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 3.4 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.

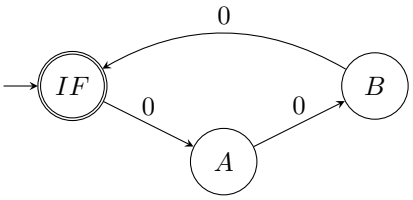
Einen solchen Computer nennen wir einen **endlichen Automaten** oder auch **deterministischen endlichen Automaten**, um ihn von später definierten Modellen abzugrenzen. Wir schreiben manchmal auch DFA für „deterministic finite automaton“. Wie bereits angekündigt, werden wir die im letzten Abschnitt entwickelte 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.

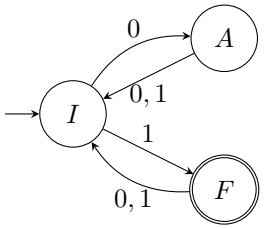
Dies reicht aus, um graphisch jede Funktionsweise des Automaten darzustellen.

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.)

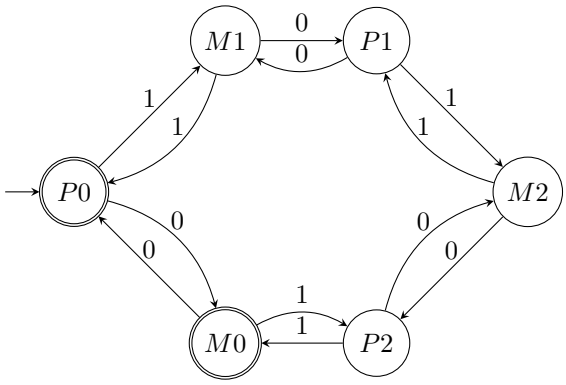
Hier ist ein weiteres Beispiel eines Automaten:



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.

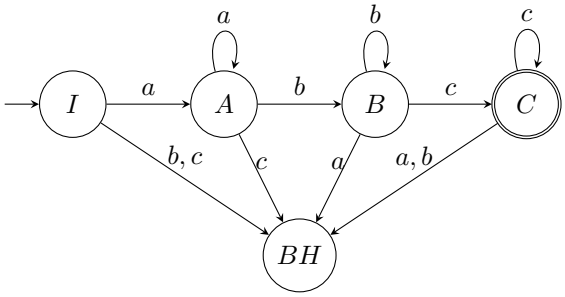
Partielle Automaten. Auch wenn die effektivere Methode der Vereinfachung erst später kommt, können wir auch jetzt schon einen Trick lernen, um manche Automaten zumindest ein bisschen übersichtlicher zu gestalten. Gelegentlich hat ein Automat Zustände, von denen aus kein Endzustand mehr zu erreichen ist. 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.

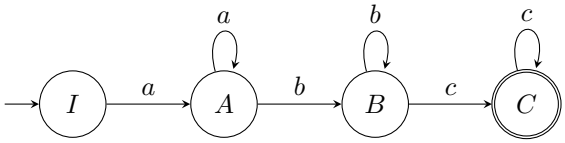
Dies führt dazu, dass es Knoten gibt, von denen aus nicht zu jedem Zeichen eine Kante ausgeht. Befinden wir uns in einem solchen Zustand und erhalten als Eingabe ein Zeichen, zu dem es keine Kante gibt, können wir sofort davon ausgehen, dass die Eingabe nicht akzeptiert wird. (Vor dem Entfernen der unnötigen 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:



