

Edmund Weitz

THEORETISCHE INFORMATIK

(Skript für den Studiengang *Medieninformatik* an der *HAW Hamburg*)

27. August 2026, 20:03

Dieses Skript ist momentan noch „work in progress“ und wird regelmäßig aktualisiert. Sie sollten ab und zu nachschauen, ob es eine neue Version (siehe Datum oben) gibt. Den jeweils letzten Stand finden Sie unter der Adresse <https://weitz.de/files/ti-mi-skript.pdf>.

Falls Sie Fehler entdecken (auch „kleine“ Tippfehler), freue ich mich über eine E-Mail an edmund.weitz@haw-hamburg.de. Für bereits entdeckte Fehler bedanke ich mich bei Finn Lunderup.

Der gesamte Text ist unter der Creative-Commons-Lizenz [BY-NC-ND 3.0 DE](https://creativecommons.org/licenses/by-nc-nd/3.0/de/) verfügbar. Eine vereinfachte Zusammenfassung dieser Lizenz findet man unter der folgenden Adresse:
<https://creativecommons.org/licenses/by-nc-nd/3.0/de/>

Inhaltsverzeichnis

Gebrauchsanweisung	1
I Formale Sprachen	5
1 Grundlegende Begriffe	5
2 Grammatiken	11
★3 Die Anzahl der Sprachen	15
II Reguläre Sprachen	17
4 Reguläre Grammatiken	17
5 Deterministische endliche Automaten	20
6 Nichtdeterministische endliche Automaten	25
7 Reguläre Ausdrücke	30
8 Relationen	36
9 Der Satz von Myhill-Nerode	40
III Kontextfreie Sprachen	47
10 Kontextfreie Grammatiken	47
11 Kellerautomaten	53
IV Rekursiv aufzählbare Sprachen	63
12 Turingmaschinen	63
13 Die Church-Turing-These	73
14 Akzeptierende Turingmaschinen und formale Sprachen	81
V Berechenbarkeitstheorie	91
15 Berechenbarkeit und Entscheidbarkeit	91
16 Das Halteproblem und der Satz von Rice	95
17 Sprachen und Entscheidbarkeit	99
VI Komplexitätstheorie	103
18 Polynomiale Laufzeit	103
19 Realistische Probleme	110
20 NP-vollständige Probleme	120
Lösungen zu ausgewählten Aufgaben	127
Literatur	165
Index	167
Mathematische Symbole	171

Gebrauchsanweisung

Ich weiß, dass viele Studierende schriftliche Materialien meiden wie der Teufel das Weihwasser. Ich kann Ihnen jedoch nur dringend raten, sich *regelmäßig* jede Woche nach der Vorlesung mit den jeweiligen Teilen des Skripts zu beschäftigen und es inklusive der Aufgaben durchzuarbeiten. Wenn Sie meinen, Sie kämen auch ohne Lesen aus, dann haben Sie das Konzept des Studierens falsch verstanden und werden sich vielleicht am Ende des Semesters wundern.

Mathematik

In der Theoretischen Informatik werden häufig mathematische Schreibweisen verwendet. Sie ist nämlich – zumindest nach Meinung vieler Mathematikerinnen und Mathematiker – ein Teilgebiet der Mathematik. Da ich davon ausgehe, dass Sie die Mathevorlesungen absolviert haben, werden deren Inhalte hier vorausgesetzt. Insbesondere die *Mengenlehre* wird eine wichtige Rolle spielen. Wenn Ihnen etwas unbekannt vorkommt oder Sie es vergessen haben, dann schauen Sie bitte im Skript (das man unter <https://weitz.de/files/mi-skript.pdf> findet) nach oder in meinem Buch *Konkrete Mathematik (nicht nur) für Informatiker*. Zu Letzterem gibt es einen Link [am Ende des Skripts](#). Bei Verweisen auf dieses Buch werde ich die Abkürzung KMFI verwenden und beziehe mich dabei auf die zweite Auflage.

KMFI

In der Regel werden Begriffe und Konzepte in diesem Skript sowohl mathematisch präzise als auch informell und anhand von Beispielen erklärt. Die informelle Erklärung soll Ihnen natürlich beim Verständnis helfen, aber es ist wichtig, dass Sie auch lernen, mathematische Definitionen zu lesen und zu verstehen. Erstens ist diese Fähigkeit wichtig für das wissenschaftliche Arbeiten (etwa für das Lesen von Fachartikeln) und zweitens gilt im Zweifelsfall (z. B. bei Prüfungen) die mathematische Definition!

Aufgaben

Im Skript finden Sie Aufgaben. Die Bearbeitung der Aufgaben sind freiwillig, aber natürlich sollten Sie sich mit ihnen beschäftigen, damit Sie eine gewisse Routine

im Umgang mit den neuen Begriffen und Konzepten entwickeln. Die Aufgaben sind als *Ergänzung* zur Vorlesung gedacht und sollen beim Verständnis helfen. Selbst dann, wenn einzelne Aufgaben bereits besprochen wurden, kann es nicht schaden, sich noch einmal mit ihnen zu beschäftigen, um sicherzustellen, dass Ihnen auch wirklich alles klar ist.

Im **Anhang** finden Sie Lösungen für die meisten Aufgaben, aber die sollten Sie sich erst anschauen, wenn Sie selbst lange genug über den Aufgaben gegrübelt haben. (Geben Sie nicht zu früh auf!) Wenn Sie einfach nur die Lösungen lesen, lernen Sie nichts. Aber nachdem Sie selbst nachgedacht haben, sollten Sie sich die Lösungen schon anschauen. Vielleicht finden Sie dort eine andere Herangehensweise oder weitere Hinweise.

Wenn Sie sich mit den Aufgaben erst kurz vor den Prüfungen beschäftigen, dann wird es höchstwahrscheinlich zu spät sein. Außerdem sind die Aufgaben ohnehin nicht als „Training“ für die Klausur gedacht und Sie sollten nicht davon ausgehen, dass die Aufgaben in der Klausur lediglich Kopien der Aufgaben aus diesem Skript sind! In der Klausur soll geprüft werden, ob Sie den Stoff verstanden haben. Es wird nicht geprüft, ob Sie etwas, das Ihnen vorgeführt wurde, nachäffen können.

Bonusmaterial

Bei Abschnitten und Aufgaben, die mit einem blauen Stern (★) gekennzeichnet sind, handelt es sich um optionales Material, das man nicht unbedingt braucht, um den Rest des Skripts zu verstehen. Wenn Sie große Angst haben, aus Versehen etwas zu lernen, das Sie für die Prüfung nicht brauchen, dann können Sie es überspringen.¹ Aber natürlich stehen diese Sachen nicht ohne Grund im Text. Sie werden auf jeden Fall nicht dümmer, wenn Sie sich damit beschäftigen, und Ihnen wird kein bleibender Schaden entstehen.

Videos und Links

Die **QR-Codes** in der Randspalte kann man scannen oder im PDF einfach anklicken. Sie führen zu Videos einer Vorlesung, die im Wintersemester 2023/2024 aufgenommen wurden. Beachten Sie jedoch, dass die Inhalte der Videos nicht immer exakt mit den Inhalten in diesem Skript übereinstimmen! (Siehe dazu den folgenden Abschnitt.) Eine sortierte Liste aller Videos findet man unter der URL <https://weitz.de/haw-videos/>. Text in **blauer Schrift** ist ebenfalls anklickbar und verlinkt zu anderen Teilen des Skripts oder auf weiterführende Videos und Wikipedia-Artikel.

„Altes“ Skript

Unter der URL <https://weitz.de/files/ti-skript.pdf> finden Sie eine Version dieses Skripts, die bis zum Wintersemester 2025/2026 für das Modul *Mathematische Methoden der Informatik* des Studiengangs *Media Systems* verwendet

¹Allerdings sollten Sie sich dann fragen, warum Sie überhaupt studieren ...

wurde. Es werden im Prinzip dieselben Themen behandelt; allerdings gibt es unterschiedliche Schwerpunkte und Herangehensweisen. Im Zweifelsfall (z. B. bei Prüfungen) ist natürlich entscheidend, was in *diesem* Skript steht!



Formale Sprachen

In der **Informatik** beschäftigt man sich vereinfacht ausgedrückt mit Dingen, die man mit Computern (also: „Rechnern“) machen kann. Die **Theoretische Informatik** entwickelte sich Anfang des 20. Jahrhunderts aus Fragestellungen der **mathematischen Logik**, bevor es überhaupt Computer gab. In ihr geht es darum, ganz fundamentale Fragen wie „Was kann man überhaupt berechnen?“ oder „Wie aufwendig ist ein bestimmter Rechengang?“ zu beantworten.



Um definitive Antworten zu ermöglichen, muss man die Fragen präzise formulieren. Ein Großteil der Arbeit in dieser Veranstaltung wird daraus bestehen, die dafür notwendigen Begriffe und Konzepte zu verstehen. Wie man sich angesichts der Überschrift schon denken kann, wird es gerade am Anfang etwas formal zugehen und es geht auch gleich mit einem ganzen Haufen Definitionen los. Sie werden jedoch hoffentlich schnell merken, dass das alles nicht so kompliziert ist, wie es zu sein scheint.

1. Grundlegende Begriffe

Jede nichtleere endliche Menge Σ kann als ein *Alphabet* betrachtet werden. In diesem Zusammenhang nennt man die Elemente von Σ auch *Symbole* oder *Zeichen*.

Alphabet
Symbol Zeichen

Ein Beispiel für ein Alphabet ist die Menge $\Sigma_{\text{bool}} = \{0, 1\}$, die aus den beiden Symbolen 0 und 1 besteht.¹ Zu beachten ist dabei, dass wir in diesem Zusammenhang 0 und 1 nicht als *Zahlen* interpretieren, sondern lediglich als (zunächst bedeutungslose) *Zeichen*. Wir könnten stattdessen auch einen Kreis und einen Strich nehmen. Ein weiteres Beispiel ist die Menge $\Sigma_{\text{lat}} = \{a, b, c, \dots, z\}$, die aus den 26 lateinischen Buchstaben a bis z besteht und die wohl am ehesten der klassischen Vorstellung eines Alphabets entspricht.

Σ_{bool}

Σ_{lat}

Aufgabe 1.1. Warum ist die Menge $\mathbb{N}$ der natürlichen Zahlen² kein Alphabet?

¹Die Menge benennen wir nach dem englischen Mathematiker **George Boole**, dessen Name in so ziemlich jeder Programmiersprache in der einen oder anderen Form vorkommt.

²In diesem Skript gilt $0 \in \mathbb{N}$. Für $\mathbb{N} \setminus \{0\}$ schreiben wir $\mathbb{N}^+$.

Wort	Jedes Tupel , dessen Komponenten alle Elemente eines Alphabets Σ sind, nennt man ein Wort über Σ . Beispielsweise sind (0) , $(0, 1, 1, 0)$ oder $(1, 1)$ Wörter über Σ_{bool} und (z, a, p, p, a) oder (x, x) sind Wörter über Σ_{lat} . Wenn keine Gefahr von Missverständnissen besteht, lassen wir die Klammern und Kommata in der Regel weg. Für die genannten Beispiele werden wir in Zukunft also einfach 0 , 0110 , 11 , zappa bzw. xx schreiben. ³ Es sollte damit auch klar geworden sein, dass Wörter im Sinne der Theoretischen Informatik – anders als im täglichen Sprachgebrauch – einfach nur Zeichenketten sind, die keine spezifische Bedeutung haben müssen.
leeres Wort ε	Zu den Wörtern über einem Alphabet Σ gehört immer auch das 0-Tupel $()$. Das bezeichnet man als das <i>leere Wort</i> und schreibt dafür ε . ⁴ Beachten Sie, dass ε in diesem Sinne <i>kein</i> Symbol ist! (Und sinnvollerweise verwenden wir auch keine Alphabete, in denen ein Symbol ε vorkommt ...)
Länge $ w $	Ist ein Wort w ein n -Tupel, so bezeichnen wir n als seine <i>Länge</i> und schreiben dafür $ w $. Die Länge von zappa ist also 5 und wir würden $ \text{zappa} = 5$ schreiben. ⁵ Das leere Wort ist das einzige Wort, dessen Länge null ist. Außerdem soll $ w _x$ zählen, wie oft das Symbol x im Wort w vorkommt, z. B. $ 00101 _0 = 3$ und $ 00101 _1 = 2$.

Aufgabe 1.2. Wie viele Wörter der Länge 1 über einem Alphabet Σ gibt es?

Definition

Für zwei Wörter $v = (v_1, \dots, v_m)$ und $w = (w_1, \dots, w_n)$ ist ihre **Konkatenation**⁶ als das Wort

$$v \circ w = (v_1, \dots, v_m, w_1, \dots, w_n)$$

definiert. Statt $v \circ w$ schreibt man meistens einfach vw .

Ferner definieren wir w^k für $k \in \mathbb{N}$ **rekursiv** durch $w^0 = \varepsilon$ und $w^{k+1} = w^k \circ w$.

Diese Definition verwenden wir sinngemäß auch für Symbole. Wenn wir $0 \circ 0$ schreiben, dann ist es also egal, ob das Symbol 0 gemeint war oder das Wort (0) , das wir abkürzend als 0 geschrieben haben. (Siehe auch die Anmerkung in der Lösung zu Aufgabe 1.2.)

Wir verwenden bei der Konkatenation Klammern mit der üblichen Bedeutung, dass geklammerte Ausdrücke zuerst evaluiert werden. Offensichtlich ist $\circ$ **assoziativ**, d. h., es gilt immer $(u \circ v) \circ w = u \circ (v \circ w)$. Daher können wir auch einfach

³Diese Konvention ergibt nur dann Sinn, wenn man Symbole und Wörter unterscheiden kann. Weil ein Alphabet nach unserer Definition eine *beliebige* Menge sein kann, könnte es beispielsweise aus den beiden Symbolen 1 und 11 bestehen. Dann wären $(1, 11)$ und $(11, 1)$ zwei *verschiedene* Wörter über diesem Alphabet, die man in der Schreibweise 111 nicht unterscheiden könnte. Wir werden jedoch immer mit *einstelligen* Symbolen arbeiten, bei denen so etwas nicht passieren kann.

⁴In manchen Büchern wird es auch mit λ bezeichnet.

⁵Die Länge ist also *nicht* 3, denn es ist nicht die **Mächtigkeit** der Menge $\{z, a, p, p, a\}$ gemeint, obwohl wir nach wie vor auch die Bezeichnung $|A|$ für die Mächtigkeit einer Menge verwenden. Es wird jeweils aus dem Kontext eindeutig hervorgehen, was gemeint ist. Solche Mehrfachverwendung derselben Schreibweise sollte Ihnen nach ein paar Semestern Mathematik kein Kopfzerbrechen mehr bereiten.

⁶Ein Fremdwort für *Aneinanderhängen* bzw. *Verketten*.

$u \circ v \circ w$ schreiben. Die Exponentiation soll jedoch eine höhere Priorität als die Konkatenation haben, so dass sowohl $v \circ w^k$ als auch vw^k Kurzformen von $v \circ (w^k)$ sind und etwas anderes als $(vw)^k$ bedeuten.⁷

Zwischendurch eine Anmerkung zur Typographie. Da man leicht durcheinanderkommen kann, wird in diesem Skript wenn möglich die folgende Unterscheidung vorgenommen: Variablen, die für Symbole oder Wörter stehen, werden wie üblich **kursiv** gesetzt, also z. B. wie a oder w . Für konkrete Symbole wird eine **Schreibmaschinenschrift** verwendet, so dass es dann wie a oder w aussieht.

Aufgabe 1.3. Sei Σ das Alphabet $\{a, b\}$ und v das Wort ab über Σ . Geben Sie zur Übung a^4 , v^3 , $a^2 v^2$, ab^3 , $(ab)^3$ und $|v^{10}|$ an.

Aufgabe 1.4. Ist die Verknüpfung $\circ$ **kommutativ**? Gibt es ein **neutrales Element**?

Aufgabe 1.5. Was würde sich ändern, wenn man oben w^{k+1} durch $w \circ w^k$ definieren würde?

Sind X und Y Mengen von Zeichen oder Wörtern, dann definieren wir die **Konkatenation** von X und Y als die Menge

$$X \circ Y = \{vw : v \in X \text{ und } w \in Y\},$$

für die wir manchmal auch einfach XY schreiben. Ferner definieren wir X^k für $k \in \mathbb{N}$ **rekursiv** durch $X^0 = \{\varepsilon\}$ und $X^{k+1} = X^k \circ X$.

Definition

Beachten Sie, dass $\{\varepsilon\}$ *nicht* die leere Menge $\emptyset$ ist, sondern eine Menge mit einem Element, nämlich dem leeren Wort.⁸ Außerdem ist es potentiell verwirrend, dass die Schreibweise X^k in der Mathematik auch für das **Mengenprodukt** verwendet wird. Daran kann ich leider nichts ändern. Sie müssen sich darauf verlassen, dass Sie (hoffentlich) jeweils aus dem Kontext schließen können, was gemeint ist.

Aufgabe 1.6. Sei $X = \{ab, ac\}$ und $Y = \{bc, d, e\}$. Geben Sie $X \circ Y$ an. Überzeugen Sie sich, dass $X \circ Y \neq Y \circ X$ gilt.

Aufgabe 1.7. Sei $X = \{aa, b\}$, $Y = \emptyset$ und $Z = \{\varepsilon\}$. Geben Sie $X \circ Y$ und $X \circ Z$ an.

Aufgabe 1.8. Sei $\Sigma = \{a, b\}$. Geben Sie Σ^3 an.

Aufgabe 1.9. Wie kann man die Menge Σ^k mit Worten⁹ beschreiben, wenn Σ ein Alphabet ist?

⁷Das ist dieselbe Regel wie bei der normalen Multiplikation.

⁸Rein technisch könnte man aus Sicht der Mengenlehre übrigens einwenden, dass gemäß der üblichen Definition ε (also das 0-Tupel) und $\emptyset$ identisch sind (jedoch *nicht* $\{\varepsilon\}$ und $\emptyset$). Das sollte jedoch hoffentlich nicht zu verwirrend sein. Schreibt man $\emptyset$, dann *meint* man eine Menge von null Wörtern, schreibt man ε , dann *meint* man ein Wort der Länge null.

⁹Damit ist gemeint, wie Sie es ausdrücken würden, wenn Sie es jemandem erklären. Zum Glück kann man im Deutschen zwischen *Worten* und *Wörtern* unterscheiden. Den Plural *Worte* verwendet man, wenn es um den Inhalt bzw. die Bedeutung geht. Die mathematischen Objekte, über die wir in diesem Kapitel sprechen, sind hingegen *Wörter*.

Aufgabe 1.10. Wie viele Elemente enthält Σ^k , wenn Σ ein Alphabet ist?

Aufgabe 1.11. Wie kann man X^1 kürzer schreiben?

Aufgabe 1.12. Finden Sie zwei endliche Mengen X und Y von Wörtern mit der Eigenschaft $|X \circ Y| \neq |X| \cdot |Y|$. Welche der beiden Mächtigkeiten ist garantiert nie kleiner als die andere?

Definition

Für eine Menge X von Zeichen oder Wörtern sind die **kleenesche**¹⁰ und die **positive Hülle** von X definiert als

$$X^* = \{w : w \in X^n \text{ für ein } n \in \mathbb{N}\} \quad \text{und} \\ X^+ = \{w : w \in X^n \text{ für ein } n \in \mathbb{N}^+\}.$$

Falls Ihnen die Schreibweisen vertraut sind: Man hätte es auch als

$$X^* = \bigcup_{n \in \mathbb{N}} X^n \quad \text{und} \quad X^+ = \bigcup_{n \in \mathbb{N}^+} X^n$$

schreiben können.

Aufgabe 1.13. Wie kann man die Menge Σ^* mit Worten beschreiben, wenn Σ ein Alphabet ist?

Aufgabe 1.14. Sei $X = \{a\}$ und $Y = X \cup \{\varepsilon\}$. Wie sehen X^k und Y^k für $k \in \mathbb{N}$ aus und wie X^* und Y^* ?

Aufgabe 1.15. Aus Aufgabe 1.9 folgt, dass $X^+ = X^* \setminus \{\varepsilon\}$ gilt, wenn X ein Alphabet ist. Geben Sie eine Menge X von Wörtern an, für die diese Gleichung *nicht* gilt.

Aufgabe 1.16. Sei w das Wort 100 über dem Alphabet Σ_{bool} . Was ist dann w^* ?

Aufgabe 1.17. Sei Σ ein Alphabet. Wie lang ist das längste Wort, das in der kleeneschen Hülle Σ^* enthalten ist?

Definition

Ist Σ ein Alphabet, so wird jede Teilmenge L von Σ^* als eine **(formale) Sprache** über Σ bezeichnet.

Genau wie mit einem *Wort* nicht das gemeint ist, was wir im Alltag damit meinen, ist auch eine *Sprache* in der Theoretischen Informatik zunächst ein rein abstraktes Objekt ohne Bedeutung, nämlich einfach eine Menge von Wörtern. Nach dieser Definition sind beispielsweise die leere Menge und Σ^* Sprachen über Σ , $\{0, 1, 00\}$ ist eine Sprache über Σ_{bool} und die Menge aller Wörter, in denen mindestens zweimal der Buchstabe a vorkommt, ist eine Sprache über Σ_{lat} .

¹⁰Benannt nach dem amerikanischen Mathematiker [Stephen Cole Kleene](#). Übrigens habe ich, wie ich später erfahren habe, den Namen offenbar in allen Videos falsch ausgesprochen. Er stammt aus dem Friesischen und sollte wie „Klehni“ klingen.

Aufgabe 1.18. Begründen Sie, warum eine unendliche Sprache für jede Zahl $k \in \mathbb{N}$ Wörter enthalten muss, deren Länge größer als k ist.

Aufgabe 1.19. In der folgenden Liste werden fünf verschiedene Sprachen über Σ_{bool} jeweils auf zwei Arten dargestellt, einmal links mittels Konkatenation und Hüllen und einmal rechts durch die beschreibende Mengenschreibweise.

$$\begin{array}{ll} A_1 = \{00\}^* & B_1 = \{w \in \Sigma_{\text{bool}}^+ : |w|_0 \cdot |w|_1 = 0\} \\ A_2 = \{0\} \circ \{00\}^* & B_2 = \{w \in \Sigma_{\text{bool}}^* : |w| \geq 2 \wedge |w|_1 = 0\} \\ A_3 = \{0\}^+ \circ \{0\}^+ & B_3 = \{w \in \Sigma_{\text{bool}}^* : |w|_0 = |w| \wedge |w| \bmod 2 = 0\} \\ A_4 = \{0\}^* \setminus \{1\}^* & B_4 = \{0^n : n \in \mathbb{N}^+\} \\ A_5 = \{0\}^+ \cup \{1\}^+ & B_5 = \{0^{2n+1} : n \in \mathbb{N}\} \end{array}$$

Finden Sie die Paare, die dieselbe Sprache beschreiben.

Aufgabe 1.20. Schreiben Sie die Σ_{bool} -Sprachen

$$A_1 = \{000\}^* \circ \{00\}, \quad A_2 = \{\varepsilon, 0, 1\}^3, \quad A_3 = \Sigma_{\text{bool}}^3 \quad \text{und} \quad A_4 = \{0\}^+ \circ \{1\}^+$$

in korrekter Mengenschreibweise so wie die Mengen auf der rechten Seite von Aufgabe 1.19 auf.

Aufgabe 1.21. Je zwei der folgenden acht Sprachen sind identisch. Finden Sie die entsprechenden Paare.

$$\begin{array}{ll} L_1 = \{10\} \circ \{0\}^* & L_2 = \{w \in \Sigma^* : |w|_0 = 3\} \\ L_3 = \{01\}^* & L_4 = \{0, 1\}^0 \cup \{0, 1\}^1 \cup \{0, 1\}^2 \\ L_5 = \{(01)^n : n \in \mathbb{N}\} & L_6 = \{w \in \Sigma^* : |w| < 3\} \\ L_7 = \{1\} \circ \{0\}^+ & L_8 = \{w \in \Sigma^* : |w|_1 = |w| - 1\} \circ \{w \in \Sigma^* : |w|_0 = 2\} \end{array}$$

Dabei sei immer $\Sigma = \Sigma_{\text{bool}}$.

Aufgabe 1.22. In Übungen sehe ich manchmal Ausdrücke wie $\{\{0\}^+ \circ \{1\}^+\}^3$. Was ist hier falsch? Überlegen Sie sich außerdem, was wohl gemeint war und wie man es richtig schreiben müsste.

Aufgabe 1.23. Die folgenden Aufgaben stammen aus alten Klausuren und eignen sich nach meiner Meinung gut, um die Begriffe aus diesem Abschnitt zu üben. (Das gilt allerdings wie üblich nur dann, wenn Sie selbst nachdenken und geduldig herumprobieren. Es bringt wenig, sich einfach nur die Lösungen anzuschauen, weil es für solche Aufgaben kein Patentrezept gibt, das man einfach nachäffen kann.)

Es geht immer um Sprachen über Σ_{bool} . Die Lösungen sollen in korrekter Schreibweise ohne **Auslassungspunkte** angegeben werden!

