele Tasten hat - dass also die Menge der verschiedenen einzelnen Eingaben endlich ist. Ebenso gehen wir davon aus, dass unser Computer einige interne Zustände hat. Nach jeder Eingabe kann der Computer den Zustand wechseln (ein Startzustand wird festgelegt, in dem sich der Computer befindet, wenn es noch keine Eingabe gab). Der Zustand, in den gewechselt wird, soll dabei nur abhängig vom aktuellen Zustand und von der entgegengenommenen Eingabe sein - der Computer hat also zum Beispiel keinen Zugriff auf die bisher eingegangene Historie von Eingaben. Damit die Modelle übersichtlich bleiben (und weil dieses Modell sonst auch bereits viel zu mächtig wäre), hat der Computer auch nur endlich viele dieser Zustände. Zuletzt soll die Ausgabe 0/1 ebenfalls nur abhängig von dem Zustand sein, in dem der Computer nach der letzten Eingabe ist.

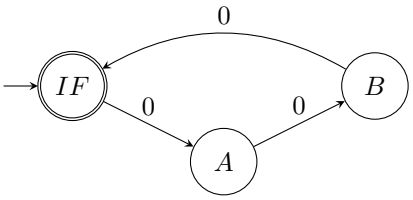
Einen solchen Computer nennen wir einen **endlichen Automaten** oder auch **deterministischen endlichen Automaten**, um ihn von später definierten Modellen abzugrenzen. Wir schreiben manchmal auch DFA für „deterministic finite automaton“. Wie bereits angekündigt, werden wir die im letzten Abschnitt entwickelte Theorie der Sprachen verwenden, um die Eingabefolgen zu beschreiben. Jede Taste des Computers bzw. jede mögliche Eingabe wird als ein Zeichen interpretiert. Die Menge aller Eingaben bildet dann also ein Alphabet. Eine Folge von Eingaben ist ein Wort. Wir sagen, dass ein endlicher Automat ein Wort w **akzeptiert**, wenn die Ausgabe des Automaten, wenn er mit w als Eingabe das Bit 1 ausgibt. Die Menge aller Wörter, die von einem bestimmten endlichen Automaten akzeptiert werden, bilden nach vorherigem Abschnitt eine Sprache. Wir bezeichnen diese Sprache als die vom Automaten **akzeptierte Sprache** oder **erkannte Sprache**. Zustände, die zur Ausgabe 1 führen, nennen wir auch **akzeptierende Zustände** oder **Endzustände**.

Das führt uns zu weiteren Fragen. Wenn ein Automat gegeben ist, welche Sprache wird von ihm erkannt? Gibt es Sprachen, die von keinem endlichen Automaten erkannt werden? Wenn ja, welche Sprachen werden denn überhaupt von irgendeinem Automaten erkannt?

Graphendarstellung. Wir können einen Automaten, wie wir ihn oben intuitiv beschrieben haben auch graphisch darstellen. Hierfür nutzen wir die graphischen Darstellungen, die aus der Graphentheorie bekannt sind und führen einen Knoten für jeden Zustand des Automaten ein. Für jeden Zustand q können wir für jedes Zeichen des Alphabets a prüfen, in welchen Zustand der Automat übergeht, wenn im Zustand q die Eingabe a gelesen wird. Wir fügen dem Graphen eine gerichtete Kante von q zu diesem Folgezustand hinzu und beschriften diese mit a. Wir kennzeichnen den Startzustand, indem wir eine Kante zu diesem Zustand hinzufügen, die keinen Startknoten hat. Zuletzt müssen wir die Ausgabe des Automaten kennzeichnen. Wir müssen also die akzeptierenden Zustände von den nicht akzeptierenden Zuständen unterscheiden können. Dafür zeichnen wir den Kreis jedes akzeptierenden Zustandes doppelt.

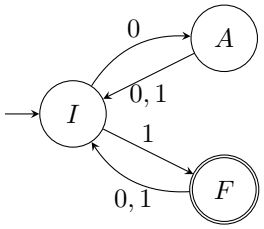
Dies reicht aus, um graphisch jede Funktionsweise des Automaten darzustellen.

Beispiele. Wir betrachten einige Beispielautomaten.



Wir können hier auch schnell sehen, wie sich dieser Automat verhält. Das einzige mögliche Zeichen, das eingegeben werden kann, ist eine 0. Die Wörter bestehen also nur aus 0en. Der Automat wechselt periodisch zwischen drei Zuständen, wobei der Startzustand der einzige akzeptierende Zustand ist. Ein Wort wird also genau dann akzeptiert, wenn die Anzahl der 0en im Wort durch drei teilbar ist. (Das leere Wort wird durch akzeptiert. Wir sehen hier 0 als eine durch 3 teilbare Zahl an.)

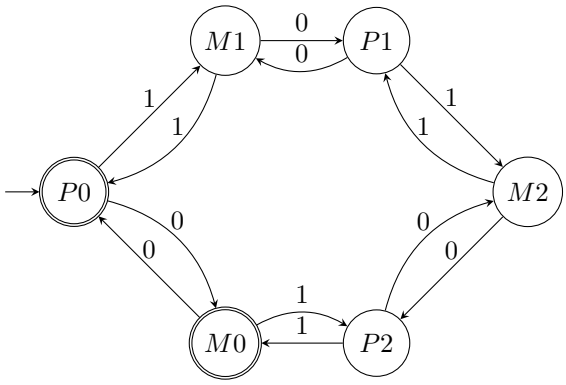
Hier ist ein weiteres Beispiel eines Automaten:



Wenn wir untersuchen wollen, welche Sprache dieser Automat akzeptiert, fällt zunächst auf, dass wir mit jedem gelesenen Zeichen den Zustand I abwechselnd verlassen und wieder betreten. Wurde eine gerade Anzahl an Zeichen gelesen, befinden wir uns in Zustand I - wurde eine ungerade Anzahl an Zeichen gelesen, befinden wir uns in Zustand A oder F. Da Zustand I nicht akzeptierend ist, wird also kein Wort gerader Länge akzeptiert. Wir sehen ebenfalls, dass bei Wörtern ungerader Länge nur das letzte Zeichen entscheidet, ob wir uns in Zustand A oder F befinden. Bei Wörtern ungerader Länge enden wir in Zustand A, wenn das letzte Zeichen eine 0 war und in Zustand F, wenn das letzte Zeichen eine 1 war. Der Automat akzeptiert also genau die Wörter ungerader Länge, die auf 1 enden.

Als nächstes Beispiel wollen wir einen Automaten konstruieren, der ein Wort aus 0en und 1en als Eingabe bekommt. Der Automat soll das Wort als Binärzahl interpretieren und genau dann akzeptieren, wenn die Zahl durch 3 teilbar ist. Hierfür nutzen wir folgendes Kriterium: Eine Binärzahl ist genau dann durch 3 teilbar, wenn ihre alternierende Quersumme durch 3 teilbar ist. Für die Zahl 1010111 erhalten wir als alternierende Quersumme zum Beispiel $1 - 0 + 1 - 0 + 1 - 1 + 1 = 3$ eine durch 3 teilbare Zahl. Also ist auch 1010111 durch 3 teilbar.

Wir merken, dass wir nicht das exakte Ergebnis der alternierenden Quersumme speichern müssen. Stattdessen reicht uns der Rest der alternierenden Quersumme modulo 3. Wir müssen ebenfalls speichern, ob das nächste Zeichen zum Ergebnis addiert oder subtrahiert werden muss. Diese Informationen können wir mit sechs Zuständen speichern. Wir nennen diese Zustände $P0, M0, P1, M1, P2, M2$. Das P bedeutet, dass das nächste Zeichen addiert wird - das M steht für Subtraktion. Die Zahl ist der Rest der bisherigen alternierenden Quersumme modulo 3. Ist bisher noch kein Zeichen eingelesen, ist der Rest der Quersumme 0 und wir wollen die nächste Ziffer addieren. Also muss $P0$ der Startzustand sein. Da genau die Zahlen akzeptieren wollen, bei denen der Rest der alternierenden Quersumme 0 ist, müssen $P0$ und $M0$ die beiden akzeptierenden Zustände sein. Die Übergänge ergeben sich dann durch einfache Rechnungen:



Man könnte die Vermutung haben, dass sechs Zustände recht viel sind, um diese Sprache zu erkennen. Damit hätte man tatsächlich recht. Die Methoden, mit denen wir ein übersichtlicheres Ergebnis bekommen können, lernen wir aber erst später kennen.

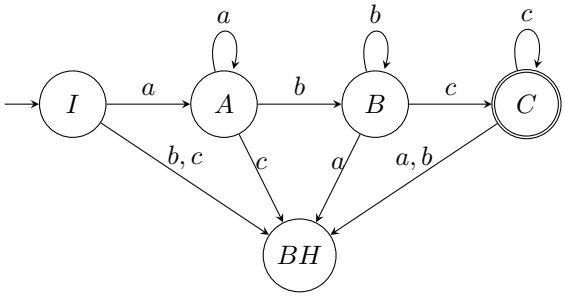
Partielle Automaten. Auch wenn die effektivere Methode der Vereinfachung erst später kommt, können wir auch jetzt schon einen Trick lernen, um manche Automaten zumindest ein bisschen übersichtlicher zu gestalten. Gelegentlich hat ein Automat Zustände, von denen aus kein Endzustand mehr zu erreichen ist. Wir können der Einfachheit zuliebe alle dieser Zustände, sowie die Kanten, die zu diesen Zuständen führen, aus dem Graphen entfernen.

Außer es handelt sich um den Startzustand. Dies ist aber nur ein Problem, wenn es sich um einen Automaten handelt, der kein einziges Wort akzeptiert und ein solcher lässt sich trivialerweise übersichtlich mit nur einem Zustand konstruieren.

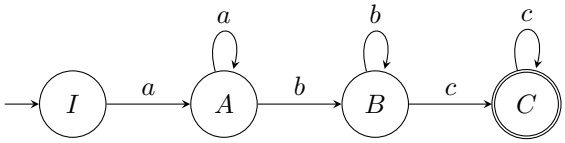
Dies führt dazu, dass es Knoten gibt, von denen aus nicht zu jedem Zeichen eine Kante ausgeht. Befinden wir uns in einem solchen Zustand und erhalten als Eingabe ein Zeichen, zu dem es keine Kante gibt, können wir sofort davon ausgehen, dass die Eingabe nicht akzeptiert wird. (Vor dem Entfernen der unnötigen Knoten wären wir ja in einen Zustand gekommen, von dem aus wir nicht mehr zu einem Endzustand hätten kommen können. Die Eingabe wäre also sowieso nicht mehr akzeptiert worden.)

Einem Automaten nennen wir **total**, wenn es in jedem Zustand zu jedem Zeichen eine ausgehende Kante gibt. Sonst nennen wir ihn **partiell**. Wir können aus einem partiellen Automaten sofort wieder zu einem äquivalenten totalen Automaten machen, indem wir einen einzigen nicht-akzeptierenden Zustand einfügen (häufig als BH - black hole - bezeichnet) und jede fehlende Kante mit BH als Folgezustand ergänzen. Insbesondere geht auch zu jedem Zeichen eine Kante von BH nach BH aus.

Beispiel. Wir betrachten die Sprache $\{a^k b^l c^m : k, l, m \geq 1\}$, also die Sprache aller Wörter, die zunächt nur a's, dann nur b's und dann nur c's als Eingabe haben. Von jedem Zeichen soll dabei mindestens eines eingegeben werden. Ein totaler Automat, der diese Sprache erkennt, sähe folgendermaßen aus:



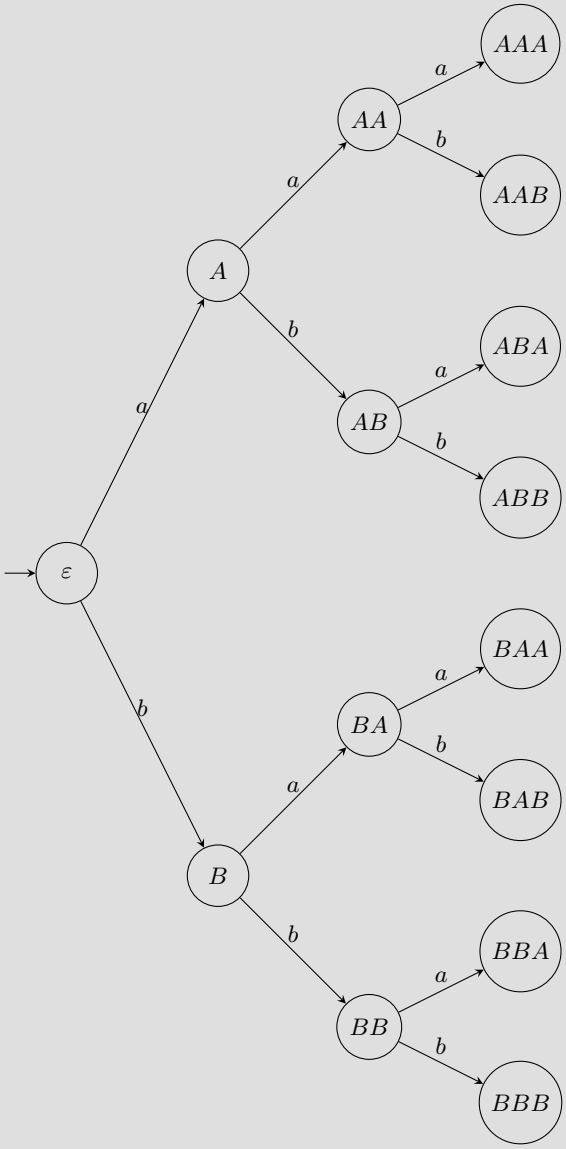
Zum Vergleich sehen wir hier den gleichen Automaten mit entferntem black hole Zustand:



Formale Definition. Bereits durch unsere intuitive Einführung endlicher Automaten wissen wir, was wir formal angeben müssen, um einen Automaten zu definieren. Wir benötigen eine Menge an Zuständen Q. Diese muss keine formalen Bedingungen erfüllen außer, dass sie nur endlich viele Elemente enthält. Es muss angegeben werden, auf welchem Alphabet Σ der Automat operiert. Zu jedem Zustand und jedem Zeichen muss ein entsprechender Folgezustand definiert werden. Formal ist diese Angabe eines Folgezustandes eine Funktion δ, die auf $Q \times \Sigma$ definiert ist und Werte in Q annimmt. Wir müssen einen Startzustand q_0 deklarieren und wir müssen angeben, welche Zustände akzeptierend sind - durch Angabe der Teilmenge $F \subseteq Q$ der Endzustände. Diese fünf Angaben definieren eindeutig den Automaten und umgekehrt. Formal können wir den Automaten also als das 5-Tupel $(Q, \Sigma, \delta, q_0, F)$ dieser Angaben betrachten.

Auch partielle Automaten können wir formal definieren. Die kennzeichnende Eigenschaft partieller Automaten ist, dass es in manchen Zuständen zu manchen Eingaben keinen Folgezustand gibt. Dies würde bedeuten, dass die Übergangsfunktion δ nicht auf dem gesamten Definitionsbereich Werte annimmt. Häufig ist dies nach der Definition einer Funktion nicht erlaubt. In dieser Vorlesung wollen wir unsere Definition des Begriffs der Funktion jedoch so wählen, dass nicht jeder Stelle des Definitionsbereiches ein Wert zugeordnet werden muss. So können wir auch partielle Automaten problemlos definieren.

Die Forderung, dass die Zustandsmenge endlich ist, ist von essenzieller Bedeutung. Sonst könnten wir zu jeder Sprache einen endlichen Automaten definieren, der diese Sprache erkennt. Man führe einfach einen Zustand für jedes mögliche Wort ein. Der Startzustand ist der Zustand, der zu ε entspricht. Die Zustandsübergänge seien so gewählt, dass wir nach einer Eingabe stets in dem Zustand sind, der der bisher gelesenen Eingabe entspricht. Für die ersten drei Eingaben könnte das in etwa folgendermaßen aussehen:



Man stelle sich vor, dass dieser Baum unendlich weit nach rechts weiterläuft. So sind wir nach jeder gelesenen Eingabe in einem für diese Eingabe einzigartigen Zustand. Wollen wir nun eine spezielle Sprache akzeptieren, müssen wir einfach die entsprechenden Zustände als Endzustände deklarieren. Deshalb könnte ein Automat, wenn wir die Bedingung fallen lassen, dass die Zustandsmenge endlich ist, bereits jede beliebige Sprache akzeptieren. Ein solches Modell ist daher derart mächtig, dass es für unsere Zwecke paradoxerweise uninteressant wird. Der Vorteil eines Modells, das nicht jede beliebige Sprache erkennen kann, ist dass wir Aussagen über verschiedene Sprachen treffen können. Sprachen, die von einem besonders simplen Modell erkannt werden, können von uns als wenig komplex angesehen werden. Ein Modell, das einfach jede Sprache erkennt, erlaubt dies nicht mehr.

Die Vorstellung dieses Automaten zeigt auch direkt, dass wir zu jeder endlichen Sprache einen DFA konstruieren können, der diese Sprache akzeptiert. Stellen wir uns diesen unendlichen Baum aus Zuständen vor. Wollen wir eine endliche Sprache akzeptieren,

müssen wir nur endlich viele dieser Zustände als akzeptierende Zustände markieren. Wir können dann alle Zustände, die zu keinem Endzustand mehr führen, entfernen. Da wir nur endlich viele Endzustände haben, bleiben uns hier auch nur noch endlich viele Zustände und wir erhalten etwas, das tatsächlich wieder als DFA zulässig ist.

2.2. **Reguläre Sprachen.**

Eine neue Fragestellung. Wie bereits gesagt, ist es eine für uns interessante Frage, welche Sprachen überhaupt von irgendeinem DFA erkannt werden. Solche Sprachen bezeichnen wir als **regulär**. Wir holen die formale Definition des Begriffs der erkannten Sprache nach:

Die Übergangsfunktion δ ist bereits definiert worden. Sie nimmt einen Zustand und ein Zeichen entgegen und liefert den Zustand, in dem der DFA nach Eingabe des Zeichens ist. Wir können diese Funktion erweitern zu einer Funktion $\hat{\delta}$, die einen Zustand und ein Wort entgegennimmt und den Zustand liefert, in dem DFA nach Eingabe des Wortes ist. Ein Wort wird genau dann akzeptiert, wenn der DFA in einem Endzustand endet, nachdem er mit dem Wort als Eingabe gestartet wurde. Anders ausgedrückt: Ein Wort w wird genau dann akzeptiert, wenn für den Startzustand q_0 der Zustand $\hat{\delta}(q_0, w)$ ein Endzustand ist. Die Menge aller akzeptierter Wörter ist die vom DFA erkannte Sprache. Eine Sprache bezeichnen wir als regulär, wenn es einen Automaten gibt, der die Sprache erkennt. Die Klasse aller regulärer Sprachen über einem Alphabet Σ wird als $REG(\Sigma)$ bezeichnet.

Wir stellen uns also die Frage, welche Sprachen regulär sind und welche nicht. Kriterien, welche Sprachen regulär sind und welche nicht, sind für uns sehr interessant.

Hinweis: Wir haben Sprachen bereits im Kontext regulärer Ausdrücke als „regulär“ bezeichnet. Diese Mehrfachdefinition des Begriffs stellt für uns aber kein Problem dar. Wir werden sehen, dass Sprachen, die im einen Sinne des Begriffs regulär sind, auch im anderen Sinne des Begriffs regulär sind.

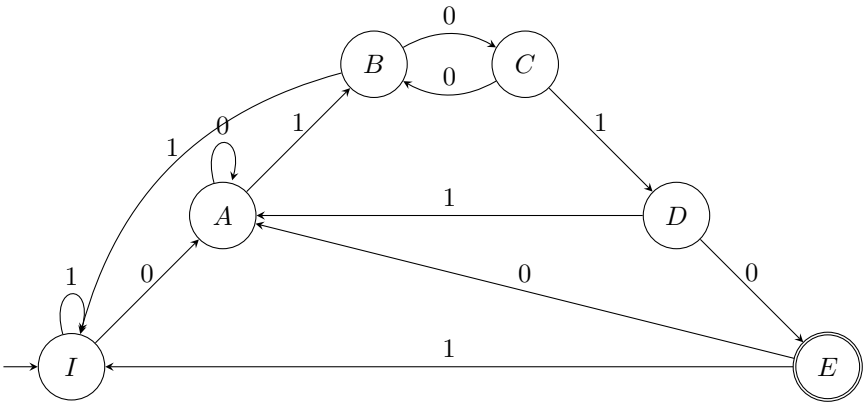
Ein notwendiges Kriterium. Eine Struktur regulärer Mengen fällt auf, wenn wir uns die Graphenstruktur anschauen. Enthält eine Sprache nur endlich viele Wörter, ist die Sprache sowieso regulär. Enthält eine Sprache unendlich viele Wörter, so gibt es insbesondere auch Wörter beliebig langer Länge. Man betrachte ein Wort, das mehr Zeichen enthält als der DFA Zustände hat. Wenn wir tracken, welche Zustände besucht werden, muss dabei mindestens ein Zustand mehrfach besucht werden. Wir nennen diesen Zustand A. Zwischen diesen zwei Besuchen des Zustandes A wurde ein bestimmtes Wort y eingelesen. Wir sehen, dass das Einlesen von y im Zustand A wieder zum Zustand A zurückführt. Wir können also das Teilwort y an dieser Stelle entweder entfernen oder beliebig oft duplizieren und ändern nichts daran, ob das gesamte Wort akzeptiert wird oder nicht.

Wir sehen, dass also Folgendes bei jeder regulären Sprache L funktioniert: Es gibt eine bestimmte Länge n (Wähle $n = |Q|$), sodass wir in jedem Wort in L, das mindestens n Zeichen enthält, ein Teilwort y finden, sodass y aus dem Wort entfernt oder beliebig oft dupliziert werden kann ohne zu ändern, dass das Wort in L enthalten ist. Schauen wir genauer hin, können wir das Kriterium noch verschärfen. Der mehrfache Besuch eines Zustandes muss bereits innerhalb der ersten n Zeichen gestehen. Auch bei deutlich längeren Wörtern müssen wir dieses Teilwort y also nicht an beliebig vielen Stellen suchen.

Diese Aussage ist als das **Pumping Lemma** bezeichnet: Zu jeder regulären Sprache L gibt es eine natürliche Zahl n, sodass wir für jedes $w \in L$ mit $|w| \geq n$

eine Zerlegung $w = xyz$ finden, sodass $xy^iz \in L$ für jede natürliche Zahl $\mathbb{N}_0$ ist, wobei $|xy| \leq n$ und $|y| \geq 1$ ist.

Beispiel. Wir betrachten den folgenden Automaten:



Wir sehen, dass zum Beispiel das Wort 0100101010 akzeptiert wird. Dieses Wort ist länger als die Anzahl an Zuständen. Irgendwo muss also ein Loop im Durchlaufen des Automaten entstehen. Die Folge der Zustände, die das Wort durchläuft, ist $I, A, B, C, B, I, A, B, C, D, E$. Der erste wiederholte Zustand ist Zustand B. Das Teilwort, das von B nach B führt, ist das folgende 00:

$$01\mathbf{00}101010$$

Das Pumping-Lemma sagt uns, dass alle der folgenden Wörter akzeptiert werden:

$$01101010, 0100101010, 010000101010, 01000000101010, 0100000000101010, \ldots$$

Mit jedem hinzugefügten 00 gehen wir einfach ein weiteres Mal von B nach C nach B.

Nützliche Kontraposition. So, wie die Aussage des Pumping-Lemmas formuliert wurde, nützt sie uns noch nicht, um zu entscheiden, ob eine gegebene Sprache regulär ist. „Wenn eine Sprache regulär ist, dann ist sie pumpbar.“ ist eben nicht das Gleiche wie „Wenn eine Sprache pumpbar ist, dann ist sie regulär.“. Können wir von einer Sprache nachweisen, dass sie pumpbar ist, sagt uns das noch nichts darüber, ob sie nun regulär ist oder nicht. Die Umkehrung ist aber der Fall. Können wir von einer Sprache nachweisen, dass sie *nicht* pumpbar ist, so können wir tatsächlich eindeutig folgern, dass die Sprache auch *nicht* regulär ist. Wäre sie regulär, dann könnten wir sie nach Pumping-Lemma ja nämlich auch pumpen. Das Pumping-Lemma hilft uns also nicht, wenn wir *nachweisen* wollen, dass eine Sprache regulär ist, aber es ist ein sehr nützliches Werkzeug, um zu *widerlegen*, dass eine Sprache regulär ist.

Beweise mit dem Pumping-Lemma. Wollen wir das Pumping-Lemma nutzen, um nachzuweisen, dass eine Sprache, zum Beispiel $L = \{a^k b^k | k \in \mathbb{N}\}$ nicht regulär ist, so müssen wir zeigen, dass L nicht pumpbar ist. „Pumpbar“ würde in diesem Fall bedeuten, dass es eine Pumpkonstante n gibt, sodass jedes Wort ein nicht-leeres Teilwort unter den ersten n Zeichen hat, das entfernt oder beliebig dupliziert werden kann, sodass das Wort danach noch immer in der Sprache enthalten ist. Wollen wir nachweisen, dass dies nicht der Fall ist, müssen wir zeigen, dass jede Wahl einer Pumpingkonstante ein Gegenbeispiel hervorbringt.

Man nehme also an, die Sprache habe eine Pumpingkonstante n. Dann müssen wir ein Wort finden, das sich als Gegenbeispiel anbieten würde. Dafür nehmen wir das Wort $a^n b^n$. Dieses scheint ein guter Kandidat für ein Gegenbeispiel zu sein. Um

zu zeigen, dass es ein Gegenbeispiel ist, müssen wir zeigen, dass das Wort nicht pumpbar ist - dass es also kein nicht-leeres Teilwort unter den ersten n Zeichen gibt, das entfernt oder dupliziert werden kann ohne das Wort aus der Sprache zu entfernen. Ein nicht-leeres Teilwort unter den ersten n Zeichen besteht nur aus a's. Das Entfernen dieses Teilwortes würde also die Anzahl der a's, aber nicht die Anzahl der b's ändern. Das Ergebnis wäre also nicht mehr in der Sprache enthalten. Dies beendet bereits den Beweis.

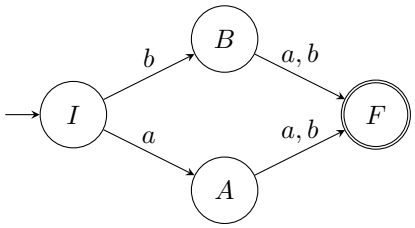
Formale Stolperfallen.

- Wir müssen wirklich zeigen, dass *jede* Wahl einer Pumpingkonstante ein Gegenbeispiel hervorbringt. Nur beispielhaft z.B. für $n = 5$ zu zeigen, dass a^5b^5 nicht pumpbar ist, reicht nicht aus. Das zeigt nur, dass die Pumpingkonstante nicht 5 sein kann - nicht dass es allgemein keine Pumpingkonstante geben kann. Zwar ist der Rechenweg in diesem Beispiel wahrscheinlich so suggestiv, dass aus diesem Beispiel wohl recht gut zu erkennen sein wird, was im Falle einer allgemeinen Pumpingkonstante zu tun ist, doch formal ist es wichtig, auch wirklich jede Pumpingkonstante auszuschließen.
- Ein Wort, das als Gegenbeispiel angegeben wird, ist niemals ganz ohne Kontext ein Pumpinglemma-Gegenbeispiel. Man kann nicht sagen, dass das Wort a^nb^n nicht pumpbar ist. Es ist konkreter nicht pumpbar *mit der Pumpkonstante* n. Andere Pumpkonstanten haben auch andere Wörter als Gegenbeispiel. Es ist also wichtig zu sagen, zu welcher Pumpkonstante das angegebene Wort nun ein Gegenbeispiel ist. (Wer ohne Kontext von dem Wort a^nb^n spricht ohne vorher zu erklären, was n überhaupt ist und zusätzlich nicht deklariert welchen Variablennamen die Pumpkonstante hat, der hat eigentlich gleich zwei formale Patzer begangen: Man hat einerseits den Variablennamen „n“ benutzt ohne vorher definiert zu haben, worauf sich dabei überhaupt bezogen wird und hat andererseits mit einer Pumpkonstante gearbeitet ohne ihr je einen Namen gegeben zu haben. Auch hier ist recht suggestiv, dass diese zwei Probleme sich wohl gegenseitig lösen. Formal ist es aber wichtig, anzugeben, dass n in diesem Kontext die Pumpkonstante sein soll.) Der Start eines Pumping-Lemma Beweises sollte also in etwa die Form „Angenommen, die Sprache habe eine Pumpingkonstante n. Dann betrachte das Wort a^nb^n.“ Der erste Satz kann hier nicht einfach so weggelassen werden.
- Der vorherige Punkt ist nicht nur eine penible formale Feinheit. Man könnte meinen, dass es doch reichen sollte, ein Wort gefunden zu haben, dass sich allgemein nie pumpen lässt. Dann habe man ja gezeigt, dass das Wort für jede Pumpkonstante ein Gegenbeispiel ist und dann müsste es ja auch egal sein, für welche konkrete Pumpkonstante das Wort nun als Gegenbeispiel angegeben wurde. Das Problem ist Folgendes: Kein Wort ist für jede Pumpingkonstante ein Gegenbeispiel. Eine grundlegende Bedingung für das Pumpinglemma ist, dass die Länge des Wortes mindestens der Pumpingkonstante entspricht. Das Wort a^nb^n enthält tatsächlich kein Teilwort, das beliebig dupliziert werden kann, ohne das Wort aus der Sprache zu entfernen. Wegen $|a^nb^n| = 2n$ ist das Wort aber trotzdem kein Gegenbeispiel für die Pumpingkonstante $2n + 1$. Das ist auch gut so. Auch reguläre Sprachen können Wörter enthalten, bei denen sich kein Teilwort entfernen oder duplizieren lässt ohne das Wort aus der Sprache zu entfernen. Dann ist bei diesen Sprachen nur eben auch die Pumpingkonstante größer.

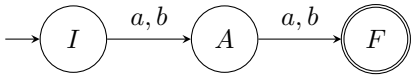
2.3. **Automatenminimierung.**

Ideen zur Minimierung. Unser Ziel ist es nun, unsere DFAs möglichst übersichtlich zu gestalten. Eine Möglichkeit dazu haben wir bereits kennengelernt, als wir die Automaten partiell dargestellt haben und somit einen black-hole Zustand auslassen konnten. In diesem Abschnitt werden wir noch radikalere Änderungen an unseren Automaten vornehmen.