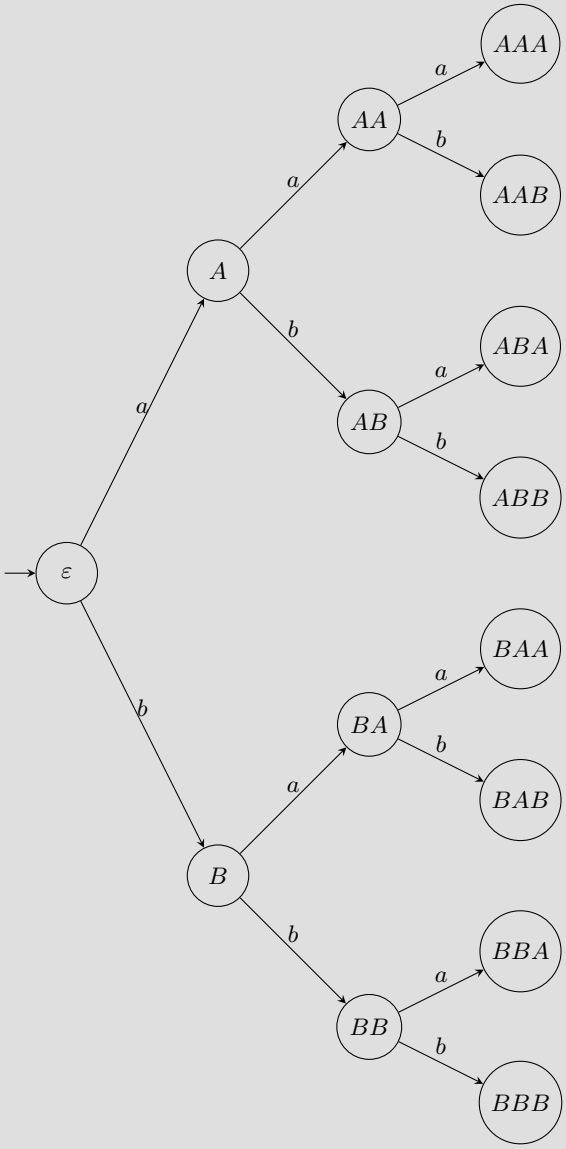
Zum Vergleich sehen wir hier den gleichen Automaten mit entferntem black hole Zustand:



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.

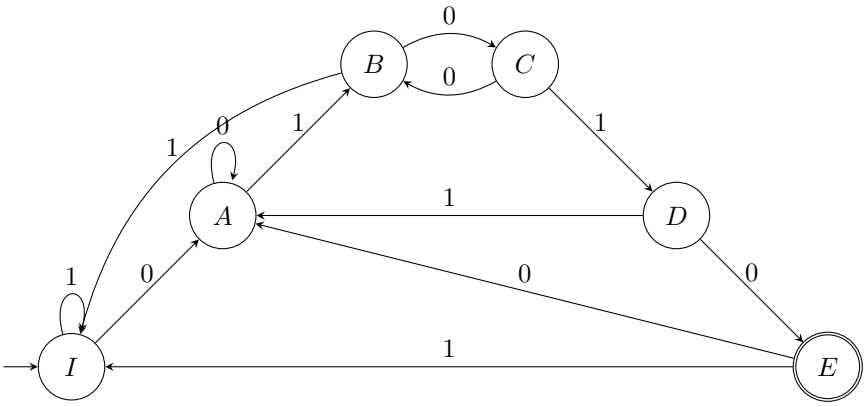
Ein notwendiges Kriterium. Eine Struktur regulärer Mengen fällt auf, wenn wir uns die Graphenstruktur anschauen. Enthält eine Sprache nur endlich viele Wörter, ist die Sprache sowieso regulär. Enthält eine Sprache unendlich viele Wörter, so gibt es insbesondere auch Wörter beliebig langer Länge. Man 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.

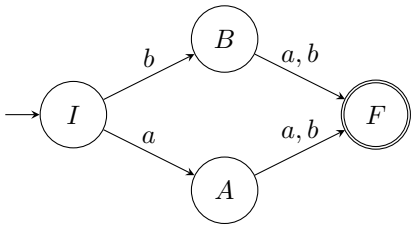
Formale Stolperfallen.

- 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.

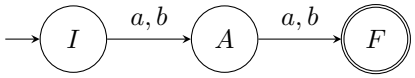
2.3. **Automatenminimierung.**

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:

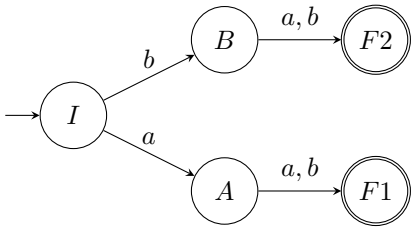


Offensichtlich ist es hier egal, ob wir uns im Zustand A oder im Zustand B befinden. Egal, in welchem der beiden Zustände wir sind, wird die Eingabe eines weiteren Zeichens uns wieder in den Zustand F führen. Es wäre also eine Idee, gar nicht mehr dazwischen zu unterscheiden, in welchem der beiden Zustände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