- (i) Geben Sie zwei *verschiedene* unendliche Sprachen L_1 und L_2 an, so dass $L_1 \circ L_2$ und $L_2 \circ L_1$ gleich sind.
- (ii) Geben Sie zwei endliche Sprachen L_1 und L_2 an, für die $L_1^+ \subseteq L_2^+$, aber *nicht* $L_1 \subseteq L_2$ gilt.
- (iii) Geben Sie zwei *verschiedene* Sprachen L_1 und L_2 an, so dass $L_1 \circ L_2 = L_1^*$ und $L_1^* \neq \Sigma_{\text{bool}}^*$ gilt.

- (iv) Geben Sie zwei unendliche Sprachen L_1 und L_2 an, so dass L_1 und L_2 disjunkt sind und $L_1 \circ L_2 = L_2 \circ L_1$ gilt.
- (v) Geben Sie drei Sprachen L_1 , L_2 und L_3 an, für die $L_1 \circ (L_2 \cap L_3) = (L_1 \circ L_2) \cap (L_1 \circ L_3)$ *nicht* gilt. [Hinweis: Denken Sie nicht an komplizierte unendliche Sprachen. Man kommt mit insgesamt vier Wörtern aus.]

Wir haben damit bereits die wichtigsten Begriffe beisammen, um – wie am Anfang des Kapitels besprochen – sehr komplizierte Fragen unmissverständlich formulieren zu können. Dafür zunächst nur zwei Beispiele, für die wir vorab n_w für $w \in \Sigma_{\text{bool}}^+$ als die Zahl definieren, deren [Binärdarstellung](#) w ist, also z. B. $n_{101010} = 42$.

Im ersten Beispiel geht es um die Sprache

$$\{w \in \Sigma_{\text{bool}}^+ : \{n_{1w}, n_{1w} + 2\} \subseteq \mathbb{P}\}. \quad (1.1)$$

Die simple Frage, ob diese Sprache endlich oder unendlich ist, konnte bis heute (Stand Juli 2026) [niemand beantworten](#). Die Frage lässt sich aber, wie man sieht, in einer Zeile hinschreiben.

Bei der zweiten Frage ist die Definition der Sprache ein bisschen aufwendiger. Ihr Alphabet besteht aus Σ_{bool} zusammen mit den Zeichen $\wedge$ und $\vee$ sowie den Klammern (und). Wir definieren

$$\begin{aligned} B_1 &= \{1\} \circ \Sigma_{\text{bool}}^*, & B_0 &= \{0\} \circ B_1, & B &= B_0 \cup B_1, \\ D_0 &= B \cup \{(w_1 \vee w_2) : w_1, w_2 \in B\} \cup \{(w_1 \vee w_2 \vee w_3) : w_1, w_2, w_3 \in B\}, \\ D_{k+1} &= D_k \circ \{\wedge\} \circ D_0 && \text{für } k \in \mathbb{N} \text{ und schließlich} \\ D &= \bigcup_{k=0}^{\infty} D_k = \{w : w \in D_k \text{ für ein } k \in \mathbb{N}\}. \end{aligned} \quad (1.2)$$

Interpretiert man ein Wort w aus B_1 als den Ausdruck x_{n_w} und ein Wort $0w$ aus B_0 als $\neg x_{n_w}$, dann sind die Elemente von D bestimmte Formeln der [Aussagenlogik](#). Die Beschäftigung mit Algorithmen, die für jede dieser Formeln ermitteln können, ob sie [erfüllbar](#) sind, hängt mit [einem der wichtigsten offenen Probleme der Informatik](#) zusammen. Mehr dazu in Kapitel VI.

Aufgabe 1.24. Warum steht in (1.1) n_{1w} und nicht einfach n_w ?

Aufgabe 1.25. Begründen Sie, dass das Wort $01 \wedge 010 \wedge (1 \vee 10)$ zur Menge D aus der Definition (1.2) gehört. Welche aussagenlogische Formel wird dadurch repräsentiert? Bonusfrage: Ist diese Formel erfüllbar?

Aufgabe 1.26. Zum Üben einiger Grundbegriffe aus diesem Kapitel eignet sich die Lernsoftware [FLACI](#),¹¹ die wir auch in späteren Kapiteln einsetzen werden. Wenn Sie sich noch unsicher im Umgang mit Begriffen wie Alphabet, Wort oder Sprache fühlen, dann schauen Sie sich den Abschnitt *Formale Sprachen* von FLACI an.

¹¹ Siehe dazu auch das Buch von Wagenknecht und Hielscher [in der Literaturliste](#).

★**Aufgabe 1.27.** Schreiben Sie für jede der folgenden Sprachen L ein Programm in einer Programmiersprache Ihrer Wahl, das entscheiden¹² kann, ob ein Wort über dem Alphabet Σ zu L gehört.

$\Sigma = \Sigma_{\text{bool}}$	$L = \{0^n 1^{2n} 0^n : n \in \mathbb{N}\}$
$\Sigma = \Sigma_{\text{lat}}$	$L = \{w \in \Sigma^+ : w \text{ ist ein Palindrom}\}$
$\Sigma = \{0, 1, 2\}$	$L = \{w \in \Sigma^* : w _0 \cdot w _1 \text{ ist ungerade}\}$
$\Sigma = \{0, 1, 2, \dots, 9, X\}$	$L = \{w \in \Sigma^* : w \text{ ist eine zehnstellige ISBN}\}$

Zum Thema **ISBN** siehe Kapitel 6 in **KMFI**.

2. Grammatiken

Von dem amerikanischen Linguisten **Noam Chomsky** stammt die Idee, Sprachen durch formale Grammatiken zu beschreiben. Seine erste Veröffentlichung dazu stammt aus dem Jahr 1956. Chomsky ging es eigentlich um natürliche Sprachen, aber seine Theorie wird heutzutage hauptsächlich in der Informatik eingesetzt.



Eine **(formale) Grammatik** ist ein 4-Tupel $G = (N, T, P, S)$ mit den folgenden Eigenschaften:

- (i) N ist ein Alphabet, dessen Elemente man **Nichtterminalsymbole** nennt.
- (ii) T ist ein Alphabet, dessen Elemente man **Terminalsymbole** nennt.
- (iii) N und T sind disjunkt.
- (iv) P ist eine endliche Teilmenge von $((N \cup T)^* \setminus T^*) \times (N \cup T)^*$, deren Elemente man **Produktionen** nennt.
- (v) S ist ein Element von N , das man **Startsymbol** nennt.

Die Menge $N \cup T$ bezeichnet man auch als **Vokabular** von G .

Definition

Ein Beispiel für eine Grammatik ist $G = (\{X, Y\}, \{a, b\}, P, X)$ mit

$$P = \{(X, \varepsilon), (X, a), (Xa, bY), (Y, Xa)\}.$$

Aufgabe 2.1. Wieso wäre das kein Beispiel für eine Grammatik mehr, wenn man in P das Paar (Xa, bY) durch (a, bY) ersetzen würde?

Aufgabe 2.2. Noch eine Verständnisfrage: Kann die erste Komponente einer Produktion das leere Wort sein?

Wir verwenden im Zusammenhang mit Grammatiken immer kursive Großbuchstaben für Nichtterminalsymbole und Kleinbuchstaben (sowie evtl. Ziffern und Sonderzeichen) in Schreibmaschinenschrift für Terminalsymbole. ►

Konvention

¹²Damit ist gemeint, dass das Programm als Eingabe ein Wort w über Σ erhält und mit *ja* bzw. *nein* (oder *True* bzw. *False*) antwortet, wenn $w \in L$ bzw. $w \notin L$ gilt.

Ferner schreiben wir Produktionen (α, β) in der Form $\alpha \rightarrow \beta$ auf und fassen mehrere Produktionen $\alpha \rightarrow \beta_1$ bis $\alpha \rightarrow \beta_n$ mit derselben linken Seite abkürzend als $\alpha \rightarrow \beta_1 \mid \dots \mid \beta_n$ zusammen. Die Mengen der Terminal- bzw. Nichtterminalsymbole werden sich implizit durch das Aufschreiben der Produktionen ergeben.¹³ Wir legen außerdem fest, dass S das Startsymbol ist, wenn nicht explizit ein anderes Zeichen als Startsymbol deklariert wird.

Die Beispielgrammatik G von oben würden wir also in der Form

$$\begin{aligned} X &\rightarrow \varepsilon \mid a \\ Xa &\rightarrow bY \\ Y &\rightarrow Xa \end{aligned} \tag{2.1}$$

aufschreiben und dann noch erwähnen, dass das Startsymbol in diesem Fall X ist.

Wofür Grammatiken verwendet werden, beschreibt die nächste Definition.

Definition

Sei $G = (N, T, P, S)$ eine Grammatik. Für $\alpha, \beta \in (N \cup T)^*$ schreiben wir $\alpha \Rightarrow_G \beta$, wenn es $\gamma, \delta, \varphi, \chi \in (N \cup T)^*$ mit $\alpha = \varphi\gamma\chi$, $\beta = \varphi\delta\chi$ und $(\gamma, \delta) \in P$ gibt. Ferner schreiben wir $\alpha \Rightarrow_G^* \beta$, wenn es $\gamma_1, \dots, \gamma_n \in (N \cup T)^*$ mit $\gamma_1 = \alpha$, $\gamma_n = \beta$ und $\gamma_i \Rightarrow_G \gamma_{i+1}$ für $i = 1, \dots, n-1$ gibt.¹⁴ Die von G **erzeugte Sprache** wird durch

$$L(G) = \{ w \in T^* : S \Rightarrow_G^* w \}$$

definiert.

Weil diese Definition sicher etwas gewöhnungsbedürftig ist, zerlegen wir sie in ihre Bestandteile. $\alpha \Rightarrow_G \beta$ bedeutet, dass β aus α dadurch entstanden ist, dass ein Teilwort γ von α durch δ ersetzt wurde und dass $\gamma \rightarrow \delta$ eine der Produktionen ist. Im Beispiel (2.1) würde unter anderem $bYb \Rightarrow_G bXab$ gelten, weil wir die Produktion $Y \rightarrow Xa$ haben. Und auch $bXab \Rightarrow_G bbYb$ wegen der Produktion $Xa \rightarrow bY$. Der Sinn von Produktionen ist also, dass man ihre linke Seite durch ihre rechte ersetzen kann. $\Rightarrow_G^*$ steht dafür, dass eine oder mehrere solcher Ersetzungen stattgefunden haben oder dass sich gar nichts geändert hat. In unserem Beispiel haben wir uns gerade überlegt, dass $bYb \Rightarrow_G^* bbYb$ gilt, weil wir in zwei Schritten vom einen Wort zum anderen kommen. Wir sagen in so einem Fall, dass $bbYb$ in G aus bYb *hergeleitet* werden kann. $L(G)$ ist somit die Menge aller Wörter, die nur aus Terminalsymbolen bestehen und die man aus dem Startsymbol herleiten kann.

herleiten

Für ein etwas interessanteres Beispiel schauen wir uns die Grammatik einer sogenannten *Dyck-Sprache*¹⁵ an, auf die wir später auch wieder zurückkommen

Dyck-Sprache

¹³ N und T sollen also nur aus Zeichen bestehen, die auch in Produktionen vorkommen.

¹⁴Wir setzen dabei $n \geq 1$ voraus, erlauben also auch $\alpha = \beta$ ohne Anwendung einer Produktion.

¹⁵Benannt nach dem Mathematiker [Walther von Dyck](#).

werden. Die Terminalsymbole sind hier die beiden Klammerzeichen [und], das einzige Nichtterminalsymbol ist S und es gibt drei Produktionen:

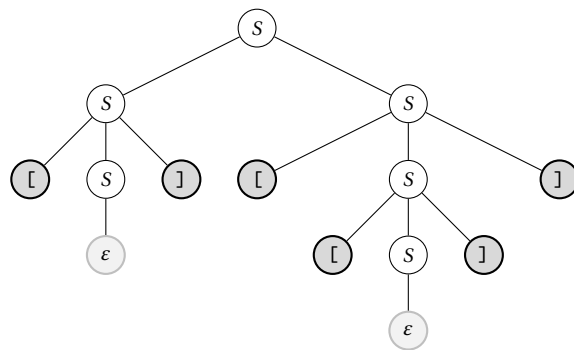
$$S \rightarrow \varepsilon \mid SS \mid [S] \quad (2.2)$$

Ein Wort, das man herleiten kann, ist [] [[]], und zwar z. B. folgendermaßen:

$$S \Rightarrow_G SS \Rightarrow_G [S] S \Rightarrow_G [] S \Rightarrow_G [] [S] \Rightarrow_G [] [[S]] \Rightarrow_G [] [[]] \quad (2.3)$$

Wenn die Grammatik nicht zu kompliziert ist, kann man so einen Prozess mit einem *Ableitungsbaum* illustrieren. So etwas kennen Sie vielleicht aus den Informatik-Vorlesungen. Es sollte auf jeden Fall größtenteils selbsterklärend sein. Nach den Regeln für Grammatiken muss die Wurzel das Startsymbol sein. Die Kinder eines Knotens entstehen immer durch eine der Produktionen und die Blätter des Baums sind Terminalsymbole (oder ε).

Ableitungsbaum



Eine Grammatik, die einen ersten Eindruck davon vermittelt, wie man so etwas konkret in der Informatik einsetzen könnte, sieht folgendermaßen aus:¹⁶

$$\begin{aligned} S &\rightarrow V=E \\ E &\rightarrow V \mid E+E \mid E * E \mid (E) \\ V &\rightarrow x \mid y \mid z \end{aligned} \quad (2.4)$$

Damit kann man arithmetische Zuweisungen für eine fiktive Programmiersprache generieren, die nur drei Variablennamen kennt und nur addieren und multiplizieren kann, z. B. $x=y$, $z=x+y*x$ oder $y=(x+x)*z$.

Aufgabe 2.3. Wie sieht im Beispiel (2.1) die Sprache $L(G)$ aus?

Aufgabe 2.4. Geben Sie eine Grammatik an, die die Sprache $\emptyset$ erzeugt.

Aufgabe 2.5. Geben Sie für das Wort [] [[]] eine andere Herleitung als (2.3) an. Ergibt sich dadurch auch ein anderer Ableitungsbaum?

¹⁶Beachten Sie, dass nach unserer Konvention die drei Zeichen =, + und * sowie die beiden Klammern terminale Symbole sind. V soll hier für Variablen stehen und E für arithmetische Ausdrücke (engl. *expressions*).

D_1

Aufgabe 2.6. Die durch die Grammatik (2.2) erzeugte Sprache wird D_1 genannt. Sie erzeugt alle Wörter, in denen „korrekt geklammert“ wird. Versuchen Sie mit Ihren eigenen Worten zu beschreiben, wie das gemeint ist.

Aufgabe 2.7. Geben Sie ein Wort aus der Dyck-Sprache D_1 an, für das es mehr als einen Ableitungsbaum gibt.

Aufgabe 2.8. Geben Sie eine Grammatik an, die die Sprache $\{a\}^*$ erzeugt.

Aufgabe 2.9. Geben Sie die Sprache an, die von dieser Grammatik erzeugt wird:

$$S \rightarrow 0S \mid 1S \mid 0A$$

$$A \rightarrow 1B$$

$$B \rightarrow 0C$$

$$C \rightarrow \varepsilon$$

Aufgabe 2.10. Erzeugt die Grammatik $S \rightarrow \varepsilon \mid abS$ die Sprache $\{a, b\}^*$?

Aufgabe 2.11. Geben Sie für die Grammatik (2.4) möglichst viele verschiedene Herleitungen für das Wort $x=y+(z*x)$ an.

Aufgabe 2.12. Mit der Lernsoftware [FLACI](#) (siehe Aufgabe 1.26) kann man auch mit Grammatiken rumexperimentieren. Ich empfehle Ihnen, das auch zu tun, um eine gewisse Routine zu entwickeln.¹⁷ Eine Alternative zu FLACI ist das Programm [JFLAP](#), das FLACI nach meiner Meinung in einigen Bereichen sogar überlegen ist. Man kann es allerdings nicht bequem im Browser verwenden. FLACI wird in diesem Skript noch öfter erwähnt werden. Sie können stattdessen immer auch JFLAP verwenden.

Aufgabe 2.13. Geben Sie Grammatiken für die folgenden Sprachen an:

(i) $\{1, 0\} \circ \{1010\}^*$

(ii) $\{w \in \Sigma_{\text{bool}}^* : |w|_1 \equiv 0 \pmod{3}\}$

(iii) $\{w \in \Sigma_{\text{bool}}^* : |w|_0 = 2|w|_1\}$

Hinweis: Man kommt in allen Fällen mit Grammatiken aus, bei denen auf der linken Seite der Produktionen immer nur ein Symbol steht. Bei der dritten Sprache ist es nicht so einfach, ein Grammatik zu finden. (Aber versuchen Sie es trotzdem!) Denken Sie daran, dass Sie das Verhalten solcher Grammatiken mit [FLACI](#) überprüfen können.

★Aufgabe 2.14. Beschäftigen Sie sich intensiver mit der Grammatik (2.4). Leiten Sie einige Ausdrücke ab und zeichnen Sie dazu die Ableitungsbäume. Wie müsste man die Grammatik erweitern, damit in den Ausdrücken auch ganze Zahlen wie in $x=23*y$ vorkommen können? (Natürlich dürfen die nicht links vom Gleichheitszeichen stehen. Und was ist mit negativen Zahlen?) Wie müsste man die Grammatik ändern, damit an der Hierarchie im Ableitungsbaum eines Ausdrucks die Regel „[Punktrechnung vor Strichrechnung](#)“ erkennbar ist? (Das sind lauter Fragen, die beim Entwerfen einer Programmiersprache tatsächlich eine Rolle spielen.)

¹⁷Allerdings kann das Programm nur mit Grammatiken umgehen, in denen die linken Seiten aller Produktionen jeweils nur aus einem nichtterminalen Zeichen bestehen, also z. B. mit den Grammatiken (2.2) und (2.4), aber *nicht* mit (2.1). Das wird jedoch für die folgenden Abschnitte ausreichen und wir werden [später](#) noch sehen, wie diese Einschränkung zustande kommt.

Aufgabe 2.15. Falls Ihnen Grammatiken theoretisch und esoterisch vorkommen, dann schauen Sie sich beispielsweise einmal die [Syntax-Spezifikation](#) von [JAVA](#) an. Das ist eines von vielen Beispielen dafür, dass formale Grammatiken in der [Praktischen Informatik](#) regelmäßig eingesetzt werden. Ein weiteres Beispiel ist der [Standard](#) der Programmiersprache [SCHEME](#).¹⁸

★**Aufgabe 2.16.** Geben Sie eine Grammatik an, die die Sprache $\{0^n 1^n 0^n : n \in \mathbb{N}\}$ über Σ_{bool} erzeugt.

Aufgabe 2.17. In der folgenden Grammatik sind $\langle \text{Subjekt} \rangle$, $\langle \text{Prädikat} \rangle$ und so weiter Nichtterminalsymbole, $\langle \text{Satz} \rangle$ ist das Startsymbol und Bär, Kekse, dicke und so weiter sind Terminalsymbole.¹⁹

$\langle \text{Satz} \rangle \rightarrow \langle \text{Subjekt} \rangle \langle \text{Prädikat} \rangle \langle \text{Objekt} \rangle$
 $\langle \text{Subjekt} \rangle \rightarrow \langle \text{Artikel} \rangle \langle \text{Adjektiv} \rangle \langle \text{Substantiv} \rangle$
 $\langle \text{Artikel} \rangle \rightarrow \text{Der} \mid \text{Die} \mid \text{Das}$
 $\langle \text{Adjektiv} \rangle \rightarrow \text{dicke} \mid \text{hübsche} \mid \text{schnuckelige}$
 $\langle \text{Substantiv} \rangle \rightarrow \text{Bär} \mid \text{Hund} \mid \text{Rhinozeros} \mid \text{Frosch}$
 $\langle \text{Prädikat} \rangle \rightarrow \text{bevorzugt} \mid \text{jagt} \mid \text{verspeist} \mid \text{sucht}$
 $\langle \text{Objekt} \rangle \rightarrow \text{Kekse} \mid \text{Kuchen} \mid \text{Pizza} \mid \text{Käse}$

Sowas in der Art (nur viel komplizierter) schwebte [Chomsky](#) wohl einmal vor. Wie viele verschiedene „Wörter“ kann man mit dieser Grammatik bilden? Wie viele davon sind grammatisch korrekte deutsche Sätze?

★3. Die Anzahl der Sprachen

Bevor wir uns in den nächsten Kapiteln bestimmte Klassen von Sprachen genauer anschauen, wollen wir uns zunächst überlegen, ob Grammatiken ein Werkzeug sind, mit dem man beliebige Sprachen beschreiben kann. Gibt es zu *jeder* Sprache eine Grammatik, die sie erzeugt?

Aufgabe 3.1. Was meinen Sie? Nach einigen Semestern Hochschulmathematik haben Sie vielleicht schon ein Gefühl für die Antwort.

Die Antwort ist: nein. Und es liegt vereinfacht formuliert daran, dass es *viel zu viele* Sprachen gibt. Für die „allermeisten“ Sprachen gibt es keine Grammatik. Ich werde dafür eine Beweisskizze liefern, die hoffentlich ausführlich genug ist, dass Sie die fehlenden Details bei Bedarf selbst ergänzen können. Voraussetzung ist allerdings, dass Ihnen die Begriffe *abzählbar* und *überabzählbar* etwas sagen.²⁰

Zu jedem Alphabet Σ gibt es überabzählbar viele Sprachen, aber nur abzählbar viele davon werden durch Grammatiken erzeugt.

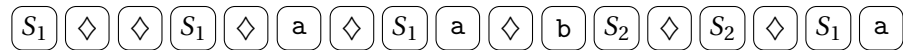
Satz 3.1

¹⁸In der Praxis werden Grammatiken für Programmiersprachen meistens in [Backus-Naur-Form](#) angegeben, während die im Skript verwendete Notation eher in akademischen Welt üblich ist.

¹⁹Anders als im Rest des Skripts bestehen Symbole hier also aus mehreren Buchstaben.

²⁰Siehe dazu die Kapitel 18 und 20 in [KMFI](#).

Beweis. Wir überlegen uns zuerst, dass es nur abzählbar viele durch Grammatiken erzeugte Sprachen über Σ geben kann. Ist L so eine Sprache und $G = (N, \Sigma, P, S)$ eine Grammatik mit $L = L(G)$, so spielt die Benennung der Nichtterminalsymbole offenbar keine Rolle für die resultierende Sprache. Wir können also o. B. d. A.²¹ davon ausgehen, dass diese S_1, S_2 und so weiter heißen, wobei der Einfachheit halber S_1 das Startsymbol S sein soll. Wir legen nun eine möglichst platzsparende Schreibweise fest, mit der solche Grammatiken notiert werden können. Wie die genau spezifiziert ist, spielt keine Rolle, aber es könnte für (2.1) z. B. so aussehen:²²



Dabei fungiert das Zeichen $\diamond$ abwechselnd als Produktionspfeil und als Trennzeichen zwischen Produktionen. Wie viele Grammatiken gibt es, die in dieser Notation m Zeichen lang sind? Eine genaue Zahl brauchen wir gar nicht. Wir können jedoch ganz großzügig abschätzen, dass so eine Grammatik auf jeden Fall weniger als m Nichtterminalsymbole hat. Zusammen mit dem Trennzeichen $\diamond$ und dem Alphabet Σ kann man mit diesen Symbolen höchstens $(m + |\Sigma|)^m$ Zeichenketten bilden (von denen die meisten gar keine Grammatiken darstellen werden). Entscheidend ist: es sind „nur“ endlich viele und wir könnten sie theoretisch alle der Reihe nach aufzählen. Und das wiederum können wir der Reihe nach für $m = 1, m = 2$ et cetera machen. Die Menge der Grammatiken ist also abzählbar.

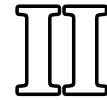
Andererseits ist die Menge Σ^* auf jeden Fall unendlich:²³ Σ enthält mindestens ein Zeichen a und damit bereits die Wörter a, a^2, a^3 und so weiter. Daher ist die Menge $\mathcal{P}(\Sigma^*)$ nach dem Satz von Cantor überabzählbar. ■

Das ist gewissermaßen das „Schicksal“ der Theoretischen Informatik. Wir werden neben Grammatiken noch weitere Techniken zum Klassifizieren von Sprachen kennenlernen. Aber wenn wir es mit realistischen Modellen von echten Rechenprozessen zu tun haben, dann werden die modellierten Ressourcen (Zeichenvorrat, Speicherzellen, Zustände) zwangsläufig immer endlich sein und wir werden daher immer an die Grenze der Abzählbarkeit stoßen. Wenn man es poetisch ausdrücken möchte, dann stehen wir am Strand und halten die Füße ins Wasser, werden aber nie erfahren, was sich in dem Meer befindet, das wir vor uns sehen.

²¹Diese Abkürzung steht für *ohne Beschränkung der Allgemeinheit* und damit ist gemeint, dass man – in der Regel zur Vermeidung von Schreibarbeit – eine Einschränkung vornimmt, die aber faktisch keine ist. Im konkreten Fall könnten die Symbole andere Namen haben, aber das ist irrelevant für die Korrektheit des Beweises.

²²Vereinbarungsgemäß „heißt“ X nun S_1 und Y heißt S_2 .

²³Man überlegt sich leicht, dass Σ^* auch abzählbar ist, weil Σ endlich ist. Das spielt für diesen Beweis aber keine Rolle.