Anstatt nur einzelne Zustände nicht ins Diagramm einzuzeichnen werden wir im Folgenden die Zustandsmenge selbst grundlegend verändern. Dabei ist unser Ziel, dass wir einen übersichtlicheren Automaten (d.h. ein Automat mit weniger Zuständen) erhalten. Die erste Idee ist hierbei recht einleuchtend. Betrachten wir folgenden Automaten:

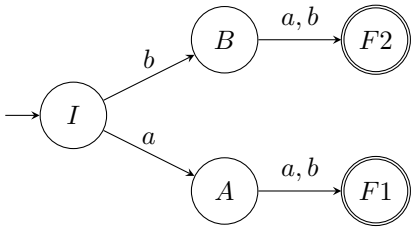


Offensichtlich ist es hier egal, ob wir uns im Zustand A oder im Zustand B befinden. Egal, in welchem der beiden Zustände wir sind, wird die Eingabe eines weiteren Zeichens uns wieder in den Zustand F führen. Es wäre also eine Idee, gar nicht mehr dazwischen zu unterscheiden, in welchem der beiden Zustände wir sind. Setzen wir diese Idee um, erhalten wir folgenden Automaten:



Man bemerke, dass dies **nicht** der gleiche DFA wie zuvor ist. Die Zustandsmengen der beiden DFAs sind völlig verschieden. Die DFAs sind aber äquivalent in dem Sinne, dass jedes Wort, das von einem der beiden Automaten akzeptiert wird, auch vom anderen akzeptiert wird. An einem DFA interessiert uns quasi nie der tatsächliche Aufbau der Zustände und Übergangsfunktion, sondern eigentlich nur, welche Sprache der DFA akzeptiert. Daher ist es für unsere Zwecke sinnvoll, zwischen zwei DFAs, die die gleiche Sprache akzeptieren, nicht mehr so streng zu unterscheiden. Auf diese Weise können wir dann auch sagen, dass der zweite DFA eine Vereinfachung des ersten DFAs ist.

Folgender DFA akzeptiert ebenfalls die gleiche Sprache:

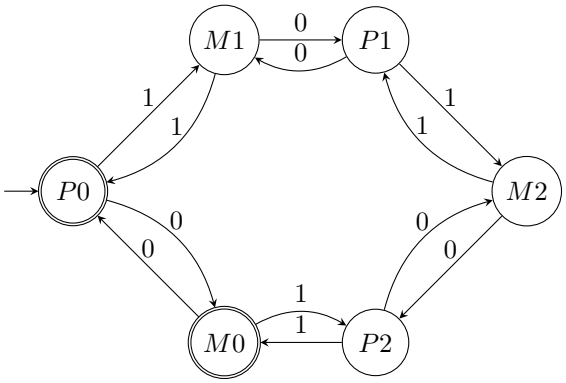


Hier gibt es keine zwei Zustände mit gleichen Folgezuständen. (Man könnte argumentieren, dass $F1$ und $F2$ beide den gleichen nicht eingezeichneten black-hole Zustand als Folgezustand haben. Nehmen wir also an, dass Eingaben aus $F1$ und $F2$ in zwei verschiedende black-hole Zustände führen.) Trotzdem wird hier offenbar

wieder die gleiche Sprache erkannt. Die Erkenntnis ist, dass wir hier gleichzeitig Zustand A und B als auch Zustand $F1$ und $F2$ miteinander verschmelzen können.

Die allgemeine Erkenntnis ist: Seien A und B zwei Zustände eines DFAs. Wenn jedes Wort, das von Zustand A aus zu einem akzeptierenden Zustand führt, auch von Zustand B aus zu einem akzeptierenden Zustand führt und jedes Wort, das von Zustand A aus zu einem nicht-akzeptierenden Zustand führt, auch von Zustand B aus zu einem nicht-akzeptierenden Zustand führt, dann müssen wir nicht mehr dazwischen unterscheiden, in welchem der beiden Zustände wir uns befinden und können diese miteinander verschmelzen. Diese Idee führt zu einem Algorithmus, mit der wir tatsächlich einen möglichst kleinen äquivalenten DFA konstruieren können.

Beispiel. Wir werden diese Idee des Verschmelzens nun nutzen, um zu dem zuvor definierten Automaten einen möglichst kleinen äquivalenten Automaten definieren. Dafür betrachten wir den Automaten, den wir bereits in Abschnitt 2.1 betrachtet haben:



Die Idee ist, dass wir alle möglichen Kombinationen zweier Zustände betrachten und ausschließen werden, welche Zustände wir garantiert nicht miteinander verschmelzen können. Alle Zustände, die wir nicht ausgeschlossen haben, werden wir dann verschmelzen. Dafür verfolgen wir die Paare von Zuständen in einer Tabelle:

	P0	M0	P1	M1	P2	M2
P0						
M0						
P1						
M1						
P2						
M2						

Offensichtlich werden wir nie einen akzeptierenden Zustand mit einem nicht-akzeptierenden Zustand verschmelzen. Alle dieser Paare werden wir also direkt ausschließen können.

	P0	M0	P1	M1	P2	M2
P0			X	X	X	X
M0			X	X	X	X
P1	X	X				
M1	X	X				
P2	X	X				
M2	X	X				

Die nächste Idee besagt, dass wir zwei Zustände A und B nicht miteinander verschmelzen können, wenn es ein Zeichen gibt, das von A aus in einen akzeptierenden Zustand und von B aus in einen nicht-akzeptierenden Zustand führt.

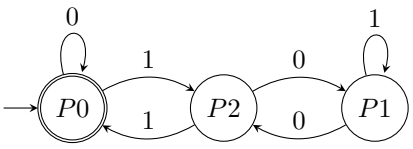
Allgemeiner können wir A und B nicht verschmelzen, wenn es ein Zeichen gibt, sodass wir bereits ausgeschlossen haben, dass die jeweiligen Folgezustände von A und B miteinander verschmolzen werden können. So können wir iterativ weitere Paare ausschließen. Wenn wir in einer Iteration keine neuen Paare mehr ausgeschlossen haben, können die noch nicht ausgeschlossenen Paare miteinander verschmolzen werden.

Zum Beispiel haben wir noch kein Kreuz bei dem Paar $(P1, M1)$ gesetzt. Wir prüfen die jeweiligen Folgezustände nach Einlesen von 0/1. Eine 0 führt $P1$ nach $M1$ und $M1$ nach $P1$. Das Folgepaar von $(P1, M1)$ via 0 ist also $(M1, P1)$. Dieses Paar hat auch noch kein Kreuz in der Tabelle. Das ist also in Ordnung. Eine 1 führt $P1$ nach $M2$ und $M1$ nach $P0$. Das Folgepaar von $(P1, M1)$ via 1 ist also $(M2, P0)$. Das Paar $(M2, P0)$ hat bereits ein Kreuz in der Tabelle. Da $(P1, M1)$ über ein Zeichen in ein Paar überführt werden kann, von dem wir wissen, dass es nicht verschmolzen werden kann, können wir auch $(P1, M1)$ selbst nicht verschmelzen und setzen ein Kreuz in der Tabelle. Für die weiteren Paare, bei denen bisher kein Kreuz gesetzt wurde, fahren wir mit analogen Überlegungen fort.

	P0	M0	P1	M1	P2	M2
P0			X	X	X	X
M0			X	X	X	X
P1	X	X		X	X	
M1	X	X	X			X
P2	X	X	X			X
M2	X	X		X	X	

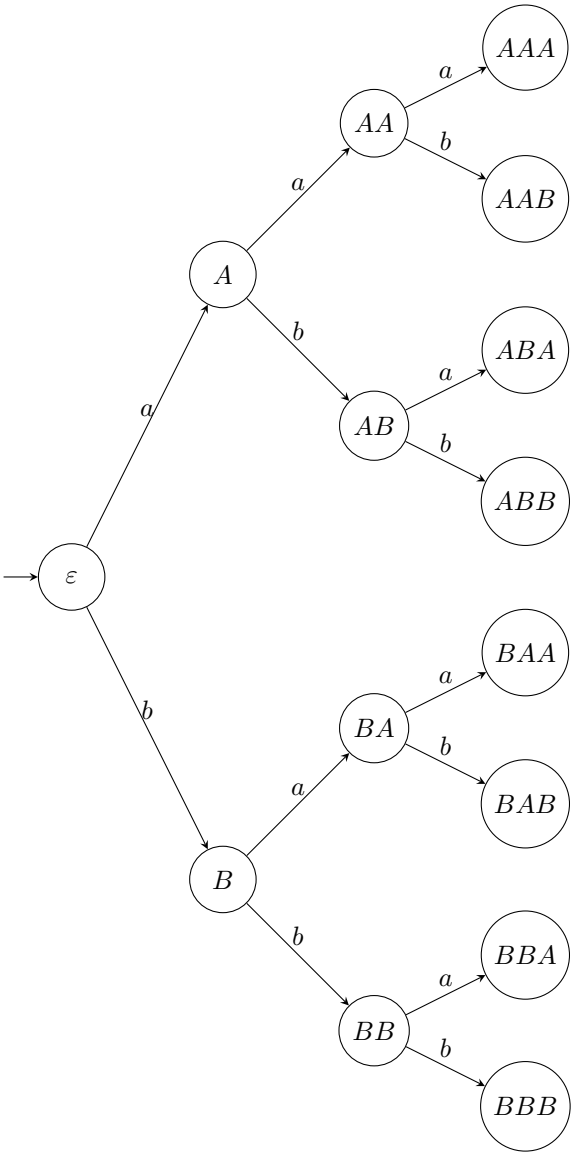
Da wir nun neue Kreuze gesetzt haben, kann sein, dass es Zustandspaare gibt, die vorher nicht in ein angekreuztes Paar überführt wurden, jetzt aber schon. Deshalb müssen wir nun noch einmal alle Paare durchgehen, die kein Kreuz haben. In diesem Fall kommt es nicht vor, dass es dann noch weitere Kreuze geben muss. Allgemein kann es aber passieren, dass ein Zustandspaar (A, B) in (C, D) überführt wird, wobei C und D selber beide nicht akzeptierend sind. Wenn dann aber (C, D) in (E, F) überführt werden kann, wobei von E und F ein Zustand akzeptierend und einer nicht-akzeptierend ist, wird in der ersten Iteration (C, D) angekreuzt und in der zweiten Iteration dann (A, B) angekreuzt.

In diesem Fall sind wir an dieser Stelle fertig. Auch nach weiteren Iterationen werden keine weiteren Paare markiert. Wir können also alle nicht markierten Paare miteinander verschmelzen:



DFA-Äquivalenztest. Dieser Algorithmus kann ebenfalls genutzt werden, um zu prüfen, ob zwei DFAs äquivalent sind. Wir zeichnen die DFAs nebeneinander und tun so, als handle es sich um einen Automaten mit zwei Zusammenhangskomponenten. Dann wenden wir den Markierungsalgorithmus an, um die Automaten, wenn möglich, zu verschmelzen. Wenn auf diese Weise die beiden Startzustände miteinander verschmolzen werden, waren die beiden Automaten äquivalent. Sonst nicht.

Ein neues Kriterium. Die Idee, dass wir zwei Zustände verschmelzen, wenn jedes Folgewort entweder beide Zustände in einen akzeptierenden Zustand oder beide Zustände in einen nicht-akzeptierenden Zustand führt, führt uns direkt zu einem neuen Kriterium für reguläre Sprachen. Wir erinnern uns an den theoretischen DFA, mit unendlich vielen Zuständen, mit dem (je nach Endzustandsmenge) jede Sprache erkannt werden kann:



Die Endzustände seinen hier so gewählt, dass die Sprache L erkannt wird. Wir wenden auf diesen Automaten die obere Art des Verschmelzens von Zuständen an: Zwei Zustände werden genau dann verschmolzen, wenn jedes Folgewort entweder von beiden Zuständen aus in einen akzeptierenden Zustand oder von beiden Zuständen aus in einen nicht-akzeptierenden Zustand führt.

Wir überlegen uns, was es bedeutet, wenn der Automat nach diesem Verschmelzen nur noch endlich viele Zustände hat. Wir haben dann immer noch einen Automaten, der die Sprache L erkennt (Das Verschmelzen auf diese Art ändert ja nichts an der erkannten Sprache.) Dieser Automat hat nun endlich viele Zustände. Der einzige

Grund, warum der vorherige Automat kein DFA war, lag an der unendlichen Zustandsmenge. Liefert das Verschmelzen auf diese Art also einen Automaten mit nur endlich vielen Zuständen, erhalten wir einen DFA, der L erkennt. Insbesondere ist L dann regulär.

Umgekehrt ist es intuitiv auch einleuchtend, dass L nicht regulär ist, wenn der Automat auch nach diesem Verschmelzen noch unendlich viele Zustände hat. Intuitiv könnte man sagen, dass wir keinen DFA erhalten konnten - selbst nachdem wir die Zustandsmenge so klein gemacht haben, wie es uns nur irgend möglich war. Die Schlussfolgerung liegt also nahe, dass es keinen DFA gibt, der L erkennen könnte. Trotz unserer etwas informellen Lösung, ist es tatsächlich möglich, diese Argumentation auch rigide durchzuführen und die Schlussfolgerung liegt nicht nur intuitiv nahe, sondern ist auch tatsächlich korrekt.

Wir erhalten so ein konkretes Kriterium, mit dem entschieden werden kann, ob eine Sprache regulär ist oder nicht. Anders als beim Pumping-Lemma, mit dem wir nur in manchen Fällen beweisen konnten, dass eine Sprache *nicht* regulär ist, können wir dieses Kriterium sowohl nutzen, um Regularität als auch nicht-Regularität nachzuweisen.

Die Vorstellung des Automaten mit unendlich vielen Zuständen war sehr hilfreich, um intuitiv zu verstehen, warum dieses Kriterium korrekt ist. Es ist jedoch für den praktischen Nutzen nicht sonderlich hilfreich. Wir können das Kriterium aber leicht umformulieren, indem wir die Konstruktion des Automaten umgehen und nur die Begriffe der Sprachen und Wörter nutzen.

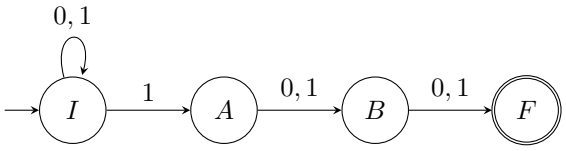
Wir erinnern uns daran, dass jeder Zustand zu genau einem Wort korrespondiert. (Der Zustand, der zu dem Wort w korrespondiert, ist der Zustand, in dem wir uns nach Einlesen des Wortes w befinden.) Wir können so die Zustandsmenge mit Σ^* identifizieren. Das Verschmelzen von Zuständen können wir als das Zusammenfassen von Wörtern aus Σ^* in Äquivalenzklassen verstehen. Das Kriterium, dass wir zwei Zustände dann verschmelzen, wenn jedes Folgewort entweder von beiden in einen akzeptierenden Zustand oder von beiden in einen nicht-akzeptierenden Zustand führt, können wir auf Ebene der Wörter so verstehen, dass wir die den Zuständen entsprechenden Wörter x und y genau dann in einer Äquivalenzklasse zusammenfassen, wenn für jedes mögliche Folgewort z genau dann $xz \in L$ ist, wenn auch $yz \in L$ ist. Das Kriterium besagt dann, dass L genau dann regulär ist, wenn Σ^* unter dieser Relation in nur endlich viele Äquivalenzklassen zerfällt. Diese Aussage ist auch als der **Myhill-Nerode-Satz** bekannt.

Wir sehen ebenfalls schnell, dass die Anzahl dieser Äquivalenzklassen genau der Mindestanzahl an Zuständen eines erkennenden DFAs entspricht - also der Anzahl der Zustände des Minimalautomaten.

Anwendungen des Myhill-Nerode-Satz. Der Myhill-Nerode-Satz kann theoretisch verwendet werden, um nachzuweisen, dass eine gegebene Sprache regulär ist. Dafür müsste man zeigen, dass Σ^* in nur endlich viele Äquivalenzklassen zerfällt. Das ist häufig unnötig kompliziert, da wir meistens recht leicht einen DFA angeben können, der die Sprache akzeptiert, was ebenfalls zum Nachweis der Regularität genügt. Wir haben hier aber noch den Vorteil gegenüber dem Pumpinglemma, dass wir auch bei tatsächlich jeder nicht regulären Sprache mit Myhill-Nerode tatsächlich nachweisen können, dass die Sprache nicht regulär ist.

Man nehme zum Beispiel die Sprache $L = \{a^m b^n c^n | m, n \geq 1\} \cup \{b^m c^n | m, n \geq 0\}$. Zu den Worten $x = ab^i$ und $y = ab^j$ mit $i \neq j$ können wir das Folgewort $z = c^i$ betrachten. Dann ist $xz \in L$ aber $yz \notin L$. Somit müssen die Wörter ab^i und ab^j stets in unterschiedlichen Äquivalenzklassen liegen, sodass es unendlich viele Äquivalenzklassen geben muss. Die Sprache ist also nicht regulär.

2.4. **Nichtdeterministische Automaten.** In diesem Abschnitt lernen wir eine neue Variation von DFAs kennen. In Graphenschreibweise werden diese Automaten genauso wie DFAs aussehen. Der einzige Unterschied wird darin liegen, dass manchmal mehrere Kanten mit der gleichen Beschriftung von einem Zustand ausgehen können.



In diesem Automaten gehen von dem Zustand I zwei Kanten mit der Beschriftung 1 aus. Die eine Kante führt wieder nach I zurück, während die andere Kante in den Zustand A führt. Wie ist nun also zum Beispiel ein Lauf des Wortes 1101 durch diesen Automaten zu verstehen?

Die Idee ist, dass es mehrere verschiedene Läufe gibt, die alle als „möglich“ betrachtet werden. Jede Kreuzung, an der wir mehrere verschiedene Kanten zu dem aktuellen Zeichen haben, gibt uns weitere mögliche Pfade. Konkret im Falle des Wortes 1101 bedeutet das: Wir starten im Zustand I. Nach der ersten 1 können wir entweder im Zustand I bleiben oder in den Zustand A weitergehen. Wir haben die Pfade $I- > I$ und $I- > A$.

Lesen wir im Pfad $I- > I$ nun die nächste 1 ein, können wir wieder entweder bei I bleiben oder nach A weitergehen. Im Pfad $I- > A$ haben wir nur die Möglichkeit nach B weiterzugehen. Wir haben bisher also die drei Pfade $I- > I- > I$ und $I- > I- > A$ und $I- > A- > B$.

Beim nächsten Zeichen 0 haben wir jeweils nur eine Möglichkeit. Es bleibt also bei drei Pfaden: $I- > I- > I- > I$ und $I- > I- > A- > B$ und $I- > A- > B- > F$.

Bei der nächsten 1 spaltet sich der Pfad $I- > I- > I- > I$ wieder in zwei weitere Pfade auf. Die anderen beiden Pfade haben jeweils nur einen möglichen Nachfolger. Wir haben insgesamt also die vier Pfade $I- > I- > I- > I- > I$ und $I- > I- > I- > I- > A$ und $I- > I- > A- > B- > F$ und $I- > A- > B- > F- > BH$.

Dies klärt, wie der Automat zu durchlaufen ist. Wie klären wir nun aber, wann ein Wort als „akzeptiert“ zu betrachten ist? Hierfür haben wir die einfache Regelung: Der Automat akzeptiert ein Wort genau dann, wenn **einer** der möglichen Pfade in einem akzeptierenden Zustand endet. In unserem Beispiel endete der Pfad $I- > I- > A- > B- > F$ in einem akzeptierenden Zustand. Das Wort 1101 wird also akzeptiert. Das Wort 1001 würde aber nicht akzeptiert werden, wie recht leicht nachzuprüfen ist.

Formelle Definition. Wir wollen nun einen nichtdeterministischen endlichen Automaten - kurz NFA - formell definieren. Wir erinnern uns daran, wie wir einen DFA definiert haben. Dies geschah durch die Angabe von fünf verschiedenen Informationen: Die Zustandsmenge Q, das Eingabealphabet Σ, die Übergangsfunktion $\delta : Q \times \Sigma \to Q$, der Startzustand q_0 und die Endzustandsmenge F.

Wir können die Anforderungen an diese Informationen nun leicht abwandeln, indem wir Nichtdeterminismus erlauben. Zunächst können wir die Übergangsfunktion ändern. Zuvor gab es zu jeder Kombination von Zustand und Zeichen immer genau einen Folgezustand. Nun kann es mehrere mögliche Folgezustände geben. Da eine Funktion an einer Stelle aber immer nur einen Wert annehmen kann, fassen wir alle möglichen Folgezustände in einer Menge zusammen. Diese ist dann der Wert

der Übergangsfunktion. Die Werte sind also nicht mehr Elemente von Q, sondern Teilmengen von Q. Die Wertemenge ist also die Menge aller Teilmengen von Q - in der Mathematik wird diese auch als 2^Q geschrieben.

Auch an einer anderen Stelle führen wir Nichtdeterminismus in der Definition ein. Wir erlauben mehrere Startzustände. Dies sollte recht intuitiv verständlich sein. Ein Wort wird akzeptiert, wenn es einen entsprechenden Pfad von **einem** der Startzustände zu **einem** der Endzustände gibt. Die Angabe des Startzustandes geschieht hier also auch nicht mehr durch Angabe eines Elementes von Q, sondern durch Angabe einer Teilmenge von Q.

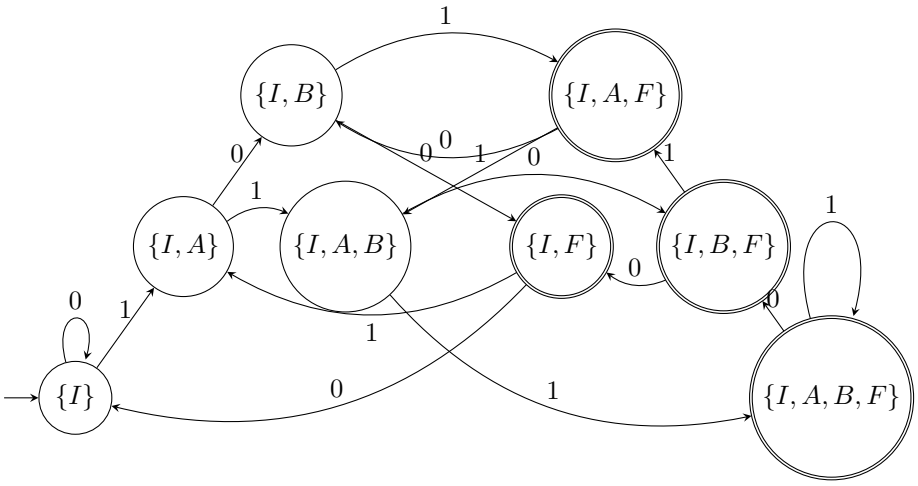
Insgesamt ist ein NFA also definiert durch Angabe von fünf Informationen: Die Zustandsmenge Q, das Eingabealphabet Σ, die Übergangsfunktion $\Delta : Q \times \Sigma \to 2^Q$, die Startzustandsmenge $Q_0 \in 2^Q$ und die Endzustandsmenge F.

Determinisieren durch Potenzmengenkonstruktion. Es drängt sich nun sofort eine Frage auf. Da wir mit NFAs Dinge tun, können, die wir mit DFAs nicht tun können: Gibt es Sprachen, die ein NFA erkennen kann, die jedoch von keinem DFA erkannt werden können? Tatsächlich ist dies nicht der Fall. Zu jedem NFA können wir einen entsprechenden DFA konstruieren, der die selbe Sprache erkennt.

Die Idee der Konstruktion ist quasi, alle möglichen Pfade auf deterministische Weise „gleichzeitig“ abzulaufen. Zu einem gegebenen (Teil-)Wort w können wir genau prüfen, zu welchen Zuständen es einen Pfad im NFA gibt, der nach Verarbeitung von w zu diesem Zustand führt. So haben wir eine mögliche Menge an Zuständen, in denen sich der NFA nach Einlesen von w befinden kann. Diese „möglichen Mengen an Zuständen, in denen sich der NFA zu diesem Zeitpunkt des Einlesens befinden kann“ können tatsächlich deterministisch getrackt werden.

Wir haben also folgende Idee: Die Zustände unseres DFAs sind benannt nach den möglichen Mengen an Zuständen des NFAs. Der Startzustand ist dann der Zustand, der der Startzustandmenge entspricht. Zu einer Menge an Zuständen U und einem Zeichen a ist die Folgezustandsmenge definiert als die Menge der Zustände, die wir von **einem** der Zustände in U aus über eine Kante nach Einlesen von a erreichen können. Akzeptierende Zustände sind diejenigen Zustandsmengen, die mindestens einen akzeptierenden Zustand des NFAs enthalten.

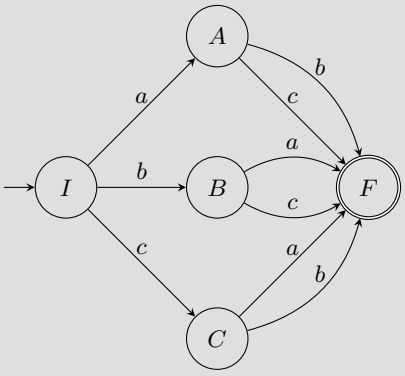
Nutzen von NFAs. NFAs helfen uns also nicht dabei, neue Sprachen zu erkennen. Warum haben wir NFAs also definiert? Ein Nutzen von NFAs ist die deutlich größere Übersichtlichkeit. Betrachten wir noch einmal den NFA vom Anfang dieses Abschnitts. Dieser war ein sehr übersichtlicher Automat, der aus nur vier Zuständen bestand. Ein minimaler äquivalenter DFA, der die gleiche Sprache erkennt, sieht folgendermaßen aus:

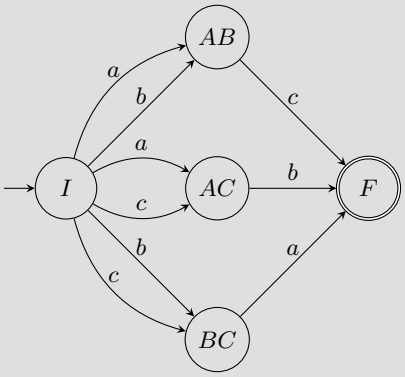


Wir erhalten einen sehr unübersichtlichen DFA mit acht Zuständen. Allgemein können wir zwar immer einen äquivalenten DFA konstruieren - die Anzahl der Zustände kann dabei aber exponentiell wachsen. Somit können NFAs häufig eine sehr viel übersichtlichere Darstellung der Lösung sein.

Für DFAs haben wir bereits ein einfaches Verfahren kennengelernt, das uns zu einem gegebenen DFA ein minimalen äquivalenten DFA konstruiert. Man könnte sich fragen ob wir ebenfalls zu einem NFA einen äquivalenten NFA mit möglichst wenig Zuständen konstruieren können. Hierfür ist tatsächlich kein effizientes Verfahren bekannt.

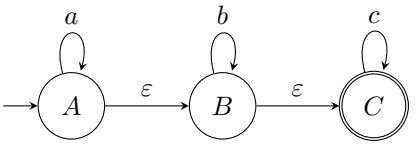
Ein Problem, auf das wir hierbei stoßen, ist die Tatsache, dass es nicht immer einen eindeutigen minimalen NFA zu einer Sprache gibt. Man betrachte die folgenden zwei NFAs:





Beide NFAs akzeptieren genau die Sprache $\{ab, ac, ba, bc, ca, cb\}$ und haben die gleiche Anzahl an Zuständen. Sie sind aber offenbar nicht gleich. Es lässt sich ebenfalls gut zeigen, dass kein NFA mit weniger Zuständen die gleiche Sprache akzeptieren kann. Diese Sprache hat also keinen eindeutigen minimalen NFA. Dieses Beispiel wurde `https://www.youtube.com/watch?v=Nyzwq4CA3KE` entnommen.

ε-NFA. Der nächste Schritt ist es, Übergänge im NFA zu erlauben, die ohne das Einlesen eines Zeichens getan werden können. In Graphenschreibweise fügen wir Kanten ein, die wir mit ε beschriften. Angenommen es existiert eine solche Kante vom Zustand A zum Zustand B. Wenn wir die möglichen Pfade bei der Verarbeitung eines Wortes überprüfen, ist zu jedem Pfad, der im Zustand A endet auch derjenige Pfad möglich, der diesen Pfad um den Übergang nach B ergänzt.



Formelle Definition der ε-Übergänge. Für die ε-Übergänge müssen wir die Definition des NFAs nur geringfügig anpassen. Es reicht aus, die Möglichkeit der neuen Kanten in die Definition der Übergangsfunktion einbauen, indem wir zu jedem Zustand ebenfalls noch angeben, welche Zustände per ε-Übergang erreichbar sind. Dafür erweitern wir die Definitionsmenge von $Q \times \Sigma$ auf $Q \times (\Sigma \cup \{\varepsilon\})$.

Konstruktion äquivalenter DFAs. Auch die Konstruktion eines äquivalenten DFAs verläuft nach der Einführung von ε-Übergängen noch analog. Die Zustandsmenge soll 2^Q sein. Eine Kante gibt an, welche Zustände nach Einlesen eines Zeichens möglicherweise erreichbar sind. Bedacht werden muss hier nur, dass nach dem vorgegebenen Zeichen auch noch beliebig viele ε-Übergänge möglich sind. Wir sprechen davon, dass wir den ε-Abschluss der Folgemenge bilden.

Mehrere Startzustände. Es ist interessant zu bemerken, dass ε-NFAs keine Notwendigkeit zur Benutzung mehrerer Startzustände haben. Zu einem ε-NFA mit mehreren Startzuständen kann ebenfalls ein NFA mit einem zusätzlichen Zustand definiert werden, der als einziger Startzustand deklariert wird und einen ε-Übergang zu jedem der vorherigen Startzustände erlaubt.

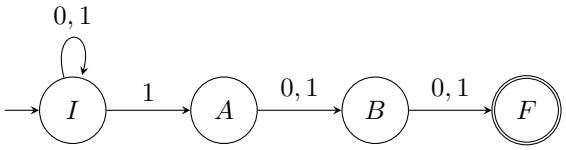
2.5. **Abschlusseigenschaften.** Wir werden uns nun endlich der Behauptung widmen, die schon seit längerem im Raum steht. Nämlich, dass unsere zwei Definitionen regulärer Mengen tatsächlich das Gleiche definieren. Wir haben einerseits die Definition, die reguläre Ausdrücke nutzt und die Definition, die endliche Automaten nutzt.

Vorher werden wir uns in einem ersten Schritt anschauen, was die Definition über endliche Automaten für Operationen auf regulären Sprachen erlaubt. Tatsächlich sind auch nach dieser Definition Vereinigungen, Produkte, Kleensche Hüllen, Komplemente, Reflexionen, Schnitte, Quotienten und Shuffleprodukte regulärer Sprachen, sowie Bilder und Urbilder regulärer Sprachen unter Homomorphismen wieder regulär.