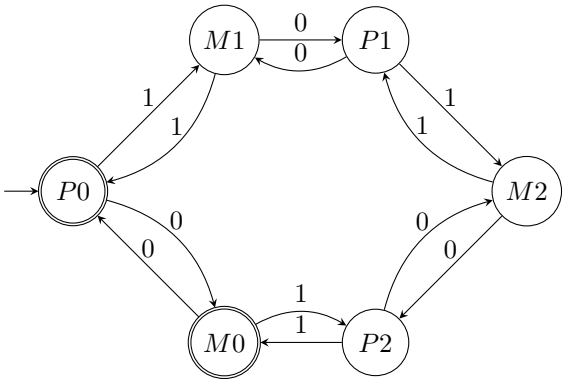


Hier gibt es keine zwei Zustände mit gleichen Folgezuständen. (Man könnte argumentieren, dass $F1$ und $F2$ beide den gleichen nicht eingezeichneten black-hole Zustand als Folgezustand haben. Nehmen wir also an, dass Eingaben aus $F1$ und $F2$ in zwei verschiedende black-hole Zustände führen.) Trotzdem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

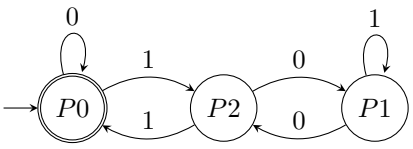
Allgemeiner können wir A und B nicht verschmelzen, wenn es ein Zeichen gibt, sodass wir bereits ausgeschlossen haben, dass die jeweiligen Folgezustände von A und B miteinander verschmolzen werden können. So können wir iterativ weitere Paare ausschließen. Wenn wir in einer Iteration keine neuen Paare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

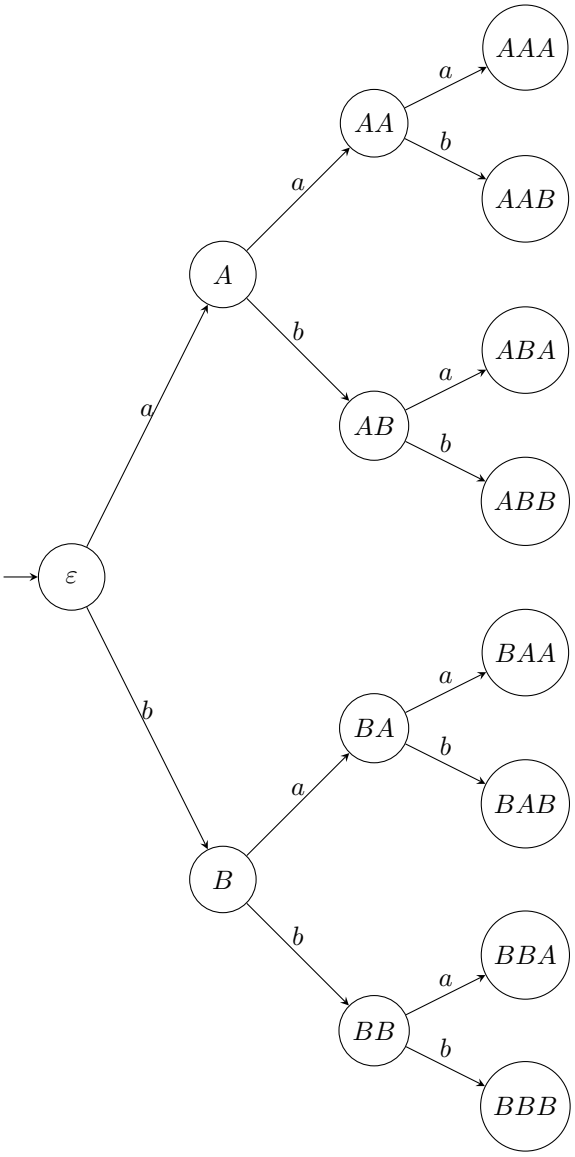
Da wir nun neue Kreuze gesetzt haben, kann sein, dass es Zustandspaare gibt, die vorher nicht in ein angekreuztes Paar überführt wurden, jetzt aber schon. Deshalb müssen wir nun noch einmal alle Paare durchgehen, die kein Kreuz haben. In diesem Fall kommt es nicht vor, dass es dann noch weitere Kreuze 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.

