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 $A\#B$.