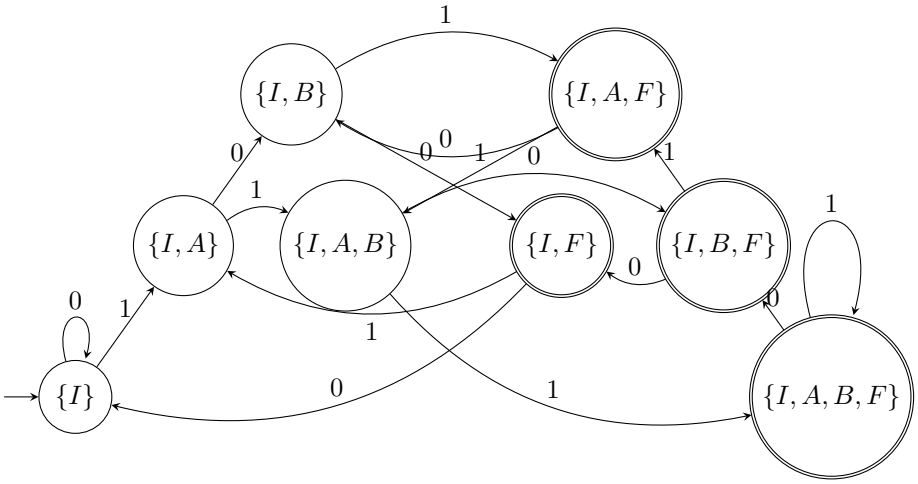
Wir werden im Folgenden einige dieser Eigenschaften genauer betrachten und anhand eines Beispiels durchgehen.

Komplement. Um zu sehen, dass Komplemente regulärer Sprachen wieder regulär sind, können wir in einem gegebenen Automaten jeden akzeptierenden Zustand zu einem nicht-akzeptierenden Zustand und jeden nicht-akzeptierenden Zustand zu einem akzeptierenden Zustand ändern. Der resultierende Automat wird genau das Komplement erkennen.

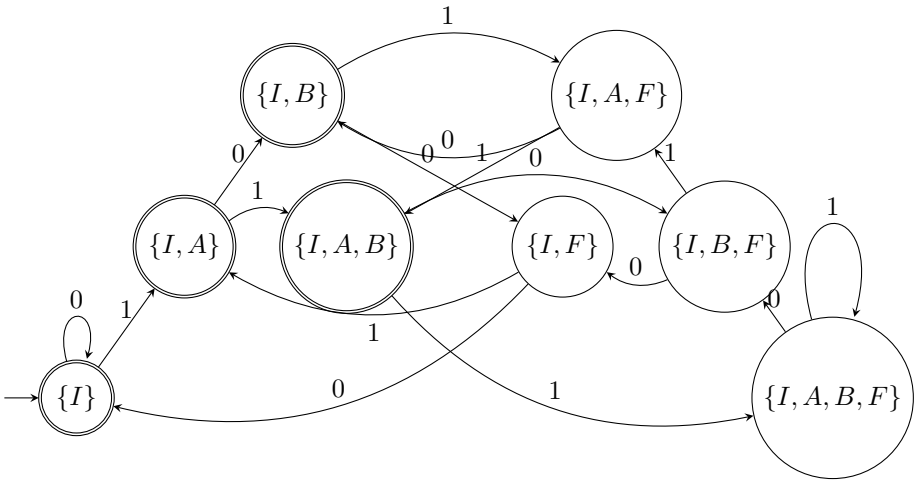
Wir betrachten die Sprache L_{-3}, die Sprache aller Wörter über $\{0,1\}$, deren drittletztes Zeichen eine 1 ist. Die Sprache wird von folgendem bereits bekannten NFA erkannt:



Einen äquivalenten DFA haben wir ebenfalls bereits gesehen:

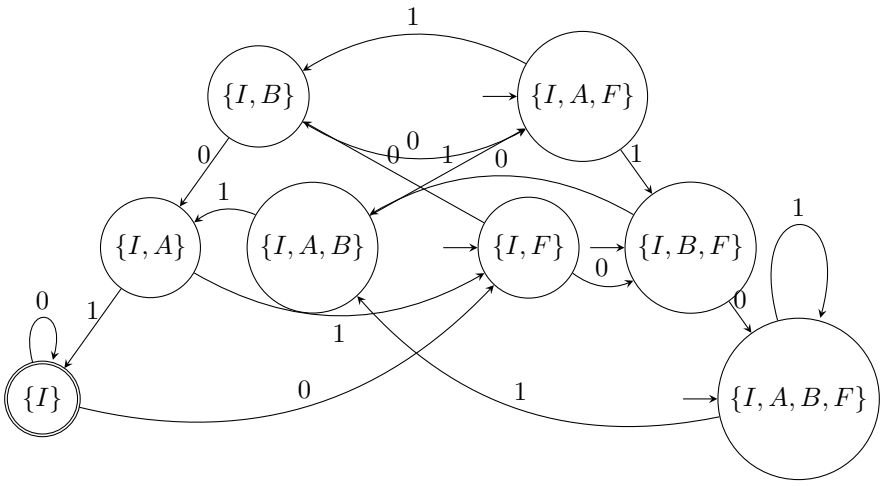


Das Komplement akzeptieren wir nun, indem wir die Rollen von akzeptierenden und nicht-akzeptierenden Zuständen tauschen:

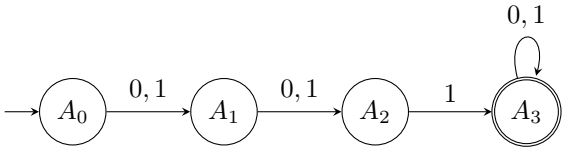


Es reicht nicht aus, akzeptierende und nicht-akzeptierende Zustände in einem NFA zu tauschen. In einem NFA kann es passieren, dass es zu einem Wort sowohl einen Pfad gibt, der in einem akzeptierenden Zustand endet, als auch einen Pfad, der in einem nicht-akzeptierenden Zustand endet. Ein solches Wort würde sowohl von dem ursprünglichen NFA als auch von dem NFA mit komplementärer Endzustandsmenge akzeptiert werden. Daher führt die komplementäre Endzustandsmenge nur bei DFAs garantiert zur komplementären akzeptierten Sprache.

Reflexion. Betrachten wir nun die Reflexion. Für diese liegt es recht nahe, den DFA „rückwärts“ durchlaufen zu wollen. Dabei wird der DFA zu einem NFA werden. Wir können aus jedem Endzustand einen Startzustand machen. (Der NFA kann dann mehrere Startzustände haben, was bei einem NFA ja erlaubt war.) Wir können dann bei jeder Kante die Richtung umdrehen und die Beschriftung behalten. Zuletzt wird der ursprüngliche Startzustand zum einzigen akzeptierenden Zustand. Der resultierende NFA akzeptiert dann die Reflexion der Sprache.



Der zugehörige deterministische Minimalautomat ist erfreulicherweise schön übersichtlich und es ist offensichtlich, dass er die gespiegelte Sprache $L_{-3}^R = L_3$ - die Sprache aller Wörter, deren drittes Zeichen eine 1 ist - erkennt:

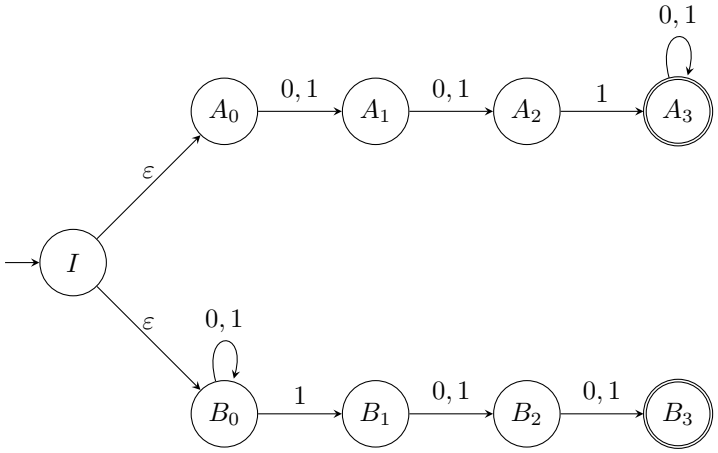


Anders als bei der Konstruktion des Komplements können wir die Konstruktion des Automaten für die Reflexion auch auf einen NFA anwenden. In diesem Beispiel wird der NFA sogar zu einem DFA, wenn wir die Konstruktion anwenden. Es fällt auf, dass der angegebene Minimalautomat genau der DFA ist, den wir bekommen, wenn wir die Konstruktion direkt auf den NFA anwenden.

Vereinigung. Wir können ebenfalls die Vereinigung zweier Sprachen mit einem NFA akzeptieren. Wir können hierfür die DFAs für die zu vereinigenden Sprachen zusammenfassen, indem wir einen neuen Startzustand einführen und von diesem aus ε-Übergänge zu den jeweiligen Startzuständen einfügen. Wird ein Wort dann von einem der beiden DFAs akzeptiert, so wird es auch von dem konstruierten NFA akzeptiert, wobei der Pfad zunächst einen ε-Übergang in den entsprechenden DFA macht und anschließend normal den DFA durchläuft.

Auch hier kann die Konstruktion wieder auch mit den NFAs der Sprachen durchgeführt werden. Der Übersichtlichkeit zuliebe werden wir dies in der folgenden Beispielgrafik auch tun. Der DFA zu L_{-3} war bereits unübersichtlich groß. Der entstehende Automat für die Vereinigung wäre noch größer. (Der minimale DFA der Vereinigung $L_3 \cup L_{-3}$ hat sogar 22 Zustände.)

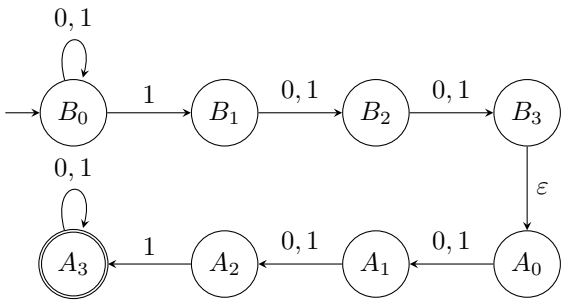
Ein Automat für die Sprache $L_{-3} \cup L_3$ sieht somit folgendermaßen aus:



Produkt. Das Produkt zweier Sprachen können wir mit einem NFA umsetzen, indem wir die entsprechenden DFAs auf folgende Weise zusammenfassen: Die Worte müssen zunächst den DFA der ersten Sprache durchlaufen. Anstatt dann von den Endzuständen akzeptiert zu werden, werden ε-Übergänge von den ursprünglichen Endzuständen des ersten Automaten zum Startzustand des zweiten Automaten eingefügt. Ein Wort, das von diesem Automaten akzeptiert wird, muss also beide Automaten nacheinander durchlaufen.

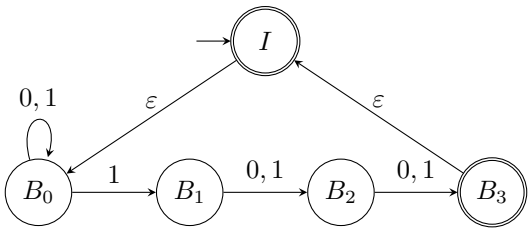
Auch hier können wir die Konstruktion mit den NFAs durchführen und werden dies für die Übersichtlichkeit auch tun. (Der minimale DFA hätte 39 Zustände!)

Das Produkt $L_{-3} \cdot L_3$ liefert folgenden Automaten:



Kleensche-Hülle. Für die Kleensche Hülle müssen wir es den Worten erlauben, den NFA auch keinmal oder mehrfach zu durchlaufen. Wir starten also in einem Endzustand, von dem aus ein ε-Übergang zum ursprünglichen Startzustand des Automaten führt. Von den Endzuständen führt dann wieder ein ε-Übergang zu diesem Zustand zurück.

Wir erhalten folgenden Automaten für L_3^*:



Schnitt. Um zu zeigen, dass der Schnitt zweier regulärer Sprachen wieder regulär ist, benötigen wir keine neue Konstruktion. Wir können den Schnitt nämlich mithilfe von Vereinigung und Komplement folgendermaßen ausdrücken:

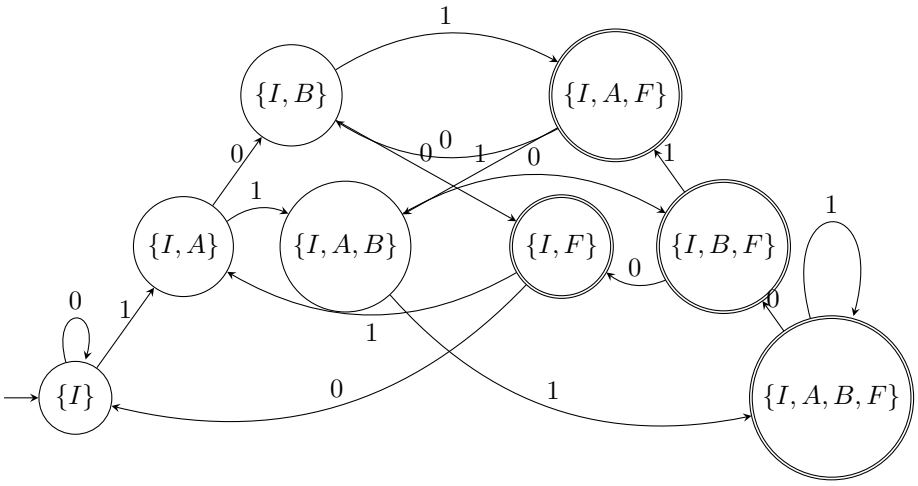
$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

$\overline{L_1}$ enthält genau die Elemente, die nicht in L_1 sind. Genauso mit L_2. Die Vereinigung $\overline{L_1} \cup \overline{L_2}$ enthält also genau die Elemente, die in mindestens einer der Mengen nicht enthalten sind - also genau die Elemente, die nicht im Schnitt enthalten sind. Daher ist $\overline{\overline{L_1} \cup \overline{L_2}}$ genau der Schnitt. Wir wissen, dass Komplemente und Vereinigungen regulärer Sprachen wieder regulär sind. Sind also L_1, L_2 regulär, sind es auch $\overline{L_1}$ und $\overline{L_2}$. Da $\overline{L_1}$ und $\overline{L_2}$ regulär sind, ist es auch $\overline{L_1} \cup \overline{L_2}$ und damit auch das Komplement $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$. Also ist der Schnitt regulärer Mengen regulär.

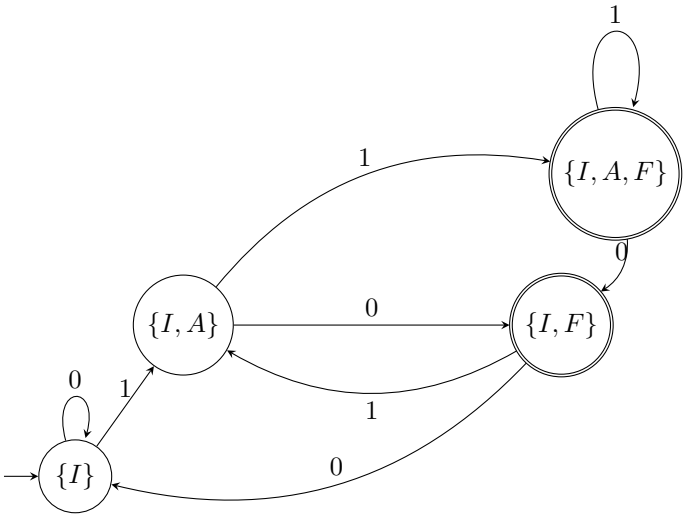
Bilder/Urbilder von Homomorphismen. Wir haben bereits in Kapitel 1.6 gezeigt, dass Bilder regulärer Mengen wieder regulär sind.

Betrachten wir also Urbilder unter Homomorphismen. Gegeben sei ein DFA D, der L akzeptiert und ein Homomorphismus φ. Wir wollen nun einen DFA konstruieren, der ein Wort w genau dann akzeptiert, wenn $\varphi(w)$ von D akzeptiert wird. Wir haben folgende Idee: Wir behalten die Zustände von D. Wenn wir im Zustand q ein Zeichen a einlesen, springen wir in denjenigen Zustand, in den wir kommen würden, wenn wir in D das Wort $\varphi(a)$ einlesen würden. So können wir für jeden Zustand q und jedes Zeichen a also zu Beginn einmal prüfen, in welchem Zustand wir enden, wenn der Automat D im Zustand q das Wort $\varphi(a)$ einliest und unserem neuen Automaten so eine neue Übergangsfunktion definieren.

Nehmen wir den Homomorphismus $\varphi : \{0,1\}^* \to \{0,1\}^*$ mit $\varphi(0) = 00$ und $\varphi(1) = 01$. Wir erinnern uns an den DFA von L_{-3}:



Wir wollen nun einen DFA konstruieren, der $\varphi^{-1}(L_{-3})$ akzeptiert. Vom Startzustand $\{I\}$ aus landen wir wieder in $\{I\}$, wenn wir zwei 0en einlesen. Wir landen in $\{I, A\}$, wenn wir eine 0 und dann eine 1 einlesen. Im neuen DFA führt von $\{I\}$ aus also die Kante via 0 nach $\{I\}$, während die Kante via 1 nach $\{I, A\}$ führt. Von $\{I, A\}$ aus führen zwei 0en nach $\{I, F\}$. Eine 0 und eine 1 führen nach $\{I, A, F\}$. Im neuen DFA führt von $\{I, A\}$ aus also die Kante via 0 nach $\{I, F\}$, während die Kante via 1 nach $\{I, A, F\}$ führt. So fahren wir fort und merken, dass wir im neuen DFA nur vier Zustände erreichen können. Der DFA des Urbilds ist also etwas kleiner:



Vom regulären Ausdruck zum NFA. Wie bereits erwähnt, ist es unser Ziel, zu zeigen, dass unsere zwei Definitionen regulärer Mengen gleich sind. Hierfür müssen wir eigentlich gleich zwei Dinge begründen. Wir müssen zeigen, dass jede Menge, die von einem regulären Ausdruck beschrieben wird, auch von einem DFA akzeptiert wird **und**, dass jede Menge, die von einem DFA akzeptiert wird, auch von einem regulären Ausdruck beschrieben wird.

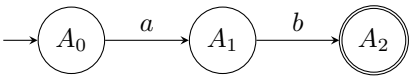
Die erste Richtung haben wir durch unsere Vorüberlegungen bereits erhalten. Wir erinnern uns daran, dass wir im Kontext regulärer Ausdrücke bereits gesehen haben, dass jede Sprache die durch reguläre Ausdrücke beschrieben wird, aus endlichen

Mengen durch endliche Anwendung von Produkt, Vereinigung und Kleenscher Hülle gewonnen werden kann. Zu jeder endlichen Menge lässt sich leicht ein DFA erkennen, der diese Menge erkennt. Wir haben ebenfalls oben gesehen, wie wir diese Automaten miteinander verbinden können, um Anwendungen von Produkt, Vereinigung und Kleenscher Hülle umzusetzen. So können wir zu einem regulären Ausdruck also einen NFA erhalten, der die beschriebene Sprache akzeptiert. Im vorherigen Abschnitt haben wir bereits gesehen, dass es dann auch einen NFA gibt, der die Sprache akzeptiert.

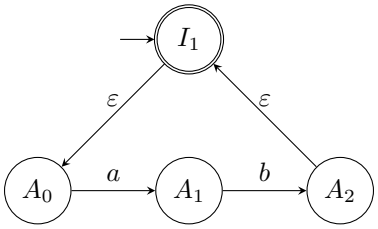
Beispiel. Man nehme den regulären Ausdruck $(ab)^*|ab^*$. Einen Automaten dazu können wir schrittweise aufbauen:

(1) Wir konstruieren einen Automaten, der ab akzeptiert.
(2) Wir konstruieren einen Automaten, der davon die Kleensche Hülle akzeptiert.
(3) Wir konstruieren einen Automaten, der b akzeptiert.
(4) Wir konstruieren einen Automaten, der davon die Kleensche Hülle akzeptiert.
(5) Wir konstruieren einen Automaten, der a akzeptiert.
(6) Wir konstruieren einen Automaten, der das Produkt von a und b^* akzeptiert, indem wir die Automaten aus Schritt 5 und 4 nutzen.
(7) Wir konstruieren einen Automaten, der die Vereinigung von $(ab)^*$ und ab^* akzeptiert, indem wir die Automaten aus Schritt 2 und 6 nutzen.

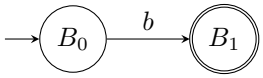
Folgender Automat akzeptiert ab:



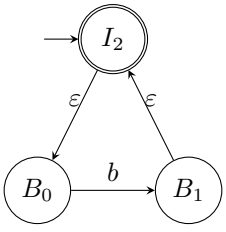
Folgender Automat akzeptiert die Kleensche Hülle davon:



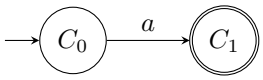
Folgender Automat akzeptiert b:



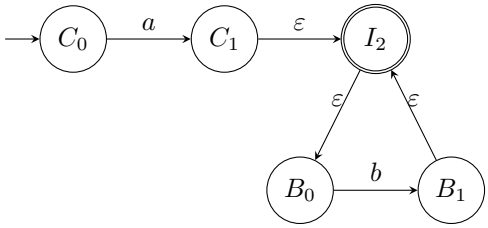
Folgender Automat akzeptiert die Kleensche Hülle davon:



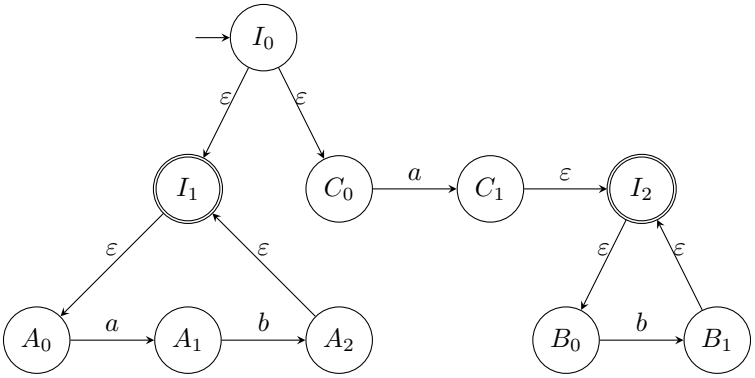
Folgender Automat akzeptiert a:



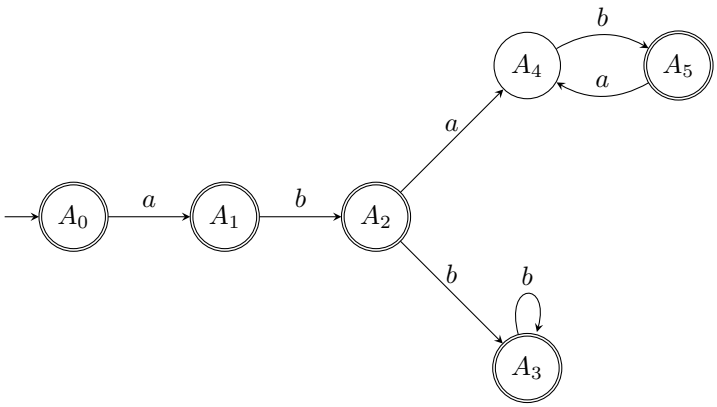
Folgender Automat akzeptiert das Produkt von a und b^*:



Folgender Automat akzeptiert die Vereinigung von $(ab)^*$ und ab^*:



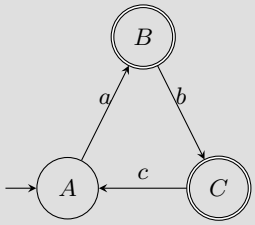
Dieses Verfahren liefert offensichtlich sehr große Automaten. Selbst deterministisch lässt sich ein sehr viel kleinerer Automat angeben:



Der Nutzen dieses Verfahrens liegt also nicht in der Übersichtlichkeit der entstehenden Automaten. Stattdessen hilft uns dieses Verfahren, weil wir so algorithmisch zu **jedem** regulären Ausdruck einen entsprechenden Automaten bekommen.

Rückrichtung. Der Beweis der Rückrichtung, also dass es zu jeder Sprache, die von einem DFA akzeptiert wird, auch einen regulären Ausdruck gibt, der sie beschreibt, ist mathematisch schwieriger und nicht sonderlich gut intuitiv nachzuvollziehen. Es weder für die Übungsblätter noch für die Klausur relevant, die Konstruktion zu verstehen oder gar durchführen zu können. Es ist nur wichtig, zu wissen, dass diese Konstruktion existiert und dass deswegen jede Sprache, die von einem DFA erkannt wird, auch von einem regulären Ausdruck beschrieben wird. Für die Interessierten ist die Konstruktion hier trotzdem gegeben.

Beispielhaft gehen wir die Konstruktion an folgendem Automaten durch:



Wir indizieren zunächst unsere Zustände. (Das heißt, wir „nummerieren“ die Zustände durch.) A sei q_1, B sei q_2, C sei q_3. Nun konstruieren wir „Hilfssprachen“, die wir $R^k_{i,j}$ nennen. Die Sprache $R^k_{i,j}$ bestehe aus allen Wörtern w, sodass der Automat, wenn er w in Zustand q_i einliest, in Zustand q_j landet, ohne dass einer der Zwischenzustände einen Index hat, der größer als k ist. Man bedenke, dass Start- und Endzustand hier nicht als Zwischenzustände zählen. Es ist also erlaubt, dass i und j größer als k sind, solange q_i und q_j nicht auch noch als Zwischenzustand besucht werden.

Es wirkt nicht ansatzweise intuitiv, weshalb es zielführend sein sollte, sich diese Sprachen anzuschauen. Wir werden aber einerseits sehen, dass sich für diese Sprachen recht leichte reguläre Ausdrücke konstruieren lassen, aus denen sich dann das Ergebnis zusammensetzen lässt.

Überlegen wir uns zunächst, was es bedeutet, eine der Mengen $R^0_{i,j}$ zu konstruieren. Keiner der Zwischenzustände darf einen Index haben, der größer als 0 ist. Es ist aber jeder Index größer als 0. Dies bedeutet also, dass es keinen Zwischenzustand geben darf. Wörter in $R^0_{i,j}$ dürfen also maximal die Länge 1 haben. Es sind alle Zeichen, die direkt in einem Schritt von q_i nach q_j führen. (Ist $i = j$, so führt auch ε von q_i nach q_j.) Für diese endlichen Sprachen lassen sich auch sehr leicht reguläre Ausdrücke finden. Den regulären Ausdruck zur Sprache $R^k_{i,j}$ bezeichnen wir als $\gamma^k_{i,j}$.

$\gamma^0_{1,1} = \varepsilon$
$\gamma^0_{1,2} = a$
$\gamma^0_{1,3} = \emptyset$
$\gamma^0_{2,1} = \emptyset$
$\gamma^0_{2,2} = \varepsilon$
$\gamma^0_{2,3} = b$
$\gamma^0_{3,1} = c$
$\gamma^0_{3,2} = \emptyset$
$\gamma^0_{3,3} = \varepsilon$

Nun wollen wir nutzen, dass wir die Sprachen $R^0_{i,j}$, kennen, um die Sprachen $R^1_{i,j}$ zu konstruieren. Betrachten wir diesen Schritt etwas allgemeiner. Angenommen wir kennen bereits die Ausdrücke der Sprachen $R_(i,j)^k$ für ein gegebenes k. Wir wollen nun Ausdrücke für die Sprachen $R^{k+1}_{i,j}$ finden. Wörter, die von q_i nach q_j führen, ohne dabei Zustände mit größerem Index als $k+1$ zu besuchen, fallen in zwei Kategorien: Entweder sie führen einen Pfad entlang, der q_{k+1} besucht oder sie tun es nicht. Die letzteren Wörter werden offenbar bereits durch den Ausdruck $\gamma^k_{i,j}$ beschrieben.

Die ersteren besuchen q_{k+1}. Vor dem ersten Besuch von q_{k+1} wurden nur Zwischenzustände besucht, deren Index maximal k ist. Der Teil des Wortes, der zum ersten Besuch

von q_{k+1} führt, wird also durch $\gamma^k_{i,k+1}$ beschrieben. Der Teil des Wortes vom ersten bis zum letzten Besuch von q_{k+1} führt, besteht aus beliebig vielen Wörtern, die durch $\gamma^k_{k+1,k+1}$ beschrieben werden. Der gesamte Teil kann also durch $(\gamma^k_{k+1,k+1})^*$ beschrieben werden. Der Teil nach dem letzten Besuch von q_{k+1} führt wieder nur über Zwischenzustände, deren Index kleiner als k ist und wird somit durch $\gamma^k_{k+1,j}$ beschrieben. Das gesamte Wort erhalten wir also durch $\gamma^k_{i,k+1}(\gamma^k_{k+1,k+1})^*\gamma^k_{k+1,j}$.

Alle Wörter erhalten wir also durch die Vereinigung $\gamma^k_{i,j}|\gamma^k_{i,k+1}(\gamma^k_{k+1,k+1})^*\gamma^k_{k+1,j}$.

$\gamma^1_{1,1} = \varepsilon|\varepsilon(\varepsilon)^*\varepsilon$
$\gamma^1_{1,2} = a|\varepsilon(\varepsilon)^*a$
$\gamma^1_{1,3} = \emptyset|\varepsilon(\varepsilon)^*\emptyset$
$\gamma^1_{2,1} = \emptyset|\emptyset(\varepsilon)^*\varepsilon$
$\gamma^1_{2,2} = \varepsilon|\emptyset(\varepsilon)^*a$
$\gamma^1_{2,3} = b|\emptyset(\varepsilon)^*\emptyset$
$\gamma^1_{3,1} = c|c(\varepsilon)^*\varepsilon$
$\gamma^1_{3,2} = \emptyset|c(\varepsilon)^*a$
$\gamma^1_{3,3} = \varepsilon|c(\varepsilon)^*\emptyset$

Wir könnten mit diesen Ausdrücken fortfahren. Rein algorithmisch würden wir dies auch tun. Der Einfachheit zuliebe werden wir aber einige Vereinfachungen durchführen. Wir können $(\varepsilon)^*$ auslassen, Produkte mit $\emptyset$ werden zu $\emptyset$ und Vereinigungen mit $\emptyset$ können ausgelassen werden.