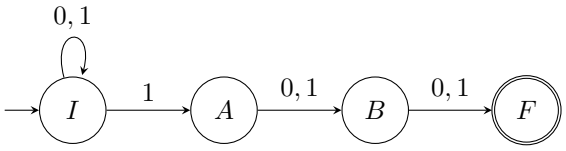
Wir erinnern uns daran, dass jeder Zustand zu genau einem Wort korrespondiert. (Der Zustand, der zu dem Wort w korrespondiert, ist der Zustand, in dem wir uns nach Einlesen des Wortes w befinden.) Wir können so die Zustandsmenge mit Σ^* identifizieren. Das Verschmelzen von Zuständen können wir als das 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.

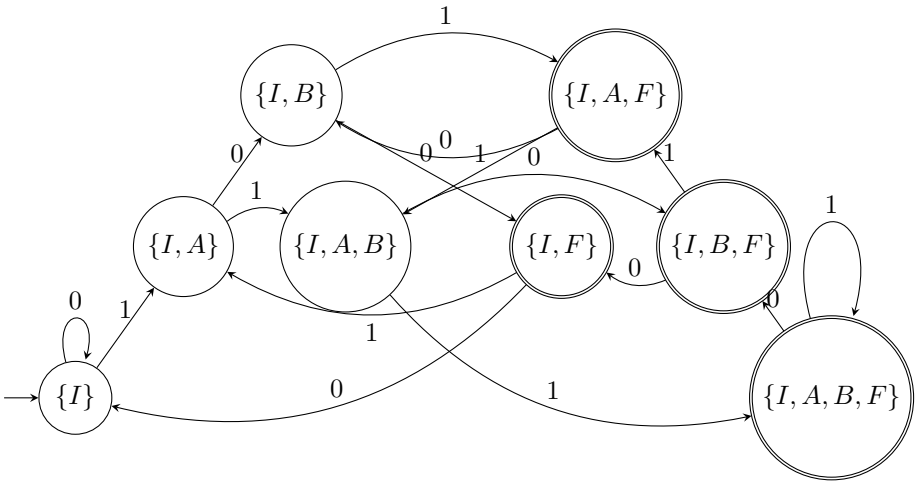
Insgesamt ist ein NFA also definiert durch Angabe von fünf Informationen: Die Zustandsmenge Q, das Eingabealphabet Σ, die Übergangsfunktion $\Delta : Q \times \Sigma \to 2^Q$, die Startzustandsmenge $Q_0 \in 2^Q$ und die Endzustandsmenge F.

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.

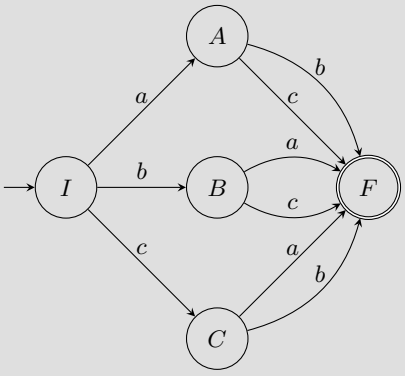
Nutzen von NFAs. NFAs helfen uns also nicht dabei, neue Sprachen zu erkennen. Warum haben wir NFAs also definiert? Ein Nutzen von NFAs ist die deutlich größere Übersichtlichkeit. Betrachten wir noch einmal den NFA vom Anfang dieses Abschnitts. Dieser war ein sehr übersichtlicher Automat, der aus nur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

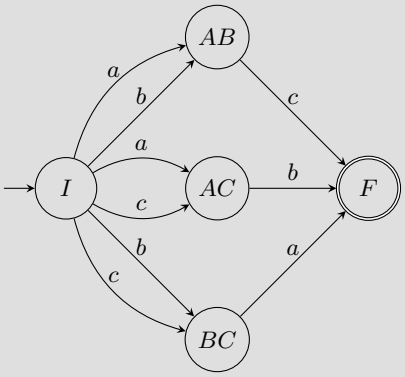


Wir erhalten einen sehr unübersichtlichen DFA mit acht Zuständen. Allgemein können wir zwar immer einen äquivalenten DFA konstruieren - die Anzahl der Zustände kann dabei aber exponentiell wachsen. Somit können NFAs häufig eine sehr viel übersichtlichere Darstellung der Lösung sein.