Reguläre Sprachen

Die Klasse der regulären Sprachen ist die einfachste Sprachenklasse, die in der Theoretischen Informatik untersucht wird. Wir werden uns mit ihr recht ausführlich beschäftigen, weil man auf diesem Wege verschiedene Aspekte der Theorie mit vergleichsweise geringem Aufwand kennenlernen kann.

4. Reguläre Grammatiken

In Chomskys Theorie werden Grammatiken danach klassifiziert, wie rigide die Regeln für zulässige Produktionen sind. Das führt zu einer Abstufung, die man *Chomsky-Hierarchie* nennt.¹ Wir beginnen mit der einfachsten Kategorie von Grammatiken, für die die Regeln besonders streng sind.

Chomsky-Hierarchie

Eine Grammatik (N, T, P, S) wird **regulär** oder **Typ-3-Grammatik** genannt, wenn für alle Produktionen $(\alpha, \beta) \in P$ die beiden Bedingungen $\alpha \in N$ und $\beta \in (T \circ N) \cup \{\varepsilon\}$ erfüllt sind. Eine Sprache wird **regulär** genannt, wenn es eine reguläre Grammatik gibt, die sie erzeugt.

Definition

Bei solchen Grammatiken darf also links vom Pfeil immer nur genau ein Nichtterminalsymbol stehen, während rechts entweder das leere Wort steht oder eine Kombination aus einem Terminalsymbol gefolgt von einem Nichtterminalsymbol. Das ist eine ziemlich starke Einschränkung der Regeln. Die Grammatik aus der Lösung von Aufgabe 2.8 ist beispielsweise regulär. Ein weiteres Beispiel ist diese Grammatik:



$$\begin{aligned} S &\rightarrow \varepsilon \mid aB \\ B &\rightarrow bC \\ C &\rightarrow bS \end{aligned} \tag{4.1}$$

¹Manchmal auch *Chomsky-Schützenberger-Hierarchie*, weil deren endgültige Form von Chomsky gemeinsam mit dem französischen Mathematiker [Marcel Schützenberger](#) publiziert wurde.

Aufgabe 4.1. Welche Sprache erzeugt die Grammatik (4.1)?

Aufgabe 4.2. Zeichnen Sie für die Grammatik (4.1) einen Ableitungsbaum für das Wort $(abb)^3$. Sie werden sehen, dass der eine ganz bestimmte Form hat. Man kann sich leicht überlegen, dass Ableitungsbäume für reguläre Grammatiken immer so aussehen müssen.

Aufgabe 4.3. Ist die Dyck-Sprache D_1 eine reguläre Sprache?

Aufgabe 4.4. Die Grammatik

$$\begin{aligned} S &\rightarrow \varepsilon \mid aS \mid bT \\ T &\rightarrow c \end{aligned}$$

ist *nicht* regulär. (Warum?) Geben Sie eine reguläre Grammatik an, die dieselbe Sprache erzeugt.

Aufgabe 4.5. Die Grammatik

$$\begin{aligned} S &\rightarrow \varepsilon \mid aT \\ T &\rightarrow abS \end{aligned}$$

ist *nicht* regulär. (Warum?) Geben Sie eine reguläre Grammatik an, die dieselbe Sprache erzeugt.

Aufgabe 4.6. Begründen Sie, dass jede endliche Sprache regulär ist.

Nach dem Bearbeiten von Aufgabe 4.3 stellen Sie sich vielleicht die Frage, wie man beweisen soll, dass eine Sprache *nicht* regulär ist. Ein Werkzeug dafür lernen wir nun kennen. Es ist gleichzeitig das erste richtige mathematische Theorem in diesem Skript.

Lemma 4.1

Pumping-Lemma für reguläre Sprachen

Ist L eine reguläre Sprache, dann gibt es eine Zahl $n \in \mathbb{N}$, so dass sich für jedes Wort $x \in L$ mit $|x| \geq n$ Wörter u , v und w finden lassen, für die die folgenden vier Bedingungen gelten.²

- (i) $x = uvw$
- (ii) $v \neq \varepsilon$
- (iii) $|uv| \leq n$
- (iv) $uv^k w \in L$ für alle $k \in \mathbb{N}$

Beweis. Weil L regulär ist, gibt es eine reguläre Grammatik $G = (N, T, P, S)$, die L erzeugt. Bei der Herleitung eines Wortes aus L wird das Wort in jedem Schritt genau ein Symbol länger und alle Zwischenschritte sind Wörter aus $T^* \circ N$. Für ein Wort der Länge m braucht man also $m + 1$ Herleitungsschritte der Form $\Rightarrow_G$ und in jedem Schritt kommt rechts ein Nichtterminalsymbol vor. Ist $m > |N|$, so muss

²Da die vierte Bedingung auch für $k = 0$ gilt, kann man v also auch weglassen.

dabei mindestens ein Symbol $A \in N$ mehr als einmal vorkommen, d. h., es wird so aussehen (wobei σ_1 bis σ_m Terminalsymbole sein sollen und $j \leq |N| + 1$ gilt):

$$\begin{aligned} S &\Rightarrow_G^* \underbrace{\sigma_1 \dots \sigma_i}_u A \\ &\Rightarrow_G^* \underbrace{\sigma_1 \dots \sigma_i}_u \underbrace{\sigma_{i+1} \dots \sigma_j}_v A \\ &\Rightarrow_G^* \underbrace{\sigma_1 \dots \sigma_i}_u \underbrace{\sigma_{i+1} \dots \sigma_j}_v \underbrace{\sigma_{j+1} \dots \sigma_m}_w \end{aligned}$$

Das bedeutet, dass sowohl $A \Rightarrow_G^* vA$ als auch $A \Rightarrow_G^* w$ gelten muss, wobei v nicht das leere Wort sein kann. Man kann also die mittlere Zeile auch weglassen oder zwischen ihr und der letzten A erneut (ggf. mehrfach) durch vA ersetzen. (Denn die erlaubten Herleitungsschritte hängen in regulären Grammatiken offenbar immer nur vom letzten Zeichen ab.) Die Rolle von n in der Formulierung des Lemmas kann $|N| + 1$ spielen. ■

Das Pumping-Lemma trägt diesen etwas seltsamen Namen, weil es besagt, dass man in regulären Sprachen bei allen Wörtern, die lang genug sind, einen Teil des Wortes „aufpumpen“ kann, um weitere Wörter der Sprache zu erhalten. Wie kann man es nun benutzen, um von einer Sprache zu zeigen, dass sie *nicht* regulär ist?

Nehmen wir als Beispiel die Sprache $L = \{a^k b^k : k \in \mathbb{N}^+\}$. Wenn L regulär wäre, dann würde uns das Pumping-Lemma eine Zahl n mit den dort genannten Eigenschaften liefern. Für das Wort $a^n b^n$ aus L müsste dabei der „aufpumpbare“ Teil v komplett in der linken Hälfte des Wortes liegen und damit ein Element von $\{a\}^+$ sein. Ersetzt man nun v z. B. durch v^2 , so gilt für das dadurch entstehende Wort w nicht mehr $|w|_a = |w|_b$, d. h., es gehört nicht zu L . Das ist ein Widerspruch, also kann L nicht regulär sein.

Beachten Sie den feinen Unterschied zur Sprache $\{(ab)^k : k \in \mathbb{N}\}$. Die ist nach den Aufgaben 2.10 und 4.5 offensichtlich regulär!

Die Aussage des Pumping-Lemmas gilt mit denselben Bezeichnungen für alle Wörter der Sprache, die sich in der Form yxz mit $|x| \geq n$ darstellen lassen.

Korollar 4.2

Beweis. Diese leichte Verallgemeinerung besagt nur, dass man irgendwo mitten im Wort anfangen kann, denn es geht ja nur darum, dass das Teilwort x , mit dem man arbeitet, lang genug ist, um Wiederholungen zu erzwingen. Am Beweis ändert sich ansonsten nichts. ■

Beachten Sie, dass das Pumping-Lemma eine **notwendige, aber keine hinreichende Bedingung** dafür liefert, dass eine Sprache regulär ist.³ So ein Kriterium werden wir **später** kennenlernen.

³Dass es sich nicht um ein hinreichendes Kriterium handelt, wird in Aufgabe 9.17 gezeigt.

Aufgabe 4.7. Unter der URL <https://weitz.de/pump/> finden Sie eine Realisation des Pumping-Lemmas als eine Art Spiel, das man zum Üben verwenden kann. Sie spielen gegen den Computer, der von einer Sprache behauptet, sie sei regulär. Er gibt zu der Sprache auch die Zahl n aus dem Pumping-Lemma an.⁴ Sie dürfen nun ein Wort auswählen und von dem wiederum einen Teil, der mindestens die Länge n hat. Das ist das Teilwort, das in Korollar 4.2 x heißt. Der Computer markiert im Gegenzug in diesem Teilwort das Stück v und Sie müssen nun k so wählen, dass das Ergebnis *nicht* zur Sprache gehört – denn Sie sollen zeigen, dass die Sprache *nicht* regulär ist. (Das wird Ihnen nicht immer gelingen, denn einige der Sprachen *sind* regulär ...)

Aufgabe 4.8. Begründen Sie mithilfe des Pumping-Lemmas 4.1, dass die Sprache

$$L = \{ w \in \Sigma_{\text{bool}}^* : |w| \text{ ist ungerade und das mittlere Symbol ist } 0 \}$$

nicht regulär ist. Schreiben Sie eine vollständige Begründung in ganzen Sätzen auf, so dass jemand, der sich mit der Materie auskennt, sie nachvollziehen kann.

Aufgabe 4.9. Suchen Sie sich eine der Sprachen aus Aufgabe 4.7 aus, die nicht regulär ist, und schreiben Sie mit Ihren eigenen Worten eine vollständige Begründung dafür auf, warum die Sprache nicht regulär ist.

★Aufgabe 4.10. Begründen Sie mithilfe des Pumping-Lemmas 4.1, dass die Sprache⁵ $\{1^p : p \in \mathbb{P}\}$ über dem Alphabet $\{1\}$ nicht regulär ist.

5. Deterministische endliche Automaten



Die sogenannten *Automaten*, mit denen wir uns im Folgenden beschäftigen werden, sind keine wirklichen Automaten, die Fahrkarten oder Kaffee ausgeben können – obwohl sie mit denen gewisse Gemeinsamkeiten haben. Es sind theoretische Konstrukte, mit denen man – wie mit Grammatiken – formale Sprachen kategorisieren kann. Während Grammatiken Regeln beschreiben, nach denen die Wörter einer Sprache *produziert* werden, kann man sich Automaten als Vorrichtungen vorstellen, die Wörter *konsumieren*.

Wie bei den Grammatiken fangen wir mit der einfachsten Kategorie an. Wir werden feststellen, dass wir auf genau dieselben Sprachen kommen.

Definition

Ein **endlicher Automat**⁶ (englisch *finite state machine* oder *finite automaton*) ist ein 5-Tupel $A = (S, I, \delta, s_0, F)$ mit den folgenden Eigenschaften:

- (i) S ist eine nichtleere endliche Menge, deren Elemente man als **Zustände** (engl. *states*) bezeichnet.
- (ii) I ist ein Alphabet, das **Eingabealphabet** genannt wird.
- (iii) δ ist eine Funktion von $S \times I$ nach S , die sogenannte **Übergangsfunktion** (*transition function*).
- (iv) Der **Startzustand** s_0 ist ein Element von S .

⁴Im Spiel ist von Automaten die Rede. Das können Sie vorerst ignorieren.

⁵Mit $\mathbb{P}$ ist die Menge der **Primzahlen** gemeint. Siehe Kapitel 8 in [KMFI](#).

- (v) F ist eine Teilmenge von S , deren Elemente **akzeptierenden Zustände** oder **Endzustände** genannt werden.

Die Vorstellung dabei ist, dass ein endlicher Automat im Startzustand beginnt und in jedem Schritt genau ein Zeichen der Eingabe liest und dabei jeweils zu einem neuen Zustand wechselt. Dieses Verhalten wird durch die Übergangsfunktion festgelegt: Gilt $\delta(s, a) = t$, dann wechselt der Automat in den Zustand t , wenn er im Zustand s das Symbol a liest.

Wir machen aus der Übergangsfunktion δ für *Zeichen* eine Funktion δ^* für *Wörter*, indem wir für alle $s \in S$, $a \in I$ und $w \in I^*$ **rekursiv**

$$\begin{aligned}\delta^*(s, \epsilon) &= s \quad \text{und} \\ \delta^*(s, aw) &= \delta^*(\delta(s, a), w)\end{aligned}$$

definieren. Die *von A akzeptierte Sprache* sei dann als

$$L(A) = \{w \in I^* : \delta^*(s_0, w) \in F\}$$

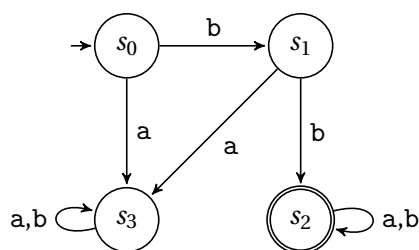
akzeptierte Sprache

$L(A)$

definiert. Ein Wort gehört also per definitionem zu $L(A)$, wenn die Symbole des Wortes den Automaten der Reihe nach vom Startzustand zu einem akzeptierenden Zustand überführen.

Aufgabe 5.1. Überprüfen Sie, dass nach der obigen Definition $\delta^*(s, a) = \delta(s, a)$ für alle $s \in S$ und $a \in I$ gilt, indem Sie links $a = a\epsilon$ ausnutzen.

Einfache endliche Automaten kann man vollständig grafisch darstellen, indem man die Zustände als Kreise und die Übergänge als Pfeile abbildet. Der Startzustand wird durch einen kleinen Pfeil markiert, die Endzustände durch einen doppelten Rand:⁷



	s_0	s_1	s_2	s_3
a	s_3	s_3	s_2	s_3
b	s_1	s_2	s_2	s_3

In diesem Beispiel wäre S die Menge $\{s_0, s_1, s_2, s_3\}$, I das Alphabet $\{a, b\}$, F die Menge $\{s_2\}$ und s_0 der Startzustand. Auch die Übergangsfunktion δ kann man direkt der Grafik entnehmen. Sie wird rechts daneben als Tabelle dargestellt.

Im obigen Automaten gilt z. B.

$$\delta^*(s_0, bb) = \delta^*(\delta(s_0, b), b) = \delta^*(s_1, b) = s_2,$$

⁶Noch genauer müsste man eigentlich von *deterministischen* endlichen Automaten sprechen. In der (englischen) Fachliteratur werden die Abkürzungen DFA und FSM verwendet.

DFA FSM

⁷Und wir sparen uns etwas Arbeit, indem wir an einen Pfeil a,b schreiben statt zwei Pfeile – einen für a und einen für b – zu zeichnen.

$$\delta^*(s_1, ba) = \delta^*(\delta(s_1, b), a) = \delta^*(s_2, a) = s_2 \quad \text{und} \\ \delta^*(s_0, bba) = \delta^*(\delta(s_0, b), ba) = \delta^*(s_1, ba) = s_2.$$

Damit ist klar, dass bb und bba zu der von ihm akzeptierten Sprache gehören. Offenbar ist es jedoch wesentlich einfacher, den Pfeilen in der Grafik zu folgen. Man sieht dann beispielsweise, dass man vom Startzustand s_0 aus zum Endzustand s_2 gelangen kann, indem man dem Automaten sukzessive mit den Zeichen b , b und a „füttert“.

Aufgabe 5.2. Welche Sprache wird von diesem Automaten akzeptiert? Geben Sie sie in beschreibender Mengenschreibweise an.

Da der Definitionsbereich der Übergangsfunktion δ die Menge $S \times I$ ist, muss im Prinzip in der grafischen Darstellung von *jedem* Zustand aus S für *jedes* Zeichen aus I ein Pfeil ausgehen. Die Darstellung von umfangreicheren Automaten wird dadurch jedoch oft unnötig unübersichtlich. Viele Pfeile führen evtl. zu Zuständen, von denen aus man nie zu akzeptierenden Zuständen gelangen kann, und sind daher in gewissem Sinne überflüssig. Im obigen Beispiel ist s_3 offenbar so eine „Sackgasse“. Daher vereinbaren wir:

Konvention

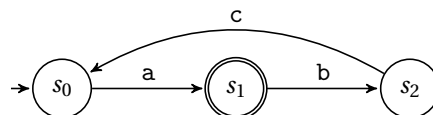
Fehlerzustand

Wenn in der grafischen Darstellung⁸ eines endlichen Automaten Pfeile fehlen, dann sollen diese auf einen (nicht eingezeichneten) „Fehlerzustand“ s_e führen, der alle Zeichen „schluckt“, für den also $\delta(s_e, a) = s_e$ für alle $a \in I$ gilt.

Nach dieser Konvention vereinfacht sich die obige Grafik deutlich, wodurch es nun ein Kinderspiel ist, die akzeptierte Sprache zu erkennen:



Aufgabe 5.3. Geben Sie für den folgenden Automaten die Übergangsfunktion in der Form einer Tabelle an. Die Tabelle soll vollständig in dem Sinne sein, dass sie einen Fehlerzustand beinhaltet und komplett gefüllt ist. Geben Sie außerdem die von diesem Automaten akzeptierte Sprache in beschreibender Mengenschreibweise oder in der Notation aus Abschnitt 1 an.

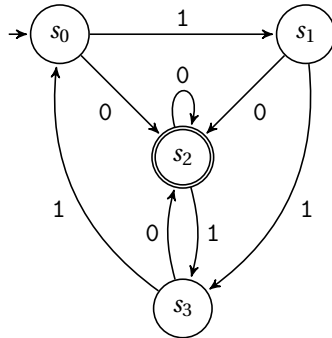


Aufgabe 5.4. Wie ändert sich in Aufgabe 5.3 die akzeptierte Sprache, wenn $\{s_0\}$ bzw. $\{s_0, s_1\}$ die Menge der akzeptierenden Zustände ist?

⁸Man beachte, dass es sich hierbei lediglich um eine Konvention für die *grafische* Darstellung handelt. Wenn wir später zum Beispiel in Beweisen über Automaten sprechen, dann sollen die immer der mathematischen Definition entsprechen: S enthält *alle* Zustände und die Übergangsfunktion ist auf ganz $S \times I$ definiert.

Aufgabe 5.5. Geben Sie einen Automaten an, der die Sprache aus Aufgabe 4.1 akzeptiert.

Aufgabe 5.6. Geben Sie für den folgenden endlichen Automaten die von ihm akzeptierte Sprache in korrekter Schreibweise an.



Aufgabe 5.7. Mit FLACI kann man auch Automaten simulieren und natürlich sollten Sie das machen, bis Sie sich sicher sind, die Inhalte dieses Abschnitts verstanden zu haben. Unter der URL <http://automatonsimulator.com/> findet man eine Alternative zu FLACI.

Aufgabe 5.8. Konstruieren Sie für die folgenden Sprachen über Σ_{bool} jeweils einen endlichen Automaten, der sie akzeptiert.

- (i) $\{0(10)^n 1 : n \in \mathbb{N}\}$
- (ii) Sprache (i) aus Aufgabe 2.13
- (iii) Sprache (ii) aus Aufgabe 2.13

Aufgabe 5.9. Konstruieren Sie für

$$L = \{w_1 a w_2 a : w_1 \in \Sigma^* \wedge w_2 \in \{b\}^* \wedge |w_2| \text{ gerade}\}$$

einen endlichen Automaten, der L akzeptiert. (Hinweis: Beachten Sie, dass die Sprache $\{waa : w \in \Sigma^*\}$ eine Teilmenge von L ist, weil 0 eine gerade Zahl ist.)

Aufgabe 5.10. Ein Kaffeeautomat kann drei Sorten Münzen erkennen: 50 Cent, 1 Euro und 2 Euro. Wir kürzen diese mit den Symbolen f , e und z ab. Man kann einen Kaffee auswählen, wenn man *mindestens* zwei Euro eingeworfen hat. Den Automaten „stört“ es aber auch nicht, wenn man zu viel Geld einschmeißt. Konstruieren Sie einen endlichen Automaten, der genau die Wörter aus den obigen drei Symbolen akzeptiert, die einem Betrag von zwei Euro oder mehr entsprechen, also z. B. z , efe oder f^4 .

Aufgabe 5.11. Sei A ein endlicher Automat mit dem Eingabealphabet I . Welche Möglichkeiten für $L(A)$ gibt es, wenn A nur einen einzigen Zustand hat?

Die folgende Aussage stellt einen Teil des am Anfang des Abschnitts angekündigten Zusammenhangs her.

Zu jedem endlichen Automaten A gibt es eine reguläre Grammatik G mit $L(G) = L(A)$.

Lemma 5.1

Beweis. Der Beweis ist naheliegend und ergibt sich quasi von selbst. Zum Automaten⁹ $A = (Z, I, \delta, s_0, F)$ definieren wir $G = (N, T, P, S)$ zunächst durch $N = Z$, $T = I$, $S = s_0$. Für jeden Übergang (Pfeil) $\delta(B, a) = C$ fügen wir zu P die Produktion $B \rightarrow aC$ hinzu. Für jeden Zustand $B \in F$ fügen wir außerdem $B \rightarrow \varepsilon$ hinzu. ■

Aufgabe 5.12. Überzeugen Sie sich anhand von Beispielen, dass die Konstruktion im Beweis von Lemma 5.1 wirklich immer eine reguläre Grammatik liefert, die dieselbe Sprache erzeugt.

★**Aufgabe 5.13.** Warum kann man die Konstruktion aus dem Beweis von Lemma 5.1 nicht einfach „umkehren“, um zu zeigen, dass es zu jeder regulären Grammatik G einen endlichen Automaten A mit $L(A) = L(G)$ gibt?

Schließlich noch ein paar Lemmata, die wir später noch brauchen werden. Außerdem wollen wir für alle Sprachenklassen auch gewisse *Abschlusseigenschaften* festhalten. Weil beispielsweise die Summe zweier natürlicher Zahlen wieder eine natürliche Zahl ist, sagt man, dass $\mathbb{N}$ *abgeschlossen* gegen die Addition ist. In diesem Sinne ist z. B. die Klasse der regulären Sprachen abgeschlossen gegen die Vereinigung von Mengen. Wir werden später sehen, dass das aus Lemma 5.3 folgt.

Lemma 5.2

Zu jedem endlichen Automaten A mit Eingabealphabet I gibt es einen Automaten, der die Sprache $I^* \setminus L(A)$ akzeptiert.¹⁰

Aufgabe 5.14. Der Beweis dieser Aussage ist so „billig“, dass Sie eigentlich selbst darauf kommen sollten. Wenn Sie besser in konkreten Beispielen denken können, überlegen Sie sich zunächst, was Sie an dem allerersten Automaten aus diesem Abschnitt ändern müssten, damit er alle Wörter aus $\{a, b\}^*$ *außer* denen akzeptiert, die mit bb anfangen. (Wohlgemerkt: Sie sollen den vorhandenen Automaten mit möglichst geringem Aufwand ändern, nicht etwa einen neuen bauen.)

Lemma 5.3

Sind L_1 und L_2 Sprachen über demselben Alphabet I , die von endlichen Automaten akzeptiert werden, dann gibt es auch einen Automaten, der $L_1 \cup L_2$ akzeptiert.

Beweis. Sei jeweils $A_i = (S_i, I, \delta_i, s_i, F_i)$ der Automat mit $L_i = L(A_i)$. Wir laufen nun „parallel“ durch beide Automaten durch. Dafür definieren wir einen neuen Automaten, dessen Zustandsmenge $S_1 \times S_2$ ist. Sein Startzustand soll (s_1, s_2) sein und seine Übergangsfunktion wird definiert durch

$$\delta((s, t), a) = (\delta_1(s, a), \delta_2(t, a))$$

für $a \in I$, $s \in S_1$ und $t \in S_2$. Als Menge der akzeptierenden Zustände nehmen wir $\{(s, t) \in S_1 \times S_2 : s \in F_1 \text{ oder } t \in F_2\}$. Es sollte hoffentlich nicht schwer zu verstehen

⁹Im Beweis nennen wir die Menge der Zustände Z , weil der Buchstabe S auch für das Startsymbol der Grammatik verwendet wird.

¹⁰Man nennt $I^* \setminus L(A)$ auch das *Komplement* von $L(A)$ (bezüglich I^*).

sein, dass dieser Automat – den man übrigens den *Produktautomaten* von A_1 und A_2 nennt – die Vereinigung von L_1 und L_2 akzeptiert. ■

Produktautomat

Sind L_1 und L_2 Sprachen über demselben Alphabet I , die von endlichen Automaten akzeptiert werden, dann gibt es auch Automaten, die $L_1 \cap L_2$ bzw. $L_1 \setminus L_2$ akzeptieren.

Lemma 5.4

Aufgabe 5.15. Denken Sie sich zwei einfache endliche Automaten aus (oder bedienen Sie sich bei den bisherigen Beispielen) und führen Sie den Beweis von Lemma 5.3 anhand dieser beiden konkret durch.

Aufgabe 5.16. Überlegen Sie sich einen Beweis für Lemma 5.4. (Hinweis: Sie müssen gar keine Automaten konstruieren. Verwenden Sie einfach die [Regeln von De Morgan](#), die Sie z. B. in Kapitel 15 von [KMFI](#) nachschlagen können.)

6. Nichtdeterministische endliche Automaten

In diesem Abschnitt begegnen wir zum ersten Mal einem wichtigen Konzept der Theoretischen Informatik,¹¹ dem *Nichtdeterminismus*. Die Automaten, die wir bisher kennen, arbeiten *deterministisch*. Damit ist gemeint, dass sie bei derselben Eingabe immer dasselbe machen; ihr Verhalten ist vollständig vorhersehbar. Bei nichtdeterministischen Automaten ist das anders.

