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 3.2 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 $A\#B$.

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

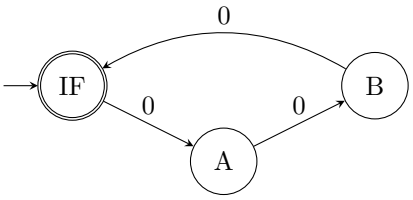
aller Wörter, die von einem bestimmten endlichen Automaten akzeptiert werden, bilden nach vorherigem Abschnitt eine Sprache. Wir bezeichnen diese Sprache als die vom Automaten **akzeptierte Sprache** oder **erkannte Sprache**. Zustände, die zur Ausgabe 1 führen, nennen wir auch **akzeptierende Zustände** 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.

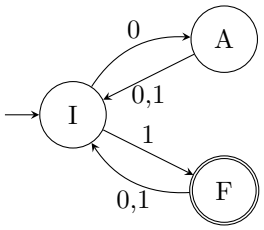
Dies reicht aus, um graphisch jede Funktionsweise des Automaten darzustellen.

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:

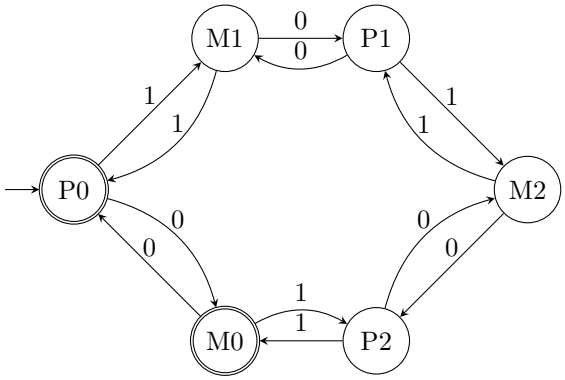


Wenn wir untersuchen wollen, welche Sprache dieser Automat akzeptiert, fällt zunächst auf, dass wir mit jedem gelesenen Zeichen den Zustand I abwechselnd verlassen und wieder betreten. Wurde eine gerade Anzahl an Zeichen gelesen, befinden wir uns in Zustand I - wurde eine ungerade Anzahl an Zeichen gelesen, 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.

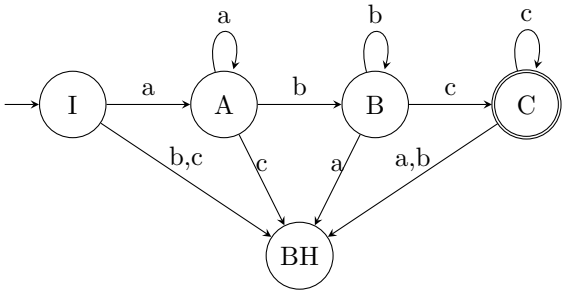
Partielle Automaten. Auch wenn die effektivere Methode der Vereinfachung erst später kommt, können wir auch jetzt schon einen Trick lernen, um manche Automaten zumindest ein bisschen übersichtlicher zu gestalten. Gelegentlich hat ein Automat Zustände, von denen aus kein Endzustand mehr zu erreichen ist.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

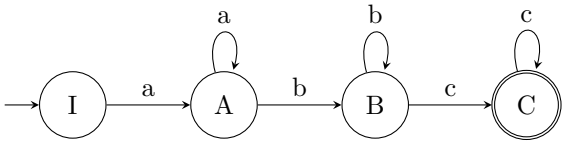
Dies führt dazu, dass es Knoten gibt, von denen aus nicht zu jedem Zeichen eine Kante ausgeht. Befinden wir uns in einem solchen Zustand und erhalten als Eingabe ein Zeichen, zu dem es keine Kante gibt, können wir sofort davon ausgehen, dass die Eingabe nicht akzeptiert wird. (Vor dem Entfernen der unnötigen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