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.

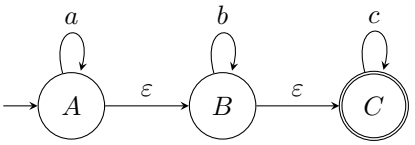
Ein Problem, auf das wir hierbei stoßen, ist die Tatsache, dass es nicht immer einen eindeutigen minimalen NFA zu einer Sprache gibt. Man betrachte die folgenden zwei NFAs:





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.

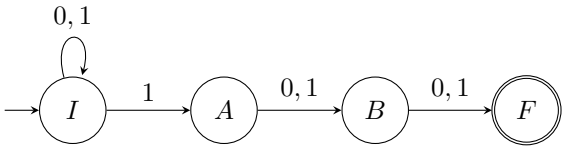
2.5. **Abschlusseigenschaften.** Wir werden uns nun endlich der Behauptung widmen, die schon seit längerem im Raum steht. Nämlich, dass unsere zwei Definitionen regulärer Mengen tatsächlich das Gleiche definieren. Wir haben einerseits die Definition, die reguläre Ausdrücke nutzt und die Definition, die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

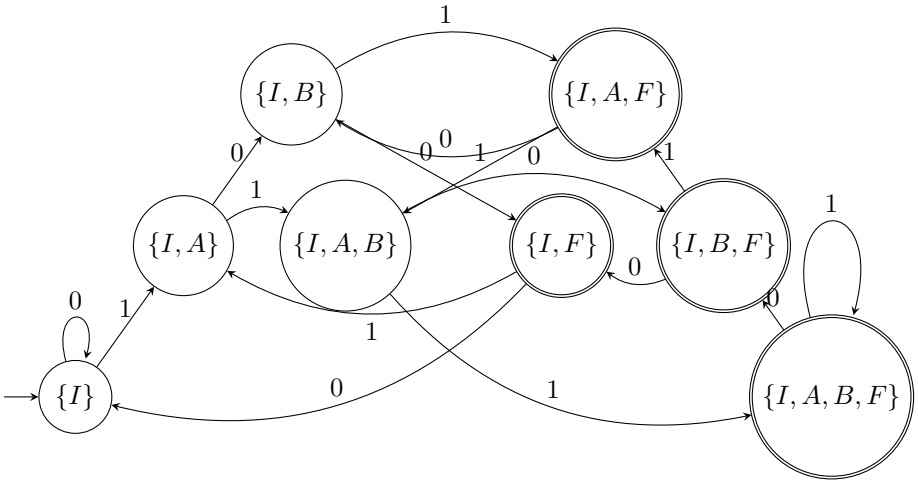
Wir werden im Folgenden einige dieser Eigenschaften genauer betrachten und anhand eines Beispiels durchgehen.

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.

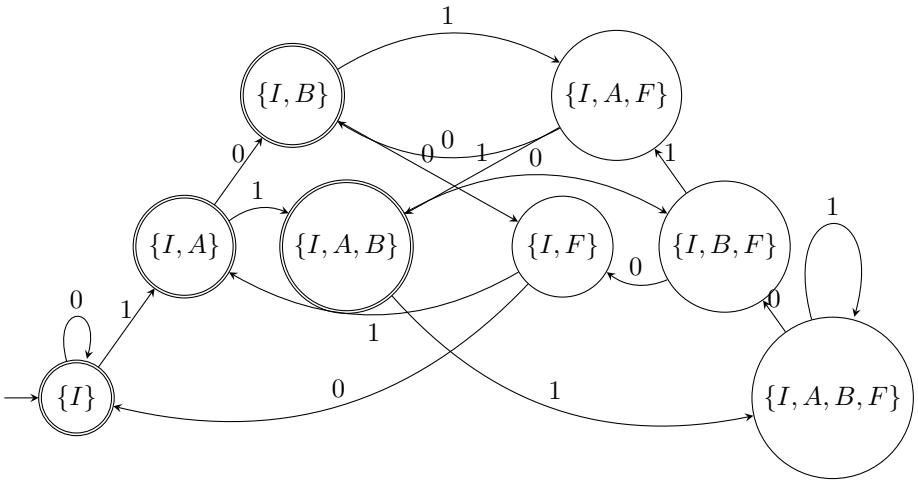
Wir betrachten die Sprache L_{-3}, die Sprache aller Wörter über $\{0,1\}$, deren drittletztes Zeichen eine 1 ist. Die Sprache wird von folgendem bereits bekannten NFA erkannt:



Einen äquivalenten DFA haben wir ebenfalls bereits gesehen:

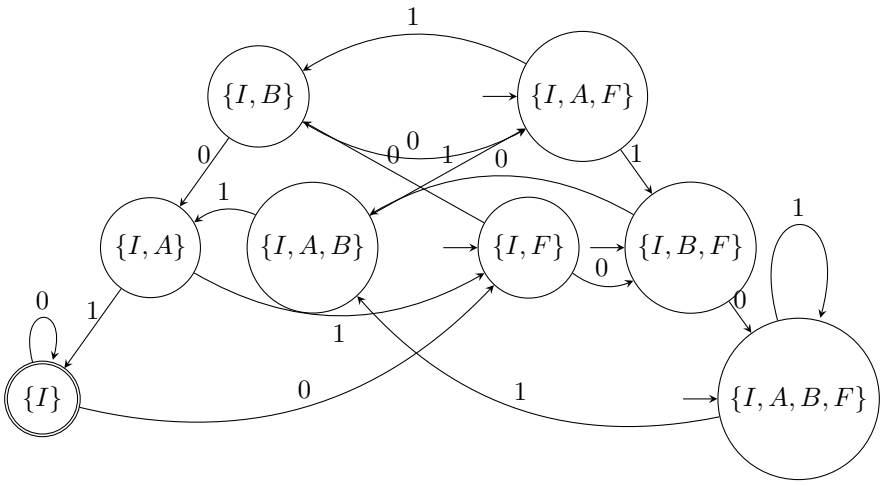


Das Komplement akzeptieren wir nun, indem wir die Rollen von akzeptierenden und nicht-akzeptierenden Zuständen tauschen:

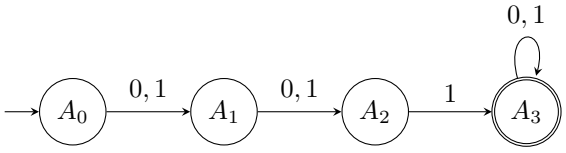


Es reicht nicht aus, akzeptierende und nicht-akzeptierende Zustände in einem NFA zu tauschen. In einem NFA kann es passieren, dass es zu einem Wort sowohl einen Pfad gibt, der in einem akzeptierenden Zustand endet, als auch einen Pfad, der in einem nicht-akzeptierenden Zustand endet. Ein solches Wort 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:

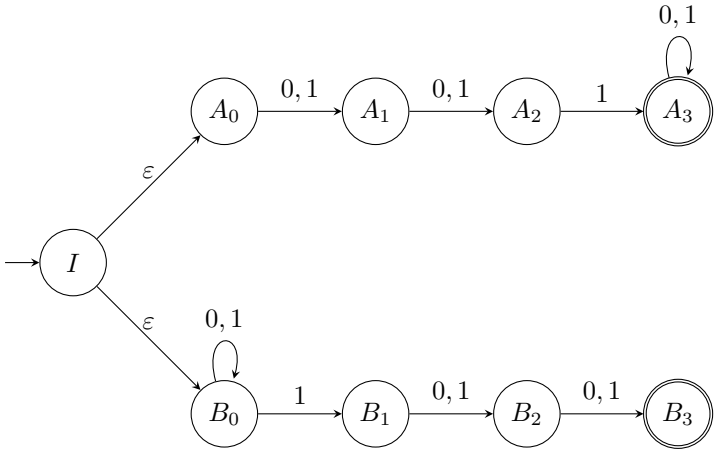


Anders als bei der Konstruktion des Komplements können wir die Konstruktion des Automaten für die Reflexion auch auf einen NFA anwenden. In diesem Beispiel wird der NFA sogar zu einem DFA, wenn wir die Konstruktion anwenden. Es fällt auf, dass der angegebene Minimalautomat genau der DFA ist, den wir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

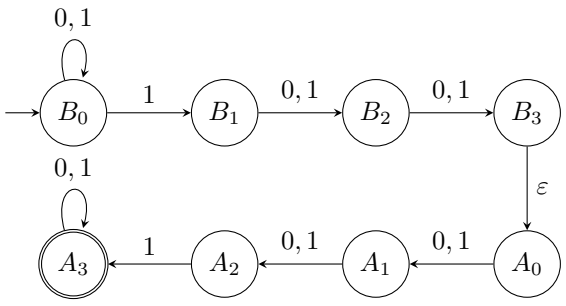
Ein Automat für die Sprache $L_{-3} \cup L_3$ sieht somit folgendermaßen aus:



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.