Nichtdeterminismus

Ein **nichtdeterministischer endlicher Automat** (englisch *nondeterministic finite automaton*, NFA) ist ein 5-Tupel $A = (S, I, \delta, s_0, F)$, das sich von einem deterministischen endlichen Automaten lediglich in einem Detail unterscheidet: δ ist eine Funktion von $S \times I$ nach $\mathcal{P}(S)$ statt nach S .

Definition

NFA

Die erweiterte Übergangsfunktion δ^* bildet nun von $\mathcal{P}(S) \times I^*$ nach $\mathcal{P}(S)$ ab und wird für $T \subseteq S$, $a \in I$ und $w \in I^*$ definiert durch

$$\begin{aligned} \delta^*(T, \varepsilon) &= T \quad \text{und} \\ \delta^*(T, aw) &= \bigcup_{s \in T} \delta^*(\delta(s, a), w). \end{aligned}$$

Und die von einem NFA A akzeptierte Sprache ist jetzt

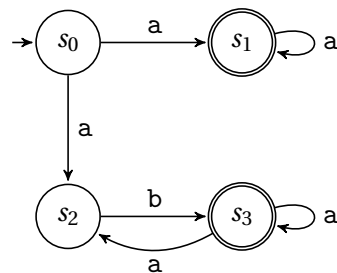
$$L(A) = \{ w \in I^* : \delta^*({s_0}, w) \cap F \neq \emptyset \}.$$

Aufgabe 6.1. Analog zu Aufgabe 5.1 kann man durch Einsetzen in die Definition nachrechnen, dass sinnvollerweise $\delta^*({s}, a) = \delta(s, a)$ für $s \in S$ und $a \in I$ gilt. Machen Sie das bitte!

¹¹Für die Einführung dieser Idee wurden der Amerikaner [Dana Scott](#) und der Israeli [Michael Rabin](#) mit dem [Turing Award](#) ausgezeichnet. Das ist gleichsam der Nobelpreis der Informatik. Den Namen Rabin haben Sie vielleicht schon einmal im Zusammenhang mit dem [Miller-Rabin-Test](#) gehört.



Wie ist diese neue Definition gemeint? Die Übergangsfunktion gibt für einen bestimmten Zustand und ein in diesem Zustand konsumiertes Zeichen nun nicht mehr *einen* Zustand an, in den der Automat wechseln *muss*, sondern sie gibt eine *Menge* von Zuständen an, aus denen sich der Automat gewissermaßen einen Folgezustand „aussuchen“ kann. Man kann es auch so interpretieren, dass es für ein vom Automaten konsumiertes Wort ggf. mehrere Wege vom Startzustand durch den Automaten gibt. Das Wort wird akzeptiert, wenn *mindestens einer* dieser Wege zu einem akzeptierenden Zustand führt.¹² Für die grafische Darstellung heißt das einfach, dass es potentiell mehr Pfeile gibt: Haben wir beispielsweise $\delta(s_0, a) = \{s_1, s_2\}$, dann gibt es vom Zustand s_0 aus zwei Pfeile, an denen das Symbol a steht: einen nach s_1 und einen nach s_2 .



	s_0	s_1	s_2	s_3
a	$\{s_1, s_2\}$	$\{s_1\}$	$\emptyset$	$\{s_2, s_3\}$
b	$\emptyset$	$\emptyset$	$\{s_3\}$	$\emptyset$

An der Grafik hier kann man gut ablesen, dass es zwei Wege für Wörter gibt, die mit dem Symbol a anfangen. Ein Wort wie a^3 wird im oberen Teil akzeptiert, wobei der Weg durch den Automaten im Zustand s_1 endet. Ein Wort wie $(ab)^2$ wird im unteren Teil akzeptiert, wobei der Weg im Zustand s_3 endet.

Übrigens wird in der Definition die leere Menge als Funktionswert nicht verboten. Das ist nicht schlimm und auch im obigen Beispiel zu sehen. Gilt $\delta(s, a) = \emptyset$, dann entspricht das der Situation in einem deterministischen endlichen Automaten, der vom Zustand s aus in einen Fehlerzustand übergeht, wenn er das Symbol a konsumiert. Für nichtdeterministische Automaten brauchen wir somit die Konvention mit dem Fehlerzustand nicht mehr, weil es „legal“ ist, wenn von einem Zustand nicht für jedes Symbol aus dem Eingabealphabet (mindestens) ein Pfeil ausgeht.

Aufgabe 6.2. Welche Sprache wird von dem Automaten im Beispiel akzeptiert?

Aufgabe 6.3. Zeichnen Sie einen (möglichst einfachen) nichtdeterministischen endlichen Automaten mit dem Eingabealphabet Σ_{bool} , in dem es für das Wort 01 mindestens zwei Wege gibt, von denen nur einer in einem akzeptierenden Zustand endet.

Aufgabe 6.4. Wie kann man zu einem vorgegebenen deterministischen Automaten $A = (S, I, \delta, s_0, F)$ einen nichtdeterministischen Automaten $A' = (S, I, \delta', s_0, F)$ konstruieren, so dass $L(A) = L(A')$ gilt? (Beachten Sie, dass sich laut Aufgabenstellung außer der Übergangsfunktion nichts ändern soll. Das ist einfacher, als Sie evtl. denken. Man muss nur die formalen Definitionen verstanden haben.)

¹²Vielleicht stellen Sie sich das wie einen [Parallelrechner](#) vor, der mehrere Wege gleichzeitig ausprobieren kann.

Man mag zunächst den Eindruck haben, dass die nichtdeterministischen Automaten „mächtiger“ sind als die deterministischen: dass sie mehr Sprachen akzeptieren können. Das ist aber nicht der Fall, denn es gilt auch die Umkehrung von Aufgabe 6.4:

Satz von Rabin und Scott

Zu jedem nichtdeterministischen endlichen Automaten $A = (S, I, \delta, s_0, F)$ gibt es einen deterministischen Automaten A' mit $L(A) = L(A')$.

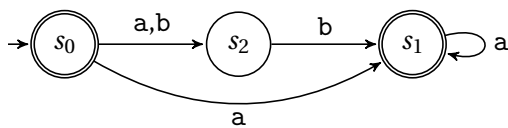
Satz 6.1

Beweis. Der „Trick“ ist, als Menge der Zustände von A' die Menge aller Zustandsmengen von A zu nehmen: $S' = \mathcal{P}(S)$. Der neue Startzustand ist $\{s_0\}$, die zugehörige Übergangsfunktion wird definiert durch¹³

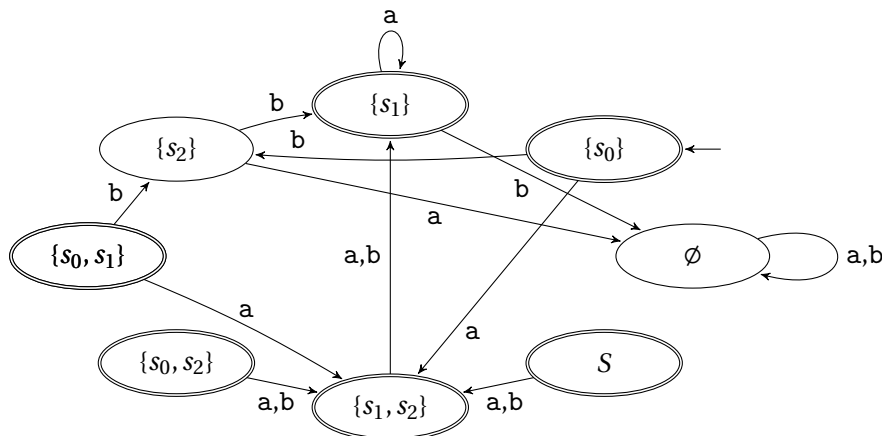
$$\delta'(s, a) = \bigcup_{t \in s} \delta(t, a) = \{r \in S : r \in \delta(t, a) \text{ für ein } t \in s\}$$

und die Menge der akzeptierenden Zustände durch $F' = \{s \in S' : s \cap F \neq \emptyset\}$. Auf den formalen Nachweis, dass dieselbe Sprache akzeptiert wird, werden wir hier verzichten. Wir schauen uns lieber gleich ein Beispiel an. ■

Zum Beispiel könnte A so aussehen:



Der im Beweis konstruierte Automat A' hat dann diese Gestalt:



A' „protokolliert“ quasi alle möglichen Wege durch A simultan. Befindet man sich beispielsweise im Zustand s_0 , so führt das Symbol a zu s_1 *oder* s_2 . Und ist man in einem von diesen beiden Zuständen, so führt b nur zu s_1 . Daher zeigt ein a -Pfeil von $\{s_0\}$ nach $\{s_1, s_2\}$ und ein b -Pfeil von dort nach $\{s_1\}$.

¹³Hier steht s für eine Menge von Zuständen von A und damit für *einen* Zustand von A' . t steht für Elemente von s , also für Zustände von A .

Beachten Sie jedoch, dass wir für die Aussage dieses Satzes einen ggf. hohen Preis zahlen müssen, weil die Anzahl der Zustände des deterministischen Automaten exponentiell mit der Anzahl der Zustände des Ausgangsautomaten ansteigt. Hat A z. B. zehn Zustände, so wird A' bereits $2^{10} = 1024$ Zustände haben.

Jedenfalls können wir nun das Pendant zu Lemma 5.1 beweisen.

Lemma 6.2

Zu jeder regulären Grammatik G gibt es einen deterministischen endlichen Automaten A mit $L(A) = L(G)$.

Beweis. Wie in Aufgabe 5.13 bereits angekündigt definieren wir zur Grammatik $G = (N, T, P, S)$ einen nichtdeterministischen Automaten $A = (Z, I, \delta, s_0, F)$ durch $Z = N$, $I = T$ und $s_0 = S$. Für jede Produktion $B \rightarrow aC$ fügen wir einen a -Pfeil von B nach C hinzu. Für jede Produktion $B \rightarrow \varepsilon$ machen wir B zu einem akzeptierenden Zustand. Nach Satz 6.1 können wir auf der Basis von A anschließend einen deterministischen Automaten für dieselbe Sprache konstruieren. ■

Damit können wir als erstes größeres Resultat festhalten, dass wir nun zwei ganz unterschiedliche Charakterisierungen derselben Menge von Sprachen haben:

Satz 6.3

Eine Sprache ist genau dann regulär (wird also von einer regulären Grammatik erzeugt), wenn sie von einem (deterministischen) endlichen Automaten akzeptiert wird.

Aufgabe 6.5. Durch Satz 6.3 erhält man einen alternativen Beweis für das Pumping-Lemma 4.1. Dabei spielt die Menge S der Zustände die Rolle, die im ursprünglichen Beweis von der Menge N der Nichtterminalsymbole gespielt wurde. Können Sie den Beweis skizzieren?

★**Aufgabe 6.6.** Die Definitionen für nichtdeterministische endliche Automaten sind in der Fachliteratur nicht immer identisch.¹⁴ In manchen Büchern dürfen solche Automaten eine *Menge* von Startzuständen haben, von denen sie sich dann ebenfalls einen „aussuchen“ dürfen. Überzeugen Sie sich davon, dass die Konstruktion im Beweis von Satz 6.1 mit dieser erweiterten Definition trotzdem noch funktionieren würde. Auch solche Automaten akzeptieren also nicht mehr Sprachen als die, die wir schon kennen.

ε -Übergang

★**Aufgabe 6.7.** Noch eine Sache, die nicht einheitlich geregelt ist: In einigen Büchern werden sogenannte ε -Übergänge zugelassen. Damit ist gemeint, dass der Definitionsbereich von δ nicht $S \times I$, sondern $S \times (I \cup \{\varepsilon\})$ ist. Gilt $\delta(s, \varepsilon) = T$, so bedeutet das, dass der Automat vom Zustand s in einen der Zustände aus T wechseln darf (aber nicht muss), ohne ein Zeichen zu konsumieren.¹⁵ Überlegen Sie sich, dass man auch dadurch nichts gewonnen hat. Mit anderen Worten: Geben sie an, wie man einen

¹⁴Und ich selbst bin auch nicht immer konsistent. In alten Videos (2015 oder früher) arbeite ich teilweise mit den Definitionen, die hier und in der folgenden Aufgabe beschrieben sind.

¹⁵Außerdem muss dann sinnvollerweise immer $s \in \delta(s, \varepsilon)$ gelten.

nichtdeterministischen endlichen Automaten *mit* ε -Übergängen zu einem *ohne* solche Übergänge umbauen kann, der dieselbe Sprache akzeptiert. (Der Einfachheit halber darf das Ergebnis mehrere Startzustände wie in Aufgabe 6.6 haben.)

Beachten Sie, dass es auch „Reihenschaltungen“ und „Zyklen“ von ε -Übergängen geben kann, also z. B. Zustände s_1 , s_2 und s_3 mit $s_2 \in \delta(s_1, \varepsilon)$ und $s_3 \in \delta(s_2, \varepsilon)$ oder $s_2 \in \delta(s_1, \varepsilon)$ und $s_1 \in \delta(s_2, \varepsilon)$.

Aufgabe 6.8. Auch an dieser Stelle empfehle ich wieder [FLACI](#), denn damit kann man auch den Umgang mit nichtdeterministischen Automaten üben. Auch mit der in Aufgabe 5.7 genannten Alternative ist das möglich. Die erlaubt sogar ε -Übergänge.

Aufgabe 6.9. In der Mathematik möchte man immer möglichst „schlanke“ Definitionen ohne überflüssigen Ballast haben. Deterministische endliche Automaten dürfen nur *einen* Startzustand haben, aber *mehrere* Endzustände. Man könnte sich fragen, ob das wirklich nötig ist oder ob *ein* Endzustand nicht evtl. reicht. Begründen Sie am Beispiel der Sprache $L = \{w \in \{0\}^* : 3 \mid |w| \text{ oder } 5 \mid |w|\}$, dass deterministische endliche Automaten mit nur *einem* Endzustand nicht jede reguläre Sprache akzeptieren würden.¹⁶

Aufgabe 6.10. Konstruieren Sie für die Sprache aus Aufgabe 6.9 einen nichtdeterministischen endlichen Automaten, der sie akzeptiert. Konstruieren Sie dann einen deterministischen Automaten, der dieselbe Aufgabe erledigt. Verwenden Sie dabei nicht den „Trick“ aus dem Beweis von Satz 6.1, sondern versuchen Sie, einen möglichst einfachen und übersichtlichen Automaten zu finden. (Hinweis: Dabei hilft es, wenn man sich an die „Uhren“ aus der [modularen Arithmetik](#) erinnert. Siehe Kapitel 3 in [KMFI](#).)

Aufgabe 6.11. Der nichtdeterministische endliche Automat

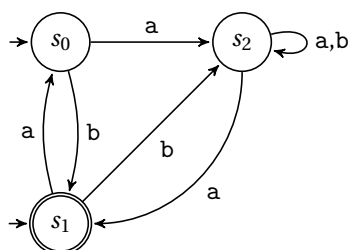
$$A = (\{s_0, s_1, s_2, s_3\}, \Sigma_{\text{bool}}, \delta, \{s_2\}, \{s_1\})$$

hat die durch die folgende Tabelle beschriebene Übergangsfunktion δ :

	s_0	s_1	s_2	s_3
0	$\{s_0\}$	$\{s_1\}$	$\{s_2\}$	$\{s_0\}$
1	$\{s_0\}$	$\{s_0\}$	$\{s_1, s_3\}$	$\{s_2\}$

Stellen Sie A grafisch dar und beschreiben Sie die von A akzeptierte Sprache $L(A)$.

Aufgabe 6.12. Konstruieren Sie wie im Beweis von Satz 6.1 zu dem folgenden nichtdeterministischen endlichen Automaten einen deterministischen, der dieselbe Sprache akzeptiert.



¹⁶Mit $k \mid n$ ist hier gemeint, dass k ein [Teiler](#) von n ist. Siehe Kapitel 4 in [KMFI](#).

Beachten Sie, dass der Automat mehr als einen Startzustand hat. (Siehe Aufgabe 6.6.)

Mithilfe der nichtdeterministischen endlichen Automaten können wir nun sehr einfach zwei Lemmata beweisen, die die vom Ende des letzten Abschnitts ergänzen.

Lemma 6.4

Sind L_1 und L_2 Sprachen über demselben Alphabet I , die von endlichen Automaten akzeptiert werden, dann gibt es einen Automaten, der $L_1 \circ L_2$ akzeptiert.

Beweis. Nach den bisherigen Überlegungen ist klar, dass es egal ist, ob wir deterministische oder nichtdeterministische Automaten betrachten. Der Einfachheit halber gehen wir von deterministischen Automaten A_1 und A_2 für L_1 bzw. L_2 aus, die die Form $A_i = (S_i, I, \delta_i, s_i, F_i)$ haben. O. B. d. A. seien S_1 und S_2 disjunkt.¹⁷ Wir konstruieren daraus einen nichtdeterministischen endlichen Automaten A für $L_1 \circ L_2$ wie folgt: Die Zustandsmenge ist $S_1 \cup S_2$, der Startzustand ist s_1 und die Menge der akzeptierenden Zustände F_2 . Wir behalten alle vorhandenen Pfeile bei, fügen aber für jeden Zustand s aus F_1 einen ε -Übergang (siehe Aufgabe 6.7) von s nach s_2 hinzu. ■

Lemma 6.5

Zu jedem endlichen Automaten A gibt es einen Automaten, der die Sprache $L(A)^*$ akzeptiert.

Aufgabe 6.13. Überlegen Sie sich, wie ein Automat aussehen müsste, mit dem man Lemma 6.5 beweisen kann.

Insgesamt haben wir durch die Lemmata 5.2, 5.3, 5.4, 6.4 und 6.5 nun die folgende Aussage bewiesen.

Satz 6.6

Die Menge der regulären Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen Vereinigung, Durchschnitt, Komplement, Konkatenation und kleenesche Hülle.

Im Folgenden werden wir sehen, dass wir dadurch eine weitere Charakterisierung der regulären Sprachen erhalten haben.

Aufgabe 6.14. Nichtdeterministische Automaten liefern uns übrigens auch einen einfacheren Beweis für Lemma 5.3. Sehen Sie ihn?



7. Reguläre Ausdrücke

Eine weitere Möglichkeit, reguläre Sprachen zu beschreiben, sind die sogenannten *regulären Ausdrücke*. Die spielen in der Praxis der Informatik eine große Rolle, seit

¹⁷Sollten S_1 und S_2 nicht disjunkt sein, ersetze man sie durch die Mengen $S_1 \times \{1\}$ und $S_2 \times \{2\}$.

sie 1973 erstmals in dem [Unix-Tool grep](#) verwendet und später (ab 1987) durch die Programmiersprache [Perl](#) popularisiert wurden.¹⁸

Wir definieren zunächst [rekursiv](#) die [Syntax](#) solcher Ausdrücke, ohne über ihre Bedeutung zu sprechen:

Sei Σ ein Alphabet. Alle Zeichen von Σ sowie die Symbole¹⁹ ε und $\emptyset$ sind **reguläre Ausdrücke** über Σ . Sind ferner α und β irgendwelche regulären Ausdrücke über Σ , so sind auch die Zeichenketten $\alpha\beta$, $(\alpha + \beta)$ und $(\alpha)^*$ reguläre Ausdrücke über Σ .²⁰ Alle Zeichenketten, die nicht nach diesen Regeln gebildet wurden, sind *keine* regulären Ausdrücke.

Definition

Reguläre Ausdrücke über Σ_{bool} sind also z. B. ε , $\emptyset$, 0 , $\emptyset 1$, 00ε , $(\emptyset 1 + 00\varepsilon)$, $(11)^*$ oder $((10 + \varepsilon))^*$. Hingegen sind nach dieser Definition (10) , $()$ oder 1^{**} keine regulären Ausdrücke über Σ_{bool} .

Aufgabe 7.1. Sind $(10)^+$ und $(10)^3$ reguläre Ausdrücke über Σ_{bool} ?

Aufgabe 7.2. Man hätte reguläre Ausdrücke auch als [formale Sprache](#) definieren können. Machen Sie das, in dem Sie eine [Grammatik](#) für reguläre Ausdrücke über Σ_{bool} angeben. Geben Sie dabei explizit an, was die terminalen und die nichtterminalen Symbole sind. Worauf muss man achten? (Achtung: Es wird Ihnen nicht gelingen, eine *reguläre* Grammatik anzugeben.)

Wenn keine Gefahr von Missverständnissen besteht, lassen wir Klammern in regulären Ausdrücken weg und folgen dabei den aus der Schule bekannten Regeln für arithmetische Ausdrücke, wobei $(\alpha + \beta)$ wie Addition („Strichrechnung“), $\alpha\beta$ wie Multiplikation („Punktrechnung“) und $(\alpha)^*$ wie Exponentiation behandelt wird. Wir würden statt $((10 + \varepsilon))^*$ also $(10 + \varepsilon)^*$ schreiben, aber *nicht* $10 + \varepsilon^*$. Insbesondere soll sich ein Stern ohne Klammern immer nur auf das vorangehende Zeichen beziehen, d. h., dass ab^* für $a(b)^*$ steht und *nicht* für $(ab)^*$. (Das ist also wie bei Potenzen von Zahlen.)

Konvention

Nun kommen wir zur [Semantik](#) – also zur intendierten Bedeutung – der regulären Ausdrücke.

Jedem regulären Ausdruck α über Σ ordnen wir folgendermaßen eine Sprache $\mathcal{L}(\alpha)$ über Σ zu: Zunächst setzen wir $\mathcal{L}(\emptyset) = \emptyset$ und $\mathcal{L}(\varepsilon) = \{\varepsilon\}$ sowie

Definition

¹⁸Die dort eingesetzten regulären Ausdrücke sind aber mächtiger als die in diesem Kapitel besprochenen, da sie Erweiterungen wie z. B. Rückwärtsreferenzen unterstützen, die in den regulären Sprachen der Chomsky-Hierarchie nicht vorgesehen sind. Siehe Aufgabe 7.9.

¹⁹Wir setzen bei dieser Definition stillschweigend voraus, dass ε und $\emptyset$ keine Zeichen von Σ sind. Außerdem sind sie auf jeden Fall als *unterschiedliche Symbole* gemeint, wenn sie auch als mathematische Objekte (siehe Fußnote 8 auf Seite 7) gleich sein sollten.

²⁰In manchen Büchern wird auch das Zeichen $|$ statt $+$ benutzt.

$\mathcal{L}(a) = \{a\}$ für alle $a \in \Sigma$. Sind α und β beliebige reguläre Ausdrücke über Σ , so definieren wir ferner rekursiv

$$\begin{aligned}\mathcal{L}(\alpha\beta) &= \mathcal{L}(\alpha) \circ \mathcal{L}(\beta), \\ \mathcal{L}(\alpha + \beta) &= \mathcal{L}(\alpha) \cup \mathcal{L}(\beta) \quad \text{und} \\ \mathcal{L}(\alpha^*) &= \mathcal{L}(\alpha)^*.\end{aligned}$$

Beachten Sie, dass nach unserer Konvention beispielsweise 0^* zwar ein regulärer Ausdruck über Σ_{bool} ist, aber nach wie vor *kein* Wort. (Siehe auch Aufgabe 1.17.) Der reguläre Ausdruck 0^* beschreibt eine *Sprache* (ist aber keine Sprache).

Aufgabe 7.3. Geben Sie die Sprachen zu den folgenden drei regulären Ausdrücken über Σ_{bool} an: 11 , $11 + 0$, $(11 + 0)^*$.

Aufgabe 7.4. Welche der folgenden Aussagen sind wahr?

$$\begin{aligned}\mathcal{L}(00^*) &= \{0\}^+ \\ \mathcal{L}(00^*) &= \{00\}^* \\ \mathcal{L}(a(b+c)) &= \{ac, ab\} \\ \mathcal{L}(a(b+c)) &= \{ba, bc\} \\ \mathcal{L}(a(b+c)) &= \{abc\} \\ \mathcal{L}(0(0+0^*)) &= \mathcal{L}(0^*0)\end{aligned}$$

$\alpha \equiv \beta$ Für $\mathcal{L}(\alpha) = \mathcal{L}(\beta)$ werden wir abkürzend $\alpha \equiv \beta$ schreiben. Einige „Rechenregeln“ sollten offensichtlich sein:

Lemma 7.1

Sind α , β und γ reguläre Ausdrücke über derselben Sprache, dann gilt:

- | | |
|--|--|
| (i) $(\alpha + \beta) + \gamma \equiv \alpha + (\beta + \gamma)$ | Assoziativität von + |
| (ii) $(\alpha\beta)\gamma \equiv \alpha(\beta\gamma)$ | Assoziativität der „Multiplikation“ |
| (iii) $\alpha + \beta \equiv \beta + \alpha$ | Kommutativität von + |
| (iv) $\alpha\beta + \alpha\gamma \equiv \alpha(\beta + \gamma)$ | Distributivität |
| (v) $\alpha + \alpha \equiv \alpha$ | Idempotenz von + |
| (vi) $\alpha + \emptyset \equiv \alpha$ | Neutrales Element bzgl. + |
| (vii) $\alpha\emptyset \equiv \emptyset$ | Absorbierendes Element der „Multiplikation“ |
| (viii) $\alpha\varepsilon \equiv \alpha$ | Neutrales Element bzgl. der „Multiplikation“ |

Aufgabe 7.5. Machen Sie sich anhand von Beispielen klar, dass die Aussagen aus Lemma 7.1 alle korrekt sind. Überlegen Sie sich dann, ob die folgenden „Rechenregeln“ für Sprachen über $\{a, b\}$ bzw. über Σ_{bool} ebenfalls richtig sind, und geben Sie ggf. Gegenbeispiele an.