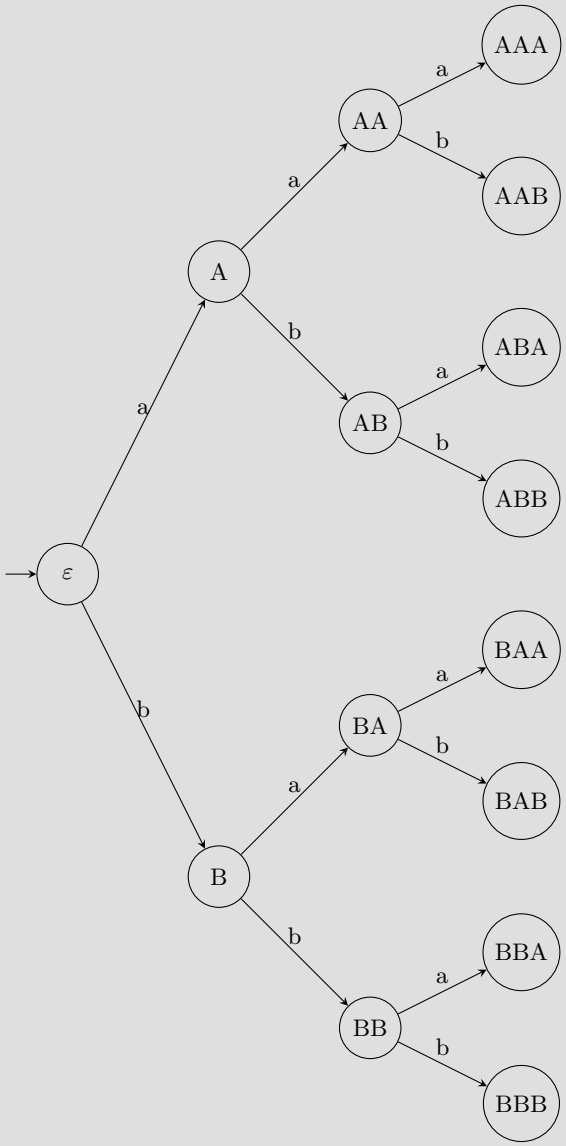


Formale Definition. Bereits durch unsere intuitive Einführung endlicher Automaten wissen wir, was wir formal angeben müssen, um einen Automaten zu definieren. Wir benötigen eine Menge an Zuständen Q. Diese muss keine formalen Bedingungen erfüllen außer, dass sie nur endlich viele Elemente enthält. Es 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.

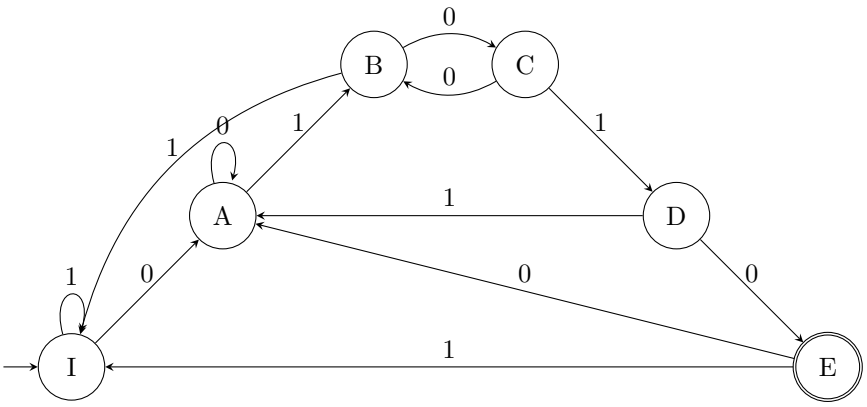
Ein notwendiges Kriterium. Eine Struktur regulärer Mengen fällt auf, wenn wir uns die Graphenstruktur anschauen. Enthält eine Sprache nur endlich viele Wörter, ist die Sprache sowieso regulär. Enthält eine Sprache unendlich viele Wörter, so gibt es insbesondere auch Wörter beliebig langer Länge. Man 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 $|xy| \geq n$ 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 $a^5 b^5$ 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 $a^n b^n$ 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 $a^n b^n$ 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 $a^n b^n$.“ 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 $a^n b^n$ enthält tatsächlich kein Teilwort, das beliebig dupliziert werden kann, ohne das Wort aus der Sprache zu entfernen. Wegen $|a^n b^n| = 2n$ 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.