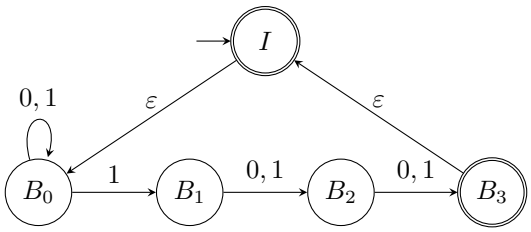
Auch hier können wir die Konstruktion mit den NFAs durchführen und werden dies für die Übersichtlichkeit auch tun. (Der minimale DFA hätte 39 Zustände!)

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:

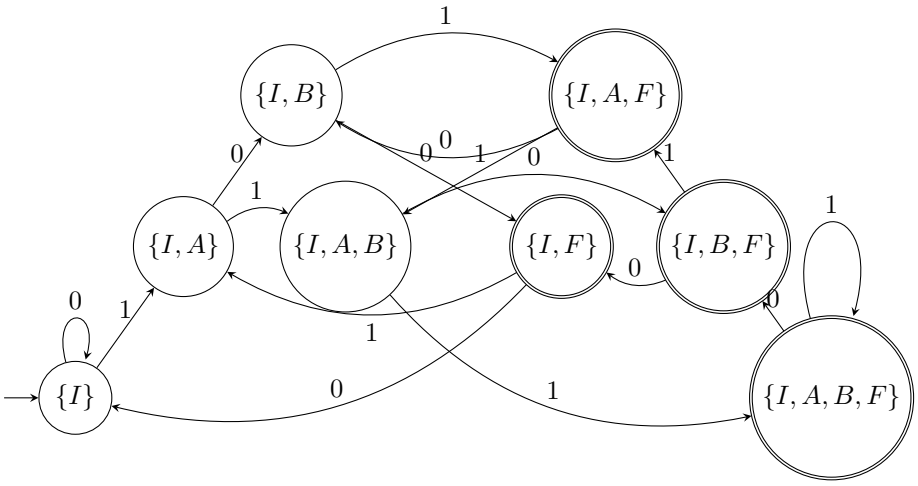
$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

$\overline{L_1}$ 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.

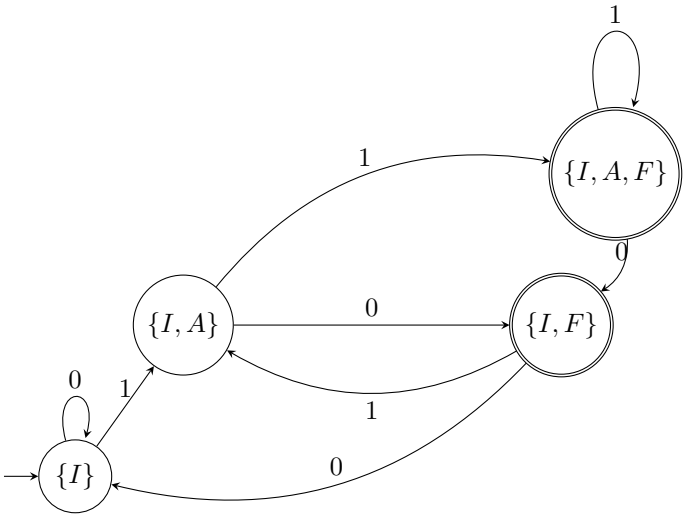
Bilder/Urbilder von Homomorphismen. Wir haben bereits in Kapitel 1.6 gezeigt, dass Bilder regulärer Mengen wieder regulär sind.

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.