$\gamma^1_{1,1} = \varepsilon$
$\gamma^1_{1,2} = a$
$\gamma^1_{1,3} = \emptyset$
$\gamma^1_{2,1} = \emptyset$
$\gamma^1_{2,2} = \varepsilon$
$\gamma^1_{2,3} = b$
$\gamma^1_{3,1} = c$
$\gamma^1_{3,2} = ca$
$\gamma^1_{3,3} = \varepsilon$

Wir fahren fort:

$\gamma^2_{1,1} = \varepsilon|a(\varepsilon)^*\emptyset$
$\gamma^2_{1,2} = a|a(\varepsilon)^*\varepsilon$
$\gamma^2_{1,3} = \emptyset|a(\varepsilon)^*b$
$\gamma^2_{2,1} = \emptyset|\varepsilon(\varepsilon)^*\emptyset$
$\gamma^2_{2,2} = \varepsilon|\varepsilon(\varepsilon)^*\varepsilon)$
$\gamma^2_{2,3} = b|\varepsilon(\varepsilon)^*b$
$\gamma^2_{3,1} = c|ca(\varepsilon)^*\emptyset$
$\gamma^2_{3,2} = ca|ca(\varepsilon)^*\varepsilon$
$\gamma^2_{3,3} = \varepsilon|ca(\varepsilon)^*b$

Vereinfacht:

$\gamma^2_{1,1} = \varepsilon$
$\gamma^2_{1,2} = a$
$\gamma^2_{1,3} = ab$
$\gamma^2_{2,1} = \emptyset$
$\gamma^2_{2,2} = \varepsilon$
$\gamma^2_{2,3} = b$
$\gamma^2_{3,1} = c$
$\gamma^2_{3,2} = ca$
$\gamma^2_{3,3} = \varepsilon|cab$

Nächster Schritt:

$\gamma^3_{1,1} = \varepsilon|ab(\varepsilon|cab)^*c$
$\gamma^3_{1,2} = a|ab(\varepsilon|cab)^*ca$
$\gamma^3_{1,3} = ab|ab(\varepsilon|cab)^*(\varepsilon|cab)$

$\gamma^3_{2,1} = \emptyset|b(\varepsilon|cab)^*c$
$\gamma^3_{2,2} = \varepsilon|b(\varepsilon|cab)^*ca$
$\gamma^3_{2,3} = b|b(\varepsilon|cab)^*(\varepsilon|cab)$
$\gamma^3_{3,1} = c|(\varepsilon|cab)(\varepsilon|cab)^*c$
$\gamma^3_{3,2} = ca|(\varepsilon|cab)(\varepsilon|cab)^*ca$
$\gamma^3_{3,3} = (\varepsilon|cab)|(\varepsilon|cab)(\varepsilon|cab)^*(\varepsilon|cab)$

Auch hier lassen sich noch Vereinfachungen durchführen. Diese sind aber nicht ganz so offensichtlich, wie die bisherigen. Deshalb (und auch um zu demonstrieren, wie groß die Ausdrücke bei rein algorithmischen abarbeiten werden,) werden wir diese Vereinfachungen nicht durchführen. Es fällt auf, dass der größte vorkommende Index bereits 3 ist. Es kann also keine Zwischenzustände mit einem Index geben, der größer als 3 ist. $\gamma^3_{i,j}$ beschreibt alle Wörter, die von q_i nach q_j führen ohne Zwischenzustände zu besuchten, deren Index größer als 3 ist. Offenbar trifft die letzte Bedingung aber immer zu, weshalb wir sie auch fallen lassen können. $\gamma^3_{i,j}$ beschreibt einfach alle Wörter, die von q_i nach q_j führen.

Wir können nun den Ausdruck final hinschreiben. Alle akzeptierten Wörter führen vom Startzustand zu einem der Endzustände. Insbesondere führen sie entweder von q_1 nach q_2 oder von q_1 nach q_3. Es handelt sich also um die Vereinigung der Sprachen, die von $\gamma^3_{1,2}$ und $\gamma 1, 3^3$ beschrieben werden. Der reguläre Ausdruck für die Sprache ist also $(a|ab(\varepsilon|cab)^*ca)|(ab|ab(\varepsilon|cab)^*(\varepsilon|cab))$.

Auch hier sehen wir, dass das Ergebnis sehr ineffizient lang ist. Der Vorteil ist aber wieder, dass wir es rein algorithmisch erhalten haben und wir das Verfahren mit **jedem** regulären Ausdruck durchführen können.

2.6. **Reguläre Grammatiken.** Wir lernen nun eine weitere Art kennen, Sprachen zu beschreiben. Eine Grammatik ist ein Werkzeug, mit dem wir Wörter einer Sprache Schritt für Schritt aufbauen können. Eine Grammatik nutzt dafür also notwendigerweise die Zeichen des Alphabets Σ, über dem die Sprache definiert ist. Zusätzlich zu diesen Zeichen nutzt eine Grammatik aber auch noch eine Menge an Hilfszeichen V. Diese Hilfszeichen werden in dem Wort, wenn es fertig aufgebaut wurde nicht mehr vorkommen, können aber sehrwohl in den Zwischenschritten vorkommen. Die Zeichen in Σ werden daher auch als **Terminale** und die Zeichen in V auch als **Nonterminale** bezeichnet. Für gewöhnlich schreiben wir Nonterminale als Großbuchstaben und Terminale als Kleinbuchstaben.

Jeder Schritt besteht daraus, dass wir einen Teil des Wortes nehmen und durch ein anderes Wort ersetzen. Später erlauben wir es, dass die Grammatiken beliebig komplizierte Teilwörter durch andere Wörter ersetzen - fürs erste betrachten wir aber nur solche Grammatiken, in denen der ersetzte Teil ein einziges Nonterminal ist.

Dieses Ersetzen kann nicht beliebig geschehen. Jede Grammatik hat eine vorgegebene Liste an Regeln (sogenannte **Produktionen**) anhand derer diese Ersetzungen stattfinden.

Am Start des Prozesses besteht das Wort nur aus einem einzigen Nonterminal. Welches Nonterminal dieses Startzeichen ist, ist durch die Grammatik vorgegeben.

Beispiel. Wir nehmen eine Grammatik mit nur einem einzigen Nonterminal. Dieses Nonterminal bezeichnen wir als S. Die Menge der Nonterminale sei $\{a, b\}$. Die Grammatik hat zwei Produktionen, die wir als $S \to \varepsilon$ und $A \to aSb$ schreiben. Wir können S also durch das leere Wort oder durch das Wort aSb ersetzen.

Wiederholte Anwendungen der Produktion $S \to aSb$ haben folgenden Effekt:
$S \to aSb \to aaSbb \to aaaSbbb \to aaaaSbbbb$

Wie erwähnt darf das Ergebnis kein Nonterminal mehr enthalten. Irgendwann müssen wir also die Produktion $S \to \varepsilon$ anwenden.

$$S \to aSb \to aaSbb \to aaaSbbb \to aaaaSbbbb \to aaaabbbb$$

Auf diese Weise haben wir nun das Wort *aaaabbbb* erzeugt.

Formal. Eine Grammatik ist formal definiert durch die Angabe von vier Informationen:

(1) Die Menge V der Nonterminale. Die einzige gestellte Bedinung ist, dass nur endlich viele verschiedene Nonterminale vorliegen.
(2) Das Alphabet Σ der Terminale. Darauf zu achten ist, dass kein Zeichen gleichzeitig in der Menge der Terminale und in der Menge der Nonterminale sein darf.
(3) Die Menge P der Produktionen. Formal ist jede Produktion ein Paar aus Wörtern, deren Zeichen aus V oder Σ sind (Es ist auch erlaubt, dass sowohl Zeichen aus V als auch Zeichen aus Σ in einem Wort vorkommen). Wir fordern, dass das erste der beiden Wörter nicht leer ist. Später interpretieren wir dieses Paar so, dass wir ein Vorkommen des ersten Wortes als Teilwort durch das zweite ersetzen können.
(4) Ein bestimmtes Nonterminal S aus V, das das Startsymbol darstellt.

Erzeugte Sprache. Wörter deren Zeichen in V oder Σ liegen, nennen wir **Satzformen** Einen Schritt, in dem wir ein Teilwort einer Satzform gemäß der Produktionen der Grammatik ersetzen, nennen wir einen **Ableitungsschritt**. Können wir mithilfe einer einzigen Anwendung einer Produktion aus der Satzform u die Satzform v erzeugen, sagen wir, dass u **direkt ableitbar** zu v ist. Können wir mithilfe einer beliebigen Anzahl an Anwendungen von Produktionen aus der Satzform u die Satzform v erzeugen, sagen wir, dass u **ableitbar** zu v ist.

Wir sagen, dass ein Wort $w \in \Sigma^*$ von der Grammatik **erzeugt** wird, wenn das Startsymbol ableitbar zu w ist. Die Sprache aller Wörter, die von einer Grammatik erzeugt werden, nennen wir die von der Grammatik **erzeugte Sprache**.

Die Grammatik des obigen Beispiels erzeugt offenbar alle Wörter $a^n b^n$ mit $n \in \mathbb{N}$. Wie wir bereits gesehen haben, ist diese Sprache nicht regulär. Es sind also auch schon recht simple Grammatiken in der Lage, nicht reguläre Sprachen zu erzeugen. Wir können aber eine recht leichte Bedingung an die Grammatik stellen, die dazu führt, dass tatsächlich alle erzeugten Sprachen regulär sind. (Und trotzdem auch immer noch jede reguläre Sprache auch erzeugt wird.)

Reguläre Grammatiken. Wir nennen eine Grammatik regulär, wenn auf der linken Seite jeder Produktion nur ein einziges Nonterminal steht und auf der rechten Seite jeder Produktion entweder das leere Wort oder ein einziges Terminal oder ein einziges Terminal gefolgt von einem einzigen Nonterminal steht. Der Fall, dass auf der rechten Seite das leere Wort steht ist nur dann erlaubt, wenn auf der linken Seite des Wortes das Startsymbol steht. **Jedes andere Nonterminal darf nicht zu dem leeren Wort abgeleitet werden.**

Beispiel. Wir betrachten die Grammatik mit $V = \{I, A, B\}, \Sigma = \{a, b\}, P = \{I \to aA, I \to bB, I \to b, A \to aI, A \to bI, B \to aI, B \to bI\}$ und dem Startsymbol I. Diese kann auf folgende Weise das Wort *aabbbab* erzeugen:

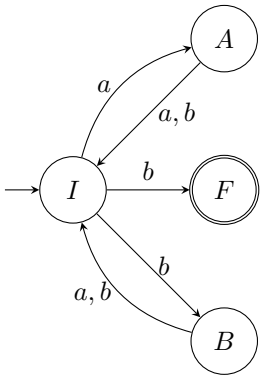
$$I \to aA \to aaI \to aabB \to aabbI \to aabbbB \to aabbbaI \to aabbbab$$

Interpretation regulärer Grammatiken. Wir sehen schnell, dass jede Satzform, die als Zwischenschritt in der Ableitung aus einer regulären Grammatik entsteht, aus einer Folge von Terminalen gefolgt von einem einzigen Nonterminal besteht. Wir können dies folgendermaßen interpretieren. Wir stellen uns vor, wir befinden uns in einem NFA, in dem die Zustände genauso heißen wie die Nonterminale unserer Grammatik. Die Satzform $aabbbB$ können wir dann zum Beispiel auffassen als „Bisher wurde das Wort $aabbb$ eingelesen und der NFA befindet sich jetzt im Zustand B“ interpretiert werden.

Tatsächlich können wir dies sehr leicht formalisieren. Wir können zu jeder regulären Grammatik einen NFA erzeugen, der genau auf diese Weise operiert. In diesem NFA ist kein Zustand ein Endzustand mit Ausnahme eines einzelnen extra zu diesem Zweck geschaffenen Zustands X. Dieser Zustand ist der einzige Zustand, der nicht zu einem der Nonterminale korrespondiert. Ansonsten gibt es zu jedem Nonterminal genau einen Zustand. Wir fügen nun für jede Produktion genau eine Kante in den NFA ein. Eine Produktion der Form $A \to aB$ führt zu einer Kante, die von Zustand A nach Zustand B führt und mit a beschriftet ist. Eine Produktion Der Form $A \to a$ beendet die Ableitung, führt also in einen akzeptierenden Zustand. Für eine solche Produktion fügen also eine Kante von A nach X ein, die mit a beschriftet ist. Existiert die Produktion $S \to \varepsilon$, fügen wir einen ε-Übergang von S nach X ein.

So erhalten wir bereits zu jeder regulären Grammatik einen NFA, der die von der Grammatik erzeugte Sprache akzeptiert. Da NFAs nur reguläre Sprachen akzeptieren, zeigt dies bereits, dass jede von einer regulären Grammatik erzeugte Sprache regulär ist.

Beispiel. Die Grammatik des oben genannten Beispiels führt zu folgendem NFA:



Umkehrung dieser Konstruktion. Diese Konstruktion können wir auch in die andere Richtung durchführen. Gegeben sei ein NFA. Wir wollen eine reguläre Grammatik konstruieren, die die vom NFA erzeugte Sprache erzeugt. Dann wählen wir eine Grammatik, die für jeden Zustand des NFAs genau ein Nonterminal hat. Das Startsymbol sei dasjenige Nonterminal, das dem Startzustand entspricht. Für jede Kante im NFA, die vom Zustand A zum Zustand B geht und mit a beschriftet ist, fügen wir die Produktion $A \to aB$ ein. Ist B ein akzeptierender Zustand, fügen wir ebenfalls die Produktion $A \to a$ ein. Ist der Startzustand S akzeptierend, ergänzen wir noch die Produktion $S \to \varepsilon$ ein.

Wenden wir diese Konstruktion auf das Beispiel oben an, erhalten wir die Grammatik mit den Produktionen $I \to aA, I \to bB, I \to bF, I \to b, A \to aI, A \to bI, B \to aI, B \to bI$. Dies ist beinahe die Grammatik, mit der wir gestartet haben, nur dass diese Grammatik noch das zusätzliche Nonterminal F und die zusätzliche Produktion $I \to bF$ hat. Dies ändert jedoch nichts an der erzeugten Sprache.

In obigem Beispiel haben wir behauptet, die Produktion $I \to bF$ würde nichts an der erzeugten Sprache ändern. Dies liegt daran, dass es keine Produktion gibt, mit der das Nonterminal F weiterverarbeitet werden kann. Haben wir erst einmal das Nonterminal F erzeugt, gibt es keine Produktionen mehr, die uns zu einem Wort führen, das nur noch aus Terminalen besteht. Ein Wort gilt nur als erzeugt, wenn es keine Nonterminale mehr erhält. Auf diesem Weg werden wir also kein weiteres Wort mehr erzeugen. Da somit keine Ableitung, die zu einem tatsächlich erzeugten Wort führt, die Produktion $I \to bF$ nutzen kann, ist die Grammatik die gleiche, wie wenn es diese Produktion gar nicht gegeben hätte.

Dies wirkt erst überraschend. Man könnte denken, das eine Grammatik, die mehr Ableitungsschritte erlaubt, auch größere Sprachen erzeugen. Kommen wir aber wie hier in einen Fall, in dem von diesen zusätzlichen Schritten keine weiteren Schritte mehr wegführen, können diese zusätzlichen Möglichkeiten einfach ignoriert werden. Eine weitere Möglichkeit ist, dass wir zwar immer noch eine weitere Produktion anwenden können, diese Produktionen aber auch immer neue Nonterminale hinzufügen und wir so nur in eine Endlosschleife kommen und nie alle Nonterminale loswerden. Dies wäre zum Beispiel der Fall, wenn wir der vorherigen Grammatik die Produktion $F \to aF$ hinzufügen würden. Diese könnten wir immer noch anwenden, aber wir würden nie das Nonterminal F loswerden. Auch so würden wir also keine neuen Wörter erzeugen.

In der Definition einer Produktion wird nirgendwo gefordert, dass die linke Seite einer Produktion mindestens ein Nonterminal enthält. Eine mögliche Grammatik wäre also die Grammatik mit nur einem einzigen Nonterminal S, das auch das Startsymbol ist, dem Terminal a und den Produktionen $S \to a, a \to aa$. Diese Grammatik erzeugt $\{a^n : n \in \mathbb{N}\}$.

Wir können aber keine Grammatik ohne Nonterminal definieren, da die Definition voraussetzt, dass das Startsymbol ein Nonterminal ist. Insbesondere muss es also immer mindestens ein Nonterminal geben.

Typ-3 Sprachen. Wir haben oben zwei Dinge gezeigt. Wir können zu jeder regulären Grammatik einen NFA konstruieren, der genau die Sprache akzeptiert, die von der Grammatik erzeugt wird und wir können zu jedem NFA eine reguläre Grammatik konstruieren, die genau die Sprache erzeugt, die von dem NFA akzeptiert wird. Ersteres zeigt, dass jede von einer regulären Grammatik erzeugte Sprache regulär ist. Letzteres zeigt, dass jede reguläre Sprache von einer regulären Grammatik erzeugt wird. Insbesondere sind die Klasse der regulären Sprachen und die Klasse der von regulären Grammatiken erzeugten Sprachen also gleich. Wir nennen diese Sprachen auch **Sprachen von Typ-3**.

Wir haben nun drei Möglichkeiten kennengelernt, Sprachen von Typ-3 zu definieren:

- Alle Sprachen, die durch einen regulären Ausdruck beschrieben werden
- Alle Sprachen, die von einem DFA akzeptiert werden
- Alle Sprachen, die von einer regulären Grammatik erzeugt werden

Wie wir nun gezeigt haben, sind diese Definitionen völlig äquivalent und definieren das Gleiche. Es kann hilfreich sein, auf diese verschiedenen Definitionen zurückgreifen zu können, da es je nach Kontext mal einfacher sein kann, einen regulären Ausdruck, einen DFA oder eine reguläre Grammatik zu verwenden. Zum Beispiel sind DFAs am besten geeignet, um zu prüfen, ob ein gegebenes Wort tatsächlich zu der Sprache gehört. Ist uns eine Sprache per Beschreibung durch einen regulären Ausdruck gegeben, kann es hilfreich sein, einen äquivalenten DFA zu konstruieren und diesen zu nutzen, um zu klären, ob ein gegebenes Wort zu der Sprache gehört.

2.7. **String Matching mit endlichen Automaten.** In diesem Abschnitt werden wir ein konkretes Beispiel behandeln. Die Frage ist folgende: Gegeben sei ein Wort

m. Wir wollen einen DFA finden, der zu jeder Eingabe entscheidet, ob sie das Wort m als Teilwort enthält. Zunächst ist nicht klar, ob es so einen DFA überhaupt gibt. Wir wissen erst, dass die Sprache der Wörter, die m als Teilwort enthalten, regulär ist, wenn wir einen solchen DFA gefunden haben. Wie wir sehen werden, wird dies aber möglich sein.

Wir stellen einige Beobachtungen auf. Zunächst fällt auf, dass unser DFA nur ein einziges Vorkommen von m in der Eingabe finden muss. Kommt m bereits einmal in der Eingabe vor, ist für den DFA irrelevant, welche Zeichen darauf folgen. Nichts, was auf m folgen kann, kann etwas daran ändern, dass das m einmal in dem Wort vorgekommen ist. Es reicht also, wenn der DFA bereits nach dem ersten Vorkommen von m in einen akzeptierenden Zustand wechselt und diesen dann nicht mehr verlässt.

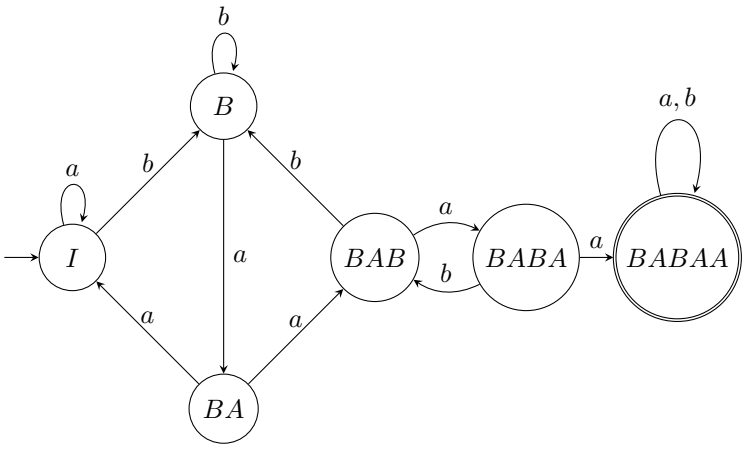
Die nächste Beobachtung lässt sich gut an einem Beispiel erkennen. Sei das gesuchte Wort $m = babaa$. Angenommen, der DFA hat bereits einen Teil der Eingabe eingelesen. Sei dieser Teil das Wort *abbababbbbab*. Ein DFA hat nur sehr begrenzten Speicher. Die einzige Art, wie Informationen über die bisherige Eingabe gespeichert werden können, läuft über die Zustände. Der DFA kann sich also nicht die gesamte bisherige Eingabe merken. Wir sehen aber, dass ein großer Teil der bisherigen Eingabe für das Ergebnis vollkommen irrelevant ist. Zum Beispiel enthält die bisherige Eingabe zwar das Teilwort *baba*, was bereits fast das gesuchte Teilwort *babaa* ist, aber da auf dieses Teilwort in der Eingabe ein *b* folgt, kann diese Stelle der Eingabe, unabhängig von den Zeichen, die noch folgen werden, am Ende nicht mehr Teil eines Vorkommens von *babaa* sein. Der DFA kann diesen Teil also vergessen - er hat keinen Einfluss auf das Ergebnis. Relevant für das Ergebnis sind nur diejenigen Teile des Wortes, die, abhängig von der weiteren Eingabe, noch Teil eines Vorkommens von *babaa* sein könnten.

In diesem Fall kann der ganze Teil mit *abbababbb* verworfen werden. Nur der Teil *bab* am Ende könnte noch Teil von *babaa* sein. (Wenn nämlich der Rest der Eingabe mit *aa* anfängt.) Allgemein muss der DFA sich also nur die letzten wenigen Zeichen der Eingabe merken - und zwar nur den Teil, der noch ein Anfang von m sein könnte. Im Beispiel $m = babaa$ gibt es also sechs mögliche Dinge, die der DFA gespeichert haben kann:

- Kein Teil der Eingabe könnte Teil von *babaa* werden.
- Die bisherige Eingabe endet auf *b* und das ist der einzige Teil, der noch Teil von *babaa* werden könnte.
- Die bisherige Eingabe endet auf *ba* und das ist der einzige Teil, der noch Teil von *babaa* werden könnte.
- Die bisherige Eingabe endet auf *bab* und das ist der einzige Teil, der noch Teil von *babaa* werden könnte.
- Die bisherige Eingabe endet auf *baba* und das ist der einzige Teil, der noch Teil von *babaa* werden könnte.
- Es kam das Wort *babaa* in der Eingabe vor.

Wir können so einen DFA mit sechs Zuständen bauen, wobei jeder dieser Zustände für eine dieser Informationen steht. Der Zustand gibt uns dann an, was die letzten relevanten Zeichen waren. Wird nun ein neues Zeichen eingelesen, reicht uns diese Information auch aus, um den nächsten Zustandsübergang zu bestimmen. Wird das nächste „korrekte“ Zeichen eingelesen, rücken wir einen Zustand weiter vor. Folgt als nächstes ein falsches Zeichen, wirft uns das zurück - aber nicht notwendigerweise bis ganz an den Anfang. Angenommen, wir befinden uns in dem Zustand, der uns sagt, dass die bisherige Eingabe auf *baba* endet. Lesen wir nun ein *b*, ein endet die bisherige Eingabe auf *babab*. Der hintere Teil *bab* kann durchaus noch Teil von *babaa* werden. Wir rücken daher nur bis zu dem Zustand zurück, der *bab* speichert und

nicht ganz zurück zum Startzustand. Der gesamte DFA sieht dann folgendermaßen aus:



Der Automat, den wir auf diese Weise erhalten, ist auch tatsächlich minimal. Wir erhalten genau einen Zustand für jeden möglichen Präfix von m. (Inklusive dem leeren Wort als Startzustand und m selbst als Endzustand.) Anhand der Myhill-Nerode Klassen können wir auch sehen, dass diese Zustände auch tatsächlich gebraucht werden. Zu einem Präfix x von m sei y gegeben, sodass $xy = m$ ist. Ist x' dann ebenfalls ein Präfix von m, der kürzer als x ist, so ist $x'y$ kürzer als m. Insbesondere kann $x'y$ nicht m als Teilwort enthalten. Dagegen wird $xy = m$ sich selbst als Teilwort enthalten. Ein DFA, der alle Wörter akzeptiert, die m als Teilwort enthalten, wird also xy akzeptieren und $x'y$ nicht akzeptieren. Deshalb können x und x' nicht in der gleichen Äquivalenzklasse bezüglich der Myhill-Nerode Relation liegen. Der Automat braucht also auch mindestens einen Zustand pro Präfix von m.

2.8. **Entscheidungsprobleme für reguläre Sprachen.**

Das Wortproblem. Wir haben uns nun sehr ausgiebig mit regulären Sprachen und ihren Eigenschaften beschäftigt. Bei verschiedenen Sprachklassen werden wir uns immer wieder ähnliche Fragen stellen und werden uns fragen, ob sie entscheidbar sind. Das heißt: Können wir überhaupt rein algorithmisch die Antwort auf die Frage herausfinden?

Zum Beispiel: Gegeben sei ein Wort w. Uns ist eine reguläre Sprache gegeben - durch einen regulären Ausdruck, DFA oder reguläre Grammatik. Wir stellen die Frage, ob die Sprache das Wort w enthält. Dies scheint uns recht einfach zu klären sein. Wir nehmen uns einen DFA, der die Sprache akzeptiert. (Möglicherweise ist uns die Sprache über einen regulären Ausdruck oder eine reguläre Grammatik gegeben. Wir haben aber bereits rein algorithmische Konstruktionen kennengelernt, um aus diesen einen DFA zu erhalten.) An diesem DFA können wir leicht ablesen, welche Zustände er mit der Eingabe w durchläuft und ob er dabei in einem akzeptierenden Zustand endet.

Dies wirkte nun vielleicht sehr trivial. Selbstverständlich können wir prüfen, ob der DFA das Wort w akzeptiert. Wir werden aber noch weitere Klassen von Sprachen kennenlernen, bei denen uns nicht so simple Beschreibungsmöglichkeiten zur Verfügung stehen, wie ein DFA. Bei diesen wird selbst eine so leichte Frage wie „enthält die Sprache das Wort w?“ nur komplizierter oder teilweise sogar gar nicht mehr algorithmisch entscheidbar sein. Diese Frage, ist also interessanter, als sie auf den ersten Blick wirken mag. Wir nennen diese Frage auch das **Wortproblem**.

Das Leerheitsproblem. Das **Leerheitsproblem** ist die Frage, ob eine gegebene Sprache leer ist - also ob sie überhaupt ein Wort enthält oder nicht. Auch hier wirkt es anhand eines DFAs sehr trivial entscheidbar. Existiert im DFA ein Pfad vom Startzustand zu einem Endzustand, ist die Sprache nicht leer, sonst ist sie es. Dies können wir zum Beispiel mit Tiefensuche oder Breitensuche entscheiden.

Auch hier werden wir noch Sprachklassen sehen, bei denen dieses Problem nicht mehr algorithmisch entscheidbar sein wird.

Das Endlichkeitsproblem. Das **Endlichkeitsproblem** ist die Frage, ob eine gegebene reguläre Sprache nur endlich viele Wörter enthält. Auch dies können wir anhand eines DFAs entscheiden, wenn auch nicht ganz so trivial, wie die vorherigen Fragen. Wir erinnern uns an das Pumping Lemma. Bei unendlichen Sprachen haben wir Wörter, deren Länge größer ist als die Anzahl der Zustände. Bei diesen Wörtern durchläuft der DFA einen Zyklus an Zuständen, der beliebig auf- und abgepumpt werden kann. Insbesondere existiert ein Zyklus von Zuständen, der vom Startzustand aus erreicht werden kann, von dem aus auch wieder ein Endzustand erreichbar ist.

Umgekehrt sehen wir: Wenn ein solcher Zyklus existiert, ist die Sprache auch unendlich, da wir diesen Zyklus beliebig oft durchlaufen können und so auf beliebig viele verschiedene akzeptierte Wörter kommen.

Die Frage nach der Endlichkeit der Sprache kann also auf die Frage nach der Existenz eines solchen Zyklus zurückgeführt werden. Mit Tiefensuche können wir vom Startzustand aus nach Zyklen suchen und dann ebenfalls danach suchen, ob von den gefundenen Zyklen auch ein Endzustand erreichbar ist. So können wir bei DFAs das Endlichkeitsproblem entscheiden.

Das Schnittproblem. Das **Schnittproblem** ist die Frage, ob es zu zwei gegebenen Sprachen ein Wort gibt, das in beiden Sprachen enthalten ist. Hierfür müssen wir uns keine neuen Ideen einfallen lassen, sondern können uns auf unsere Lösung des Leerheitsproblems verlassen. Es gibt genau dann ein Wort, dass in beiden Sprachen enthalten ist, wenn der Schnitt der Sprachen nicht leer ist. Wir wissen bereits, wie wir algorithmisch einen DFA konstruieren, der den Schnitt zweier Sprachen erkennt. Auf diesen können wir nun unsere Lösung des Leerheitsproblems anwenden.

Wir haben es hier ohne große Probleme geschafft, einen Algorithmus zur Entscheidung des Schnittproblems zu finden. Tatsächlich wird die Klasse der regulären Sprachen aber die einzige Sprachklasse dieser Vorlesung sein, bei der das Schnittproblem algorithmisch entscheidbar ist. Für alle Sprachklassen, die wir im Folgenden kennenlernen werden, gibt es keinen solchen Algorithmus mehr.

Das Äquivalenzproblem. Das Äquivalenzproblem ist die Frage, ob zwei Sprachen gleich sind. In Abschnitt 2.3 haben wir bereits eine algorithmische Lösung für dieses Problem gefunden. (Abschnitt: DFA Äquivalenztest) Die Lösung des Leerheitsproblems gibt uns nun aber auch noch einen weiteren Weg. Wir können Gleichheit auch folgendermaßen formulieren: Zwei Sprachen A und B sind genau dann gleich, wenn es kein Wort gibt, das in A, aber nicht in B liegt **und** kein Wort, das in B, aber nicht in A liegt. Die Menge der Wörter die in A, aber nicht in B liegen, ist $A \cap \overline{B}$. Die Menge der Wörter, die in B, aber nicht in A liegen, ist $B \cap \overline{A}$. Die Menge der Wörter, die in einer der beiden Mengen liegen, ist $(A \cap \overline{B}) \cup (B \cap \overline{A})$. Diese Menge ist genau dann leer, wenn A und B gleich sind. Wir können einen DFA für diese Menge konstruieren, indem wir die Konstruktionen von Komplement, Schnitt und Vereinigung hintereinanderschalten. Auf diesen DFA können wir dann die Lösung für das Leerheitsproblem anwenden und somit prüfen, ob A und B gleich sind.

3. Kontextfreie Sprachen und Kellerautomaten

3.1. **Chomsky-Typen.** Wir haben reguläre Sprachen als Typ-3 Sprachen bezeichnet. Dies legt nahe, dass es auch noch andere Typen von Sprachen gibt. Tatsächlich ist dies der Fall. Wir definieren nun auch noch Sprachen vom Typ-0 bis 2. Diese Sprachtypen geben uns einen Plan für einen großen Teil des Rests der Vorlesung. Die Typ-3 Sprachen (regulären Sprachen) haben wir bereits sehr ausführlich behandelt. Wir werden uns in diesem Kapitel mit Typ-2 Sprachen beschäftigen und in dem darauffolgenden Kapitel mit Sprachen von Typ-1 und Typ-0.

Definition der Grammatiktypen. Wir bezeichnen jede Grammatik als eine Grammatik von Typ-0.