- (i) $a(a + a^*) \equiv a^*a$
- (ii) $a(b + a^*) \equiv a^*b$
- (iii) $(ab^* + ba^*) \equiv (a + b)(a + b)^*$
- (iv) $(a + b)^* \equiv (a^* + b^*)^*$

- (v) $(01)^* \equiv 0^* 1^*$
- (vi) $(10)^* \equiv (1^* 0^*)^*$
- (vii) $(0 + 1)^* \equiv 0^* + 1^*$
- (viii) $1^* \equiv (1^*)^*$

Aufgabe 7.6. Beschreiben Sie die drei folgenden Sprachen jeweils durch reguläre Ausdrücke:

$$L_1 = \{w \in \Sigma_{\text{bool}}^* : |w| = 3\}$$

$$L_2 = \{w \in \Sigma_{\text{bool}}^* : |w|_1 \text{ ist gerade}\}$$

$$L_2 = \Sigma_{\text{bool}}^+ \setminus \{0\}$$

Reguläre Ausdrücke sind so definiert, dass man mit ihnen nur reguläre Sprachen beschreiben kann. Das lässt sich ganz einfach begründen.

Zu jedem regulären Ausdruck α über Σ gibt es einen endlichen Automaten, der die Sprache $\mathcal{L}(\alpha)$ akzeptiert.

Lemma 7.2

Beweis. Für die einfachen regulären Ausdrücke $\emptyset$, ε und $a \in \Sigma$ ist es trivial, Automaten zu konstruieren:



Und wenn es Automaten für $\mathcal{L}(\alpha)$ und $\mathcal{L}(\beta)$ gibt, dann gibt es nach Lemma 5.3 einen für $\mathcal{L}(\alpha + \beta)$, nach Lemma 6.4 einen für $\mathcal{L}(\alpha\beta)$ und nach Lemma 6.5 einen für $\mathcal{L}(\alpha^*)$. Das war's schon. ■

Aufgabe 7.7. Geben Sie einen deterministischen endlichen Automaten an, der die durch den regulären Ausdruck $(a^* + b)b$ beschriebene Sprache akzeptiert. (Sie müssen dabei nicht das umständliche Verfahren aus dem Beweis von Lemma 7.2 anwenden.)

Reguläre Ausdrücke sind allerdings auch expressiv genug, um *alle* reguläre Sprachen zu beschreiben.

Die regulären Sprachen über einem Alphabet Σ sind genau die Sprachen, die von regulären Ausdrücken über Σ beschrieben werden.

Satz 7.3

Beweis. Eine Hälfte dieser Aussage wurde in Lemma 7.2 bereits bewiesen. Für die andere Hälfte geben wir uns einen Automaten $A = (\{s_1, \dots, s_n\}, \Sigma, \delta, s_1, F)$ vor und definieren für $i, j \in \{1, \dots, n\}$ und $k \in \{0, \dots, n\}$ die Sprache $R(i, j, k)$ als die Menge aller Wörter, die vom Zustand s_i zum Zustand s_j führen, ohne dass dazwischen Zustände s_m mit $m > k$ vorkommen.

Wenn man diese Definition verstanden hat, kann man sich leicht die folgenden Zusammenhänge klarmachen:

$$R(i, j, 0) = \begin{cases} \{a \in \Sigma : \delta(s_i, a) = s_j\} & i \neq j \\ \{a \in \Sigma : \delta(s_i, a) = s_j\} \cup \{\varepsilon\} & i = j \end{cases} \quad (7.1)$$

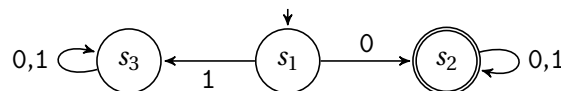
$$R(i, j, k+1) = R(i, j, k) \cup (R(i, k+1, k) \circ R(k+1, k+1, k)^* \circ R(k+1, j, k)) \quad (7.2)$$

$$L(A) = \bigcup_{\substack{j \in \{1, \dots, n\} \\ s_j \in F}} R(1, j, n) \quad (7.3)$$

Im Detail besagt (7.1), dass $R(i, j, 0)$ nur aus Symbolen bestehen kann, die an Pfeilen stehen, die direkt von s_i auf s_j zeigen, weil die Null im dritten Argument „Umwege“ über weitere Zustände verbietet. Wir müssen nur beachten, dass der Fall $i = j$ nicht verboten ist und dann noch das leere Wort hinzukommt. (7.2) besagt, dass wir einen Weg von s_i nach s_j , der (evtl. mehrfach) über den Zustand s_{k+1} führt, zerlegen können in Teile, die bei diesem Zustand starten oder enden (oder beides) und zwischendurch nur Zustände mit kleineren Indizes besuchen. Und Gleichung (7.3) ist schlicht und einfach eine andere Art, die Definition von $L(A)$ aufzuschreiben.

Entscheidend sind die folgenden Beobachtungen: Wir können $L(A)$ durch endlich viele Sprachen der Form $R(1, j, n)$ beschreiben und diese wiederum mittels (7.2) in andere Sprachen dieser Art zerlegen, die wir analog erneut zerlegen können und so weiter. Bei jeder solchen Zerlegung wird aber das dritte Argument kleiner, so dass der Prozess irgendwann beendet sein wird. Das geschieht dann, wenn nur noch Sprachen der Form $R(i, j, 0)$ verbleiben, die nach (7.1) eine einfache Form haben. Solche Sprachen lassen sich offensichtlich durch reguläre Ausdrücke beschreiben. Und bei der Rekonstruktion von $L(A)$ mittels (7.2) und (7.3) werden nur Konkatenation, kleenesche Hülle und Vereinigung benötigt, was wir auch jeweils mithilfe von regulären Ausdrücken beschreiben können. So erhalten wir am Ende einen regulären Ausdruck für die vom Automaten akzeptierte Sprache. ■

Wir gehen den Beweis noch einmal anhand des folgenden Automaten A durch:



Weil es nur den einen akzeptierenden Zustand s_2 gibt, ist nach (7.3) die gesuchte Sprache, für die wir einen regulären Ausdruck konstruieren wollen, $R(1, 2, 3)$. Die-

sen Ausdruck wandeln wir mittels (7.2) um, auf die so erhaltenen Teilausdrücke wenden wir wieder (7.2) an und so weiter. Das sieht so aus:

$$\begin{aligned}
 R(1,2,3) &= R(1,2,2) \cup (R(1,3,2) \circ R(3,3,2)^* \circ R(3,2,2)) \\
 R(3,3,2) &= R(3,3,1) \cup (R(3,2,1) \circ R(2,2,1)^* \circ R(2,3,1)) \\
 R(3,2,2) &= R(3,2,1) \cup (R(3,2,1) \circ R(2,2,1)^* \circ R(2,2,1)) \\
 R(1,3,2) &= R(1,3,1) \cup (R(1,2,1) \circ R(2,2,1)^* \circ R(2,3,1)) \\
 R(1,2,2) &= R(1,2,1) \cup (R(1,2,1) \circ R(2,2,1)^* \circ R(2,2,1)) \\
 R(3,3,1) &= R(3,3,0) \cup (R(3,1,0) \circ R(1,1,0)^* \circ R(1,3,0)) \\
 R(3,2,1) &= R(3,2,0) \cup (R(3,1,0) \circ R(1,1,0)^* \circ R(1,2,0)) \\
 R(2,3,1) &= R(2,3,0) \cup (R(2,1,0) \circ R(1,1,0)^* \circ R(1,3,0)) \\
 R(2,2,1) &= R(2,2,0) \cup (R(2,1,0) \circ R(1,1,0)^* \circ R(1,2,0)) \\
 R(2,1,1) &= R(2,1,0) \cup (R(2,1,0) \circ R(1,1,0)^* \circ R(1,1,0)) \\
 R(1,3,1) &= R(1,3,0) \cup (R(1,1,0) \circ R(1,1,0)^* \circ R(1,3,0)) \\
 R(1,2,1) &= R(1,2,0) \cup (R(1,1,0) \circ R(1,1,0)^* \circ R(1,2,0))
 \end{aligned} \tag{7.4}$$

Nun haben wir nur noch Ausdrücke der Form $R(i, j, 0)$ vor uns, auf die wir (7.3) anwenden, um das Ergebnis dann sofort in einen regulären Ausdruck zu konvertieren. Das ergibt

$$\begin{aligned}
 R(1,1,0) &= \{\varepsilon\} = \mathcal{L}(\varepsilon), \\
 R(1,2,0) &= \{0\} = \mathcal{L}(0), \\
 R(1,3,0) &= \{1\} = \mathcal{L}(1), \\
 R(3,3,0) &= R(2,2,0) = \Sigma_{\text{bool}} \cup \{\varepsilon\} = \mathcal{L}(0 + 1 + \varepsilon) \quad \text{und} \\
 R(i, j, 0) &= \emptyset = \mathcal{L}(\emptyset) \quad \text{für alle anderen Fälle.}
 \end{aligned}$$

Das setzen wir nun „rückwärts“ in die Gleichungen aus (7.4) wieder ein, wandeln in reguläre Ausdrücke um und vereinfachen diese gleich:

$$\begin{aligned}
 R(1,2,1) &\rightarrow 0 + \varepsilon \varepsilon^* 0 \equiv 0 \\
 R(1,3,1) &\rightarrow 1 + \varepsilon \varepsilon^* 1 \equiv 1 \\
 R(2,1,1) &\rightarrow \emptyset + \emptyset \varepsilon^* \varepsilon \equiv \emptyset \\
 R(2,2,1) &\rightarrow (0 + 1 + \varepsilon) + \emptyset \varepsilon^* 0 \equiv 0 + 1 + \varepsilon \\
 R(2,3,1) &\rightarrow \emptyset + \emptyset \varepsilon^* 1 \equiv \emptyset \\
 R(3,2,1) &\rightarrow \emptyset + \emptyset \varepsilon^* 0 \equiv \emptyset \\
 R(3,3,1) &\rightarrow (0 + 1 + \varepsilon) + \emptyset \varepsilon^* 1 \equiv 0 + 1 + \varepsilon \\
 R(1,2,2) &\rightarrow 0 + 0(0 + 1 + \varepsilon)^*(0 + 1 + \varepsilon) \equiv 0(0 + 1)^* \\
 R(1,3,2) &\rightarrow 1 + 0(0 + 1 + \varepsilon)^* \emptyset \equiv 1 \\
 R(3,2,2) &\rightarrow \emptyset + \emptyset \circ (0 + 1 + \varepsilon)^*(0 + 1 + \varepsilon) \equiv \emptyset \\
 R(3,3,2) &\rightarrow (0 + 1 + \varepsilon) + \emptyset(0 + 1 + \varepsilon)^* \emptyset \equiv 0 + 1 + \varepsilon \\
 R(1,2,3) &\rightarrow 0(0 + 1)^* + 1(0 + 1 + \varepsilon) \emptyset \equiv 0(0 + 1)^*
 \end{aligned}$$

Der letzte Schritt liefert $L(A) = \mathcal{L}(0(0+1)^*)$ und das Beispiel demonstriert gleichzeitig, wie aufwendig dieser Prozess bereits für einen Automaten mit nur drei Zuständen ist. Aber das macht nichts, weil es ja nur darum ging, die prinzipielle Machbarkeit zu beweisen. Wir haben nun jedenfalls drei verschiedene Möglichkeiten, dieselben Sprachen zu beschreiben: durch Grammatiken, durch Automaten oder durch reguläre Ausdrücke.

Aufgabe 7.8. Kfz-Kennzeichen bestehen aus dem sogenannten *Unterscheidungszeichen* (bis zu drei Buchstaben, die im allgemeinen für den Verwaltungsbezirk der Zulassungsbehörde stehen) gefolgt von einem Bindestrich gefolgt von ein oder zwei Buchstaben gefolgt von einem Leerzeichen gefolgt von ein bis vier Ziffern. Die Gesamtanzahl der Buchstaben und Ziffern darf jedoch acht nicht überschreiten. Es sind nur deutsche Großbuchstaben außer den Umlauten zugelassen und die erste Ziffer darf keine Null sein.

Geben Sie einen regulären Ausdruck für zulässige Kennzeichen des Landkreises Regen an. Dabei gehen wir davon aus, dass alle Kennzeichen, deren Unterscheidungszeichen REG ist und die den obigen Regeln entsprechen, zulässig sind.²¹

Setzen Sie Ihren regulären Ausdruck aus einfacheren regulären Ausdrücken zusammen, damit er nicht zu lang und unübersichtlich wird.

★**Aufgabe 7.9.** Die meisten Programmiersprachen, die reguläre Ausdrücke unterstützen, verwenden die *Syntax von Perl*.²² (Es gibt diverse Tools, mit denen man diese doch recht gewöhnungsbedürftige „Sprache“ lernen kann, z. B. *The Regex Coach*.) Geben Sie einen regulären Ausdruck in Perl-Syntax an, durch den eine Sprache beschrieben wird, die *nicht* regulär im Sinne des Skripts ist.

★**Aufgabe 7.10.** Implementieren Sie den regulären Ausdruck aus Aufgabe 7.8 in der in PERL-Syntax und probieren Sie ihn aus. (Siehe dazu die Links in Aufgabe 7.9.)

Aufgabe 7.11. *FLACI* (siehe Aufgabe 5.7) kann reguläre Ausdrücke in Automaten umwandeln und umgekehrt. (Beachten Sie allerdings, dass dort eine andere Syntax verwendet wird. Siehe Aufgabe 7.9.) Experimentieren Sie damit. Wenn Sie beispielsweise einen regulären Ausdruck in einen Automaten und diesen wieder in einen regulären Ausdruck verwandeln, wird der resultierende Ausdruck manchmal komplizierter als der ursprüngliche sein.

★**Aufgabe 7.12.** Ein „klassisches“ Programm, das reguläre Ausdrücke verwendet, ist *lex*. Wenn Sie daran interessiert sind, wie diese Ausdrücke in der Praxis der Informatik eingesetzt werden, dann schauen Sie sich das vielleicht einmal an.

8. Relationen

Aus der Mathevorlesung kennen Sie den Begriff der *Relation* für eine Menge von Paaren.²³ Man macht sich jedoch häufig nicht klar, wie oft man es mit Relationen

²¹In der Realität ist das ein bisschen komplizierter. Siehe http://de.wikipedia.org/wiki/Kfz-Kennzeichen_%28Deutschland%29.

²²Beachten Sie jedoch, dass diese Syntax in Prüfungen nicht verwendet werden kann!

²³Falls Sie nicht mehr wissen, was mit Begriffen wie *Identität* oder *Komposition* gemeint ist, schauen Sie im *Matheskript* nach.

zu tun hat. Schreibt man beispielsweise $a < b$ für zwei reelle Zahlen a und b , so meint man damit eigentlich, dass das Paar (a, b) ein Element der Relation

$$\{(x, y) \in \mathbb{R} \times \mathbb{R} : y - x \text{ ist positiv}\} \quad (8.1)$$

ist. Und im Falle einer Grammatik (N, T, P, S) hatten wir uns auf die Schreibweise $\alpha \rightarrow \beta$ für Produktionen geeinigt, wobei das lediglich eine Abkürzung dafür war, dass (α, β) ein Element der Relation P ist. Wir legen ein paar grundlegende Begriffe zu Relationen fest.

Ist A irgendeine Menge und $R \subseteq A \times A$, so nennt man R eine **Relation auf A** . Statt $(x, y) \in R$ schreibt man häufig xRy .²⁴ Gilt xRx für alle $x \in A$, so nennt man R **reflexiv auf A** . Folgt aus xRy immer yRx , so nennt man R **symmetrisch**. Folgt aus xRy und yRz immer xRz , so nennt man R **transitiv**.

Definition

Beispielsweise ist die Teilbarkeitsrelation

$$\{(m, n) \in \mathbb{N}^2 : m \text{ teilt } n\}$$

reflexiv auf $\mathbb{N}$ (denn jede Zahl teilt sich selbst) und – Sie entsinnen sich hoffentlich noch – transitiv. Sie ist aber *nicht* symmetrisch, denn beispielsweise ist 3 ein Teiler von 27, aber 27 kein Teiler von 3. Und die Relation aus (8.1) ist transitiv, aber weder symmetrisch noch reflexiv auf $\mathbb{R}$.

Für ein weiteres Beispiel nennen wir zwei natürliche Zahlen „ziffernfremd“, wenn ihre Dezimaldarstellungen keine gemeinsame Ziffer haben. 42 und 378 sind beispielsweise ziffernfremd, 189 und 378 aber nicht, weil in beiden Zahlen die Ziffer 8 vorkommt. Die Relation

$$\{(m, n) \in \mathbb{N}^2 : m \text{ und } n \text{ sind ziffernfremd}\}$$

ist offensichtlich symmetrisch, aber weder reflexiv auf $\mathbb{N}$ noch transitiv.

Aufgabe 8.1. Ist Ihnen klar, warum Ziffernfremdheit nicht transitiv ist? Können Sie ein Beispiel angeben?

Aufgabe 8.2. Sei $\Sigma = \{a, b, c\}$. Wir betrachten die folgenden Relationen auf Σ^* :

$$R_1 = \{(w, wv) : w, v \in \Sigma^*\}$$

$$R_2 = \{(w, wv) : w \in \Sigma^* \wedge v \in \Sigma^+\}$$

$$R_3 = \{(w, w^R) : w \in \Sigma^*\}$$

$$R_4 = \{(w_1, w_2) \in \Sigma^+ \times \Sigma^+ : |w_1| = |w_2|\}$$

Dabei soll w^R das Wort sein, das sich ergibt, wenn man die Reihenfolge der Symbole in w umkehrt. $(abb)^R$ ist also bba und $(aba)^R$ ist aba . ϵ^R soll natürlich ϵ sein.

Welche dieser Relationen sind symmetrisch, welche sind transitiv und welche sind reflexiv auf Σ^* ?

²⁴Das Zeichen zwischen x und y ist also gleichzeitig die Relation.

Reflexivität einer Relation R auf einer Menge A bedeutet offenbar, dass id_A eine Teilmenge von R ist. Daher ist $R \cup \text{id}_A$ die *kleinste* Relation oberhalb von R , die reflexiv auf A ist.²⁵ Es gibt auch eine kleinste Relation oberhalb von R , die transitiv ist. Man erhält sie, indem man sukzessive Kompositionen von R mit sich selbst zu R hinzufügt: $R \circ R$, $R \circ R \circ R$ und so weiter. Insgesamt ergibt sich:

Lemma 8.1

Ist R eine Relation auf A , so gibt es eine eindeutig bestimmte Relation R^* auf A mit den folgenden Eigenschaften:

- (i) R^* ist eine Obermenge von R .
- (ii) R^* ist reflexiv auf A .
- (iii) R^* ist transitiv.
- (iv) Jede Relation, die ebenfalls die ersten drei Eigenschaften hat, ist eine Obermenge von R^* .

R^* wird die **reflexive und transitive Hülle von R** genannt.

Aufgabe 8.3. Auf Seite 12 wurde die Relation $\Rightarrow_G$ definiert. Machen Sie sich klar, dass die dort ebenfalls definierte Relation $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$ ist.

★**Aufgabe 8.4.** Ist A endlich, so lässt sich die transitive Hülle effizient berechnen. Informieren Sie sich über den [Algorithmus von Floyd und Warshall](#) und realisieren Sie ihn in einer Programmiersprache Ihrer Wahl.

Definition

Eine Relation R auf einer Menge A , die reflexiv auf A , symmetrisch und transitiv ist, nennt man eine **Äquivalenzrelation auf A** . Für $a \in A$ nennt man die Menge

$$[a]_R = \{b \in A : aRb\}$$

dann die **Äquivalenzklasse** von a (bezüglich R).

Aus der Mathevorlesung kennen Sie das schon. Beispielsweise wurde dort durch

$$R = \{(m, n) \in \mathbb{Z} : m \bmod 7 = n \bmod 7\} \quad (8.2)$$

eine Äquivalenzrelation auf $\mathbb{Z}$ definiert. Für die Äquivalenzklassen hätten wir in diesem Fall $[m]_7$ statt $[m]_R$ geschrieben. $[3]_7$ enthält beispielsweise die Elemente 3, 10, 143, -4 und noch unendlich viele andere. Insgesamt hat diese Relation sieben Äquivalenzklassen.

Aufgabe 8.5. Welche der Relationen aus Aufgabe 8.2 ist eine Äquivalenzrelation auf der Menge Σ^* ?

Aufgabe 8.6. Ist $R = \{(v, w) \in \Sigma_{\text{bool}}^* \times \Sigma_{\text{bool}}^* : |v|_0 = |w|_0 \vee |v|_1 = |w|_1\}$ eine Äquivalenzrelation auf Σ_{bool}^* ?

²⁵Damit ist gemeint, dass jede Relation auf A , die reflexiv auf A und eine Obermenge von R ist, eine Obermenge von $R \cup \text{id}_A$ sein muss.

Aufgabe 8.7. Ist die durch²⁶

$$R = \{(v, w) \in \Sigma_{\text{bool}}^* \times \Sigma_{\text{bool}}^* : ||v| - |w|| \leq 1\}$$

definierte Relation eine Äquivalenzrelation auf Σ_{bool}^* ?

Für jede Menge A gibt es immer mindestens zwei Äquivalenzrelationen. Die eine ist die „triviale“ Relation $A \times A$. Die andere ist die Gleichheitsrelation

$$\{(x, y) \in A \times A : x = y\},$$

die bekanntlich als *Identität* id_A bezeichnet wird. Letztere ist gewissermaßen der Prototyp für den Begriff der Äquivalenzrelation. Mit den drei in der Definition genannten Eigenschaften wird die Essenz dessen, was das Konzept der *Gleichheit* ausmacht, herausgearbeitet.

Aufgabe 8.8. Wie viele Äquivalenzklassen hat $A \times A$ und wie sehen diese aus? Und wie sieht das für id_A aus?

Aufgabe 8.9. Sei $A = \{w \in \Sigma_{\text{bool}}^* : |w| \leq 2\}$ und $R = \{(v, w) \in A \times A : |v| = |w|\}$. Überzeugen Sie sich, dass R eine Äquivalenzrelation auf A ist, und geben Sie alle Äquivalenzklassen von R an.

Aufgabe 8.10. Gehen Sie die bisherigen Beispiele und Aufgaben durch und finden Sie mindestens einen Fall, in dem die Grundmenge unendlich ist, es aber nur endlich viele Äquivalenzklassen gibt. Geben Sie außerdem eine Äquivalenzrelation an, die unendlich viele Äquivalenzklassen hat.

Im Folgenden benötigen wir diese Eigenschaft von Äquivalenzrelationen:

Ist R eine Äquivalenzrelation auf der Menge A , dann ist jedes Element von A in genau einer Äquivalenzklasse enthalten und die Äquivalenzklassen sind paarweise disjunkt.

Lemma 8.2

Beweis. Ist $a \in A$, so ist wegen der Reflexivität a natürlich in $[a]_R$ enthalten. Sei nun a auch Element von $[b]_R$. Ist c nun ein beliebiges Element von $[a]_R$, so gilt nach Definition aRc . Wegen $a \in [b]_R$ gilt ferner bRa . Wegen der Transitivität folgt bRc , was nach Definition $c \in [b]_R$ impliziert. Damit haben wir $[a]_R \subseteq [b]_R$ gezeigt. Mit der Symmetrie folgt aus $a \in [b]_R$ aber auch $b \in [a]_R$ und man kann analog $[b]_R \subseteq [a]_R$ zeigen. Damit hat man insgesamt $[a]_R = [b]_R$, d. h., $[b]_R$ ist dieselbe Menge wie $[a]_R$. Es gibt somit keine andere Äquivalenzklasse, in der a liegt.

Gibt es umgekehrt ein Element c von A , das Element von $[a]_R$ und auch von $[b]_R$ ist, so gilt nach Definition aRc und bRc und wegen der Symmetrie cRb . Mit der Transitivität folgt aRb und damit wie im ersten Teil des Beweises $[a]_R = [b]_R$. Haben also zwei Äquivalenzklassen auch nur ein gemeinsames Element, dann sind sie

²⁶Es sollte offensichtlich sein, dass mit den inneren vertikalen Strichen die *Länge* von Wörtern gemeint sein muss und mit den äußeren der *Betrag* einer Zahl. Es wird zwar dasselbe Zeichen verwendet, aber aus dem Kontext geht klar hervor, dass es nur so Sinn ergibt.

schon gleich. Daher müssen zwei *verschiedene* Äquivalenzklassen immer disjunkt sein. ■

Es gilt auch die Umkehrung von Lemma 8.2: Wenn eine Menge in lauter disjunkte Teilmengen zerfällt, dann gibt es eine Äquivalenzrelation auf dieser Menge, die sozusagen dafür verantwortlich ist – ihre Äquivalenzklassen sind genau diese disjunkten Teilmengen. Das brauchen wir aber nicht. Weitergehende Informationen zu Relationen (die auch für die Theorie der [relationalen Datenbanken](#) wichtig sind) nebst Beweisen finden Sie im "alten" Matheskript.