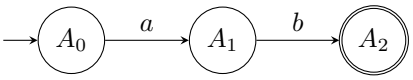
Die erste Richtung haben wir durch unsere Vorüberlegungen bereits erhalten. Wir erinnern uns daran, dass wir im Kontext regulärer Ausdrücke bereits gesehen haben, dass jede Sprache die durch reguläre Ausdrücke beschrieben wird, aus endlichen

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.

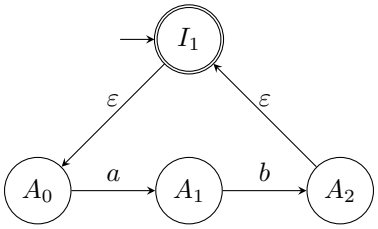
Beispiel. Man nehme den regulären Ausdruck $(ab)^*|ab^*$. Einen Automaten dazu können wir schrittweise aufbauen:

(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.

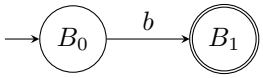
Folgender Automat akzeptiert ab:



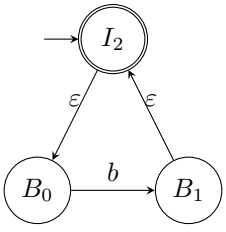
Folgender Automat akzeptiert die Kleensche Hülle davon:



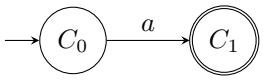
Folgender Automat akzeptiert b:



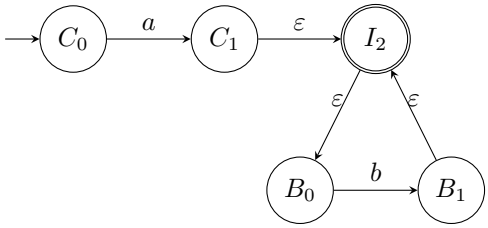
Folgender Automat akzeptiert die Kleensche Hülle davon:



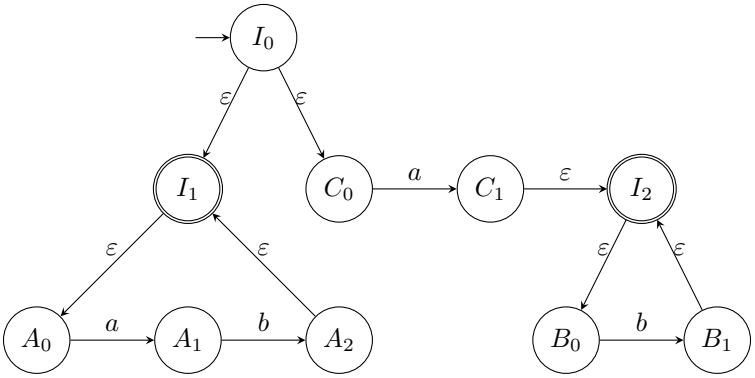
Folgender Automat akzeptiert a:



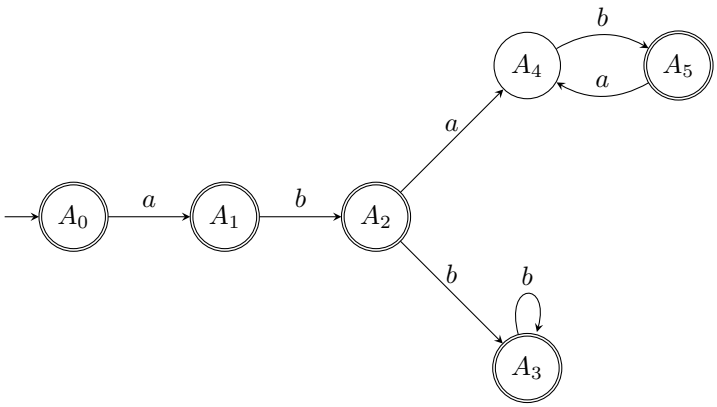
Folgender Automat akzeptiert das Produkt von a und b^*:



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



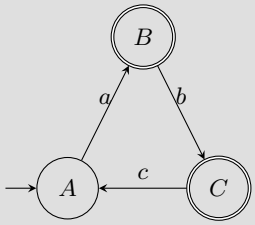
Dieses Verfahren liefert offensichtlich sehr große Automaten. Selbst deterministisch lässt sich ein sehr viel kleinerer Automat angeben:



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.