Wenn die Grammatik keine Produktionen hat, bei denen die rechte Seite kürzer als die linke Seite ist, bezeichnen wir die Grammatik zusätlich als eine Grammatik von Typ-1.

Besteht zusätzlich die linke Seite jeder Produktion nur aus einem einzigen Nichtterminal, bezeichnen wir die Grammatik zusätzlich als eine Grammatik von Typ-2.

Besteht die rechte Seite jeder Produktion zusätzlich nur aus einem Terminal oder einem Terminal gefolgt von einem Nichtterminal, bezeichnen wir die Grammatik zusätzlich als eine Grammatik von Typ-3.

Offenbar können Grammatiken von Typ 1 das leere Wort nicht erzeugen. Wir führen daher die Ausnahmeregel ein, dass das Startsymbol auf das leere Wort produziert werden kann, wenn das Startsymbol bei keiner Produktion auf der rechten Seite steht.

Wir definieren diese Typen auch für Sprachen. Eine Sprache heißt Typ-i Sprache, wenn es eine Grammatik von Typ-i gibt, die diese Sprache erzeugt.

Formale Stolperfalle. Wir betrachten die Grammatik mit den Produktionen $A \to AA, AA \to A, A \to a$. Aufgrund der Produktion $AA \to A$, bei der die rechte Seite kürzer als die linke Seite ist, kann es sich nicht um eine Typ-1, Typ-2 oder Typ-3 Grammatik handeln. Die erzeugte Sprache ist aber $\{a\}^+$, eine reguläre Sprache und somit eine Sprache von Typ-3. Dieses Beispiel zeigt, dass eine Sprache einen Typ haben kann, den die erzeugende Grammatik nicht hat. Die Definition des Typs einer Sprache setzt nämlich nur voraus, dass es **eine** erzeugende Grammatik des entsprechenden Typs gibt - nicht aber, dass **jede**, Grammatik den entsprechenden Typ hat. Zu zeigen, dass eine Grammatik einen bestimmten Typ nicht hat, reicht also nicht aus, um zu zeigen, dass auch die erzeugte Sprache diesen Typ nicht hat.

Wortproblem. Wir haben bereits gesehen, dass sich das Wortproblem bei regulären Sprachen algorithmisch entscheiden lässt. D.h. zu einem gegebenen Wort w und einer Typ-3 Grammatik können wir algorithmisch entscheiden, ob w von der Grammatik erzeugt wird. Dies gilt sogar für Typ-1 und Typ-2 Grammatiken.

Wir wissen, dass keine Produktion einer Typ-1 Grammatik die Länge einer erzeugten Satzformen verkürzt. Wollen wir am Ende das Wort $|w|$ erzeugen, wissen wir dass keine Satzform, die als Zwischenschritt auftaucht, länger als $|w|$ sein kann. Wenn wir nach möglichen Ableitungen suchen, können wir die Suche abbrechen, wenn die erzeugte Satzform länger als $|w|$ geworden ist. Da es zu jeder Länge nur eine endliche Anzahl an Satzformen gibt, gibt es also nur endlich viele Wörter, die als Zwischenschritt auftreten können.

Wir betrachten den Graphen, in dem es zu jeder Satzform, die nicht länger als w ist, genau einen Knoten gibt. In diesem Graphen gibt es gerichtete Kanten, die einzelne Ableitungsschritte darstellen. Dieser Graph ist endlich und lässt sich

algorithmisch in endlicher Zeit konstruieren. Mit Breitensuche oder Tiefensuche finden wir heraus, ob es einen Weg vom Startsymbol zu w gibt.

ε-Elimination. Um das leere Wort erzeugen zu können, haben wir die Zusatzregel $S \to \varepsilon$ erlaubt, wenn das Startsymbol auf keiner rechten Seite einer Produktion steht. Bei manchen Grammatiken können wir aber auch entspannter sein. Konkreter können wir bei Typ 2 Grammatiken auch beliebige Regeln der Form $A \to \varepsilon$ hinzufügen. Die entstehende Grammatik wird dann immer noch eine Sprache erzeugen, die auch nach unseren strengeren Regeln vom Typ-2 ist. Das gleiche gilt auch für Typ-3 Grammatiken.

Wir wollen also zu einer Typ-2 Grammatik in der Nichtterminale auf das leere Wort abgeleitet werden können, eine äquivalente Typ-2 Grammatik konstruieren, in der dies nicht mehr der Fall ist. Betrachte die Grammatik mit:

$$S \to AB, A \to a, A \to CD, B \to b, C \to c, C \to \varepsilon, D \to d, D \to \varepsilon$$

Zunächst wollen wir herausfinden, welche der Nichtterminale überhaupt auf das leere Wort abgeleitet werden können. Offensichtlich ist dies bei C und D der Fall. Wir sehen aber auch, dass wir ausgehend vom A folgende Ableitung haben. $A \to CD \to D \to \varepsilon$. Also kann ebenfalls das A auf ein ε abgeleitet werden. Bei S und B ist dies nicht der Fall.

Allgemein bestimmen wir zunächst alle Nichtterminalen, von denen ausgehend wir eine Ableitung haben, die zum leeren Wort führt. (Sogenannte nullierbare Variablen) Dies beginnt mit denjenigen Nichtterminalen, die direkt auf ε abgeleitet werden können. Daraufhin werden rekursiv all jene Nichtterminale aufgenommen, die auf eine Satzform abgeleitet werden können, die nur aus Nullierbaren Variablen besteht. In unserem Beispiel erkennen wir also zunächst C und D als nullierbare Variablen und sehen dann, dass die Satzform CD, auf die A abgeleitet werden kann, nur aus nullierbaren Variablen besteht, weshalb auch A nullierbar ist.

Die Idee ist nun, dass wir uns bereits beim Erzeugen eines Nichtterminals entscheiden, ob wir es später auf ε ableiten oder nicht. Wenden wir also die Produktion $A \to CD$ an und wissen, dass wir C später auf ε ableiten wollen, können wir stattdessen auch gleich nur $A \to D$ ableiten. Dies führt dann zum selben Ergebnis ohne dass eine Produktion $C \to \varepsilon$ verwendet werden müsste. Dafür muss es aber $A \to D$ als erlaubte Regel in unserer Grammatik geben. Zu jeder Produktion, in der auf der rechten Seite eine nullierbare Variable erzeugt wird, müssen wir also eine Variante einfügen, in der diese nullierbare Variable nicht vorkommt. (Um so zu simulieren, dass diese auf ε abgeleitet wurde.) Dies erweitert unsere Beispielgrammatik auf folgende Weise:

$$S \to AB, S \to B, A \to a, A \to CD, A \to C, A \to D, B \to b, C \to c, D \to d$$

3.2. **Eigenschaften kontextfreier Grammatiken.** Da wir uns nun gut mit Sprachen von Typ-3 auskennen, gehen wir nun einen Typ höher und widmen uns Sprachen vom Typ-2. Wir werden Fragen, die wir zu regulären Sprachen bereits geklärt haben nun auch für Typ-2 Sprachen klären. Wir werden Automaten definieren, die Typ-2 Sprachen akzeptieren, werden prüfen, unter welchen Konstruktionen die Klasse der Typ-2 Sprachen abgeschlossen ist, werden eine abgewandelte Form des Pumping Lemmas für Typ-2 Sprachen definieren und werden uns erzeugende Grammatiken von Typ-2 Sprachen anschauen. Wir starten mit dem letzten Punkt dieser Liste.

Kontextfreie Grammatiken und Sprachen. Eine Typ-2 Grammatik hat auf der linken Seite jeder Produktion nur ein einziges Nichtterminal. Taucht also ein Nichtterminal in einer Satzform auf, gibt uns die Grammatik genau an, durch welche Satzformen dieses Nichtterminal ersetzt werden kann. Dies ist unabhängig davon, von welchen Zeichen dieses Nichtterminal umgeben ist. Daher nennen wir Typ-2 Grammatiken auch **kontextfreie Grammatiken**. Wodurch ein Nichtterminal ersetzt wird, ist unabhängig von den Zeichen links und rechts - kann also völlig frei vom Kontext des Zeichens gewählt werden.

Eine Typ-2 Sprache wird daher auch als **kontextfreie Sprache** bezeichnet. Die Klasse aller kontextfreier Sprachen bezeichnen wir als CFL (context-free-languages) und die Klasse aller kontextfreier Sprachen über dem Alphabet Σ bezeichnen wir als $CFL(\Sigma)$.

Eine vereinfachte Schreibweise. Eine Liste aller Produktionen wird schnell unübersichtlich. Häufig gibt es viele Satzformen, durch die ein Nichtterminal ersetzt werden kann. Anstatt diese alle getrennt aufzulisten ($A \to AA, A \to B, A \to CD, A \to BD, A \to a$), fassen wir diese gerne auch in eine Produktion zusammen, wobei wir die möglichen Satzformen auf der rechten Seite durch ein | trennen. ($A \to AA|B|CD|BD|a$) Wir sagen, dass eine Grammatik, die diese verkürzte Schreibweise nutzt, in **Backus-Naur-Form** ist.

Beispiele einfacher kontextfreier Grammatiken. Wir haben gesehen, dass die Sprache $\{a^n b^n | n \in \mathbb{N}\}$ nicht regulär ist. Wir sehen aber direkt, dass die kontextfreie Grammatik mit den Produktionen $S \to \varepsilon|aSb$ diese Sprache erzeugt. Diese Sprache ist also kontextfrei, aber nicht regulär.

Es ist hervorzuheben, dass es nicht sofort klar war, dass solche Sprachen überhaupt existieren. Selbstverständlich existieren Grammatiken, die nicht regulär, aber kontextfrei sind. Wie jedoch bereits im letzten Abschnitt erwähnt wurde, kann es auch nicht reguläre Grammatiken geben, die reguläre Sprachen erzeugen. Bisher war noch die Frage ungeklärt, ob es überhaupt eine kontextfreie Grammatik gibt, deren erzeugte Sprache nicht regulär ist. Dies ist jetzt geklärt. Auch für die anderen Sprachtypen werden wir uns diese Frage stellen müssen. Existiert eine Typ-1 Sprache, die keine Typ-2 Sprache ist? Existiert eine Typ-0 Sprache, die keine Typ-1 Sprache ist? Die Antwort auf diese Fragen wird „ja“ sein, aber ohne diese Beispiele zu sehen, ist dies nicht sofort offensichtlich.

Als weiteres Beispiel kontextfreier Sprachen, betrachten wir Dycksprachen. Wollen wir Ausdrücke mit offenen und geschlossenen Klammern betrachten, so ist wichtig, dass, wenn wir das Wort von links nach rechts durchgehen, wir zu keinem Zeitpunkt mehr geschlossene als offene Klammern gesehen haben und, dass es insgesamt genausoviele offene wie geschlossene Klammern gibt. Die Sprache all dieser Wörter wird durch die Sprache mit den Produktionen $S \to \varepsilon|SS|(S)$ erzeugt. Wollen wir einen geklammerten Ausdruck wie z.B. ((())())(()) erzeugen, so gehen wir wie folgt vor: Wir finden diejenige Klammer, die die linkeste Klammer des Ausdrucks schließt (Das Klammerpaar ist hier groß hervorgehoben):

$$\big(((())())\big)(())$$

Wir prüfen, ob dieses Klammerpaar den gesamten Ausdruck umschließt. In diesem Fall ist dies nicht der Fall. Wir wenden also zunächst die Produktion $S \to SS$ an und werden in den folgenden Produktionen das linke S auf den Teil des Wortes produzieren, der von diesem Klammerpaar begrenzt ist:

$$\big(((())())\big)$$

Dieser Teil ist nun von diesem Klammerpaar umschlossen. Wir wenden also die Produktion $S \to (S)$ an und müssen nun nur noch den umschlossenen Teil erzeugen. Dies geschieht rekursiv. Auf diese Weise können wir jeden geklammerten Ausdruck erzeugen:

$$S \to SS \to (S)S \to (SS)S \to ((S)S)S \to (((S))S)S \to ((())S)S \to ((()))S)S$$
$$\to ((())())S \to ((())())(S) \to ((())())((S)) \to ((())())(())$$

Haben wir verschiedene Klammertypen, die jeweils den Regeln der korrekten Klammerung unterworfen sind, können wir je eine Produktion für jeden Klammertyp hinzufügen. Bei zwei Klammertypen haben wir zum Beispiel: $S \to \epsilon|SS|(S)|[S]$, was mit einer analogen Konstruktion alle korrekt geklammerten Ausdrücke mit zwei Klammertypen erzeugt. Solche Sprachen bezeichnen wir auch als **Dycksprachen**.

Ableitungsbäume. Wir können den Ableitungsprozess graphisch über einen Baum darstellen. Wir konstruieren diesen Baum, indem wir zunächst das Startsymbol als Wurzel des Baumes notieren. Im ersten Ableitungsschritt werden wir das Startsymbol durch eine Satzform ersetzen. Wir fügen dem Baum Kanten hinzu, die von der Wurzel ausgehen. Für jedes Zeichen in der neu erzeugten Satzform fügen wir dem Baum einen neuen Knoten und eine Kante von der Wurzel bis zu diesem Knoten hinzu. Diese neuen Knoten sind in der Reihenfolge sortiert, in der sie auch in der Satzform vorkommen. Wir fahren so mit jedem Ableitungsschritt fort. Ersetzen wir ein Nonterminal durch eine Satzform fügen wir je einen Knoten für jedes Zeichen der neuen Satzform ein und ergänzen eine Kante vom ersetzten Nonterminal zu dem neuen Knoten. Wir sortieren die neuen Knoten nach der Reihenfolge der Zeichen in der Satzform. Ersetzen wir in einem Ableitungsschritt ein Nonterminal durch das leere Wort, fügen wir dem entsprechenden Knoten genau einen Nachfolgerknoten hinzu, der mit ε beschriftet ist. Führen wir dies für jeden Ableitungsschritt einer Ableitung durch, erhalten wir einen Baum, der uns Informationen über die Struktur der Ableitung gibt. Diesen Baum bezeichnen wir als **Ableitungsbaum** der Ableitung.

Betrachten wir zum Beispiel die Grammatik mit

$$S \to A|B, A \to a, AA, B \to b, BB$$

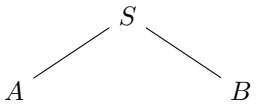
können wir das Wort *aabbb* zum Beispiel durch die Ableitung

$$S \to AB \to AAB \to aAB \to aaB \to aaBB \to aabB \to aabBB \to aabbB \to aabbb$$

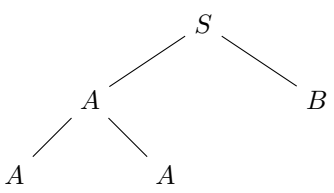
erhalten. Wir fangen mit der Wurzel an:

$$S$$

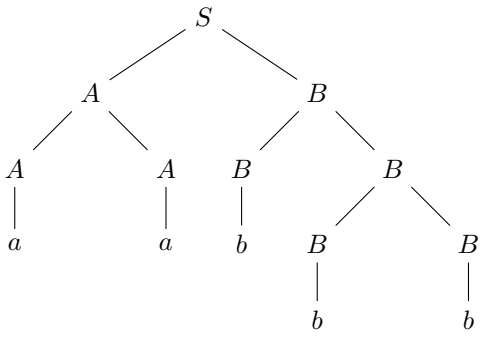
Im ersten Ableitungsschritt ersetzen wir S durch AB:



Wir ersetzen das A durch AA:



So fahren wir mit den restlichen Ableitungsschritten fort und erhalten folgenden Ableitungsbaum:

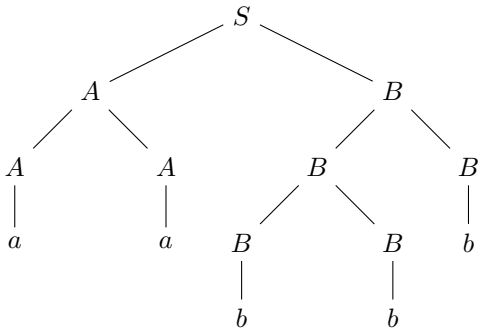


Lesen wir die Blätter des Baumes von links nach rechts, erhalten wir genau das erzeugte Wort. Dies gilt für jeden Ableitungsbaum.

Mehrdeutigkeit. Im obigen Beispiel würde auch folgende Ableitung zu dem Wort *aabbb* führen:

$$S \to AB \to AAB \to aAB \to aaB \to aaBB \to aaBb \to aaBBb \to aabBb \to aabbb$$

Mit dieser Ableitung erhalten wir den folgenden Ableitungsbaum:



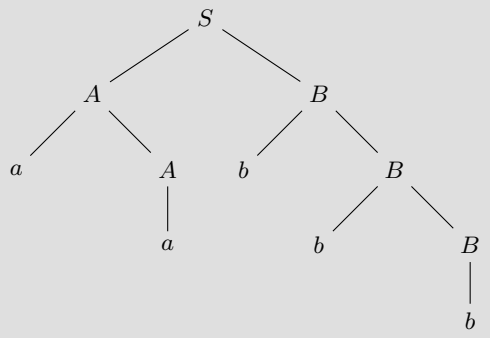
Mit dieser Grammatik können also zwei verschiedene Ableitungsbäume für das gleiche Wort konstruieren. Wir nennen Grammatiken, die so etwas ermöglichen, **mehrdeutig**. Grammatiken, in denen jedes Wort nur einen einzigen Ableitungsbaum hat, nennen wir **eindeutig**.

Eindeutigkeit trotz uneindeutiger Ableitung. Wir können die gleiche Sprache auch mit der Grammatik mit den Produktionen $S \to AB, A \to a|aA, B \to b|bB$ erzeugen. Auch hier hat das Wort *aabbb* verschiedene mögliche Ableitungen. Zum Beispiel gibt es:

$$S \to AB \to aAB \to aaB \to aabB \to aabbB \to aabbb$$

$$S \to AB \to AbB \to AbbB \to Abbb \to aAbbb \to aabbb$$

Konstruieren wir jeweils den Ableitungsbaum, erhalten wir jedoch in beiden Fällen den Baum:



Wir bemerken, dass wir für unsere Definition von Eindeutigkeit nur gefordert haben, dass es zu einem Wort nur einen einzigen *Ableitungsbaum* gibt. Es wurde nicht gefordert, dass es auch nur eine einzige *Ableitung* gibt. Die Exitenz dieser zwei verschiedenen Ableitungen führt also noch nicht dazu, dass wir diese Grammatik als mehrdeutig bezeichnen. Tatsächlich gibt es für diese Grammatik auch überhaupt kein Wort mit mehr als einem Ableitungsbaum. Diese Grammatik ist also eindeutig, obwohl es die Ableitungen nicht sind.

Inhärent mehrdeutig. Die Grammatiken, die wir in den vorherigen Beispielen betrachtet haben, beschreiben die Sprache $L \coloneqq \{a^m b^n | m, n \in \mathbb{N}\}$. Wir haben zunächst eine mehrdeutige Grammatik gesehen, die diese Sprache erzeugt. Im grauen Kasten haben wir jedoch eine Grammatik für diese Sprache gefunden, die eindeutig ist. Dies bringt uns zu einer wichtigen Erkenntnis zum Begriff der Mehrdeutigkeit: Mehrdeutigkeit ist eine Eigenschaft der Grammatik und keine Eigenschaft der erzeugten Sprache. Damit ist gemeint: Die Tatsache, dass wir L mit einer mehrdeutigen Grammatik erzeugt haben, bedeutet nicht, dass wir sagen können, dass L mehrdeutig ist. Wir haben L später ja auch mit einer eindeutigen Grammatik erzeugt und das würde bedeuten, dass L eindeutig ist. Dies widerspricht sich. Wir können also nur sagen, dass die Grammatik mit den Produktionen $S \to A|B, A \to a, AA, B \to b, BB$ mehrdeutig ist und dass die Grammatik mit den Produktionen $S \to AB, A \to a|aA, B \to b|bB$ eindeutig ist, aber diese Begriffe lassen sich nicht so direkt auf Mehrdeutigkeit der Sprache übertragen.

Wenn wir versuchen, einen Begriff zur Mehrdeutigkeit einer Sprache zu definieren, müssen wir also die Gesamtheit aller Grammatiken, die die Sprache erzeugen, in Betracht ziehen. Da wir jede Grammatik noch beliebig komplizierter machen können, gibt es zu jeder Sprache (außer $\emptyset$) eine mehrdeutige Grammatik. Die Frage, ob es eine mehrdeutige Grammatik gibt, ist also nicht besonders aussagekräftig. Interessanter ist die Frage, ob es zu einer gegebenen Sprache eine eindeutige Grammatik gibt. Dies führt uns zu einem Begriff der Mehrdeutigkeit, der auf Sprachen angewendet werden kann. Gibt es zu einer Sprache keine eindeutige Grammatik, die diese Sprache erzeugt, bezeichnen wir die Sprache als **inhärent mehrdeutig**.

Die Sprache $\{a^i b^j c^k | i = j \text{ oder } j = k\}$ ist zum Beispiel inhärent mehrdeutig. Der Beweis dieser Tatsache geht aber auf jeden Fall weit über die Anforderungen dieser Vorlesung hinaus. Im Allgemeinen ist es recht leicht, zu beweisen, dass eine Grammatik mehrdeutig ist. Wir müssen ein Wort mit zwei Ableitungen angeben, die zu verschiedenen Ableitungsbäumen führen. Es ist komplizierter, zu beweisen, dass eine Grammatik eindeutig ist. Hier müssen wir mehr von der Struktur der Grammatik verstehen. Bei vielen Grammatiken ist dies aber auch gut möglich. Beweise zu inhärenter Mehrdeutigkeit sind dagegen komplizierter. Zu beweisen, dass eine Grammatik nicht inhärent mehrdeutig ist, erfordert nur eine Angabe einer eindeutigen Grammatik und den Beweis der Eindeutigkeit. Zum Beweis inhärenter

Mehrdeutigkeit müssen wir uns aber überlegen, warum jede noch so komisch angelegte Grammatik zur Erzeugung der Sprache mehrdeutig sein muss. Dies ist sehr kompliziert und wir werden in dieser Vorlesung keine Methoden für solche Beweise kennenlernen.

Intuitiv können wir uns aber Folgendes vorstellen: Die Grammatik muss sowohl Wörter der Form $a^m b^n c^n$ als auch Wörter der Form $a^m b^m c^n$ erzeugen können. Für die erste Art von Wörtern brauchen wir ein Nonterminal A, das alle Wörter der Form A^m erzeugt und ein Nonterminal B_C, das alle Wörter der Form $b^n c^n$ erzeugt. Für die zweite Art Wörter brauchen wir ein Nonterminal B_C, das alle Wörter der Form $a^m b^m$ erzeugt und ein Nonterminal C, das alle Wörter der Form c^n erzeugt. Vom Startsymbol S ausgehend müssen wir sowohl zu AB_C als auch zu $A_B C$ kommen können. Dann können wir Wörter der Form $a^n b^n c^n$ aber sowohl über den Zwischenschritt AB_C als auch über den Zwischenschritt $A_B C$ erzeugen und erhalten so zwei verschiedene Ableitungsbäume für diese Wörter.

Es ist nicht absolut garantiert, dass eine erzeugende Grammatik so aussieht. Zum Beispiel könnten wir auch ein Nonterminal haben, das alle Wörter der Form $a^m b$ erzeugt und ein Nonterminal, das alle Wörter der Form $b^{n-1} c^n$ erzeugt. Die Tatsache, dass sich doch noch überraschend viele Alternativen finden lassen, wie diese Sprache auch erzeugt werden kann, macht den Beweis, dass tatsächlich trotzdem jede dieser Grammatiken mehrdeutig ist, so trickreich. Die grundlegende Idee bleibt aber, dass eine Grammatik zu dieser Sprache die Wörter $a^n b^n c^n$ sowohl über die Blöcke $a^n b^n, c^n$ als auch über die Blöcke $a^n, b^n c^n$ erzeugen können muss.

3.3. **Chomsky-Normalform und Pumping-Lemma.** Für reguläre Sprachen kennen wir mit dem Pumping-Lemma bereits ein Kriterium, mit dem wir bei vielen Sprachen nachweisen können, dass sie nicht regulär sind. Nun da wir uns viel mit kontextfreien Sprachen beschäftigt haben, liegt die Frage nahe, ob sich für diese auch ein ähnliches Kriterium finden lässt.

Tatsächlich wird dies möglich sein. Diese Aussage wird das Ziel dieses Kapitels sein. Um zu verstehen, warum dieses Kriterium funktioniert, müssen wir erst einmal jedoch noch einige Erkenntnisse über die Struktur kontextfreier Sprachen und Grammatiken gewinnen. Insbesondere müssen wir erkennen, dass kontextfreie Grammatiken in eine sogenannte Chomsky-Normalform umgeschrieben werden können. Diese Form wird für die Herleitung des kontextfreien Pumping-Lemmas wichtig sein, ist aber darüber hinaus auch häufig nützlich, um einfach mit kontextfreien Sprachen umgehen zu können.

Chomsky-Normalform. Wir leiten die Chomsky-Normalform nur für ε**-freie** Grammatiken her, also solche Grammatiken, die kein ε erzeugen können. Eine solche Grammatik liegt in **Chomsky-Normalform** vor, wenn in jedem Ersetzungsschritt ein Nonterminal entweder durch genau zwei Nonterminale oder genau ein Terminal ersetzt wird. Es muss also jede Ersetzungsregel die Form

$$A \to BC \text{ oder } A \to a$$

haben. Chomskynormalformen sind daher besonders nützlich, weil sich bei ihnen die Länge der betrachteten Satzformen sehr überschaubar bleibt. Wir wissen, dass die betrachtete Satzform in jedem Ableitungsschritt maximal um ein Zeichen länger wird. So lässt sich leichter verfolgen, was im Laufe einer Ableitung passiert.

Es stellt sich tatsächlich heraus, dass wir zu jeder ε-freien Grammatik auch eine äquivalente Grammatik in Chomsky-Normalform finden können. Dies funktioniert folgendermaßen:

Wir betrachten die Beispielgrammatik mit den Regeln

$$S \to ABC|AB|C, A \to Aa|\varepsilon, B \to Bb|b, C \to Cc|c$$

Diese Grammatik ist ε-frei. Es existiert zwar die Produktion $A \to \varepsilon$, aber die Grammatik kann das Wort ε nicht erzeugen. Dies liegt daran, dass wir das Nonterminal A nur erzeugen können, wenn wir auch noch ein B erzeugen, welches nicht auf das leere Wort abgeleitet werden kann. Vom Startsymbol S aus können wir also nicht das leere Wort erzeugen.

Wir haben nun mehrere Produktionen, die nicht der Form entsprechen, die wir haben wollen. Wir haben Produktionen auf das leere Wort wie $A \to \varepsilon$, wir haben Produktionen, bei denen Nonterminale und Terminale zusammen auf der rechten Seite stehen wie $B \to Bb$, wir haben Produktionen, bei denen die rechte Seite nicht genügend Nonterminale enthält wie $S \to C$ und wir haben Produktionen, bei denen die rechte Seite zu viele Nonterminale enthält wie $S \to ABC$. Diese Produktionen müssen wir nun schrittweise loswerden.

Wir kümmern uns zunächst um die Produktionen auf ε. In Abschnitt 3.1 haben wir bereits die ε-Elimination kennengelernt, mit der wir Produktionen dieser Art loswerden können. Diese können wir also sofort durchführen und erhalten die Grammatik mit den Regeln

$$S \to ABC|BC|AB|B|C, A \to Aa|a, B \to Bb|b, C \to Cc|c$$

Die andere Art von Regel, bei der auf der rechten Seite zu wenig Nonterminale stehen, sind die, bei denen ein einziges Nonterminal auf ein einziges anderes Nonterminal abgeleitet wird. Wir können uns um diese Regeln auf ähnliche Weise kümmern, wie wir es bei der ε-Elimination getan haben: Wir können die Regel entfernen und „vorwegnehmen“, was im nächsten Schritt getan wird. Wir haben zum Beispiel die Produktion $S \to B$. Wir können B dann später auf Bb oder b produzieren. Wir können die unerwünschte Regel $S \to B$ loswerden, indem wir es stattdessen erlauben, S direkt auf Bb oder b abzuleiten. Die Grammatkik, die wir erhalten, wenn wir dies auch mit $S \to C$ tun, ist:

$$S \to ABC|BC|AB|Bb|b|Cc|c, A \to Aa|a, B \to Bb|b, C \to Cc|c$$

Als nächstes kümmern wir uns darum, dass es Regeln gibt, bei denen auf der rechten Seite sowohl Terminale als auch Nonterminale stehen. Hierfür können wir für jedes Terminal a in einer solchen Regel ein Nonterminal V_a einfügen, zu dem es nur die Produktion $V_a \to a$ gibt. Diese neuen Regeln sind bereits von einer Form, die für Chomsky-Normalformen erlaubt sind. Wir werden an den neu hinzugefügten Regeln also auch nichts mehr ändern müssen. Dann können wir Regeln wie $A \to Aa$ durch $A \to AV_a$ ersetzen. Da V_a sowieso nur auf a abgeleitet werden kann, ändern wir hier nichts am erzeugten Wort. Wir haben nur eine unerwünschte Ableitungsschritte durch zwei erwünschte Schritte ersetzt.