9. Der Satz von Myhill-Nerode



Wie bereits erwähnt liefert das [Pumping-Lemma](#) nur eine notwendige Bedingung für die Regularität und wir können es daher nicht verwenden, um zu begründen, dass eine Sprache regulär *ist*. Ein notwendiges *und* hinreichendes Kriterium liefert ein Satz, der auf Arbeiten der Mathematiker [John Myhill](#) und [Anil Nerode](#) aus Großbritannien bzw. den USA basiert.

Nerode-Relation

Deren Idee war die folgende: Für eine Sprache L über einem Alphabet Σ definieren wir $w_1 \sim_L w_2$ für Wörter $w_1, w_2 \in \Sigma^*$ (die sogenannte *Nerode-Relation* für L) durch

$$\forall v \in \Sigma^* (w_1 v \in L \Leftrightarrow w_2 v \in L)$$

äquivalent

und sagen in diesem Fall, dass w_1 und w_2 *äquivalent*²⁷ sind. In Worten: Sind zwei Wörter äquivalent, dann heißt das, dass jeder Versuch, die beiden Wörter durch Anhängen eines dritten Wortes zu verlängern,²⁸ dazu führt, dass die verlängerten Wörter entweder beide zu L gehören oder beide nicht. Gibt es auch nur ein einziges Wort v , so dass $w_1 v$ in L liegt und $w_2 v$ nicht (oder umgekehrt), dann sind w_1 und w_2 nicht äquivalent.

Aufgabe 9.1. Machen Sie sich klar, dass $\sim_L$, also die Menge

$$\{(v, w) \in \Sigma^* \times \Sigma^* : v \sim_L w\},$$

eine [Äquivalenzrelation](#) auf Σ^* ist.

Für Äquivalenzklassen der Nerode-Relation schreiben wir

$$[w]_L = \{v \in \Sigma^* : w \sim_L v\},$$

wobei w ein Wort aus Σ^* sein soll.

Nehmen wir als erstes Beispiel die Sprache $L_1 = \{0\} \circ \{1\}^*$. An das Wort 0 kann man beliebig viele Einsen anhängen (auch gar keine) und erhält wieder ein Wort der Sprache. Konkateniert man hingegen ein Wort, in dem mindestens eine Null

²⁷Man müsste eigentlich hinzufügen, bezüglich welcher Sprache sie äquivalent sind, aber das wird im Folgenden immer aus dem Kontext ersichtlich werden.

²⁸Dabei kann die „Verlängerung“ auch das Konkatenieren des leeren Wortes sein.

enthalten ist, ist das Resultat kein Wort von L_1 . Die gleiche Eigenschaft haben auch Wörter wie 01, 011 und so weiter. Es gilt also beispielsweise $0 \sim_{L_1} 011$ oder auch $01 \sim_{L_1} 01^7$ und insgesamt $[0]_{L_1} = L_1$. Das leere Wort ε hingegen hat logischerweise die Eigenschaft, dass die Wörter, die man anhängen kann, um Wörter aus L_1 zu erhalten, genau die Wörter von L_1 sind. Und offenbar ist in diesem Fall ε das einzige Wort mit dieser Eigenschaft, d. h., es gilt $[\varepsilon]_{L_1} = \{\varepsilon\}$. Alle anderen Wörter können nicht durch Anhängen weiterer Symbole zur Wörtern von L_1 gemacht werden. Ein Beispiel für ein solches Wort ist 1 und allgemein ergibt das $[1]_{L_1} = \Sigma_{\text{bool}}^+ \setminus L_1$. Insgesamt zerfällt die Menge aller Wörter somit in drei Klassen:²⁹

$$\Sigma_{\text{bool}}^* = [0]_{L_1} \dot{\cup} [\varepsilon]_{L_1} \dot{\cup} [1]_{L_1}$$

Als weiteres Beispiel betrachten wir $L_2 = \{0^n 1^n : n \in \mathbb{N}\}$. Jedes Wort, das nur aus Nullen besteht, kann zu einem Wort von L_2 ergänzt werden. Aber je nach Anzahl der Nullen führen unterschiedliche Ergänzungen zum Erfolg. 0^3 kann man durch Konkatenieren von 1^3 oder 01^4 zu einem Wort von L_2 machen, aber für 0^5 oder 0^2 würde das nicht funktionieren. Ebenso können Wörter der Form $0^n 1^m$ mit $0 < m \leq n$ zu Wörtern von L_2 gemacht werden, aber es gibt jeweils nur *eine* Möglichkeit: man muss 1^{n-m} konkatenieren. Alle anderen Wörter – solche, die mit einer Eins anfangen oder in denen eine Null auf eine Eins folgt – können nicht „gerettet“ werden: *jedes* Wort (auch das leere), das man an sie anhängt, macht aus ihnen ein Wort, das nicht zu L_2 gehört. Die Partition sieht in diesem Fall so aus:

$$\Sigma_{\text{bool}}^* = [1]_{L_2} \dot{\cup} \bigcup_{n \in \mathbb{N}} [0^n]_{L_2} \dot{\cup} \bigcup_{n \in \mathbb{N}^+} [0^n 1]_{L_2}$$

Aufgabe 9.2. Wie viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben sich für $L = \{ab\}$ und wie sehen sie aus?

Aufgabe 9.3. Wie viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben sich für $L = \{ab\}^*$ und wie sehen sie aus?

Aufgabe 9.4. Wie viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben sich für $L = \{0(011)^{2^n} : n \in \mathbb{N}\}$ und wie sehen sie aus?

Aufgabe 9.5. Begründen Sie, dass sich für $L = \{ww^R : w \in \{a, b\}^*\}$ unendlich viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben.

Aufgabe 9.6. Begründen Sie, dass sich für $L = \{0^n 1^m 0^{n-1} : m, n \in \mathbb{N}^+\}$ unendlich viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben.

Aufgabe 9.7. Geben Sie für $L = \{0w : w \in \Sigma_{\text{bool}}^* \wedge |w| \bmod 3 = 0\}$ alle Äquivalenzklassen der Nerode-Relation an und geben Sie zu jeder Äquivalenzklasse mindestens zwei Elemente an, wenn das möglich ist.

²⁹Der Punkt über dem Vereinigungszeichen ist kein Fliegendreck, sondern er steht für die *disjunkte Vereinigung*.

Aufgabe 9.8. Sei $L = \{0^n 10^n : n \in \mathbb{N}\}$. Welche der folgenden Aussagen sind wahr, welche sind falsch? Begründen Sie jeweils Ihre Antwort.

- (i) Für jedes Wort $w \in \Sigma_{\text{bool}}^*$ gibt es eine natürliche Zahl n mit $w \in [0^n 1]_L$.
- (ii) Alle Wörter aus Σ_{bool}^* , in denen keine Null vorkommt, befinden sich in derselben Äquivalenzklasse.
- (iii) Die Äquivalenzklasse $[01]_L$ enthält unendlich viele Wörter.
- (iv) Sind m und n zwei verschiedene natürliche Zahlen, dann sind die Äquivalenzklassen $[0^m 10^m]_L$ und $[0^n 10^n]_L$ unterschiedliche Mengen.

Das versprochene Kriterium liefert nun das folgende Theorem:

Satz 9.1

Satz von Myhill-Nerode

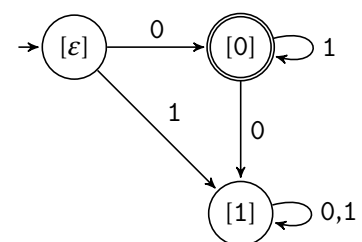
Eine Sprache ist genau dann regulär, wenn sie bezüglich der Nerode-Relation nur endlich viele Äquivalenzklassen hat.

Beweis. Zunächst der einfache Teil der Aussage: Ist eine Sprache über Σ regulär, dann gibt es einen deterministischen endlichen Automaten $(S, \Sigma, \delta, s_0, F)$, der sie akzeptiert. Jedes Wort $w \in \Sigma^*$ landet beim Durchlaufen des Automaten bei einem bestimmten Zustand, nämlich bei $s = \delta^*(s_0, w)$. Und offensichtlich sind alle Wörter, die ebenfalls bei s landen, bezüglich der Nerode-Relation äquivalent zu w . Da es nur endlich viele Zustände gibt, gibt es auch nur endlich viele Äquivalenzklassen.

Umgekehrt kann man zu einer Sprache L , die nur endlich viele Äquivalenzklassen hat, einen Automaten bauen, der diese Sprache akzeptiert. Die entscheidende Idee dabei ist, dass man die Klassen zu den Zuständen des Automaten macht. Der Rest ergibt sich dann mehr oder weniger von selbst. Der Startzustand ist natürlich $[\varepsilon]_L$ und akzeptierend müssen alle Zustände der Form $[w]_L$ für $w \in L$ sein. Ist a ein Symbol von Σ , dann setzt man $\delta([w]_L, a) = [wa]_L$ für alle $w \in \Sigma^*$. Es muss nur noch gezeigt werden, dass δ dadurch wohldefiniert ist, dass wir also keine widersprüchlichen Definitionen vorgenommen haben. Das folgt aber daraus, dass $w_1 \sim_L w_2$ offensichtlich $w_1 a \sim_L w_2 a$ impliziert, wie man sich leicht überlegt. ■

Den zweiten Teil des Beweises gehen wir noch anhand eines Beispiels durch. Dazu betrachten wir die Sprache $L_1 = \{0\} \circ \{1\}^*$ von oben. Deren Äquivalenzklassen $[0]_{L_1}$, $[\varepsilon]_{L_1}$ und $[1]_{L_1}$ werden zu Zuständen. $[\varepsilon]_{L_1}$ wird der Startzustand und $[0]_{L_1}$ ist der einzige akzeptierende Zustand, weil ε und 1 nicht zur Sprache L_1 gehören. Die Übergangsfunktion sieht folgendermaßen aus (wobei wir zur Abkürzung den Index an der rechten eckigen Klammer weglassen):

	$[\varepsilon]$	$[0]$	$[1]$
0	$[\varepsilon 0] = [0]$	$[00] = [1]$	$[10] = [1]$
1	$[\varepsilon 1] = [1]$	$[01] = [0]$	$[11] = [1]$



Aufgabe 9.9. Machen Sie dasselbe für die Sprache aus Aufgabe 9.3.

Aufgabe 9.10. Mit Satz 9.1 ist klar, dass die Sprache aus Aufgabe 9.5 nicht regulär ist. Begründen Sie auf die gleiche Art, dass $L = \{a^k b^k : k \in \mathbb{N}\}$ nicht regulär ist.³⁰

Ganz nebenbei liefert uns Satz 9.1 noch eine hilfreiche Folgerung. Es ist offensichtlich, dass es verschiedene Automaten gibt, die dieselbe Sprache akzeptieren. Es ist allerdings auch klar, dass es zu jeder Sprache einen sie akzeptierenden Automaten mit minimaler Anzahl von Zuständen geben muss, den sogenannten *Minimalautomaten* für diese Sprache. Warum man von *dem* (und nicht von *einem*) Minimalautomaten spricht, begründet die folgende Aussage.

Minimalautomat

Für jede reguläre Sprache gibt es (bis auf die Benennung der Zustände) nur *einen* Minimalautomaten. Er kann mittels der Nerode-Relation konstruiert werden.

Korollar 9.2

Beweis. Der im Beweis von Satz 9.1 konstruierte Automat muss aufgrund der Argumentation im ersten Absatz des Beweises eine minimale Anzahl von Zuständen haben, denn ein Automat mit weniger Zuständen hätte auch weniger Äquivalenzklassen. Dass es keinen anderen Automaten mit derselben Anzahl von Zuständen geben kann, folgt durch diesen Gedankengang ebenfalls: Da alle Wörter, die im selben Zustand landen, äquivalent sind und es in einem Automaten mit minimaler Anzahl von Zuständen nicht mehr Zustände als Äquivalenzklassen gibt, sind die Zustände faktisch die Äquivalenzklassen und für jedes Wort steht damit sein „Ziel“ fest. Dadurch ist die Übergangsfunktion eindeutig bestimmt. ■

Dieser Beweis klärt allerdings noch nicht, wie man aus einem vorhandenen Automaten (S, I, δ, s_0, F) den Minimalautomaten konstruieren kann, ohne den Umweg über die akzeptierte Sprache zu gehen. Dafür gibt es jedoch einen Algorithmus, der direkt die Nerode-Relation aufbaut und der aus zwei Teilen besteht:³¹

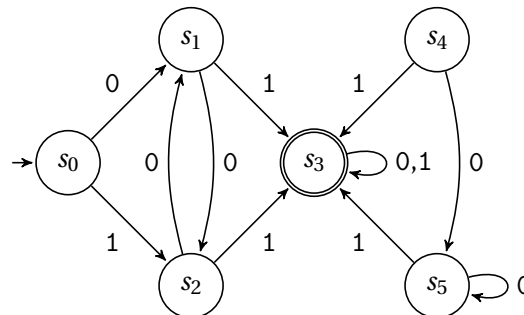
- (i) Erstelle eine Liste aller Zustände und streiche s_0 durch.
- (ii) Streiche alle Zustände durch, auf die ein Pfeil von einem durchgestrichenen Zustand zeigt, also alle $s \in S$, für die es ein durchgestrichenes $s' \in S$ und ein $a \in I$ mit $\delta(s', a) = s$ gibt.
- (iii) Wiederhole den letzten Schritt, bis sich nichts mehr ändert.
- (iv) Alle *nicht* gestrichenen Zustände können nun entfernt werden, weil sie vom Startzustand aus nicht erreicht werden können. Die Restmenge nennen wir S' .
- (v) Erstelle eine Tabelle für $S' \times S'$. Die Tabelle hat also für jeden verbleibenden Zustand sowohl eine Zeile als auch eine Spalte.
- (vi) Markiere alle Paare (s, s') aus $S' \times S'$, für die $s \in F$ und $s' \notin F$ oder $s' \in F$ und $s \notin F$ gilt.

³⁰Ja, das wissen wir eigentlich schon, weil sie als Beispiel hinter Lemma 4.1 vorkam.

³¹Hierbei geht es immer um den *ganzen* Automaten inklusive Fehlerzustand. Siehe Fußnote 8 auf Seite 22.

- (vii) Markiere alle Paare (s, s') und (s', s) aus $S' \times S'$, für die $(\delta(s, a), \delta(s', a))$ für ein $a \in I$ bereits markiert ist.
- (viii) Wiederhole den letzten Schritt, bis sich nichts mehr ändert.
- (ix) Alle Zustandspaare (s, s') mit $s \neq s'$, die nun *nicht* markiert sind, sind äquivalent³² und können durch einen der beiden Zustände ersetzt werden. Wird s' entfernt, so müssen alle Pfeile, die auf s' zeigten, nun auf s „umgeleitet“ werden. Aus zwischen s und s' verlaufenden Pfeilen werden Pfeile von s auf sich selbst.³³

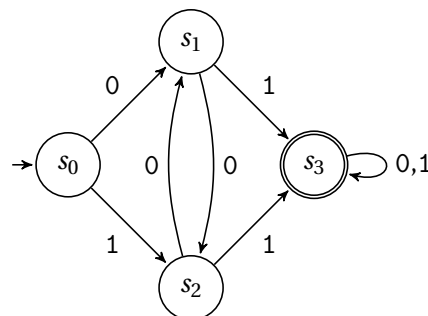
Wir ersparen uns den (nicht schweren, aber technischen) Beweis, dass dieser Algorithmus immer funktioniert, und schauen uns ein Beispiel an.



Im ersten Teil des Algorithmus sieht unsere Liste folgendermaßen aus:

- ~~s_0~~ , s_1 , s_2 , s_3 , s_4 , s_5
- ~~s_0~~ , ~~s_1~~ , ~~s_2~~ , s_3 , s_4 , s_5
- ~~s_0~~ , ~~s_1~~ , ~~s_2~~ , ~~s_3~~ , s_4 , s_5

Weil es danach keine Änderungen mehr gibt, können wir s_4 und s_5 entfernen und es verbleibt der folgende Automat.



Jetzt erstellen wir die Tabelle, die uns die Nerode-Relation liefern wird. (Da die Relation **symmetrisch** ist, können wir uns etwas Arbeit sparen. Das sind die grauen Quadrate.) Da nur s_3 ein Endzustand ist, kann s_3 zu keinem der anderen Zustände äquivalent sein.

³²Das ist etwas flapsig formuliert. Die Zustände sind in dem Sinne „äquivalent“, dass Wörter, die bei diesen Zuständen landen, bzgl. der Nerode-Relation äquivalent sind.

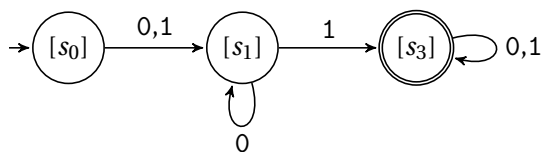
³³Dabei werden Dopplungen natürlich vermieden. Um die Pfeile, die von s' ausgehen, müssen wir uns nicht kümmern, weil s und s' äquivalent sind.

	s_0	s_1	s_2	s_3
s_0				×
s_1	■			×
s_2	■	■		×
s_3	■	■	■	

Man sieht nun, dass das Zeichen 1 von s_0 zu s_2 und von s_1 zu s_3 führt. Weil (s_2, s_3) markiert ist, muss (s_0, s_1) markiert werden. Analog muss (s_0, s_2) markiert werden, denn von den beiden führt 1 zum markierten Paar (s_2, s_3) .

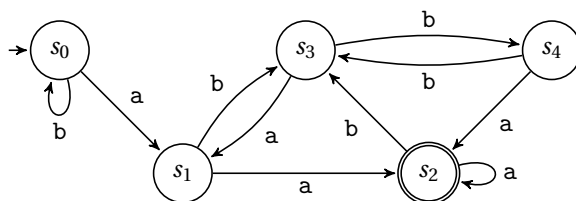
	s_0	s_1	s_2	s_3
s_0		×	×	×
s_1	■			×
s_2	■	■		×
s_3	■	■	■	

Ab hier gibt es keine weiteren Veränderungen und wir können der Tabelle entnehmen, dass s_1 und s_2 äquivalent sind. Wir entfernen s_2 und machen aus dem 1-Pfeil von s_0 nach s_2 einen von s_0 nach s_1 . Außerdem wird aus den 0-Pfeilen zwischen s_1 und s_2 ein Pfeil, der von s_1 auf sich selbst zeigt. Das ergibt schließlich den Minimalautomaten.



Aufgabe 9.11. FLACI (siehe Aufgabe 5.7) kann für Sie Minimalautomaten konstruieren. Damit können Sie den obigen Algorithmus an eigenen Beispielen ausprobieren.

Aufgabe 9.12. In einer (schon länger zurückliegenden) Stundenübung wurde dieser Automat zur Lösung einer Aufgabe vorgeschlagen:



Konstruieren Sie einen Minimalautomaten dazu und geben Sie für die akzeptierte Sprache einen regulären Ausdruck an.

Aufgabe 9.13. Die Sprache L wird durch den regulären Ausdruck $b + ab(bb)^*$ beschrieben. Wie viele Zustände hätte der Minimalautomat für diese Sprache? Beantworten Sie die Frage, ohne einen Automaten für L zu konstruieren.

Aufgabe 9.14. Geben Sie für die Sprache aus Aufgabe Aufgabe 9.4 einen regulären Ausdruck und die Anzahl der Zustände des Minimalautomaten an.

Aufgabe 9.15. War Ihre Lösung von Aufgabe 6.9 minimal? Können Sie Ihre Antwort begründen?

Aufgabe 9.16. Konstruieren Sie nach dem vorgestellten Algorithmus den Minimalautomaten für den Automaten $(\{s_0, \dots, s_5\}, \Sigma_{\text{bool}}, \delta, s_0, \{s_0, s_5\})$, bei dem die Übergangsfunktion δ durch

	s_0	s_1	s_2	s_3	s_4	s_5
0	s_3	s_1	s_1	s_5	s_1	s_4
1	s_1	s_4	s_4	s_1	s_4	s_5

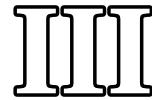
gegeben ist. Idealerweise arbeiten Sie nur mit der Tabelle, ohne Automaten zu zeichnen.

★Aufgabe 9.17. Über dem Alphabet $\Sigma = \{a, b, c\}$ sei mit

$$L_1 = \{a^m b^n c^n : m, n \in \mathbb{N}^+\} \quad \text{und} \quad L_2 = \{b^m c^n : m, n \in \mathbb{N}\}$$

die Sprache $L = L_1 \cup L_2$ gegeben.

- (i) Begründen Sie mit dem **Satz von Myhill-Nerode**, dass L nicht regulär ist.
- (ii) Begründen Sie, dass man mit dem **Pumping-Lemma** *nicht* beweisen kann, dass L nicht regulär ist.



Kontextfreie Sprachen

Wie wir im letzten Kapitel gesehen haben, sind die regulären Sprachen ziemlich einfach „gestrickt“. Für viele Anwendungen braucht man mehr Komplexität. Und die kontextfreien Sprachen, um die es nun gehen wird, sind in der Praxis – z. B. bei der Spezifikation der Syntax von Programmiersprachen – häufig das Mittel der Wahl. Sie erlauben mehr Flexibilität, sind aber gleichzeitig noch einfach genug, dass Computer gut mit ihnen umgehen können.



10. Kontextfreie Grammatiken

Die kontextfreien Grammatiken bilden in der Chomsky-Hierarchie die nächste Stufe nach den regulären.

Eine Grammatik (N, T, P, S) wird **kontextfrei** oder **Typ-2-Grammatik** genannt, wenn für alle Produktionen $(\alpha, \beta) \in P$ die Bedingung $\alpha \in N$ erfüllt ist. Eine Sprache wird **kontextfrei** genannt, wenn es eine kontextfreie Grammatik gibt, die sie erzeugt.

Definition

Offensichtlich bilden die regulären Grammatiken bzw. Sprachen über einem festen Alphabet eine Teilmenge ihrer kontextfreien Gegenstücke. Die Bezeichnung „kontextfrei“ rührt daher, dass jede Produktion unabhängig davon ausgeführt werden darf, in welchem Kontext ihre linke Seite auftritt. Hat eine Grammatik beispielsweise eine Produktion der Form $aX \rightarrow ab$, dann darf nach dieser Regel das nichtterminale Symbol X nur dann durch b ersetzt werden, wenn es rechts vom Symbol a steht. In kontextfreien Grammatiken sind solche Produktionen nicht erlaubt, weil links immer nur ein einziges nichtterminales Symbol stehen darf.

Die beiden ersten ernsthaften Beispiele in Abschnitt 2 – die Dyck-Sprache und die Sprache für primitive arithmetische Zuweisungen – sind offensichtlich kontextfrei. Gleichzeitig kann man sich mithilfe des [Satzes von Myhill-Nerode](#) leicht überzeugen, dass die Dyck-Sprache nicht regulär ist. (Siehe Aufgabe 10.1.) Die regulären Sprachen bilden also sogar eine *echte* Teilmenge der kontextfreien.

Aufgabe 10.1. Begründen Sie die Korrektheit der obigen Behauptung, dass die Dyck-Sprache nicht regulär ist.

Aufgabe 10.2. Geben Sie für die Sprache $\{a^m b^m c^n : m, n \in \mathbb{N}\}$ eine kontextfreie Grammatik an.

Für den Umgang mit kontextfreien Grammatiken ist eine einfachere Darstellung manchmal hilfreich:

Definition

Eine Grammatik (N, T, P, S) ist in **Chomsky-Normalform**, wenn alle Produktionen von einer der Formen $S \rightarrow \varepsilon$, $A \rightarrow a$ oder $A \rightarrow BC$ sind, wobei $a \in T$, $A \in N$ und $B, C \in N \setminus \{S\}$ gelten muss.

Aufgabe 10.3. Überlegen Sie sich, wie **Ableitungsbäume** für Grammatiken in Chomsky-Normalform aussehen.

Lemma 10.1

Zu jeder kontextfreien Sprache gibt es eine Grammatik in Chomsky-Normalform, die sie erzeugt.

Beweis. Wir geben ein Verfahren an, um eine kontextfreie Grammatik (N, T, P, S) sukzessive in eine in Chomsky-Normalform umzuwandeln, die dieselbe Sprache erzeugt. Das Verfahren besteht aus fünf Teilschritten, die in der angegebenen Reihenfolge abgearbeitet werden.

- (i) Füge ein neues Startsymbol S' sowie die Produktion $S' \rightarrow S$ hinzu.
- (ii) Alle Produktionen der Form $A \rightarrow \varepsilon$ mit $A \neq S'$ werden entfernt. Ist $A \rightarrow \varepsilon$ die einzige Produktion, auf deren linker Seite A steht, dann wird auf den rechten Seiten einfach überall A durch ε ersetzt. Gibt es noch weitere Produktionen mit linker Seite A , so werden zu jeder Produktion, in deren rechter Seite A vorkommt, ein oder mehrere ansonsten identische Produktionen hinzugefügt, bei denen A entfernt wurde.¹
Dieser Schritt muss evtl. mehrfach durchgeführt werden. Bei der Grammatik mit den Produktionen $S' \rightarrow S$, $S \rightarrow A \mid B$, $A \rightarrow \varepsilon$ und $B \rightarrow 1$ wird z. B. zunächst $A \rightarrow \varepsilon$ entfernt und aus $S \rightarrow A$ wird $S \rightarrow \varepsilon$. Dann muss aber die neue Produktion $S \rightarrow \varepsilon$ ebenfalls entfernt und die Produktion $S' \rightarrow \varepsilon$ hinzugefügt werden. Erst dann ist der Prozess beendet.
- (iii) Jedes terminale Symbol x , das nicht allein auf einer rechten Seite steht, wird dort durch ein neues nichtterminales Symbol V_x ersetzt und dafür wird die Produktion $V_x \rightarrow x$ hinzugefügt.
- (iv) Alle Produktionen der Form $A \rightarrow B_1 B_2 \dots B_n$ mit $n > 2$ werden mithilfe neuer nichtterminale Symbole A_i in Produktionen $A \rightarrow A_{n-1} B_n$, $A_{n-1} \rightarrow A_{n-2} B_{n-1}$ und so weiter bis $A_2 \rightarrow B_1 B_2$ zerlegt.