In diesem Fall haben wir so bereits jede Regel, bei der sich Teminale und Nonterminale auf der rechten Seite mischen in die gewünschte Form gebracht. Hätten wir eine Regel wie $S \to Abc$ gehabt, hätten wir sie durch $S \to AV_bV_c$ ersetzt und hätten immer noch eine Regel mit drei Nonterminalen auf der rechten Seite. Dies wäre immer noch eine Regel, die wir eigentlich nicht wollen. Das ist aber kein Problem, da wir uns in einem späteren Schritt sowieso noch um Regeln mit mehr als zwei Nonterminalen auf der rechten Seite kümmern werden. In unserem Beispielfall erhalten wir zu diesem Zeitpunkt die folgenden Regeln:

$$S \to ABC|BC|AB|BV_b|b|CV_c|c, A \to AV_a|a, B \to BV_b|b, C \to CV_c|c,$$
$$V_a \to a, V_b \to b, V_c \to c$$

Die einzigen Regeln, die uns jetzt noch stören, sind diejenigen mit mehr als zwei Nonterminalen auf der rechten Seite. Wir können diese umgehen, indem wir eine Anwendung der Regel in mehrere Schritte aufteilen. In unserem Beispiel stört

uns die Regel $S \to ABC$. Wir können ein Nonterminal einführen, das wir A_B nennen und diesem die Ableitung $A_B \to AB$ erlauben. Dann ersetzen wir $S \to ABC$ durch $S \to A_BC$. So haben wir wieder eine unerwünschte Regel durch zwei erwünschte ersetzt. Dies funktioniert mit Regeln beliebiger Länge. Hätten wir die Regel $S \to ABCDE$, könnten wir diese durch die vier folgenden Schritte ersetzen: $S \to A_{BCD}E, A_{BCD} \to A_{BC}D, A_{BC} \to A_BC, A_B \to AB$. Bei jedem Schritt stehen nur zwei Terminale auf der rechten Seite. Doch sobald wir S durch $A_{BCD}E$ ersetzt haben, haben wir keine andere Wahl, als in den nächsten Ableitungsschritten $ABCDE$ zu erzeugen. Insgesamt haben wir also nichts an der erzeugten Sprache verändert. In unserem Beispiel erhalten wir die folgende Chomsky-Normalform:

$$S \to A_BC|BC|AB|BV_b|b|CV_c|c, A \to AV_a|a, B \to BV_b|b, C \to CV_c|c,$$
$$V_a \to a, V_b \to b, V_c \to c, A_B \to AB$$

Insgesamt können wir eine Chomsky-Normalform also mit den folgenden Schritten konstruieren:

(1) Wir löschen Regeln der Form $A \to \varepsilon$. Für alle Nonterminale A, die in einem oder mehreren Schritten auf ε abgeleitet werden können, fügen wir Produktionen, in denen A auf der rechten Seite vorkommt, außerdem eine Produktion hinzu, bei der A bereits fehlt.
(2) Wir löschen Regeln der Form $A \to B$, indem wir es der Grammatik bereits erlauben, A auf alle Satzformen abzuleiten, auf die B in einem Schritt ableitet werden kann.
(3) Wir vereinfachen Regeln, in denen Terminale und Nonterminale auf der rechten Seite gemischt vorkommen, indem wir für jedes Terminal a ein Nonterminal V_a einfügen, das auf a abgeleitet werden kann und in jeder „gemischten“ rechten Seite das Terminal a durch V_a ersetzen.
(4) Wir teilen Regeln $A \to B_1 \dots B_n$ mit mehr als zwei Nonterminalen auf der rechten Seite in mehrere Schritte auf. Dafür nutzen wir spezielle Nonterminale $A_2, A_3, \dots, A_{n-1}$ und fügen die Kette von Regeln aus $A \to A_{n-1}B_n, A_{n-1} \to A_{n-2}B_{n-1}, \dots, A_3 \to A_2B_2, A_2 \to B_1B_2$ hinzu.

Klausurrelevant ist wahrscheinlich nur die Fähigkeit, das Pumping-Lemma anwenden zu können und nicht, zu wissen, warum es funktioniert. Ich habe mich aber dagegen entschieden, die Herleitung in einen grauen Kasten zu setzen, da ich glaube, dass es hilft, zu verstehen, *warum* eine Aussage gilt, um sich merken zu können, *was* die Aussage ist. Wer kurz vor der Klausur noch einmal dieses Dokument durchgeht, um auf den letzten Drücker noch zu lernen, was noch nicht verstanden war, muss die Herleitung nicht bis ins kleinste Detail durchdrungen haben, sondern kann zum letzten Absatz der Herleitung springen, in dem die Aussage formal fertig da steht. Wer dieses Dokument aber während des Semesters begleitend zur Vorlesung liest, sei hiermit wirklich stark ermuntert, sich die Herleitung zumindest einmal anzuschauen.

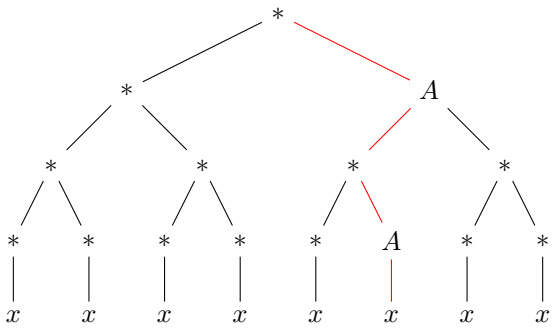
Herleitung des Pumping-Lemmas. Wir kennen bereits das Pumping-Lemma für reguläre Sprachen. Die Aussage war, dass wir in regulären Sprachen in allen Wörtern gewisser Länge ein Teilwort finden, das wir beliebig auf und abpumpen können ohne die Sprache zu verlassen. Wir wollen nun eine ähnliche Aussage für kontextfreie Sprachen finden. Hierfür stellt sich die Frage, in welchen Situationen wir in einer kontextfreien Sprache ein Wort auf- und abpumpen können.

Uns fällt Folgendes auf: Sei G eine Grammatik und A ein Nonterminal, mit der Bedingung, das es Ableitungsschritte gibt, mit denen A auf eine Satzform abgeleitet wird, die wieder A enthält. Es existiert also eine Satzform v und eine Satzform x sodass A auf vAx abgeleitet werden kann. Insbesondere interessiert uns der Fall,

in dem v und x nur aus Terminalen bestehen. Was bedeutet es nun, wenn A im Ableitungsbaum eines Wortes z vorkommt? Wenn A im Ableitungsbaum vorkommt, wird es irgendwann auf das Wort w abgeleitet. Wir wissen aber auch, dass es Ableitungsschritte gibt, die A auf vAx ableiten. Anstatt A direkt auf w abzuleiten, können wir also auch erst vAx herstellen und dann erst A auf w ableiten. Dann haben wir statt w gleich das Teilwort vwx erzeugt. Wir könnten A auch zwei mal auf vAx und dann erst auf w ableiten und erhalten v^2wx^2. Es scheint, als würden solche Nonterminale es uns erlauben, kontextfreie Wörter aufzupumpen.

Wir wollen also eine Aussage finden, die besagt, dass wir in jeder kontextfreien Sprache bei genügend langen Wörtern ein solches Nonterminal im Ableitungsbaum finden. Hierbei hilft es uns, wenn wir von unserer Grammatik wissen, dass sie sich in Chomsky-Normalform befindet. Da in einer Chomsky-Normalform in jedem Ableitungsschritt die Länge der Satzform nur um 1 erhöht wird, haben lange Wörter auch lange Ableitungen, weshalb wir mehr Chancen haben, dass ein Nonterminal sich selbst erzeugt.

Wir beschreiben konkreter, was wir suchen, indem wir einen Ableitungsbaum betrachten:



Wenn wir im Ableitungsbaum den Pfad eines Zeichens zur Wurzel zurückverfolgen und dabei ein Nonterminal A mehr als einmal antreffen, wissen wir, dass wir eine Situation haben, in der wir das Wort aufpumpen können. Wir wissen ebenfalls, dass es nur eine begrenzte Menge an Nonterminalen gibt. Da es nur $|V|$ viele verschiedene Nonterminale gibt, muss in jedem Pfad der Länge $|V|+1$ mindestens ein Nonterminal doppelt vorkommen. Hat der Ableitungsbaum die Höhe $|V|+1$, gibt es mindestens einen solchen Pfad und das Wort ist pumpbar. Nun ist es hilfreich, dass wir davon ausgehen können, dass die Grammatik in Chomsky-Normalform vorliegt. Da aus jedem Nonterminal stets entweder ein Terminal oder zwei Nonterminale erzeugt werden, wird sich die Anzahl der Knoten des Baumes von einer Stufe zur nächsten maximal verdoppeln. Ein Ableitungsbaum der Höhe $|V|$ kann also keine Wörter erzeugen, die länger als $2^{|V|-1}$ sind. (Der Baum im Beispielbild oben hat Höhe 4, da die Blätter selbst auch Teil des Baums sind. Im letzten Schritt wird die Anzahl der Knoten nicht mehr verdoppelt, da ein Nonterminal nur durch ein einziges Terminal ersetzt wird. Daher wird im Exponent eine 1 abgezogen.) Das bedeutet umgekehrt, dass ein Wort, das länger als $2^{|V|-1}$ ist, einen Ableitungsbaum hat, der eine Höhe von mindestens $|V|+1$ hat, weshalb das Wort pumpbar ist. Dies ist genau das Resultat, das wir haben wollten: Für jede kontextfreie Sprache gibt es eine vorgegebene Länge, sodass jedes Wort, das diese Länge überschreitet, pumpbar ist.

Wir müssen das Pumping-Lemma nun noch formell definieren. Wir haben bereits gesehen, dass wir für jede kontextfreie Sprache eine Wortlänge n finden können, sodass sich bei jedem Wort der Sprache mit mindestens n Zeichen ein sich selbst erzeugendes Nonterminal im Ableitungsbaum vorkommt. (Wir werden sehen, dass

es hilfreich ist, n sogar noch ein bisschen größer zu wählen.) Weiter oben haben wir bereits beschrieben, dass in einer solchen Situation das Teilwort, das vom oberen A erzeugt wird, als vwx beschrieben werden kann, wobei das w das Teilwort ist, das vom unteren A erzeugt wird. Wir bezeichnen den Teil des Wortes, der vor vwx kommt, als u und den Teil hinter vwx als y. Das gesamte Wort ist also $uvwxy$. Wie ebenfalls oben beschrieben, können wir die Ableitungsschritte zwischen den beiden A's beliebig oft wiederholen und erhalten so uv^iwx^iy mit $i \geq 2$ oder wir können die Ableitungsschritte zwischen den A's weglassen und erhalten uwy, also uv^0wx^0y. Im Pumping-Lemma für reguläre Sprachen war es sehr hilfreich, die Länge des pumpbaren Teilwortes mithilfe der Pumpkonstante n nach oben abschätzen zu können. Dies ist auch bei kontextfreien Sprachen möglich. Angenommen, die Länge von vwx ist deutlich größer als n. Dann ist die Höhe des erzeugenden Baumes auch deutlich größer als $|V|$. Insbesondere haben vermutlich auch die Teilbäume, die von den Kindern der Wurzel ausgehen, noch eine Höhe, die größer als $|V|$ ist und so lässt sich auch in diesen ein sich selbst erzeugendes Nonterminal finden. In diesem Fall können wir auch einfach dieses Nonterminal wählen und haben so ein kleineres pumpbares Teilwort gefunden. Diesen Vorgang können wir iterieren bis die Wurzel im Ableitungsbaum des Wortes das einzige selbsterzeugende Nonterminal ist. Dies ist nur bei Bäumen bis zu einer Höhe von $|V| + 1$ möglich. Die maximale Länge von vwx ist dann $2^{|V|}$. Wählen wir also $n \coloneqq 2^{|V|}$, können wir garantieren, dass uvw nicht länger als n ist. Außerdem wissen wir, dass mindestens eines der gepumpten Teilwörter v und x nicht leer ist. Ist A das Nonterminal, das sich im Ableitungsbaum selbst erzeugt, kann das erste A nicht *nur* das zweite A erzeugt haben, da unsere Grammatik in Chomsky-Normalform ist. Wir können also sagen, dass vx mindestens ein Zeichen lang ist.

Insgesamt finden wir also für jede kontextfreie Sprache L eine Pumpingkonstante n, sodass sich jedes Wort z, das mindestens n Zeichen lang ist, zerlegen lässt als $z = uvwxy$ mit $|vwx| \leq n$ und $|vx| \geq 1$, sodass für jedes $i \in \mathbb{N}$ auch $uv^iwx^iy \in L$ ist.

Man könnte sich fragen, warum wir das Pumping-Lemma für alle kontextfreie Sprachen formuliert haben. Immerhin existieren Chomsky-Normalformen nur für ε-freie Grammatiken und unsere Herleitung hat sich sehr stark darauf verlassen, dass es zu der betrachteten Sprache eine Grammatik in Chomsky-Normalform gibt. Warum gilt das Pumping-Lemma also nicht nur für alle ε-freie kontextfreie Sprachen?

Ähnlich wie in Kapitel 3.1 beschrieben, können wir Sprachen die ε enthalten, mit einer Grammatik erzeugen, in der das Startsymbol auf ε abgeleitet werden kann und das Startsymbol außerdem auf keiner rechten Seite einer Produktion vorkommt. Entfernen wir in dieser Grammatik die Produktion $S \to \varepsilon$, bringen dann die verbliebende Grammatik in Chomsky-Normalform und fügen dann die Produktion $S \to \varepsilon$ wieder hinzu, haben wir eine Grammatik für die Sprache, die fast in Chomsky-Normalform ist. Alle Wörter der Sprache außer dem leeren Wort haben Ableitungsbäume wie wir sie in der Herleitung des Pumping-Lemmas betrachtet haben. Da für das Pumping-Lemma ja sowieso erst Wörter ab einer gewissen Länge relevant sind, ist es egal, dass der Ableitungsbaum von ε noch eine andere Art Ableitungsschritt nutzt. Die Herleitung funktioniert also auch für Sprachen mit ε so wie oben beschrieben.

Beweise mit dem Pumping-Lemma. Wie beim Pumping-Lemma für reguläre Sprachen sagt auch das kontextfreie Pumping-Lemma nur, dass eine Sprache pumpbar ist, *wenn* sie kontextfrei ist. Es besagt nicht, dass eine Sprache kontextfrei ist, wenn sie pumpbar ist. Wir können also auch das kontextfreie Pumping-Lemma nicht nutzen, um nachzuweisen, *dass* eine Sprache kontextfrei ist. Aber wie auch schon beim regulären Pumping-Lemma können wir das kontextfreie Pumping-Lemma nutzen,

um zu zeigen, dass eine Sprache *nicht* kontextfrei ist. Ist eine Sprache nämlich nicht pumpbar, ist sie nicht kontextfrei.

Zu einer gegebenen Sprache L müssen wir also nachweisen, dass sich bei jeder Pumpkonstante n ein Wort z in L finden lässt, das mindestens die Länge n hat und sodass für jede Zerlegung $z = uvwxy$ von z mit $|vx| \geq 1$ und $|vwx| \leq n$ es mindestens ein $i \in \mathbb{N}$ gibt, sodass uv^iwx^iy nicht in L enthalten ist.

Wir können dies zum Beispiel mit der Sprache $L \coloneqq \{a^kb^kc^k | k \in \mathbb{N}\}$ tun. Zu gegebener Pumpingkonstante n betrachten wir das Wort $a^nb^nc^n$. Offenbar ist $|a^nb^nc^n| = 3n \geq n$. Angenommen, es existiert eine Zerlegung $a^nb^nc^n = uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$. Anders als beim regulären Pumping-Lemma muss der pumpbare Teil nun nicht am Anfang des Wortes stehen. Wir müssen als mehrere mögliche Fälle betrachten:

Fall 1: vwx enthält kein c. In diesem Fall enthält vx mindestens ein a oder mindestens ein b. Pumpen wir vx ab, haben wir die Anzahl der c's also nicht geändert, doch wir haben weniger als n a's oder weniger als n b's (oder von beiden weniger als n). In jedem Fall ist uv^0wx^0y nicht in L enthalten.

Fall 2: vwx enthält kein a. In diesem Fall enthält vx mindestens ein b oder mindestens ein c. Pumpen wir vx ab, haben wir die Anzahl der a's also nicht geändert, doch wir haben weniger als n b's oder weniger als n c's (oder von beiden weniger als n). In jedem Fall ist uv^0wx^0y nicht in L enthalten.

Fall 3: vwx enthält mindestens ein a und mindestens ein c. Da vwx ein Teilwort von $a^nb^nc^n$ ist, muss vwx auch alle n dazwischenliegenden b's enthalten. Dann ist vwx aber länger als n Zeichen, was nicht erlaubt ist. Fall 3 kann daher nicht eintreten.

Da das Wort in keinem der auftretenden Fälle pumpbar ist, kann die Sprache nicht regulär sein.

Im Vergleich zu Beweisen mit dem regulären Pumping-Lemma tauchen in Beweisen mit dem kontextfreien Pumping-Lemma mehr Fallunterscheidungen auf, da nicht von Anfang an bekannt ist, an welcher Stelle des Wortes der pumpbare Teil liegen muss. Der grundsätzliche Ablauf von Beweisen mit den beiden Lemmata ist aber sehr ähnlich. Für Dinge, die bei Beweisen mit dem kontextfreien Pumpinglemma zu beachten sind, sei daher auch einfach auf die Liste der formalen Stolperfallen hingewiesen, die am Ende von Kapitel 2.2 zu finden ist.

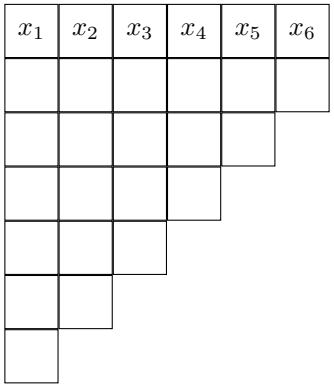
3.4. **Der CYK-Algorithmus.** Wir wollen uns in diesem Kapitel mit der Entscheidung des Wortproblems beschäftigen. Das Wortproblem hatte die folgende Problemstellung: Gegeben ist eine Sprache L und ein Wort x. Ist $x \in L$? Für reguläre Sprachen war das Wortproblem leicht zu beantworten, da wir L über einen DFA gegeben hatten und den Durchlauf auf der Eingabe x einfach simulieren konnten. Wie können wir eine Entscheidung zum Wortproblem treffen, wenn uns L über eine kontextfreie Grammatik gegeben ist? Können wir das überhaupt algorithmisch lösen?

Zunächst fällt auf, dass es einen solchen Algorithmus geben muss. Wenn wir nicht vor Brute-Force-Algorithmen zurückschrecken, können wir folgenden Algorithmus in Betracht ziehen. Wir wissen, dass x eine gewisse Länge hat. Eine Ableitung wird mit dem Startsymbol S starten. Wir sammeln alle Satzformen, von denen wir wissen, dass wir sie erzeugen können, in der Menge W. In jedem Schritt wenden wir auf jedes Element aus W jeden möglichen Ableitungsschritt an und speichern die Ergebnisse wieder in W. Wir brechen ab, wenn wir das Wort x erzeugt haben oder, wenn alle in einem Schritt hinzugefügten Satzformen länger als x waren. Dies funktioniert, da wir wissen, dass kontextfreie Grammatiken keine Ableitungsschritte erlauben, bei denen die rechte Seite kürzer als die linke Seite ist. Von einer Satzform, die länger als x ist, kann also keine Ableitung zu dem kürzeren Wort x führen.

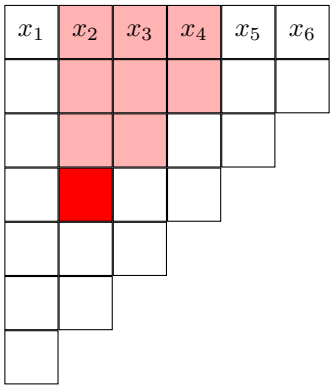
Sind also alle neu entdeckten Satzformen länger als x, werden auch alle zukünftig entdeckten Satzformen länger als x sein. Wenn wir x zu diesem Zeitpunkt noch nicht gefunden haben, werden wir es also auch in Zukunft nicht mehr tun.

Dieser Algorithmus zeigt, dass das Wortproblem für kontextfreie Grammatiken entschieden werden *kann*. Er ist jedoch kein Algorithmus, der sich in vertretbarer Zeit per Hand durchführen lässt. Wir werden in diesem Kapitel den CYK-Algorithmus kennenlernen, der das Wortproblem in einer Laufzeit löst, die sich tatsächlich gut manuell durchführen lässt.

Idee des CYK-Algorithmus. Für den CYK-Algorithmus nutzen wir eine Tabelle, bei der in der ersten Zeile je ein Tabelleneintrag für ein Zeichen aus x vorliegt. Darunter befinden sich dreieckig angeordnete, zu Beginn leere, Tabellenkästen.



Die leeren Tabelleneinträge stehen für bestimmte Teilwörter. Um zu erkennen, welchem Teilwort ein bestimmtes Kästchen entspricht, gehen wir von einem Kästchen einmal direkt nach oben und einmal nach oben rechts bis wir die zweite Tabellenspalte erreicht haben. (Also die erste Zeile, die nicht mit Terminalen gefüllt ist.) Das Kästchen entspricht dem Teilwort, das in dem so eingegrenzten Abschnitt zu finden ist. In der folgenden Tabelle entspricht das stark rot eingefärbte Kästchen dem schattierten Bereich, also insbesondere dem Teilwort $x_2x_3x_4$:



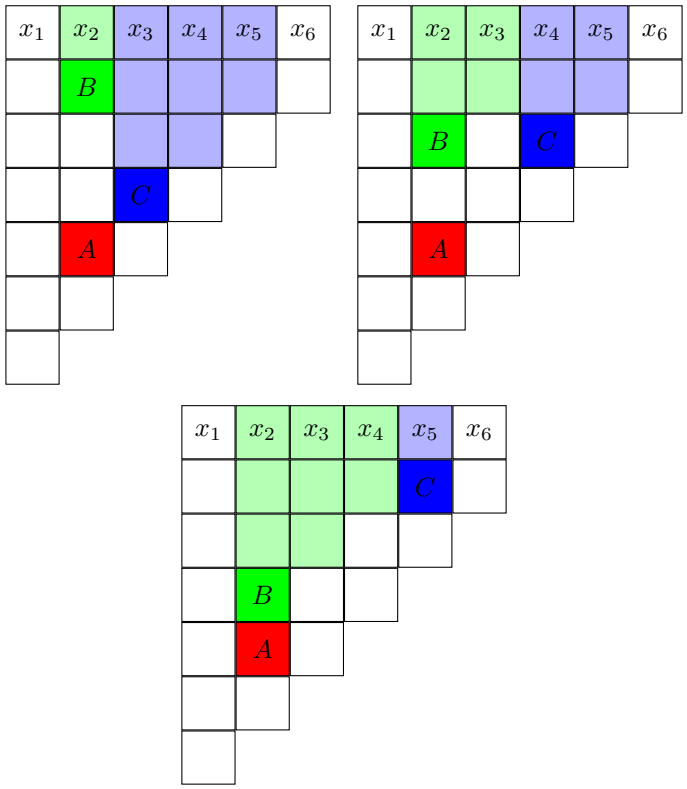
Das unterste Kästchen entspricht also insbesondere dem gesamten Wort.

Für den CYK-Algorithmus formen wir zunächst unsere Grammatik in Chomsky-Normalform um. Unser Ziel ist es nun, die Tabelle mit Nonterminalen der Grammatik zu füllen. Wir schreiben ein Nonterminal A in ein Kästchen, wenn es eine Ableitung gibt, mit der A in das Teilwort umgeformt wird, das dem Kästchen entspricht.

Am Ende enthält jedes Kästchen also eine Liste aller Nonterminale, aus denen das entsprechende Teilwort erzeugt werden kann. Ist ein Kästchen leer, gibt es kein solches Nonterminal. Das Ziel des Wortproblems ist es, herauszufinden, ob eine Grammatik ein bestimmtes Wort x erzeugen kann - wir wollen wissen, ob es eine Ableitung gibt, mit der das Startsymbol S auf x abgeleitet wird. Stellen wir also diese Tabelle her, können wir diese Frage direkt beantworten, da das unterste Kästchen ja dem gesamten Wort x entspricht. Enthält das unterste Kästchen das Startsymbol, so gibt es eine Ableitung und x kann erzeugt werden. Steht S am Ende nicht im untersten Kästchen, kann x von der Grammatik nicht erzeugt werden.

Durchführung des Algorithmus. Wie können wir die Tabelle also korrekt ausfüllen? Wir gehen rekursiv von oben nach unten vor. Die erste leere Zeile können wir direkt ausfüllen. Jedem Kästchen entspricht genau das Terminal im darüber liegenden Kästchen. In das Kästchen unter x_i tragen wir also genau die Nonterminale A ein, für die es eine Produktion $A \to x_i$ gibt.

Gehen wir also davon aus, dass wir die Tabelle bereits einige Zeilen weit korrekt gefüllt haben. Wie füllen wir nun die nächste Zeile? Wir wollen für jedes Kästchen beantworten, welche Nonterminale das entsprechende Teilwort erzeugen können. Da unsere Grammatik in Chomsky-Normalform vorliegt, muss ein Ableitungsschritt ein solches Nonterminal auf genau zwei verschiedene weitere Nonterminale ableiten. Die wichtigste Erkenntnis für den CYK-Algorithmus ist folgende: Steht ein Nonterminal A in einem Kästchen, finden wir die beiden Nonterminale, auf die A abgeleitet wird, in höheren Zeilen wieder. Steht A zum Beispiel in dem Kästchen für $x_2x_3x_4x_5$ und wird A in der entsprechenden Ableitung auf BC abgeleitet, muss auch BC auf $x_2x_3x_4x_5$ abgeleitet werden können. Hier gibt es folgende drei Möglichkeiten: B wird auf x_2 abgeleitet und C wird auf $x_3x_4x_5$ abgeleitet. B wird auf x_2x_3 abgeleitet und C wird auf x_4x_5 abgeleitet. B wird auf $x_2x_3x_4$ abgeleitet und C wird auf x_5 abgeleitet. In jedem der drei Fälle finden wir B und C weiter oben in der Tabelle wieder:



Für jedes Kästchen gehen wir also folgendermaßen vor: Wir prüfen jedes Paar an Kästchen, in denen die nachfolgenden Nonterminale stehen könnten. Für jedes dieser Paare prüfen wir, ob in beiden Kästchen ein Eintrag steht und, wenn ja, welche Satzformen aus zwei Nonterminalen wir so erhalten können. Zuletzt prüfen wir für jede so erhaltene Satzform BC, für welche einzelnen Nonterminale A es eine Produktion $A \to BC$ gibt. Wir tragen genau diese Nonterminale in das Kästchen ein. So können wir die ganze Tabelle bis zum untersten Kästchen füllen.

Die Anzahl an leeren Kästchen, die bei einem Wort der Länge n zu füllen sind, liegt in $\mathcal{O}(n^2)$. Wir können aber nicht davon ausgehen, dass alle Kästchen die gleiche Laufzeit benötigen, um gefüllt zu werden, da für ein Kästchen in der i-ten leeren Zeile auch $i-1$ Paare von Kästchen geprüft werden müssen. Daher liegt die Laufzeit des CYK-Algorithmus in $\mathcal{O}(n^3)$.

Beispiel. Beispielhaft gehen wir die Grammatik mit den folgenden Produktionen durch:

$$S \to AB_C, A \to AA|a, B_C \to B'C|BC, B' \to BB_C, B \to b, C \to c$$

Wir wollen entscheiden, ob die Grammatik das Wort *abbcc* erzeugen kann.
Wir legen die Tabelle an:

a	b	b	c	c

Das Nonterminal A kann auf a abgeleitet werden. Das Nonterminal B kann auf b abgeleitet werden. Das Nonterminal C kann auf c abgeleitet werden.

a	b	b	c	c
A	B	B	C	C

In der nächsten Zeile sehen wir, dass kein Nonterminal auf AB, BB oder CC abgeleitet werden kann. Kästchen 1,2 und 4 bleiben daher frei. Da B_C auf BC abgeleitet werden kann, wird das dritte Kästchen gefüllt.

a	b	b	c	c
A	B	B	C	C
		B_C		

In der nächsten Zeile kann das erste Kästchen nicht gefüllt werden, da es keine Kombination möglicher Folgekästchen gibt, in der kein leeres Kästchen vorkommt. Für das zweite Kästchen finden wir die Folgekombination BB_C. Tatsächlich existiert auch das Nonterminal B', das auf BB_C abgeleitet werden kann. Für das dritte Kästchen finden wir die Folgekombination B_CC, doch es existiert kein Nonterminal, das auf B_CC abgeleitet werden kann. Daher bleibt dieses Kästchen leer.

a	b	b	c	c
A	B	B	C	C
		B_C		
	B'			

In der nächsten Zeile finden wir für das erste Kästchen die Folgekombination AB', doch es existiert kein Nonterminal, das auf AB' abgeleitet werden kann. Das erste Kästchen bleibt leer. Für das zweite Kästchen finden wir die Folgekombination $B'C$ und das Nonterminal, B_C, das auf $B'C$ abgeleitet werden kann.

a	b	b	c	c
A	B	B	C	C
		B_C		
	B'			
	B_C			

Für das letzte Kästchen ist die einzige mögliche Folgekombination AB_C und tatsächlich ist S das einzige Nonterminal, das auf diese Satzform abgeleitet werden kann.

a	b	b	c	c
A	B	B	C	C
		B_C		
	B'			
	B_C			
S				

Da ein S im untersten Kästchen der CYK-Tabelle steht, können wir das Wort mit der Grammatik erzeugen.

Würden wir den Algorithmus auf dem Wort *bbcc* starten, bekämen wir folgende Tabelle:

b	b	c	c
B	B	C	C
	B_C		
B'			
B_C			

Auch hier ist das untere Kästchen nicht leer. Das Nonterminal, das im unteren Kästchen steht, ist jedoch nicht das Startsymbol. Die Schlussfolgerung ist daher, dass wir das Wort *bbcc* nicht mit der Grammatik erzeugen können.

Wortproblem mit Ableitungsbaum entscheiden. Beim CYK-Algorithmus tragen wir ein Nonterminal in ein Kästchen ein, wenn es das entsprechende Teilwort erzeugen kann. Dabei entscheiden wir aber nicht nur, *dass* das Nonterminal das entsprechende Teilwort erzeugen kann, sondern auch, über welche Folge-Nonterminale eine entsprechende Ableitung läuft.

Wir können beim Algorithmus also ohne zusätzliche Laufzeit beim Eintrag eines Nonterminals in die Tabelle auch speichern, welche Nonterminale aus welchen Kästchen in einer Ableitung folgen würden. Wenn ein Wort erzeugt werden kann, können wir mit der vollständigen Tabelle dann vom Startsymbol ausgehend herausfinden, mit welchen Ableitungsschritten das Wort erzeugt werden kann und erhalten so einen Ableitungsbaum. Der CYK-Algorithmus kann so also auch eingesetzt werden, wenn wir bereits wissen, dass ein gegebenes Wort erzeugt werden kann und wir daran interessiert sind, wie ein Ableitungsbaum des Wortes aussieht.

3.5. **Abschlusseigenschaften.** Für reguläre Sprachen haben wir bereits einige Operationen kennengelernt, unter denen die Klasse der regulären Sprachen abgeschlossen ist. Wir werden nun für kontextfreie Sprachen untersuchen, unter welchen dieser Operationen die Klasse der kontextfreien Sprachen abgeschlossen ist. Danach werden wir noch genauer die Struktur kontextfreier Sprachen untersuchen und ein besseres intuitives Verständnis dafür erlangen, welche Sprachen sich tatsächlich mit kontextfreien Grammatiken erzeugen lassen können.

Verschiedene Abschlusseigenschaften. Wir werden Stück für Stück sehen, dass Vereinigungen, Produkte, Kleensche Hüllen und Bilder unter Homomorphismen kontextfreier Sprachen selbst wieder kontextfrei sind. Ohne, dass wir hier auf die Begründung eingehen, sei auch noch erwähnt, dass ebenfalls Reflexionen und Urblider unter Homomorphismen von kontextfreien Sprachen wieder kontextfrei sind. Die Klasse der regulären Mengen war ebenfalls unter Bildung von Schnitt oder Komplement abgeschlossen. Wir werden jedoch sehen, dass dies bei den kontextfreien Sprachen nicht der Fall ist.

Zunächst widmen wir uns der Bildung der Vereinigung. Gegeben sind zwei kontextfreie Sprachen L_1 und L_2. Wir wollen zeigen, dass auch $L_1 \cup L_2$ eine kontextfreie Sprache ist. Hierfür betrachten wir die zugehörigen Grammatiken G_1, G_2 und konstruieren eine neue Grammatik G, die $L_1 \cup L_2$ erzeugt. Wir können erst einmal davon ausgehen, dass kein Nonterminal, das in G_1 auftaucht, auch in G_2 auftaucht. Sei S_1 das Startsymbol von G_1 und S_2 das Startsymbol von G_2. Wir führen ein neues Startsymbol S ein und erlauben die Produktionen $S \to S_1|S_2$. Im ersten Ableitungsschritt wird S also auf eines der Startsymbole abgeleitet. Nachdem wir S zum Beispiel auf S_1 abgeleitet haben, sind die weiteren Produktionen aus G_1 und somit wird ein Wort aus L_1 erzeugt. Umgekehrt können wir auch jedes Wort aus

L_1 erzeugen, indem wir $S \to S_1$ ableiten und dann die entsprechenden Ableitungsschritte aus G_1 durchgehen. Das Gleiche gilt für S_2 und L_2. Die neue Grammatik erzeugt also tatsächlich genau $L_1 \cup L_2$.

Für das Produkt können wir eine ähnliche Konstruktion durchführen. Statt den Produktionen $S \to S_1|S_2$ nutzen wir die Produktion $S \to S_1S_2$. Mit den nachfolgenden Ableitungsschritten wird das Teilwort, das aus S_1 erzeugt wird, aus L_1 stammen und das Teilwort, das aus L_2 erzeugt wird, wird aus L_2 stammen. Wir erzeugen also Wörter aus L_1L_2. Haben wir umgekehrt Wörter w_1, w_2 aus L_1 und L_2, können wir auf S_1 die Ableitungsschritte anwenden, die w_1 erzeugen, und auf L_2 die Ableitungsschritte anwenden, die w_2 erzeugen. So lässt sich auch jedes w_1w_2 aus L_1L_2 erzeugen. Die erzeugte Sprache ist also tatsächlich genau L_1L_2.

Wir wollen nun die Kleensche Hülle L_1^* erzeugen. Wir führen wieder ein externes Startsymbol S ein. Unser Ziel ist es, beliebig viele, aber auch gar keine Wörter aus L_1 hintereinander hängen zu können. Dafür müssen wir aus S die Satzformen $\varepsilon, S_1, S_1S_1, S_1S_1S_1, \ldots$ erzeugen können. Dies erreichen wir, indem wir die Produktionen $S \to \varepsilon|SS|S_1$ erlauben. Mithilfe der Produktion $S \to SS$ können wir das Startsymbol beliebig oft duplizieren bevor wir es auf S_1 ableiten. Die Produktion $S \to \varepsilon$ erlaubt es uns, auch das leere Wort zu erzeugen.

Das Hauptskript konstruiert die Kleensche Hülle mit den Zusatzproduktionen $S \to \varepsilon, S_1$ und $S_1 \to S_1S_1$. Dies ist tatsächlich auch eine korrekte Konstruktion, wenn das Startsymbol S_1 vorher auf keiner rechten Seite einer Produktion vorkam. Kommt S_1 auf der rechten Seite einer Produktion vor, kann dies jedoch zu Fehlern führen. Der hier angegebene Weg funktioniert für jede kontextfreie Grammatik.

Als nächstes konstruieren wir eine Grammatik für das Bild unter einem Homomorphismus. Sei $\varphi : \Sigma^* \to \Gamma^*$ ein Homomorphismus und G eine Grammatik über Σ. Wenn wir in jeder Produktion auf der rechten Seite ein Vorkommen eines Terminals a durch $\varphi(a)$ ersetzen, erzeugen die Produktionsschritte statt einem Wort $a_1a_2 \ldots a_n$ nun das Wort $\varphi(a_1)\varphi(a_2) \ldots \varphi(a_n)$, was wegen der Homomorphismuseigenschaften auch $\varphi(a_1a_2 \ldots a_n)$ ist. Die Grammatik erzeugt also genau die Bilder der Wörter, die G erzeugt hatte.

Wir überlegen nun, weshalb der Schnitt kontextfreier Sprachen nicht wieder kontextfrei ist. Hierfür betrachten wir die beiden Sprachen $\{a^mb^nc^n|n, m \in \mathbb{N}\}$ und $\{a^mb^mc^n|n, m \in \mathbb{N}\}$. Es gibt eine kontextfreie Grammatik, die $\{a^mb^nc^n|n, m \in \mathbb{N}\}$ erzeugt:

$$S \to AB_C, A \to \varepsilon|AA|a, B_C \to \varepsilon|bB_Cc$$

Es sollte recht leicht zu sehen sein, dass hiermit tatsächlich genau die Wörter erzeugt werden, bei denen erst a's dann b's und dann genauso viele c's, wie es b's gab, kommen. Die Sprache ist also kontextfrei. Mit einer sehr ähnlichen kontextfreien Grammatik können wir $\{a^mb^nc^n|n, m \in \mathbb{N}\}$ erzeugen. Beide Sprachen sind kontextfrei. Wir überlegen uns, wie der Schnitt $\{a^mb^nc^n|n, m \in \mathbb{N}\} \cap \{a^mb^mc^n|n, m \in \mathbb{N}\}$ aussieht. In einem Wort im Schnitt müssen offenbar erst a's, dann b's, dann c's kommen. Da es in $\{a^mb^nc^n|n, m \in \mathbb{N}\}$ liegt, muss es genauso viele b's wie c's geben. Da es in $\{a^mb^mc^n|n, m \in \mathbb{N}\}$ liegt, muss es genauso viele a's wie b's geben. Der Schnitt ist also die Sprache $\{a^nb^nc^n|n \in \mathbb{N}\}$, bei der die Anzahl aller drei Zeichen gleich ist. Von dieser Sprache haben wir bereits mit dem kontextfreien Pumping-Lemma gezeigt, dass sie nicht kontextfrei ist. Wir sehen hier also zwei kontextfreie Sprachen, deren Schnitt nicht kontextfrei ist. Die Klasse der kontextfreien Sprachen kann also nicht unter Bildung des Schnitts abgeschlossen sein.

Wir erinnern uns an Kapitel 2.5 zurück. Dort hatten wir gewusst, dass die Klasse der regulären Mengen unter Bildung von Vereinigung und Komplement abgeschlossen

ist. Daraus konnten wir direkt folgern, dass sie auch unter der Bildung von Schnitten abgeschlossen ist. Wüssten wir für die Klasse der kontextfreien Sprachen ebenfalls, dass sie unter Bildung von Vereinigung und Komplement abgeschlossen ist, könnten wir völlig analog folgern, dass sie unter Bildung von Schnitten abgeschlossen ist. Da wir aber bereits gesehen haben, dass diese Schlussfolgerung falsch wäre, kann die Klasse der kontextfreien Sprachen auch nicht sowohl unter Bildung von Vereinigung als auch unter Bildung von Komplementen abgeschlossen sein. Da wir bereits gesehen haben, dass die Klasse der kontextfreien Sprachen unter Bildung von Vereinigungen abgeschlossen ist, bleibt nur die Möglichkeit, dass sie nicht unter Bildung von Komplementen abgeschlossen ist.

Auch wenn der Schnitt zweier kontextfreier Sprachen nicht notwendigerweise wieder kontextfrei ist, sei hier ohne Beweis erwähnt, dass der Schnitt einer kontextfreien und einer regulären Sprache tatsächlich immer kontextfrei ist.

Kontextfreie Sprachen und Dycksprachen. Wir haben gesehen, dass die Sprache $\{a^n b^n | n \in \mathbb{N}\}$ nicht regulär ist. Intuitiv gesehen sind reguläre Sprachen nicht sehr gut darin, zwei Zeichen gleich oft vorkommen zu lassen. Wir haben aber gesehen, dass diese Sprache kontextfrei ist und darüber hinaus haben wir in Kapitel 3.2 gesehen, dass alle Dycksprachen kontextfrei sind. Intuitiv sind kontextfreie Sprachen sogar gut darin, beliebig viele Paare von Zeichen gleich oft vorkommen zu lassen. Dagegen haben wir aber auch gesehen, dass die Sprache $\{a^n b^n c^n | n \in \mathbb{N}\}$ nicht kontextfrei ist. Es wirkt, als würden kontextfreie Sprachen scheitern, sobald drei oder mehr Zeichen gleich oft vorkommen sollen. Bisher sind die Dycksprachen also das Komplizierteste, was wir mit kontextfreien Sprachen erreichen konnten. Es würde unserem intuitiven Verständnis sehr helfen, wenn wir tatsächlich wüssten, dass Dycksprachen „das Komplizierteste sind, was sich mit kontextfreien Sprachen erreichen lässt“.

Auf gewisse Weise ist dies tatsächlich der Fall. Damit wir diese Aussage wirklich nutzen können, müssen wir konkreter machen, was wir wirklich meinen. Offenbar ist ja nicht jede kontextfreie Sprache eine Dycksprache. Wie können wir also Sprachen wie $\{a^n b^n | n \in \mathbb{N}\}$ oder $\{a^{2n} b^{2n} | n \in \mathbb{N}\}$ mit dieser Anschauung erklären? Es fällt auf, dass diese Sprachen beide Teilmengen einer Dycksprache sind. Die Sprache $\{a^n b^n | n \in \mathbb{N}\}$ besteht aus genau denjenigen Elementen der Dycksprache D_1, bei der zunächst alle a's und dann alle b's kommen. Wir können also $\{a^n b^n | n \in \mathbb{N}\} = \{a^m b^n | m, n \in \mathbb{N}\} \cap D_1$ schreiben. Genauso ist auch $\{a^{2n} b^{2n} | n \in \mathbb{N}\}$ eine Teilmenge einer Dycksprache. Genauer ist $\{a^{2n} b^{2n} | n \in \mathbb{N}\} = \{a^{2m} b^{2n} | m, n \in \mathbb{N}\} \cap D_1$. Wir können aber nicht alle Teilmengen von Dycksprachen zulassen. Wir könnten die Sprache $\{a^{n^2} b^{n^2} | n \in \mathbb{N}\}$ darstellen als $\{a^{n^2} b^{n^2} | n \in \mathbb{N}\} = \{a^{m^2} b^{n^2} | m, n \in \mathbb{N}\} \cap D_1$. Das Problem ist, dass die Sprache $\{a^{n^2} b^{n^2} | n \in \mathbb{N}\}$ nicht mehr kontextfrei ist. Wenn wir zu komplizierte Teilmengen aus unserer Dycksprache nehmen, erhalten wir also keine kontextfreie Sprache. Warum können wir D_1 also mit $\{a^m b^n | m, n \in \mathbb{N}\}$ oder $\{a^{2m} b^{2n} | m, n \in \mathbb{N}\}$ schneiden, aber nicht mit $\{a^{m^2} b^{n^2} | m, n \in \mathbb{N}\}$? Auf welche Weise sind die ersten beiden Sprachen weniger kompliziert als die dritte? Die Antwort fällt recht schnell auf: Die ersten beiden Sprachen sind regulär - die dritte nicht. Das führt uns zu folgender Überlegung. Eine Teilmenge einer Dycksprache ist noch kontextfrei, solange sie ein Schnitt der Dycksprache mit einer regulären Menge ist.

Dies bringt uns unserem Ziel ein ganzes Stück näher. Wir wollten formaler sagen, dass kontextfreie Sprachen nicht komplizierter als eine Dycksprache sein können. Wir haben nun schon viele bekannte kontextfreie Sprachen als einfache Teilmenge einer Dycksprache beschrieben. Aber noch gibt es viele kontextfreie Sprachen, die sich nicht auf diese Weise darstellen lassen. Wir müssen also noch ein wenig erweitern, was wir unter „nicht komplizierter als eine Dycksprache“ verstehen. Es gibt auch kontextfreie Sprachen wie $\{(ab)^n (cd)^n | n \in \mathbb{N}\}$ oder auch $\{a^n | n \in \mathbb{N}\}$, die bisher noch nicht abgedeckt sind. Bei $\{(ab)^n (cd)^n | n \in \mathbb{N}\}$ fällt auf: Es gibt hier zwar nicht einfach

zwei Zeichen, die gleich oft vorkommen sollen, aber es gibt zwei *Wörter*, die gleich oft vorkommen sollen. Wir können solche Sprachen mithilfe einer Dycksprache leicht ausdrücken, indem wir jedes einzelne Zeichen der Sprache durch ein entsprechnedes Wort ersetzen. Dies ist genau das, was ein Homomorphismus tut. Es fällt auf, dass ein Homomorphismus auch $\{a^n | n \in \mathbb{N}\}$ erzeugt. Hierfür nehmen wir die Sprache $\{a^n b^n | n \in \mathbb{N}\}$ und bilden a auf a und b auf ε ab.

Es wirkt, als hätten wir so eine vernünftige formelle Ausdrucksweise von „nicht komplizierter als eine Dycksprache“ gefunden. Eine Sprache ist genau dann kontextfrei, wenn wir sie konstruieren können, indem wir eine Dycksprache D_k nehmen, sie mit einer regulären Menge R schneiden und das Bild vom Ergebnis unter einem Homomorphismus φ betrachten. Diese Aussage ist tatsächlich korrekt. Sie ist bekannt als der **Satz von Chomsky-Schützenberger**

Reguläre und kontextfreie unäre Sprachen. Besondere Erkenntnisse über kontextfreie Sprachen können wir auch gewinnen, wenn wir uns nur auf unäre Sprachen beschränken. Eine Sprache ist unär, wenn es nur ein einziges Zeichen gibt, das in der Sprache vorkommt - wenn es also ein Zeichen a gibt, sodass die Sprache eine Teilmenge von $\{a\}^*$ ist.

Wir überlegen uns, was es bedeutet, ein unäres Wort aufzupumpen. Ein Wort einer unären Sprache über $\{a\}$ ist nur eine Abfolge von a's - hat also die Form a^k für irgendein k. Im kontextfreien Pumpinglemma wählen wir zwei Teilworte v, x, die wir zusammen auf- oder abpumpen. Hierbei fügen wir bei jedem Aufpumpen genausoviele a's hinzu, wie in v und x zusammen enthalten sind. Man wird dem Endergebnis jedoch nicht anmerken können, an welcher Stelle die a's dupliziert wurden. Anstatt zwei Teilworte an beliebiger Stelle des Wortes auf- und abzupumpen, könnten wir auch ein einziges Teilwort am Anfang des Wortes nehmen, das so lang ist wie v und x zusammen. Das Ergebnis wäre das gleiche. Lässt sich eine unäre Sprache also pumpen, wie es im kontextfreien Pumping-Lemma passiert, lässt sie sich auch so pumpen, wie es im regulären Pumping-Lemma passiert. Kontextfreie unäre Sprachen erfüllen also auch die stärkere Pumpbedingung, die wir sonst nur von regulären Sprachen kennen.

Dies alleine reicht noch nicht aus, um zu folgern, dass jede kontextfreie unäre Sprache auch regulär ist. Das reguläre Pumping-Lemma sagt nur aus, dass eine Sprache pumpbar ist, wenn sie regulär ist und nicht, dass eine Sprache regulär ist, wenn sie pumpbar ist. Trotzdem scheint es sinnvoll, sich Gedanken darüber zu machen, ob kontextfreie unäre Sprachen auch regulär sind. Tatsächlich ist dies der Fall. Bei unären Sprachen brauchen wir nicht zwischen regulären und kontextfreien Sprachen unterscheiden, da jede kontextfreie unäre Sprache auch regulär ist.

Wir wollen auch beweisen, dass jede kontextfreie unäre Sprache L regulär ist. Wir wissen, dass L pumpbar ist - sogar wie es im regulären Pumping-Lemma gefordert wird. Es existiert also eine pumping-Konstante n. Wir wollen L nun darstellen als Vereinigung von endlich vielen Sprachen, von denen wir wissen, dass sie regulär sind. Da Vereinigungen regulärer Sprachen auch wieder regulär sind, ist dann auch L regulär. Zunächst sei K die Menge aller Wörter in L, die weniger als n Zeichen enthalten. Da die Sprache unär ist, gibt es maximal n Wörter, die weniger als n Zeichen enthalten. Insbesondere ist K endlich und somit regulär.

Wir betrachten nun die Wörter aus L, die nicht in K enthalten sind. Sei $a^k \in L \setminus K$. Nach dem Pumping-Lemma ist a^k pumpbar. Es gibt also ein Teilwort a^s, sodass $a^{k-s}, a^k, a^{k+s}, a^{k+2s}, \ldots$ in L enthalten sind. Wir sehen also, dass viele Wörter von L in einem „Streifen“ liegen, der jedes $s$te Wort enthält. Genauer gesagt handelt es sich um den Streifen aller a^n, bei denen $n \equiv k \mod s$ ist. Dieser Streifen ist auch tatsächlich regulär - wir nennen ihn $S_{k,s}$. Das Problem ist, dass nicht jedes Element aus $S_{k,s}$ auch wirklich

in L enthalten sein muss. Da wir oben gesehen haben, dass ab a^k jedes Element von $S_{k,s}$ auch in L liegen muss, kann es nur endlich viele Elemente in $S_{k,s}$ geben, die nicht in L liegen. Da wir bei der Bildung des Schnitts $S_{k,s} \cap L$ also nur endlich viele Elemente entfernen, ist auch $S_{k,s}$ regulär. Es kann aber nur endlich viele verschiedene Mengen unter den $S_{k,s} \cap L$ geben. Dies liegt daran, dass nach dem Pumping-Lemma gefordert ist, dass der gempumpte Teil maximal n Zeichen enthält. Es muss also $s \leq n$ sein. Somit gibt es nur endlich viele legitime Auswahlmöglichkeiten für s. Außerdem gibt es nur endlich viele Restklassen modulo s. Da $S_{k,s}$ nur von der Restklasse von k modulo s abhängt, gibt es auch, wenn wir die verschiedenen k's durchlaufen für jedes s nur endlich viele Ergebnisse. Wenn wir für jedes $a^k \in L$ mit $k \geq n$ also die entsprechenden Mengen $S_{k,s} \cap L$ vereinigen, haben wir alle Wörter in L, die mindestens n Zeichen enthalten, in einer Vereinigung von endlich vielen regulären Mengen eingefangen. Vereinigen wir noch mit K, haben wir ganz L als Vereinigung von endlich vielen regulären Mengen dargestellt und somit gezeigt, dass L regulär ist.

3.6. **Kellerautomaten (Pushdown-Automaten).** Für reguläre Sprachen haben wir recht früh die Beschreibung mithilfe von DFAs kennen gelernt. Erst später sind wir auf reguläre Grammatiken gestoßen. Für kontextfreie Grammatiken kennen wir bisher nur die kontextfreien Grammatiken und wir haben uns noch nicht mit einer Art akzeptierenden Automaten für kontextfreie Sprachen beschäftigt. In diesem Kapitel wollen wir das nachholen.

Wir werden Kellerautomaten definieren. Intuitiv handelt es sich hierbei um NFAs mit einem Speicher, auf den sie mit einigen Beschränkungen zugreifen können. Wir werden ebenfalls eine deterministische Variante der Kellerautomaten definieren. Für die nicht-deterministische Variante werden wir sehen, dass sie genau die kontextfreien Sprachen akzeptieren können.

Intuitive Beschreibung von Kellerautomaten. Um einen Kellerautomaten zu definieren, starten wir mit einem NFA. Wir haben also eine Zustandsmenge Q, ein Startzustand q_0, eine Endzustandsmenge F, ein Eingabealphabet Σ und eine Übergangsfunktion δ. Für die Übergänge können wir ein Zeichen einlesen, aber wir erlauben auch ε-Übergänge.

Die Neuerung ist, dass wir auch einen Speicher haben, in den wir Informationen speichern können. Wir können aber nur begrenzt auf diesen Speicher zugreifen. Wollen wir ein Zeichen aus dem Speicher - auch Keller genannt - lesen, müssen wir vorher alle Zeichen, die später gespeichert wurden, wieder gelöscht haben. Wir können uns den Keller also als einen Stapel vorstellen. Während ein Wort eingelesen wird, legen wir mehrere Zeichen auf diesen Stapel. Zu jedem Zeitpunkt können wir aber immer nur das oberste Zeichen des Stapels sehen. Unsere Übergänge sind nun nicht mehr nur abhängig vom aktuellen Zustand und dem eingelesenen Zeichen, sondern auch von dem obersten Zeichen des Stapels. Der Effekt eines Übergangs ist nun nicht mehr nur ein Wechsel des Zustands, sondern auch eine Änderung des Stapels. Unsere Übergangsfunktion hat also zusätzlich ein Zeichen aus dem Stapel als Input und liefert als zusätzlichen Output eine Anweisung, was im Keller zu tun ist. Wir können das oberste Zeichen des Stapels löschen, nichts tun oder das oberste Zeichen durch ein oder mehrere Zeichen ersetzen, die nun oben auf dem Stapel liegen.

Formale Definition von Kellerautomaten. DFAs und NFAs waren formal eindeutig definiert durch die Angabe von Zustandsmenge, Eingabealphabet, Übergangsfunktion, Startzustand und Endzustandsmenge. Wir wollen Kellerautomaten (oder auch Pushdownautomaten - im Folgenden als PDA abgekürzt) auf eine ähnliche Weise definieren. Die Neuerung liegt hierbei im Keller. Wir müssen also außerdem eine

Menge an Zeichen Γ angeben, die auf dem Speicher liegen können. Außerdem wollen wir, dass zu Beginn der Verarbeitung genau ein Zeichen auf dem Kellerstapel liegt. Dieses Zeichen muss auch bei der Definition eines PDAs angegeben werden. Ein PDA ist also eindeutig definiert durch die Angabe der folgenden sieben Informationen:

- Die Zustandsmenge Q
- Das Eingabealphabet Σ
- Das Kelleralphabet Γ
- Die Übergangsfunktion Δ
- Der Startzustand q_0
- Das Kellerstartsymbol z_0
- Die Endzustandsmenge F

Hierbei ist zu beachten, dass wir auch wollen, dass wir Zeichen aus Σ im Keller abspeichern können. Also muss jedes Zeichen aus Σ auch in Γ sein.

Wir müssen uns ebenfalls noch überlegen, wie genau die Übergansfunktion auszusehen hat. Wie bereits gesagt, hat die Übergangsfunktion drei Inputs. Der erste Input ist der aktuelle Zustand - also ein Element aus Q. Der zweite Input ist das nächste Zeichen des eingelesenen Wortes - also ein Zeichen aus Σ. Wir wollen aber auch ε-Übergänge erlauben. Der zweite Input muss daher in $\Sigma \cup \{\varepsilon\}$ liegen. Der dritte Input ist das oberste Zeichen des Kellerstapels - muss also in Γ liegen. Insgesamt ist der Definitionsbereich der Übergangsfunktion also $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$.

Im Wertebereich brauchen wir zwei Outputs. Der eine Output ist der Folgezustand - liegt also in Q. Der zweite Output sagt uns, durch welche Zeichen wir das oberste Zeichen des Stapels ersetzen müssen. Es handelt sich also um ein Wort über Γ, bzw ein Element aus Γ^*. Wir würden den Wertebereich also gerne als $Q \times \Gamma^*$ bezeichnen. Wir wollen aber auch gleich erlauben, dass unsere Automaten nichtdeterministisch sind. Dafür sollte der Output - analog zur Definition von NFAs - also kein Element, sondern eine Teilmenge von $Q \times \Gamma^*$ sein. Der Wertebereich ist also $2^{Q \times \Gamma^*}$ - die Menge der Teilmengen von $Q \times \Gamma^*$.

Wir hatten weiter oben gesagt, dass wir es erlauben wollen, das oberste Symbol im Keller zu löschen, nichts mit dem Keller zu tun oder das oberste Kellersymbol durch ein anderes Wort zu ersetzen. Wir scheinen jetzt nur das Ersetzen des obersten Symbols durch ein anderes Wort erlaubt zu haben. Die anderen Optionen sind mit dieser Definition aber auch möglich. Das oberste Kellersymbol zu löschen ist das Gleiche, wie das oberste Symbol durch das Wort ε zu ersetzen. Nichts mit dem Stapel zu tun ist das Gleiche, wie a durch a zu ersetzen. Man bedenke, dass wir die Konvention haben, dass das linkeste Zeichen des Kellers „oben“ ist. Wollen wir auf des oberste Zeichen A nun ein B legen, müssen wir A nicht durch AB, sondern BA ersetzen.

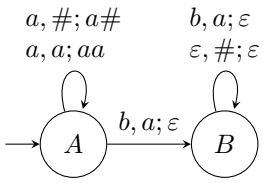
Wie ein PDA Worte akzeptiert. Wie bei einem NFA akzeptiert auch ein PDA ein Wort w genau dann, wenn es **einen** Pfad durch den PDA gibt, bei dem der PDA auf der Eingabe w in einem Endzustand endet. Um zu wissen, wie ein Pfad weitergehen kann, reicht es nicht nur, zu wissen, in welchem Zustand wir uns befinden und welche Zeichen noch eingelesen werden - wir müssen auch wissen, welche Zeichen gerade auf dem Stapel liegen. Diese drei Informationen können wir für die Übersicht in einem Tripel (q, v, w) speichern, wobei q der aktuelle Zustand, v der noch einzulesende Teil des Wortes und w das Wort auf dem Stapel ist. Dieses Tripel nennen wir auch die **Konfiguration** des Automaten. Wir können also sagen, dass ein Automat mit Startzustand q_0 und Startkellersymbol z_0 ein Wort w genau dann akzeptiert, wenn es einen Endzustand q_F gibt, sodass es einen Pfad im Automaten gibt, der von der Konfiguration (q_0, w, z_0) zu einer Konfiguration (q_F, ε, s) führt. Hier kann s ein beliebiges Wort sein.

Es gibt auch noch eine weitere Möglichkeit, wie ein PDA Wörter akzeptieren kann. Anstatt Endzustände für den PDA anzugeben, können wir auch deklarieren, dass der PDA ein Wort akzeptiert, wenn es einen Pfad gibt, bei dem alle Zeichen eingelesen werden und der Keller am Ende leer ist. Wenn es also einen Pfad von der Konfiguration (q_0, w, z_0) zu einer Konfiguration $(q, \varepsilon, \varepsilon)$ gibt.

Einen PDA, der mit Endzuständen akzeptiert, nennen wir auch **Endzustand-Akzeptierer**. Einen PDA, der mit einem leeren Keller akzeptiert, nennen wir auch **Leerkeller-Akzeptierer**.

Beispiele. Die uns bekannteste Sprache, die kontextfrei, aber nicht regulär ist, ist wohl die Sprache $\{a^n b^n | n \in \mathbb{N}\}$. Wir wollen zeigen, wie wir einen PDA konstruieren können, der diese Sprache akzeptiert. Dies wollen wir mit einem Leerkeller-Akzeptierer machen.

Während die a's eingelesen werden, müssen wir im Keller die Anzahl speichern. Wir legen also einfach jedes eingelesene a auf den Kellerstapel bis wir ein b einlesen. Wenn wir ein b eingelesen haben, wechseln wir in einen anderen Zustand. Wenn wir in diesem Zustand ein a einlesen, wissen wir, dass in dem Wort ein a hinter einem b vorkam. Wir können dann also in einen black hole Zustand wechseln. Ansonsten nehmen wir für jedes eingelesene b (auch schon für das erste) wieder ein a vom Stapel weg. Gab es mehr a's als b's wird der Keller noch nicht leer sein, wenn das letzte b eingelesen wurde. Das Wort wird also nicht akzeptiert. Gibt es mehr b's als a's werden wir noch ein b einlesen, wenn schon kein a mehr auf dem Stapel liegt. Wir können dem Automaten sagen, dass er in ein black hole wechseln soll, wenn wir noch ein b einlesen, wenn auf dem Stapel nur noch das Startsymbol liegt. Dann werden auch solche Worte nicht akzeptiert. Ein Wort wird also genau dann akzeptiert, wenn wir genausoviele a's vom Stapel nehmen, wie wir zu Beginn drauf gelegt haben - wenn es also genauso viele b's gibt, wie es a's gab. In Graphenschreibweise:



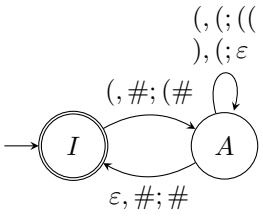
Die Kantenbeschriftungen $a; \#; a\#$ sind folgendermaßen zu verstehen: Die erste Information steht für das eingelesene Zeichen. Die zweite Information steht für das oberste Kellersymbol. Die dritte Information ist das Wort, durch das das oberste Kellersymbol ersetzt wird. Wenn wir $a; \#; a\#$ sehen, bedeutet dies: Wenn der Automat nun ein a einliest und ein # oben auf dem Keller sieht, kann er das # durch $a\#$ ersetzen.

Weitere kontextfreie Sprachen, die wir bereits kennengelernt haben, sind Dycksprachen. Da wir nun bereits einen Leerkeller-Akzeptierer kennengelernt haben, werden wir für die Dycksprache D_1 mit nur einem Klammerpaar $(,)$ nun einen Endzustand-Akzeptierer konstruieren.

Die Idee ist folgende: Der Keller speichert zu jedem Zeitpunkt, wie viele offene Klammerpaare noch geschlossen werden müssen. Immer wenn wir eine offene Klammer einlesen, legen wir sie oben auf den Stapel. Wenn wir eine geschlossene Klammer einlesen, während noch Klammerpaare zu schließen sind, nehmen wir eine der Klammern vom Stapel. Wenn wir eine geschlossene Klammer einlesen, wenn gar kein Klammerpaar mehr zu schließen ist, gehen wir in ein nicht eingezeichnetes black hole.

Wir wollen ein Wort akzeptieren, wenn kein Klammerpaar mehr zu schließen ist und wir nicht im black hole gelandet sind. Wir brauchen neben dem black hole nur

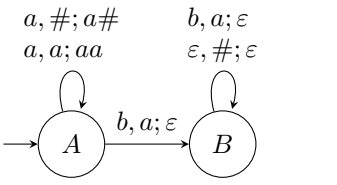
noch zwei Zustände - einen akzeptierenden und einen nicht-akzeptierenden Zustand. Wenn wir ein neues Klammerpaar öffnen, gehen wir in den nicht-akzeptierenden Zustand. Wenn wir dann sehen, dass das letzte Klammerpaar geschlossen wurde, können wir per ε-Übergang zurück in den akzeptierenden Zustand wechseln. Der gesamte Automat sieht also folgendermaßen aus:

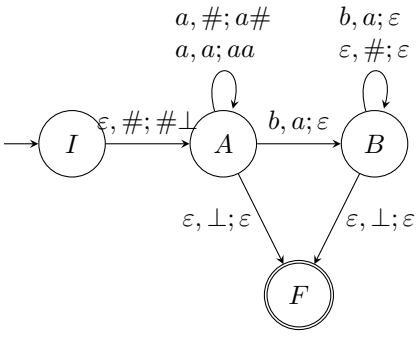


Äquivalenz der Akzeptierer. Wir haben nun zwei Arten kennengelernt, wie PDAs Sprachen akzeptieren können. Glücklicherweise müssen wir nicht zwischen diesen unterscheiden. Tatsächlich gibt es für jeden Leerkeller-Akzeptierer auch einen Endzustand-Akzeptierer und für jeden Endzustand-Akzeptierer auch einen Leerkeller-Akzeptierer, der die gleiche Sprache akzeptiert.