¹Potentiell mehrere deshalb, weil A wie z. B. in $B \rightarrow Ax A$ mehrfach auftreten kann. Man müsste dann $B \rightarrow Ax$, $B \rightarrow x A$ und $B \rightarrow x$ hinzufügen.

- (v) Produktionen der Form $A \rightarrow B$ mit $B \in N$ werden entfernt. Dafür wird für jede Produktion der Form $B \rightarrow \beta$ eine neue $A \rightarrow \beta$ hinzugefügt, wenn diese nicht bereits in diesem Schritt entfernt wurde.²

Damit ist der Prozess beendet. Man kann allerdings ggf. noch „sinnlose“ Produktionen entfernen, nämlich solche der Form $A \rightarrow \beta$ mit $A \neq S'$, für die A nie auf einer rechten Seite vorkommt. ■

Wir gehen ein Beispiel durch. Die kontextfreie Grammatik mit den sechs Produktionen $S \rightarrow AB \mid ABA$, $A \rightarrow 0A \mid 0$ und $B \rightarrow 1B \mid \varepsilon$ erzeugt die Sprache $\{0\}^+ \{1\}^* \{0\}^*$. Zuerst fügen wir $S' \rightarrow S$ hinzu und S' wird das neue Startsymbol.

$$S' \rightarrow S \quad S \rightarrow AB \mid ABA \quad A \rightarrow 0A \mid 0 \quad B \rightarrow 1B \mid \varepsilon$$

Dann wird $B \rightarrow \varepsilon$ entfernt. Dafür kommen $B \rightarrow 1$ sowie $S \rightarrow A \mid AA$ hinzu.

$$S' \rightarrow S \quad S \rightarrow AB \mid A \mid ABA \mid AA \quad A \rightarrow 0A \mid 0 \quad B \rightarrow 1B \mid 1$$

Anschließend werden Variablen für 0 und 1 eingeführt.

$$S' \rightarrow S \quad S \rightarrow AB \mid A \mid ABA \mid AA \quad A \rightarrow V_0 A \mid 0 \quad B \rightarrow V_1 B \mid 1 \\ V_0 \rightarrow 0 \quad V_1 \rightarrow 1$$

Dann wird $S \rightarrow ABA$ durch $S \rightarrow S_2 A$ und $S_2 \rightarrow AB$ ersetzt.

$$S' \rightarrow S \quad S \rightarrow AB \mid A \mid S_2 A \mid AA \quad A \rightarrow V_0 A \mid 0 \quad B \rightarrow V_1 B \mid 1 \\ V_0 \rightarrow 0 \quad V_1 \rightarrow 1 \quad S_2 \rightarrow AB$$

Jetzt geht es $S \rightarrow A$ an den Kragen.

$$S' \rightarrow S \quad S \rightarrow AB \mid S_2 A \mid AA \mid V_0 A \mid 0 \quad A \rightarrow V_0 A \mid 0 \quad B \rightarrow V_1 B \mid 1 \\ V_0 \rightarrow 0 \quad V_1 \rightarrow 1 \quad S_2 \rightarrow AB$$

Wir müssten nun noch aufwendig dasselbe mit $S' \rightarrow S$ machen, was zu diversen überflüssigen Produktionen führen würde. Wir ersparen uns das aber und machen S wieder zum Startsymbol, weil das in diesem Fall funktioniert.³ Das Ergebnis sieht dann so aus:

$$S \rightarrow AB \mid S_2 A \mid AA \mid V_0 A \mid 0 \quad A \rightarrow V_0 A \mid 0 \quad B \rightarrow V_1 B \mid 1 \\ V_0 \rightarrow 0 \quad V_1 \rightarrow 1 \quad S_2 \rightarrow AB$$

Es gibt viele verschiedene Chomsky-Normalformen zu einer vorgegebenen Sprache und man kann bei der Konstruktion auch anders vorgehen als hier gezeigt. Jede Methode hat ihre eigenen Vor- und Nachteile.

²Damit werden auch Artefakte wie $A \rightarrow A$ entfernt, die ggf. durch Schritt (ii) entstehen können. Man könnte bei der Elimination von „Ketten“ wie $A \rightarrow B$ auch wie in Schritt (ii) vorgehen, aber dann müsste man S' separat behandeln.

³Die Einführung eines neuen Startsymbols ist eigentlich nur dann nötig, wenn das leere Wort zur Sprache gehört. Wir haben den ersten Schritt hier lediglich aus didaktischen Gründen durchgeführt.

Aufgabe 10.4. Geben Sie eine Grammatik in Chomsky-Normalform für die Sprache aus Aufgabe 10.2 an.

Aufgabe 10.5. Erzeugen Sie für die Grammatik mit den Produktionen $S \rightarrow ASA \mid aB$, $A \rightarrow B \mid S$, $B \rightarrow b \mid \varepsilon$ eine Chomsky-Normalform mit dem Verfahren aus dem Beweis von Lemma 10.1.

Aufgabe 10.6. FLACI kann kontextfreie Grammatiken in Chomsky-Normalform überführen. Dabei kommen aber nicht unbedingt dieselben Grammatiken heraus, die der oben beschriebene Algorithmus erzeugt. Vielleicht probieren Sie es ja mal aus, vergleichen die Ergebnisse und überlegen sich, wie das Programm vorgeht.

Die Chomsky-Normalform wird für Verfahren wie den *CYK-Algorithmus* benötigt, die feststellen können, ob ein Wort zu einer kontextfreien Sprache gehört. So etwas werden wir uns zwar nicht anschauen, aber wir benötigen die Normalform für das folgende Lemma, das eine analoge Aussage zu Lemma 4.1 macht.

Lemma 10.2

Pumping-Lemma für kontextfreie Sprachen

Ist L eine kontextfreie Sprache, dann gibt es eine Zahl $n \in \mathbb{N}$, so dass sich für jedes Wort $z \in L$ mit $|z| \geq n$ Wörter u, v, w, x und y finden lassen, für die die folgenden vier Bedingungen gelten.

- (i) $z = uvwxy$
- (ii) $vx \neq \varepsilon$
- (iii) $|vwx| \leq n$
- (iv) $uv^kwx^ky \in L$ für alle $k \in \mathbb{N}$

Beweis. Wir wählen uns eine L erzeugende Grammatik (N, T, P, S) in Chomsky-Normalform, die es nach Lemma 10.1 geben muss, und setzen $n = 2^{|N|} + 1$. Die besondere Form der Grammatik sorgt dafür, dass für jedes Wort sein Ableitungsbaum ein Binärbaum ist. (Siehe Aufgabe 10.3.) Für Wörter, deren Länge mindestens n ist, muss so ein Baum entsprechend viele Blätter und damit mindestens die Höhe $|N| + 2$ haben.⁴ Daher gibt es einen Weg von der Wurzel S zu einem Blatt (mit einem Terminalsymbol), der über mindestens $|N| + 1$ Nichtterminalsymbole führt. Weil es nur $|N|$ solcher Symbole gibt, muss (mindestens) eines von denen auf diesem Weg mehrfach vorkommen. Es wird also schematisch so aussehen wie in der folgenden Skizze mit der Aufteilung wie in (i).

⁴Wie im verlinkten Wikipedia-Artikel (Lösung von Aufgabe 10.3) soll ein Binärbaum, der nur aus der Wurzel besteht, die Höhe 1 haben.

Der folgende Satz sagt etwas über die Abschlusseigenschaften kontextfreier Sprachen aus. Man vergleiche mit Satz 6.6.

Satz 10.3

Die Menge der kontextfreien Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen Vereinigung, Konkatenation und kleenesche Hülle.

Beweis. Wir gehen von zwei kontextfreien Sprachen L_1 und L_2 aus, wobei L_i von der kontextfreien Grammatik (N_i, Σ, P_i, S_i) erzeugt wird. O. B. d. A. seien N_1 und N_2 disjunkt. (Siehe Fußnote 17 auf Seite 30.)

Wir wählen ein neues Symbol S , das nicht in $N_1 \cup N_2$ enthalten ist, und definieren eine Grammatik durch

$$G_{\cup} = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{(S, S_1), (S, S_2)\}, S).$$

$G_{\cup}$ ist offensichtlich kontextfrei und erzeugt $L_1 \cup L_2$. Analog definieren wir

$$G_{\circ} = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{(S, S_1 S_2)\}, S)$$

als eine kontextfreie Grammatik, die $L_1 \circ L_2$ erzeugt. Zum dritten Teil des Beweises siehe Aufgabe 10.9. ■

Allerdings ist die Menge der kontextfreien Sprachen *nicht* abgeschlossen gegen Durchschnitt und Komplement! Das wird in den nun folgenden Aufgaben ausgearbeitet.

Aufgabe 10.9. Ergänzen Sie den fehlenden Teil des Beweises von Satz 10.3.

Aufgabe 10.10. Sei $L_1 = \{a^m b^m c^n : m, n \in \mathbb{N}\}$ und $L_2 = \{a^m b^n c^n : m, n \in \mathbb{N}\}$. Diese Sprachen sind kontextfrei. (Siehe Aufgabe 10.2.) Wie sieht $L_1 \cap L_2$ aus? Was schließen Sie daraus?

Aufgabe 10.11. Begründen Sie mithilfe von Aufgabe 10.10, dass $\Sigma^* \setminus L$ nicht notwendig kontextfrei ist, wenn L eine kontextfreie Sprache über dem Alphabet Σ ist. (Hinweis: Erinnern Sie sich an Aufgabe 5.16.)

Aufgabe 10.12. Falls Sie sich das gefragt haben: Wenn eine Sprache L_1 zu einer bestimmten Klasse gehört und L_2 eine Teilmenge von L_1 ist, dann muss L_1 deswegen nicht notwendig zu derselben Klasse gehören. Um dafür ein Gegenbeispiel zu finden, nehmen wir die Sprache $L_2 = \{a^k b^k : k \in \mathbb{N}\}$ über $\Sigma = \{a, b\}$, von der wir inzwischen wissen, dass sie kontextfrei, aber *nicht* regulär ist. Andererseits ist Σ^* natürlich regulär und L_2 eine Teilmenge davon.

Können Sie ein interessanteres Gegenbeispiel finden? Konkret: Geben Sie eine reguläre Sprache L_3 mit $L_2 \subseteq L_3 \neq \Sigma^*$ an.

11. Kellerautomaten

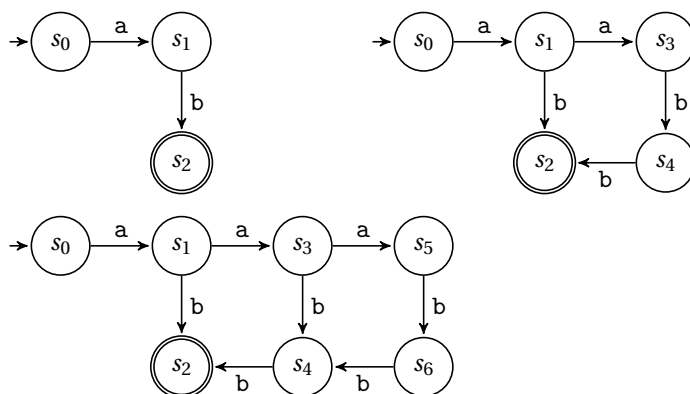
Wir wissen bereits, dass endliche Automaten nicht alle kontextfreien Sprachen akzeptieren können, z. B.

$$L_{\infty} = \{a^k b^k : k \in \mathbb{N}^+\}, \quad (11.1)$$

das erste Anwendungsbeispiel für das [Pumping-Lemma für reguläre Sprachen](#). Woran scheitert es? Schauen wir uns dafür die Sprachen der Form

$$L_n = \{a^k b^k : k \in \mathbb{N}^+ \text{ und } k \leq n\}$$

für $n \in \mathbb{N}^+$ an. Diese Sprachen *sind* offenbar alle regulär. Allerdings wächst der Aufwand für die Erstellung der Automaten mit größer werdendem n . Die folgenden Skizzen zeigen Automaten für L_1 , L_2 und L_3 .



Die Automaten müssen sich „merken“, wie oft das Symbol a schon vorkam, und die einzige Möglichkeit, das zu tun, ist ein Zustand. Im Automaten für L_3 steht s_3 beispielsweise für den Wortanfang a^2 und s_5 für a^3 . Würden wir so fortfahren, dann bräuchten wir unendlich viele Zustände für die Sprache L_{∞} und das ist bei *endlichen* Automaten natürlich nicht erlaubt. Und eine Erweiterung auf Automaten mit unendlich vielen Zuständen eignet sich zwar als mathematische Spielerei, würde aber nicht zu den Zielen der Theoretischen Informatik passen (und liefert nebenbei auch keine besonders interessante Theorie).

Stattdessen versieht man Automaten mit einem vergleichsweise primitiven „Speicher“ in Form eines [Stacks](#), den man auch *Stapel-* oder *Kellerspeicher* nennt.

Ein (**nichtdeterministischer**) **Kellerautomat** (engl. *pushdown automaton*, abgekürzt PDA) ist ein 7-Tupel $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ mit den folgenden Eigenschaften:

- (i) S ist eine nichtleere endliche Menge von sogenannten **Zuständen**.
- (ii) I ist ein Alphabet, das **Eingabealphabet** genannt wird.
- (iii) Γ ist ein Alphabet, das **Stack-** oder **Kelleralphabet** genannt wird.
- (iv) $\delta : S \times (I \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathcal{P}_{<\infty}(S \times \Gamma^*)$ ist die **Übergangsfunktion**.⁵

Definition



- (v) Der **Startzustand** s_0 ist ein Element von S .
- (vi) $\perp \in \Gamma$ ist der **initiale Stackinhalt**.
- (vii) F ist eine Teilmenge von S , deren Elemente **akzeptierenden Zustände** oder **Endzustände** genannt werden.

Kellerautomaten basieren wie endliche Automaten auf Zuständen, die in Abhängigkeit von konsumierten Zeichen aus dem Eingabealphabet durchlaufen werden. Und wie bisher geht es beim Startzustand los. Allerdings bestimmt nicht nur das Eingabezeichen den Ablauf. In jedem Schritt wird ein Zeichen vom Stack gelesen („*pop*“) und dieses entscheidet mit über das weitere Vorgehen. Zudem kann der Automat bei jedem Übergang von Zustand zu Zustand Zeichen aus dem Stackalphabet auf den Stack schreiben („*push*“). Formalisiert wird das folgendermaßen:

Definition

Eine **Konfiguration** eines Kellerautomaten $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ ist ein Tripel (s, w, α) , wobei $s \in S$, $w \in I^*$ und $\alpha \in \Gamma^*$ gilt. Wir definieren eine Relation $\Rightarrow_K$ auf der Menge dieser Konfigurationen durch

$$\begin{aligned} (s, aw, \sigma\alpha) \Rightarrow_K (s', w, \beta\alpha) & \text{ genau dann, wenn } (s', \beta) \in \delta(s, a, \sigma) \\ (s, w, \sigma\alpha) \Rightarrow_K (s', w, \beta\alpha) & \text{ genau dann, wenn } (s', \beta) \in \delta(s, \varepsilon, \sigma) \end{aligned}$$

für $s, s' \in S$, $w \in I^*$, $a \in I$ und $\alpha, \beta \in \Gamma^*$ und $\sigma \in \Gamma$.

Durch eine Konfiguration (s, w, α) wird quasi eine bestimmte Situation beim Durchlaufen des Automaten beschrieben: K ist im Zustand s , das Wort w muss noch konsumiert werden und α ist der momentane Inhalt des Stacks. $\Rightarrow_K$ beschreibt, wann der Automat von einer Konfiguration in eine andere wechseln darf. Befindet er sich im Zustand s und ist sowohl das nächste zu lesende Eingabesymbol a als auch das oberste Zeichen auf dem Stack σ , dann darf er a konsumieren, zum Zustand s' wechseln und σ durch β ersetzen,⁶ wenn δ diese Aktion erlaubt. Zudem sind auch ε -Übergänge (ähnlich wie in Aufgabe 6.7) vorgesehen. Wir sollten allerdings im Kopf behalten, dass ein „Übergang“ nicht wie bei endlichen Automaten nur ein Wechsel von Zustand zu Zustand, sondern nun ein Wechsel von Konfiguration zu Konfiguration ist.

Wie bei den Grammatiken sei $\Rightarrow_K^*$ die reflexive und transitive Hülle von $\Rightarrow_K$. Die von einem Kellerautomaten K *akzeptierte Sprache* wird dann definiert als

$$L(K) = \{ w \in I^* : (s_0, w, \perp) \Rightarrow_K^* (s, \varepsilon, \varepsilon) \text{ für ein } s \in S \}.$$

Der Automat beginnt also im Startzustand s_0 und das Zeichen $\perp$ sorgt dafür, dass der Stack nicht leer ist. Ein Wort wird akzeptiert, wenn es eine endliche Folge

⁵Mit $\mathcal{P}_{<\infty}(A)$ ist die Menge der endlichen Teilmengen einer Menge A gemeint.

⁶Man beachte, dass β kein Zeichen, sondern ein Wort ist. Z. B. kann σ einfach entfernt werden, wenn $\beta = \varepsilon$ gilt, oder es kann ein weiteres Symbol τ hinzugefügt werden, wenn $\beta = \tau\sigma$ gilt. Es ist durchaus auch möglich, dass der Stack dadurch komplett geleert wird. Dann geht es jedoch nicht weiter, denn in der Definition von $\Rightarrow_K$ wird verlangt, dass sich auf dem Stack mindestens ein Symbol σ befindet.

von Konfigurationswechseln der Form $\Rightarrow_K$ gibt, die damit enden, dass das Wort abgearbeitet und der Stack leer ist. Der dadurch erreichte Zustand s ist irrelevant.⁷

Wir schauen uns nun Beispiele und die grafische Repräsentation von Kellerautomaten an, einigen uns aber vorab auf eine Konvention, die die Lesbarkeit ein bisschen erhöht.

Ähnlich wie bei Grammatiken verwenden wir bei Kellerautomaten für Zeichen aus den Stackalphabeten (abgesehen von $\perp$) durchgehend kursive Großbuchstaben und für Eingabesymbole Kleinbuchstaben (sowie evtl. Ziffern und Sonderzeichen) in Schreibmaschinenschrift.

Konvention

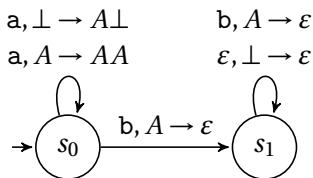
Und nun folgt unser erster Kellerautomat, der formal die Form

$$K = (\{s_0, s_1\}, \{a, b\}, \{A, \perp\}, \delta, s_0, \perp, \emptyset) \quad (11.2)$$

mit der folgenden Übergangsfunktion δ hat:⁸

$$\begin{aligned} \delta(s_0, a, \perp) &= \{(s_0, A\perp)\} & \delta(s_0, a, A) &= \{(s_0, AA)\} \\ \delta(s_0, b, A) &= \{(s_1, \varepsilon)\} & \delta(s_1, b, A) &= \{(s_1, \varepsilon)\} \\ \delta(s_1, \varepsilon, \perp) &= \{(s_1, \varepsilon)\} \end{aligned}$$

Kellerautomaten können im Vergleich zu endlichen Automaten erstaunlich viel mit sehr wenigen Zuständen erreichen, dafür wird aber die Visualisierung problematisch, weil an den Pfeilen nun auch immer notiert werden muss, was auf dem Stack passiert. Die Beschriftung ist hoffentlich selbsterklärend:



Exemplarisch schauen wir uns zwei Wege durch den Automaten an.

$$\begin{aligned} (s_0, aabb, \perp) &\Rightarrow_K (s_0, abb, A\perp) \Rightarrow_K (s_0, bb, AA\perp) \Rightarrow_K (s_1, b, A\perp) \\ &\Rightarrow_K (s_1, \varepsilon, \perp) \Rightarrow_K (s_1, \varepsilon, \varepsilon) \\ (s_0, abb, \perp) &\Rightarrow_K (s_0, bb, A\perp) \Rightarrow_K (s_1, b, \perp) \Rightarrow_K (s_1, b, \varepsilon) \end{aligned}$$

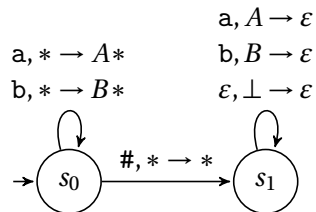
Das Wort $aabb$ wird akzeptiert, weil nach dem Abarbeiten aller Zeichen der Stack leer ist. Das Wort abb kann nicht akzeptiert werden, denn es führt schon zu einem leeren Stack, bevor alle Symbole konsumiert wurden. (Siehe Fußnote 6.)

⁷Ja, Sie haben das richtig gelesen: die Menge F der Endzustände spielt keine Rolle! Auf die kommen wir [später](#) noch zu sprechen, aber das passiert im optionalen Teil des Skripts.

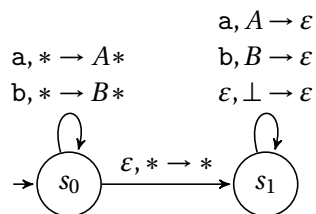
⁸Wir vereinbaren als weitere Konvention, dass δ für alle Argumente, die nicht aufgeführt sind, den Funktionswert $\emptyset$ hat. Gemäß der Definition von $\Rightarrow_K$ bedeutet das, dass es in so einem Fall „nicht weitergeht“.

Es ist hoffentlich klar geworden, dass dieser Automat die Sprache L_∞ aus (11.1) akzeptiert. Für jedes a wird ein A auf den Stack gelegt und jedes b entfernt ein A vom Stack.

Noch ein weiteres Beispiel, bei dem das Eingabealphabet die Menge $\{a, b, \#\}$ ist und das Stackalphabet aus den Zeichen $\perp$, A und B besteht. Der Automat wird lediglich grafisch dargestellt und wir ersparen uns etwas Schreibarbeit dadurch, dass das Zeichen $*$ als Platzhalter für ein beliebiges Symbol des Stackalphabets steht – natürlich auf beiden Seiten von $\rightarrow$ für dasselbe Symbol.⁹



Machen Sie sich klar, dass dieser Automat die Sprache $\{w\#w^R : w \in \{a, b\}^*\}$ akzeptiert. Für so etwas ist ein Stack natürlich perfekt geeignet. Störend ist nur das „Trennzeichen“ $\#$, das für den Wechsel von s_0 nach s_1 sorgt. Geht es auch ohne? Ja! Hier ist ein Kellerautomat für die Sprache $\{ww^R : w \in \{a, b\}^*\}$:



Moment mal, ist das nicht derselbe Automat wie eben, bei dem einfach das Zeichen $\#$ weggelassen wurde? Genau. Aber wie „erkennt“ der Automat denn nun, wann er von s_0 nach s_1 wechseln muss? Das muss er gar nicht. Wir haben bisher gar nicht ausgenutzt, dass wir Kellerautomaten von vornherein nichtdeterministisch konstruiert haben. Damit ein Wort akzeptiert wird, reicht es, wenn es einen Weg durch den Automaten gibt, der mit einem leeren Stack endet. In gewissem Sinne „rät“ der Automat also, wann er von s_0 zu s_1 wechseln soll. (Alternative Interpretation: Er probiert parallel alle Varianten durch und bei Wörtern aus der Sprache führt mindestens eine zum Erfolg.)

Aufgabe 11.1. Konstruieren Sie für $L = \{a^k b^m : k, m \in \mathbb{N}^+ \text{ und } k \geq m\}$ einen Kellerautomaten, der L akzeptiert.

Aufgabe 11.2. Konstruieren Sie einen Kellerautomaten für die Sprache $\{0^n 1^{2n} : n \in \mathbb{N}\}$.

Aufgabe 11.3. Konstruieren Sie einen Kellerautomaten für $\{w \in \Sigma_{\text{bool}}^* : |w|_0 = |w|_1\}$.

⁹Die Zeile $a, * \rightarrow A*$ ersetzt z. B. drei Zeilen, weil $*$ jedes der drei Stacksymbole sein kann.

★**Aufgabe 11.4.** Man kann Akzeptanz durch Kellerautomaten auch anders definieren, indem man für $K = (S, I, \Gamma, \delta, s_0, \perp, F)$

$$L^*(K) = \{ w \in I^* : (s_0, w, \perp) \Rightarrow_K^* (s, \varepsilon, \alpha) \text{ für ein } s \in F \text{ und ein } \alpha \in \Gamma^* \} \quad L^*(K)$$

definiert und das als die von dem Automaten akzeptierte Sprache festsetzt. Hier wird also F wie bei den endlichen Automaten verwendet, während der finale Stackinhalt keine Rolle spielt. Begründen Sie, dass beide Definitionen äquivalent sind: Für jeden Kellerautomaten K findet man einen Kellerautomaten K' mit $L(K') = L^*(K)$ und einen Kellerautomaten K'' mit $L^*(K'') = L(K)$.