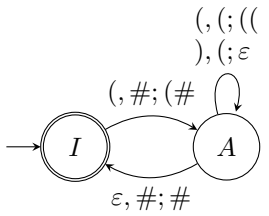
Widmen wir uns dem Leerkeller-Akzeptierer, den wir oben gesehen haben. Die Idee ist, dass wir ein neues Kellersymbol $\perp$ einführen, das uns anzeigt, dass der Keller leer ist. Statt mit z_0 wollen wir mit dem Keller $z_0\perp$ starten. Nur, wenn der Keller in der ursprünglichen Variante leer ist, sehen wir dann das Symbol $\perp$ auf dem Kellerstapel. Wir können also einen akzeptierenden Zustand hinzufügen und jedem Zustand einen ε-Übergang in den akzeptierenden Zustand erlauben, wenn $\perp$ auf dem Kellerstapel liegt. So ist genau dann ein akzeptierender Zustand erreichbar, wenn im ursprünglichen PDA der Keller geleert werden konnte.

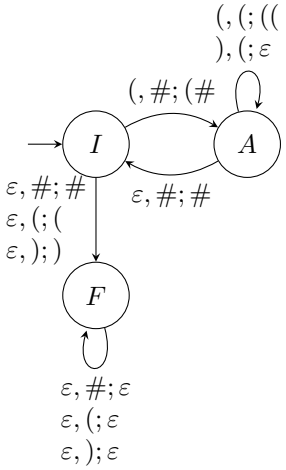
Wir müssen uns nur noch fragen, wie wir es schaffen können, den Keller mit $z_0\perp$ statt mit $\perp$ zu starten. Hierfür fügen wir zunächst einen neuen Startzustand ein. Von diesem Startzustand aus gibt es einen ε-Übergang zum ursprünglichen Startzustand, bei dem der Kellerinhalt angepasst wird. Hier sehen wir den ursprünglichen Leerkeller-Akzeptierer und den äquivalenten Endzustands-Akzeptierer im Vergleich untereinander:





Wir können auch zu einem Endzustands-Akzeptierer einen äquivalenten Leerkeller-Akzeptierer bauen. Wir wollen dafür, dass wir in jedem Endzustand den Keller vollständig leeren können. Es wäre leicht, Übergänge einzubauen, die genau dies ermöglichen. Doch wir müssen aufpassen, dass wir es dem Automaten nicht erlauben, danach oder zwischendurch noch weiter durch den Automaten zu laufen. Wenn wir uns also entscheiden wollen, in einem Endzustand den Keller zu leeren, muss dies der letzte Schritt des Automaten sein. Um dies sicherzustellen, können wir von den Endzuständen ausgehenend ε-Übergänge in einen Zustand einbauen, aus dem man nicht mehr herauskommt. Von diesem Zustand ausgehend muss kann dann jedes Zeichen des Kellers gelöscht werden. Hier ein Endzustand-Akzeptierer und der äquivalente Leerkeller-Akzeptierer:





PDAs und kontextfreie Grammatiken. Am Anfang des Kapitels wurde angekündigt, dass PDAs genau die kontextfreien Sprachen erkennen können. Um dies zu belegen, müssen wir zwei Dinge tun: Wir müssen zeigen, dass es zu jedem PDA eine kontextfreie Grammatik gibt, die die akzeptierte Sprache erzeugt und wir müssen zeigen, dass es zu jeder kontextfreien Grammatik einen PDA gibt, der die erzeugte Sprache akzeptiert. Wir werden in diesem Dokument letzteres tun. Für ersteres wollen wir nur darauf hinweisen, dass es auch möglich ist und dass die gewünschte Aussage daher korrekt ist, verzichten aber auf die Angabe der Konstruktion.

Dafür ist hier die Konstruktion, um zu einer kontextfreien Grammatik einen äquivalenten PDA zu erhalten. Wir konstruieren einen Leerkeller-Akzeptierer mit nur einem einzigen Zustand. Die einzigen Änderungen des Automaten geschehen also im Keller. Die Idee ist, dass wir im Keller die Ableitungsschritte der Grammatik simulieren. Das Kelleralphabet besteht genau aus den Terminalen und Nonterminalen der Grammatik. Das Startkellersymbol ist das Startnonterminal der Grammatik. In jedem Schritt des Automaten wollen wir im Keller auf das linkeste Nonterminal einen Ableitungsschritt anwenden.

Hierfür muss es also die Möglichkeit geben, die Nonterminale im Keller entsprechend der Ableitungsregeln der Grammatik zu ersetzen. Ist A ein Nonterminal und $A \to x$ eine Produktion der Grammatik, soll der Automat per ε-Übergang ein A im Keller durch x ersetzen können, wenn A oben liegt.

Durch diese Schritte werden auf die Nonterminale im Keller stückweise Ableitungsschritte angewandt, die zur Erzeugung eines Wortes w führen würden. Wir können leicht prüfen, ob das Wort, das der PDA als Eingabe bekommen hat, das erzeugte Wort w ist: Entspricht das erste Zeichen der Eingabe dem obersten Terminal im Keller, können wir ebenjenes Terminal aus dem Keller entfernen. Dann können wir das zweite Zeichen abgleichen, dann das dritte, usw. Entspricht irgendwann das eingelesene Zeichen nicht dem obersten Kellerterminal, so ist die Eingabe nicht das erzeugte Wort im Keller. Der Pfad kann abgebrochen werden. Ist die Eingabe vorbei bevor das Wort w vollständig aus dem Keller geleert werden konnte, bricht der Pfad ebenfalls nichtakzeptierend ab. Nur, wenn die Eingabe exakt dem im Keller erzeugten Wort entsprach, ist der Keller nach dem Einlesen der Eingabe vollkommen leer und das Wort wird akzeptiert.

Es ist zu beachten, dass diese Schritte nicht auf diese Art nacheinander passieren werden. Wir werden nicht erst das Wort erzeugen und dann die Zeichen abgleichen. Wir müssen immer das oberste Zeichen im Keller verarbeiten. Ist es ein Nonterminal, wenden wir einen Ableitungsschritt an. Ist es ein Terminal, gleichen wir es mit dem nächsten Zeichen der Eingabe ab. Somit wird der linkeste Teil des Wortes bereits mit dem Anfang der Eingabe abgeglichen worden sein bevor jedes Nonterminal ersetzt wurde und das Wort vollständig erzeugt wurde. An den nötigen Übergängen im Automaten ändert dies jedoch nichts - an unserem Verständnis, warum dieser Automat funktioniert, sollte es auch nicht viel ändern.

Deterministische Kellerautomaten. Bei regulären Sprachen haben wir sowohl DFAs als auch NFAs definiert und später gesehen, dass beide Arten von Automaten die gleiche Klasse von Sprachen beschreiben. PDAs haben wir bisher nur nichtdeterministisch definiert. Es stellt sich aber die Frage, ob deterministische und nichtdeterministische Automaten im Falle von Kellerautomaten wieder das gleiche Konzept definieren. Leider stellt sich heraus, dass dies nicht der Fall ist.

Definieren wir zunächst deterministische Kellerautomaten, auch DPDA abgekürzt: Einen Endzustands-Akzeptierer bezeichnen wir als **deterministisch**, wenn in jeder Konfiguration des Automaten maximal ein Folgeübergang gewählt werden kann. Anders: Wird im Zustand q mit oberstem Kellersymbol γ und nächstem Eingabezeichen das Zeichen σ eingelesen, darf es keine zwei Übergänge in $\Delta(q, \sigma, \gamma)$ geben. Es darf auch keine zwei ε-Übergänge $\Delta(q, \varepsilon, \gamma)$ geben. Es darf außerdem nicht sowohl einen Übergang in $\Delta(q, \sigma, \gamma)$, als auch einen ε-Übergang in $\Delta(q, \varepsilon, \gamma)$ geben. Ist dies für alle (q, σ, γ) gegeben, ist der Kellerautomat deterministisch.

Man könnte sich fragen, warum wir nur deterministische Endzustands-Akzeptierer und keine deterministischen Leerkeller-Akzeptierer zugelassen haben. Das Problem ist, dass deterministische Leerkeller-Akzeptierer viel zu viele Sprachen nicht erkennen könnten.

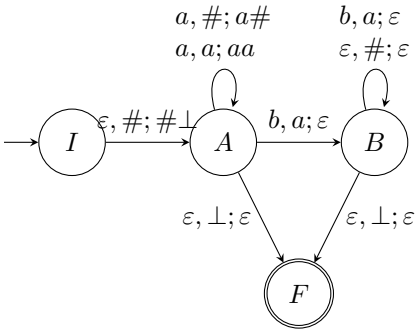
Der Definitionsbereich der Übergangsfunktion ist $Q \times \Sigma_\varepsilon \times \Gamma$. Insbesondere besteht Γ nur aus Zeichen, die im Keller liegen können. Wenn der Keller leer ist, wird kein Zeichen aus Gamma eingelesen. Der Definitionsbereich der Übergangsfunktion erlaubt es daher nicht, einen Übergang zu definieren, der begangen werden kann, wenn der Keller leer ist. Wenn ein Übergang den Keller völlig leert, muss dieser daher der letzte Schritt des Pfades sein.

Ein Leerkeller-Akzeptierer akzeptiert ein Wort, wenn die Konfiguration $(q, \varepsilon, \varepsilon)$ erreicht werden kann. Ein Leerkeller-Akzeptierer akzeptiert ein Wort also nur dann, wenn es einen Pfad gibt, der nach dem Einlesen den Keller vollständig leert.

Betrachten wir nun also deterministische Leerkeller-Akzeptierer: Nehmen wir die endliche Sprache $\{a, aa\}$. Ein deterministischer Leerkeller-Akzeptierer dieser Sprache müsste a akzeptieren können. Es muss also einen Pfad geben, der beim Einlesen von a den Keller völlig leert bevor noch ein weiteres Zeichen eingelesen wird. Da es sich um einen deterministischen Automaten handelt, muss dies aber auch der einzige Pfad sein. Wenn wir also die Eingabe aa haben, wird der Keller bereits geleert bevor das zweite a eingelesen wird. Nach der ersten Beobachtung würde das bedeuten, dass es keinen Übergang mehr gibt, der das zweite a einlesen kann, weshalb das Wort aa nicht akzeptiert werden kann. Es scheint, als gäbe es keinen Leerkeller-Akzeptierer für die endliche und somit trivial reguläre Sprache $\{a, aa\}$. Algemeiner können wir sagen, dass eine Sprache L nicht von einem deterministischen Leerkeller-Akzeptierer akzeptiert werden kann, wenn ein Wort der Sprache ein Präfix eines anderen Wortes der Sprache ist.

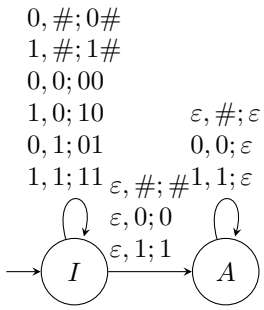
Da deterministische Leerkeller-Akzeptierer also selbst viele der einfachsten Sprachen nicht erkennen können, halten wir sie für zu uninteressant, um sie im Folgenden weiter zu betrachten. Wird im Folgenden also von einem DPDA gesprochen, setzen wir voraus, dass es sich um einen Endzustands-Akzeptierer handelt.

Beispiel. Es fällt auf, dass der oben angegebene Endzustands-Akzeptierer für die Sprache $\{a^n b^n | n \in \mathbb{N}\}$ bereits deterministisch ist:



In keinem Zustand q, gibt es ein $\sigma \in \Sigma$ und ein $\gamma \in \Gamma$, sodass wir zwei verschiedene Entscheidungsmöglichkeitern hätten, wenn wir im Zustand q das Zeichen σ einlesen während γ oben auf dem Stapel liegt.

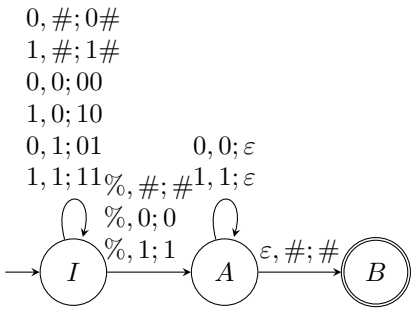
Palindrome als Trennsprache. Wir betrachten die Sprache der Palindrome gerader Länge. Dies ist die Sprache aller Wörter der Form $\{ww^R | w \in \{0, 1\}\}$. Wir finden schnell einen Leerkeller-Akzeptierer für die Sprache. Wir legen zunächst jedes eingelesene Zeichen auf den Kellerstapel. Irgendwann gehen wir per *varepsilon*-Übergang in einen anderen Zustand über, in dem wir das oberste Zeichen genau dann löschen, wenn es genau dem eingelesenen Zeichen entspricht. Ist der Keller am Ende leer bedeutet das, dass die Zeichen aus der Aufbauphase des Stapels danach in genau umgekehrter Reihenfolge in der Abbauphase wieder aufgetaucht sind. Das Wort hat dann also genau die gewünschte Form. Der Automat sieht dann folgendermaßen aus:



Wir merken jedoch, dass der Nichtdeterminismus hier eine wichtige Rolle spielt. Der Automat muss genau dann einen ε-Übergang machen, wenn bisher die Hälfte des Wortes eingelesen wurde. Sind wir bei der Mitte des Wortes angekommen gibt es aber keine Anzeichen, die uns eindeutig verraten, dass dies die Mitte des Wortes ist. Intuitiv müssen wir also nach jedem Zeichen davon ausgehen können, dass gerade die Mitte des Wortes überschritten wurde und sowohl den Pfad testen können, bei dem dies tatsächlich der Fall ist, als auch den Pfad, bei dem die Mitte des Wortes noch kommt. Daher wird ein deterministischer Automat diese Sprache nicht akzeptieren können.

Dies ist natürlich kein formeller Beweis, aber es ist tatsächlich wahr, dass kein DPDA diese Sprache akzeptieren kann. Es gibt also Sprachen, die von einem PDA, aber von keinem DPDA erkannt werden können.

Die Intuition, dass ein DPDA die Sprache erkennen könnte, wenn die Mitte markiert wäre, können wir auf folgende Weise formalisieren: Wir betrachten die Sprache $\{w\%w^R | w \in \{0,1\}^*\}$. Dies ist fast die gleiche Sprache wie die oben genannte, doch nun befindet sich ein Sonderzeichen in der Mitte des Wortes. Dieses Zeichen kann einem DPDA das Signal geben, dass nun die Mitte des Wortes gekommen ist und der Zustand gewechselt werden muss. Wir können den oberen Automaten ansonsten fast unverändert übernehmen, um einen DPDA für diese Sprache zu erhalten:



Die Sprachklasse DCFL. Da wir nun aber wissen, dass nicht jede Sprache, die von einem PDA akzeptiert wird, auch von einem DPDA akzeptiert wird, macht es Sinn, die Sprachklasse derjenigen Sprachen zu betrachten, die von einem DPDA erkannt werden können. Diese liegt zwischen der Klasse der regulären Sprachen und der Klasse der kontextfreien Sprachen. Wir kennen *Reg* als Klasse der regulären Sprachen und CFL als Klasse der kontextfreien Sprachen. Sei nun $DCFL$ die Klasse der **deterministisch kontextfreien Sprachen** - also die Klasse aller Sprachen, die von einem deterministischen Kellerautomaten akzeptiert werden können. Wir können einen DFA schnell als DPDA auffassen, in dem sich der Keller nie verändert. Daher ist jede reguläre Sprache auch deterministisch kontextfrei. Umgekehrt haben

wir aber bereits gesehen, dass wir für $\{a^n b^n | n \in \mathbb{N}\}$ einen DPDA finden konnten. Es gibt also deterministisch kontextfreie Sprachen, die nicht regulär sind. Insgesamt ist also Reg echt in $DCFL$ enthalten, während $DCFL$ echt in CFL enthalten ist:

$$Reg \subsetneq DCFL \subsetneq CFL$$

Wir können noch einige Eigenschaften von $DCFL$ betrachten. Anders als CFL ist $DCFL$ unter Bildung von Komplementen abgeschlossen. Bei DFAs konnten wir einfach jeden akzeptierenden Zustand zu einem nicht-akzeptierenden Zustand und jeden nicht-akzeptierenden Zustand zu einem akzeptierenden Zustand machen, um einen DFA zu erhalten, der die komplementäre Sprache akzeptiert. Bei DPDAs ist die Idee ähnlich. Da die Pfade eindeutig sind, landet jedes Wort, das vorher in einem akzeptierenden Zustand landete, nun in einem nicht-akzeptierenden Zustand und umgekehrt. Wir müssen jedoch einige Probleme bedenken:

Einerseits kann es passieren, dass ein Wort nie bis zum Ende eingelesen wird, weil der Keller leer ist, kein weiterer Übergang definiert ist oder der Automat in einer Endlosschleife aus ε-Übergängen landet. Solche Wörter werden weder vom ursprünglichen, noch vom komplementären Automaten akzeptiert. Andererseits kann es passieren, dass ein Wort bis zum Schluss eingelesen wird und dass dann ein ε-Übergang von einem akzeptierenden in einen nicht-akzeptierenden Zustand führt. In diesem Fall ist für das Wort sowohl im ursprünglichen, als auch im komplementären Automaten ein Pfad zu einem akzeptierenden Zustand verfügbar. Solche Wörter werden also von beiden Automaten akzeptiert. Wir müssen also zunächst Änderungen am Automaten vornehmen, die dafür sorgen, dass keiner dieser Fälle auftritt. An dieser Stelle sei ausgelassen, wie dies möglich ist, doch es sei erwähnt, *dass* es möglich ist. Danach können wir die Rollen akzeptierender und nicht-akzeptierender Zustände tauschen und erhalten einen DPDA, der die komplementäre Sprache akzeptiert.

Wir sehen, dass $DCFL$ unter der Bildung von Schnitten abgeschlossen ist. Hierfür sei auf Kapitel 3.5 verwiesen. Dort haben wir zwei kontextfreie Sprachen gesehen, deren Schnitt nicht kontextfrei ist. Wir sehen recht schnell, dass diese beiden Sprachen auch deterministisch kontextfrei sind und somit ist dies auch ein Gegenbeispiel in $DCFL$.

Analog zur Argumentation in Kapitel 3.5 sehen wir wegen

$$A \cap B = \overline{\overline{A} \cup \overline{B}}$$

dass eine Sprache, die unter der Bildung von Komplementen, aber nicht unter der Bildung von Schnitten abgeschlossen ist, ebenfalls nicht unter der Bildung von Vereinigungen abgeschlossen sein kann.

Bei CFL und $DCFL$ verhält es sich also genau umgekehrt: CFL war unter der Bildung von Vereinigungen, aber nicht unter der Bildung von Komplementen abgeschlossen - $DCFL$ ist unter der Bildung von Komplementen, aber nicht unter der Bildung von Vereinigungen abgeschlossen.

Betrachten wir nun das Wortproblem für $DCFL$, dem wir uns in CFL bereits in Kapitel 3.4 gewidmet haben. Da wir in einem DPDA stets nur einen Pfad haben, können wir das Wortproblem entscheiden, indem wir den Lauf durch den $DPDA$ simulieren. Da wir nicht zwischen vielen verschiedenen Pfaden unterscheiden müssen, läuft dies sogar in Linearzeit ab.

Dies ist sogar ein gutes Kriterium für Grammatiken, die deterministisch kontextfreie Sprachen erzeugen. Für reguläre und kontextfreie Sprachen kennen wir je eine entsprechende Art von Automaten als auch eine entsprechende Art von Grammatiken. Für $DCFL$ kennen wir bisher nur Automaten. Welche Bedingung müssen

wir an unsere Grammatiken stellen, damit genau die deterministisch kontextfreien Sprachen erzeugt werden? Das Kriterium ist nicht so leicht, wie das der bisher bekannten Grammatikarten. Die Idee ist aber, dass wir kontextfreie Grammatiken betrachten wollen, bei denen das Wortproblem in Linearzeit lösbar ist. Genauer: Wir wollen eine Ableitung für das Wort konstruieren, indem wir vom Startsymbol ausgehend in jeder Satzform das linkeste Nonterminal ersetzen wollen. Wenn wir anhand unserer Kenntnis des nächsten Terminals, das im Wort vorkommen soll, bereits eindeutig bestimmen können, welcher Ableitungsschritt anzuwenden ist, erzeugt die Grammatik eine deterministisch kontextfreie Sprache.

3.7. **Entscheidungsprobleme für Kontextfreie Grammatiken.** Den Abschnitt zu regulären Sprachen haben wir in Kapitel 2.8 mit einer Liste verschiedener Entscheidungsprobleme beendet. Bei regulären Sprachen konnten wir für jedes dieser Probleme einen Algorithmus angeben, mit dem wir eine Antwort erhalten konnten. Wir können diese Liste nun für kontextfreie Sprachen durchgehen und werden sehen, dass einige dieser Probleme nun nicht mehr per Algorithmus entscheidbar sind.

Das Wortproblem. Wir haben bereits in Kapitel 3.4 mithilfe des CYK-Algorithmus gesehen, wie wir das Wortproblem für kontextfreie Sprachen entscheiden können.

Das Leerheitsproblem. Das Leerheitsproblem war die Frage, ob eine gegebene Grammatik überhaupt ein Wort erzeugen kann oder ob die erzeugte Sprache leer ist. Diese Frage ist auch für kontextfreie Grammatiken entscheidbar.

Es kann passieren, dass eine Grammatik nur die leere Sprache erzeugt, selbst wenn die Grammatik mehrere Nonterminale und Produktionen hat. Dies ist der Fall wenn wir in den erzeugten Satzformen bei jedem Schritt neue Nonterminale erzeugen und nie eine Satzform erhalten, in der jedes Nonterminal vollständig durch Terminale ersetzt werden kann. Die Idee „Kann jedes Nonterminal vollständig durch Terminale ersetzt werden?“ ist auch die zentrale Idee für den folgenden Algorithmus. Wir führen eine Menge T ein, in der wir alle Nonterminale speichern, die (in einem oder mehreren Ableitungsschritten) vollständig durch Terminale ersetzbar sind. Solche Nonterminale nennen wir auch **terminierbar**.

In der ersten Iteration fügen wir der Menge jedes Nonterminal hinzu, das in einem einzigen Ableitungsschritt auf eine Satzform abgebildet werden kann, die nur aus Terminalen besteht. In jeder folgenden Iteration gehen wir alle Nonterminale A durch, die noch nicht in T enthalten sind. Wir prüfen, ob es eine Produktion $A \to x$ gibt, bei der x nur aus Teminalen und Zeichen aus T besteht. Wenn ja, fügen wir A auch T hinzu. Haben wir in einer Iteration kein weiteres Nonterminal zu T hinzugefügt, sind wir fertig und wir haben alle terminierbaren Nonterminale gefunden.

Die erzeugte Sprache ist genau dann leer, wenn aus dem Startsymbol S kein Wort erzeugt werden kann, das nur aus Terminalen besteht - also genau dann wenn S nicht terminierbar ist. Wir können also wie oben beschrieben die Menge T der terminierbaren Nonterminale erzeugen und dann prüfen, ob S in T enthalten ist. Wenn ja, ist die Sprache nicht leer. Wenn nein, ist sie es.

Das Endlichkeitsproblem. Wir widmen uns nun dem Endlichkeitsproblem: Gegeben sei eine kontextfreie Grammatik G. Ist die erzeugte Sprache endlich? Das Endlichkeitsproblem ist algorithmisch endscheidbar.

Ein Ansatz läuft über brute-force. Wir wissen, dass kontextfreie Sprachen pumpbar sind. In Kapitel 3.3 haben wir gesehen, dass wir bei einer Chomsky-Normalform-Grammatik die Pumpingkonstante $n = 2^{|V|}$ wählen können. Ist die erzeugte Sprache unendlich, muss es auch ein Wort w geben, das mindestens n Zeichen enthält. Wir

können dies aber noch stärker formulieren. In einer unendlichen Sprache gibt es ein Wort w mit $n \leq |w| < 2n$.

Angenommen, das kürzeste Wort w mit $n \leq |w|$ hätte mindestens $2n$ Zeichen. Da das Wort nach dem Pumping-Lemma pumpbar ist, finden wir Teilworte v, x, die wir aus w entfernen können ohne aus der Sprache zu fallen. Da nach Pumping-Lemma aber $|vx| \leq n$ ist, entfernen wir hierbei maximal n Zeichen. Die abgepumpte Version hat also immer noch mindestens n Zeichen - im Widerspruch zu der Annahme w sei das kürzeste Wort mit mindestens n Zeichen.

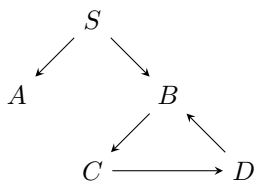
Wenn die Sprache also unendlich ist, gibt es ein Wort w in der Sprache mit $2^{|V|} \leq |w| < 2 \cdot 2^{|V|}$. Wir müssen also „nur“ bei endlich vielen Wörtern überprüfen, ob sie in der Sprache enthalten sind. Da wir das Wortproblem algorithmisch lösen können, können wir es auch mehrmals algorithmisch lösen. Wenn wir bei all diesen Tests kein Wort gefunden haben, muss die Sprache endlich sein. Wenn wir ein solches Wort gefunden haben, können wir es nach dem Pumping-Lemma beliebig oft aufpumpen, weshalb die erzeugte Sprache unendlich ist.

Diese brute-force Lösung ist aber natürlich sehr unelegant. Wir können einen sehr viel effizienteren Weg finden, wenn wir über die Struktur unendlicher Ableitungen nachdenken. Ähnlich wie beim Pumping-Lemma merken wir, dass wir in unendlichen Sprachen Ableitungen haben, die so lang sind, dass zwangsweise an einer Stelle ein Nonterminal A in einem oder mehreren Schritten auf eine Satzform abgeleitet wird, die wieder ein A enthält. Wir benötigen also zwangsweise einen Zyklus von Nonterminalen $A_1 \to A_2 \to \cdots \to A_n \to A_1$, wobei es eine Produktion gibt, mit der aus A_1 eine Satzform erzeugt wird, die A_2 enthält, A_3 in einer Satzform vorkommt, die aus A_2 erzeugt werden kann... Bis dann A_1 wieder in einer Satzform vorkommt, die aus A_n erzeugt werden kann.

Dies ist unser erster Ansatz für den Algorithmus. Wir wollen bestimmen, ob es einen solchen Zyklus gibt. Zunächst bringen wir die Grammatik in Chomsky-Normalform. Dies tun wir, damit beim erneuten Durchlaufen eines Zyklus auch tatsächlich die Länge des erzeugten Wortes erhöht wird. Dann stellen wir einen Graphen auf, bei dem jedes Nonterminal ein Knoten ist. Wir fügen genau dann eine Kante von A nach B ein, wenn es eine Produktion gibt, mit der A auf eine Satzform abgeleitet wird, die B enthält. Dann prüfen wir, ob wir in dem Graphen vom Startsymbol S aus in einen Zyklus gelangen können. Dies ist schon fast ein funktionierender Algorithmus. Leider ist er noch nicht ganz korrekt. Betrachten wir die folgende Grammatik:

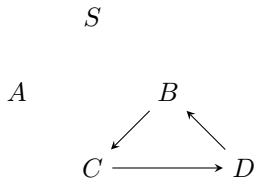
$$S \to AB, B \to CC, C \to DD, D \to d|BB$$

Wir sehen, dass aus dem Startsymbol S unweigerlich ein A erzeugt wird. Für das A gibt es aber keine weiterführenden Produktionen. Daher kann diese Grammatik keine Ableitung finden, in der alle Nonterminale durch Terminale ersetzt werden können. Die erzeugte Sprache ist $\emptyset$ und somit endlich. Da die Grammatik aber in Chomsky-Normalform ist, können wir ja trotzdem einmal die oben beschriebene Konstruktion anwenden. Wir erhalten den folgenden Graphen:



Wir können von S ausgehend den Zyklus $C \to D \to E \to C$ erreichen, weshalb wir nach den oben stehenden Überlegungen schlussfolgern müssten, dass die Sprache

unendlich ist. Wir haben aber bereits gesehen, dass die Sprache nicht nur endlich, sondern sogar leer ist. Das Problem liegt daran, dass uns ein Zyklus nichts gibt, wenn wir auf dem Weg Nonterminale erzeugen, die nicht terminierbar sind. Wir müssen in unserer Konstruktion also zunächst die Menge aller terminierbaren Nonterminale bestimmen, wie wir es für das Leerheitsproblem getan haben. Danach fügen wir eine Kante von A nach B nur ein, wenn B in einer Satzform vorkommt, die aus A erzeugt werden kann **und** in dieser Satzform keine nicht-terminierbaren Nonterminale vorkommen. Wenden wir diese Konstruktion auf die obere Grammatik an, bekommen wir folgenden Graphen:



Nun können wir von S ausgehend keinen Zyklus mehr erreichen und wir schlussfolgern korrekt, dass die Sprache endlich ist. Aus der Graphentheorie sind auch Algorithmen bekannt, mit denen wir prüfen können, ob von einem Knoten ausgehend ein Zyklus erreicht werden kann. Daher können wir uns sicher sein, dass wir auch auf diese Weise das Endlichkeitsproblem rein algorithmisch lösen können.

Unentscheidbare Probleme. Bisher konnten wir jedes der Probleme, die wir für reguläre Sprachen entscheiden konnten, auch für kontextfreie Sprachen entscheiden. Dies hört nun auf. Für reguläre Sprachen waren auch das Äquivalenzproblem und das Schnittproblem algorithmisch entscheidbar. Dies ist für kontextfreie Sprachen nicht mehr der Fall. Eine Frage, die sich bei kontextfreien Sprachen stellt, ist die Frage nach der Mehrdeutigkeit der Grammatik oder der inhärenten Mehrdeutigkeit der Sprache. Beide Fragen sind nicht algorithmisch entscheidbar.

Es sei darauf hingewiesen, dass „nicht algorithmisch entscheidbar“ nicht nur bedeutet, dass wir noch keinen Algorithmus gefunden haben, der die Frage entscheiden kann. Es bedeutet, dass mit völliger Sicherheit bewiesen wurde, dass es keinen Algorithmus geben *kann*, der die Frage entscheiden kann. Später in der Vorlesung werden wir Techniken sehen, mit denen wir formal beweisen werden, dass es zu manchen Problemen keinen Algorithmus geben kann.

Es sei aber darauf hingewiesen, dass das Äquivalenzproblem für $DCFL$ tatsächlich entscheidbar ist. Der Beweis dafür ist jedoch sehr kompliziert und wird hier nicht mit aufgenommen.