Aufgabe 11.5. FLACI kann auch Kellerautomaten simulieren, allerdings nur solche, die wie in Aufgabe 11.4 akzeptieren.¹⁰ Trotzdem sollten Sie damit vielleicht ein bisschen herumspielen.

★**Aufgabe 11.6.** Ein Kellerautomat wird *beschränkt* genannt, wenn in jedem Übergang die Länge des Stacks maximal um ein Zeichen erhöht wird. Begründen Sie, dass es zu jedem Kellerautomaten K einen beschränkten Kellerautomaten K' mit $L(K') = L(K)$ und $L^*(K') = L^*(K)$ gibt.

beschränkter
Kellerautomat

Nun stellen wir den Zusammenhang zu den kontextfreien Sprachen her.

Eine Sprache ist genau dann kontextfrei, wenn es einen Kellerautomaten gibt, der sie akzeptiert.

Satz 11.1

Beweis. Wir gehen zunächst von einer kontextfreien Grammatik $G = (N, T, P, S)$ aus und konstruieren einen Kellerautomaten, der $L(G)$ akzeptiert. Die Zustandsmenge wird überraschenderweise nur aus einem Symbol s_0 bestehen, das dann logischerweise auch das Startsymbol sein muss. Die ganze „Arbeit“ wird auf dem Stack erledigt. Das Vokabular der Grammatik wird zusammen mit dem neuen Symbol $\perp$ zum Stackalphabet $N \cup T \cup \{\perp\}$. Für alle $A \in N$ setzen wir jetzt

$$\delta(s_0, \varepsilon, A) = \{(s_0, \beta) : (A, \beta) \in P\}.$$

Das bedeutet, dass der Automat für jede Produktion der Form $A \rightarrow \beta$ vom Stapel das oberste Symbol A entfernen und durch β ersetzen darf. Für alle $a \in T$ setzen wir außerdem $\delta(s_0, a, a) = \{(s_0, \varepsilon)\}$. Befindet sich ein Terminalsymbol oben auf dem Stack, so wird es also entfernt, wenn es „dran“ ist. Schließlich fügen wir noch $\delta(s_0, \varepsilon, \perp) = \{(s_0, S)\}$ hinzu. Damit wird ganz am Anfang das Startsymbol der Grammatik als einziges Zeichen auf den Stack gelegt. Und das war's schon! Dass das funktioniert, sollte hoffentlich recht naheliegend sein. Wir gehen es aber gleich auch noch anhand eines Beispiels durch.

¹⁰Allerdings kann man mit dem Button *Transformieren* die andere Variante der Akzeptanz zumindest anzeigen lassen.

Für die andere Richtung des Beweises gehen wir von einem Kellerautomaten $K = (S, I, \Gamma, \delta, s_0, \perp, \emptyset)$ aus und wollen eine kontextfreie Grammatik G konstruieren, die $L(K)$ erzeugt. Dafür definieren wir die Menge¹¹

$$N' = \{[s\sigma t] : s, t \in S \text{ und } \sigma \in \Gamma\}$$

und wählen als Menge der Nichtterminalsymbole $N = N' \cup \{S_0\}$, wobei S_0 irgendein Symbol sein soll, das nicht zu N' gehört. S_0 soll das Startsymbol von G sein. Für alle Zustände s des Automaten fügen wir zunächst zu unserer Grammatik die Produktion $S_0 \rightarrow [s_0 \perp s]$ hinzu. Nun betrachten wir beliebige Tupel (s, a, σ) aus dem Definitionsbereich der Übergangsfunktion δ und dazu den zugehörigen Funktionswert $\Delta = \delta(s, a, \sigma)$. Ist $(t, \tau_1 \cdots \tau_k)$ ein Element von Δ ,¹² so fügen wir für alle $s_1, \dots, s_k \in S$ die folgende Produktion zur Grammatik hinzu:

$$[s\sigma s_k] \rightarrow a[t\tau_1 s_1][s_1\tau_2 s_2] \cdots [s_{k-1}\tau_k s_k]$$

Es ist sicherlich nicht ohne Weiteres klar, warum diese (sehr umfangreiche) Grammatik die gewünschten Anforderungen erfüllt und wie sie überhaupt zustande kommt. Die Idee ist, dass ein Symbol wie $[s\sigma t]$ für einen Weg durch den Automaten steht, der vom Zustand s zum Zustand t führt und dabei „netto“ das Symbol σ vom Stack entfernt. Formal muss man dafür

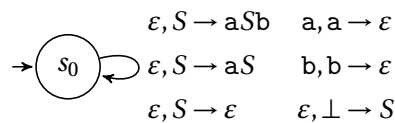
$$[s\sigma t] \Rightarrow_G^* w \quad \text{genau dann, wenn} \quad (s, w, \sigma) \Rightarrow_K^* (t, \varepsilon, \varepsilon)$$

für alle $s, t \in S$, alle $\sigma \in \Gamma$ und alle $w \in I^*$ zeigen. Das ersparen wir uns jedoch. Einen ausführlichen Beweis finden Sie z. B. im Standardwerk von Hopcroft et al., das [in der Literaturliste](#) aufgeführt ist. Siehe auch Aufgabe 11.11. ■

Hier nun das im Beweis versprochene Beispiel. Dazu betrachten wir die Grammatik

$$S \rightarrow aSb \mid aS \mid \varepsilon$$

für die Sprache $\{a^m b^n : m, n \in \mathbb{N} \text{ und } m \geq n\}$. Ein Kellerautomat K für diese Sprache könnte so aussehen.



Dabei geben die Übergänge in der linken Spalte die Produktionen der Grammatik wieder, während die anderen für das Konsumieren der Symbole sowie das initiale Befüllen des Stacks mit dem Startsymbol zuständig sind.

Aufgabe 11.7. Geben Sie einen Weg durch diesen Automaten an, durch den das Wort aab der Sprache akzeptiert wird.

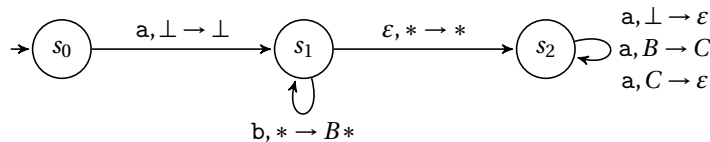
¹¹ $[s\sigma t]$ wählen wir hier als Abkürzung für (s, σ, t) . Damit sind „atomare“ Symbole wie in der Lösung von Aufgabe 11.4 gemeint.

¹²Die τ_i sollen also Stacksymbole sein. Dabei ist auch $\tau_1 \cdots \tau_k = \varepsilon$ möglich, also $k = 0$.

Aufgabe 11.8. Welche der folgenden Aussagen ist wahr? Begründen Sie Ihre Antwort.

- (i) Zu jeder regulären Sprache gibt es einen endlichen Automaten mit nur einem Zustand, der sie akzeptiert.
- (ii) Zu jeder kontextfreien Sprache gibt es einen Kellerautomaten mit nur einem Zustand, der sie akzeptiert.

Aufgabe 11.9. Geben Sie eine Grammatik für die Sprache an, die von dem folgenden Kellerautomaten akzeptiert wird.



Aufgabe 11.10. Sie haben es sich schon gedacht: **FLACI** kann Kellerautomaten in kontextfreie Grammatiken umwandeln und umgekehrt. Gelegenheit zum Vertiefen des Gelernten gibt es also wie immer reichlich.

★**Aufgabe 11.11.** Überlegen Sie sich anhand des Beispielautomaten K aus (11.2), wie die zugehörige Grammatik aus dem Beweis von Satz 11.1 aussehen würde.

★**Aufgabe 11.12.** Sind L_2 und L_3 Sprachen über demselben Alphabet Σ und ist L_2 kontextfrei und L_3 regulär, dann ist $L_2 \cap L_3$ kontextfrei. Man kann das jedoch wegen Aufgabe 10.10 nicht daraus folgern, dass L_3 kontextfrei ist. Trotzdem stimmt es. Fällt Ihnen eine Begründung ein? (Hinweis: Schauen Sie sich die Automatenkonstruktionen in Kapitel II noch einmal an.)

„Unsere“ Kellerautomaten sind per definitionem nichtdeterministisch. Kann man, wie bei endlichen Automaten, dieselben Resultate auch deterministisch erzielen? Die Antwort ist *nein*, und das sollten Sie wissen. Die technischen Details können Sie sich aber sparen. Falls Sie dennoch neugierig geworden sind, dann lesen Sie den folgenden Abschnitt.

★Deterministische Kellerautomaten

Wir haben Kellerautomaten von Anfang an als nichtdeterministisch definiert. Anders als bei endlichen Automaten gibt es in diesem Fall jedoch nicht nur einen quantitativen, sondern auch einen qualitativen Unterschied zwischen Determinismus und Nichtdeterminismus. Um das präzise zu formulieren, definieren wir zunächst einmal, was in diesem Zusammenhang mit *deterministisch* gemeint ist.

Ein Kellerautomat $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ wird **deterministisch** genannt, wenn alle Funktionswerte von δ höchstens die Mächtigkeit 1 haben. Außerdem muss für alle $s \in S$, und $\sigma \in \Gamma$ die Menge $\delta(s, \varepsilon, \sigma)$ leer sein, wenn $\delta(s, a, \sigma)$ für mindestens ein $a \in I$ nicht leer ist.

Definition

Intuitiv heißt das, dass es für jedes Wort nur einen möglichen Weg durch den Automaten gibt. (Details dazu in Aufgabe 11.14.)

Aufgabe 11.13. Überzeugen Sie sich, dass die ersten beiden Beispiele für Kellerautomaten in diesem Skript deterministisch waren.

Aufgabe 11.14. In deterministischen Kellerautomaten sind nach wie vor ε -Übergänge erlaubt, die wir im Zusammenhang mit *nichtdeterministischen* endlichen Automaten (Aufgabe 6.7) kennengelernt hatten. Können Sie hieb- und stichfest begründen, dass es trotzdem für jeden deterministischen Kellerautomaten pro Eingabewort im Wesentlichen nur *einen* Weg durch den Automaten gibt? (Und wieso die Einschränkung „im Wesentlichen“?)

Für alle regulären Sprachen gibt es einen deterministischen Kellerautomaten K mit $L^*(K)$ (siehe Aufgabe 11.4). Das geschieht ganz simpel dadurch, dass K einen deterministischen endlichen Automaten für die entsprechende Sprache simuliert und seinen Stack ignoriert. Bei Akzeptanz mit leerem Stack – also via $L(K)$ – sieht die Sache jedoch anders aus:

Aufgabe 11.15. Können Sie einen deterministischen Kellerautomaten K angeben, für den $L(K) = \{0, 01\}$ gilt?

Der oben angesprochene Unterschied zwischen Determinismus und Nichtdeterminismus lässt sich an der Sprache

$$L = L_1 \cup L_2 \text{ mit } L_1 = \{0^n 1^n : n \in \mathbb{N}^+\} \text{ und } L_2 = \{0^n 1^{2n} : n \in \mathbb{N}^+\} \quad (11.3)$$

verdeutlichen. L ist kontextfrei (siehe Aufgabe 11.16), aber es gibt keinen deterministischen Kellerautomaten K , der diese Sprache im Sinne von $L = L^*(K)$ akzeptiert.

Die Begründung dafür soll im Folgenden skizziert werden. Wir werden dafür die unendliche Version des sogenannten *Schubfachprinzips* verwenden, die wohl jedem einsichtig sein dürfte: Verteilt man unendlich viele Objekte auf endlich viele Klassen („Schubfächer“), dann müssen in mindestens einer davon unendlich viele landen. Da die interessanten Sprachen aus unendlich vielen Wörtern bestehen, Automaten und Alphabete aber endlich sind, ist die Anwendung dieses Prinzips häufig möglich.

Sei also K ein deterministischer Kellerautomat mit $L_1 \subseteq L^*(K)$. Für jedes Wort $w \in L_1$ gibt es einen eindeutig bestimmten Zustand s_w , der direkt nach dem Konsumieren der letzten Null erreicht wird. Da es nur endlich viele Zustände gibt, muss es nach dem Schubfachprinzip einen Zustand s' geben, der dieser Zwischenzustand s_w für unendlich viele Wörter $w \in L_1$ ist. Nennen wir die Menge dieser Wörter L'_1 . Da nun nur noch Einsen kommen, muss für Elemente von L'_1 ab s' der weitere Weg durch den Automaten ausschließlich vom Stackinhalt abhängen, in dem daher die Anzahl der konsumierten Nullen codiert sein muss.

Erneut nach dem Schubfachprinzip muss es unendlich viele Wörter aus L'_1 geben, die nach dem Konsumieren der letzten Eins im selben (akzeptierenden) Zustand s'' landen. Deren Stackinhalt muss zu diesem Zeitpunkt aus dem genannten Grund identisch sein. Greifen wir uns zwei dieser Wörter heraus, etwa $0^{n_1} 1^{n_1}$ und $0^{n_2} 1^{n_2}$

mit $n_1 \neq n_2$. Würde der Automat auch alle Wörter aus L_2 akzeptieren, dann müsste man von s'' aus sowohl mit n_1 als auch mit n_2 weiteren Einsen zu einem akzeptierenden Zustand kommen. Das würde aber implizieren, dass auch das Wort $0^{n_1} 1^{n_1+n_2}$ akzeptiert wird, das nicht zu L gehört.

Auch die **Palindrom**-Sprache $\{ww^R : w \in \{a,b\}^*\}$ von vorhin kann übrigens von deterministischen Kellerautomaten nicht akzeptiert werden. Der Beweis dafür ist allerdings haariger.

Aufgabe 11.16. Begründen Sie, dass die Sprache aus (11.3) kontextfrei ist.

IV

Rekursiv aufzählbare Sprachen

Wir machen nun einen großen Sprung von den kontextfreien Sprachen zu denen, die überhaupt noch in irgendeiner Form von einer (abstrakten oder konkreten) Maschine „erkannt“ werden können. Als Motivation können Sprachen wie

$$L_1 = \{a^k b^k c^k : k \in \mathbb{N}\} \quad \text{und} \quad L_2 = \{ww : w \in \{a, b\}^*\}$$

dienen, von denen wir inzwischen wissen, dass kein Kellerautomat sie akzeptieren kann. Was müsste man hinzufügen, damit ein Automat auch mit solchen Sprachen klarkommt? Für L_1 könnte das ein zweiter Stack sein. Aber bräuchte man dann nicht für noch kompliziertere Sprachen noch mehr Stacks? Für L_2 bräuchte man wohl eher eine Warteschlange statt eines Stacks (FIFO statt LIFO), aber würde man damit noch die Palindrom-Sprache $\{ww^R : w \in \{a, b\}^*\}$ akzeptieren können? Die Automaten, die das Problem lösen, sind natürlich komplexer als Kellerautomaten, aber immer noch erstaunlich simpel. Und wie wir noch sehen werden, gibt es faktisch keine „mächtigeren“ Automaten als die, die nun vorgestellt werden.

12. Turingmaschinen

Turingmaschinen sind nach dem englischen Mathematiker [Alan Turing](#) benannt, der sie sich 1936 für seine wegweisende Arbeit *On Computable Numbers, with an Application to the Entscheidungsproblem* ausdachte. Sie sind nach wie vor das wichtigste und in theoretischen Argumenten am häufigsten verwendete abstrakte Automatenmodell der Informatik. 1936 gab es allerdings noch keine Computer und keine Informatik. Turing ging es um die für die mathematische Grundlagenforschung interessante Frage, wann etwas *berechenbar* oder *entscheidbar* ist.¹ Seine „Maschinen“ (den Namen erhielten sie erst später) sollten möglichst einfach, aber gleichzeitig vollständig simulieren, wie ein *Mensch* etwas berechnet. Dafür stellte er sich eine Person vor, die sämtliche Gedanken und Zwischenschritte während eines Rechengangs mit einem Bleistift (und ggf. einem Radiergummi) auf einem Blatt Papier festhält. Diesen Vorgang vereinfachte er nach und nach, indem er

¹Dabei ist mit *Berechnung* in diesem Zusammenhang nicht nur klassisches Rechnen mit Zahlen gemeint, sondern jeder Vorgang, der in irgendeiner Form Symbole manipuliert und sich präzise formal beschreiben lässt.

- Kästchenpapier vorgab, das pro Kästchen nur ein Zeichen zuließ,
- die Kästchen eindimensional in einer Reihe (statt zweidimensional in der Fläche) anordnete,
- nur einen endlichen Zeichenvorrat (also ein [Alphabet](#)) erlaubte,
- den Ablauf in einzelne Schritte unterteilte, bei denen immer nur ein Kästchen zur Zeit betrachtet wird, und
- dem Gehirn des rechnenden Menschen endlich viele „Zustände“ zuschrieb, zwischen denen er hin und her wechseln kann.

Wir beginnen mit einer Variante der Turingmaschinen, die noch simpler als Turings Original ist. Sie beruht auf Modifikationen des im [ersten Kapitel](#) bereits erwähnten [Stephen Kleene](#).

Damit Sie mit solchen Turingmaschinen herumspielen können und als Vorbereitung für die nächsten Abschnitte habe ich ein paar Programme geschrieben. Die Anleitung dafür finden Sie unter der Adresse <https://weitz.de/files/turing.pdf>. Im Folgenden gehe ich davon aus, dass Sie die dort beschriebene Software (die sich in einem [Docker](#)-Image befindet) installiert haben und sie ggf. verwenden, um einzelne Schritte nachzuvollziehen. Im Zweifelsfall sind die Beschreibungen in dem erwähnten PDF ausführlicher als die in den folgenden Abschnitten!

Die erste Version der Turingmaschine arbeitet mit einem *Band* (engl. *tape*), das einen Anfang, aber kein festes Ende hat. (Das Band soll nicht unendlich lang sein, aber immer „lang genug“. Es wächst gewissermaßen nach rechts, wenn das nötig sein sollte.) Das Band ist in Kästchen (auch *Zellen* genannt) unterteilt, die entweder leer sind oder als Zeichen einen vertikalen Strich enthalten.



Als weitere „Zutat“ gehört zu einer Turingmaschine ein *Schreib-Lese-Kopf*, den man sich als eine Vorrichtung vorstellen muss, die immer über einem der Kästchen des Bandes steht und dessen Inhalt lesen kann – ob es sich also um ein leeres Kästchen oder um eines mit einem Strich handelt. Abgesehen davon „sieht“ der Schreib-Lese-Kopf, den wir im Folgenden meistens einfach *Kopf* nennen werden, nichts! Der Kopf kann außerdem den Inhalt der Zelle, über der er steht, ändern, d. h., er kann einen Strich „ausradieren“ oder in eine leere Zelle einen Strich einfügen.

Die Maschine befindet sich immer in einem bestimmten *Zustand* (engl. *state*) und es gibt nur endlich viele verschiedene Zustände. (Das kennen Sie schon von [endlichen Automaten](#) und [Kellerautomaten](#).) Die Maschine arbeitet in einzelnen Schritten und jeder Schritt besteht aus den folgenden Teilen:

- Inhalt des aktuellen Kästchens anschauen
- Inhalt des aktuellen Kästchens evtl. ändern
- Evtl. zum linken oder rechten Nachbarkästchen wechseln
- Evtl. in einen anderen Zustand übergehen

Was genau in den Teilen (ii) bis (iv) passiert, wird durch *Instruktionen* beschrieben. Diese beziehen sich ausschließlich auf den aktuellen Zustand und den in Teil (i) gesehenen Inhalt. Insbesondere „weiß“ die Maschine nicht, wo auf dem Band sie ist oder was der Inhalt der anderen Kästchen ist.

Bevor wir weitermachen, zwei Bemerkungen zur *Nomenklatur*:

- Informell werden die Instruktionen oft wie *Programme* interpretiert, mit denen „die“ Turingmaschine programmiert wird. (Und manchmal wird es in diesem Text auch so klingen.) Formal korrekt besteht eine Turingmaschine jedoch aus dem Band, dem Kopf und den Instruktionen. Das „Programm“ ist gewissermaßen „fest verdrahtet“. Ändert man die Instruktionen, erhält man eine *andere* Turingmaschine. Es gibt also nicht *die* Turingmaschine, sondern viele verschiedene Turingmaschinen.
- Für die Beschreibung ist es einfacher, wenn man leere Kästchen ebenfalls als Symbole ansieht. In diesem Sinne arbeiten die gerade beschriebenen Turingmaschinen mit *zwei* Symbolen – dem vertikalen Strich und dem Leerzeichen.

Nun aber zu den Instruktionen. In der *erwähnten Software* besteht eine einzelne Instruktion aus drei Zeichen, auf die eine Zahl folgt, z. B. so:

I-R2

Die ersten beiden Zeichen sind jeweils entweder I or – für den vertikalen Strich bzw. für das Leerzeichen. Das erste Zeichen steht für das Symbol, das der Kopf „sieht“, das zweite für das Symbol, das der Kopf schreiben soll. (Die beiden können identisch sein. Dann ändert der Kopf den Inhalt der Zelle nicht.) Das dritte Zeichen ist L, R oder S – für *left*, *right* oder *stay*. Es beschreibt die Bewegung des Kopfes: Er soll nach dem Schreiben nach links oder rechts gehen oder auf derselben Zelle bleiben. Die Zahl am Ende ist die Nummer des nächsten Zustands, den die Maschine einnehmen soll. Das obige Beispiel besagt also: Wenn in dem Kästchen unter dem Kopf ein Strich zu sehen ist, lösche ihn, gehe ein Kästchen nach rechts und wechsele in den Zustand mit der Nummer 2.

Die Maschine wüsste allerdings nicht, was sie tun soll, wenn das Kästchen leer ist. Wir brauchen *zwei* Instruktionen (die durch das Zeichen | getrennt werden):²

I-R2 | --L3

Wenn das Kästchen leer ist, soll es also leer bleiben, der Kopf sich eine Zelle nach links bewegen und die Maschine in den dritten Zustand wechseln.

Zwei Dinge fehlen allerdings noch. Erstens kann der dritte Buchstabe einer Instruktion auch H für *halt* sein. Das bedeutet, dass die Maschine anhalten soll. (Es gibt dann keinen „nächsten“ Zustand. Damit die *Syntax* einfacher ist, muss hinter dem H jedoch trotzdem eine Nummer stehen.)

²Die Reihenfolge – erst – oder erst I – spielt keine Rolle.

Die zweite und letzte Sache, die fehlt: Was passiert, wenn der Kopf zu weit nach links wandert? Dieses Problem wird durch die Einführung des *Pseudo-Symbols* * gelöst. Bewegt sich der Kopf vom ersten Kästchen aus nach links, so „sieht“ er das Zeichen *, das es nicht wirklich gibt, das ihm jedoch signalisiert, dass er das Band verlassen hat. (Dieses „Symbol“ kann nicht verändert werden und der Kopf kann sich in diesem Fall auch nicht weiter nach links bewegen. Außerdem kann * auch nicht in die Kästchen des Bandes geschrieben werden.)

Mit diesen Informationen können wir nun einen kompletten Satz von Instruktionen aufschreiben:

```
I-R2 | --L3 | **R7
```

Aber wo sind die Zustände? Die Zeile, die Sie gerade gesehen haben, *ist* gewissermaßen ein Zustand! Eine Turingmaschine für die hier verwendete Simulationssoftware wird durch eine [Textdatei](#) beschrieben, die aus solchen Zeilen besteht. Dabei beschreibt die erste Zeile das Verhalten im ersten Zustand, die zweite Zeile ist für den zweiten Zustand da und so weiter. Eine komplette Turingmaschine könnte so aussehen:³

```
--R2 | I-R1 | **R1 # erster Zustand
--H2 | I-R1 | **R2 # Zustand Nr. 2
```

Die Turingmaschine startet immer mit dem Kopf auf dem ersten Kästchen und sie befindet sich initial im ersten Zustand. Die *Eingabe* für die Maschine ist der Inhalt der Kästchen auf dem Band, bevor sie startet. Ihre *Ausgabe* ist das, was sich auf dem Band befindet, wenn die Maschine anhält. Die Eingabe kann nur aus endlich vielen Kästchen bestehen, die nicht leer sind. Wenn neue Kästchen hinzugefügt werden müssen, weil der Kopf nach rechts wandert, sind diese grundsätzlich erst einmal leer.

Aufgabe 12.1. Was macht die obige Turingmaschine mit dem Inhalt des Bandes? Was würde also beispielsweise passieren, wenn sich anfangs auf dem Band drei vertikale Striche befinden? Überlegen Sie sich das erst einmal mit Zettel und Stift, bevor Sie weiterlesen.

Wie dem [oben erwähnten PDF](#) zu entnehmen ist, können Sie die Turingmaschine nun wie folgt (mit dem initialen Bandinhalt III) ausprobieren:

```
echo $'--R2 | I-R1 | **R1\n--H2 | I-R1 | **R2\n' > Code/test.tm
tm --tape=III Code/test.tm
```

Aufgabe 12.2. Experimentieren Sie ein bisschen mit dem Programm tm. Lesen Sie [im PDF](#) bis zum Anfang des Abschnitts *Better Turing machines* und schauen Sie sich die dortigen Beispiele an. Bearbeiten Sie auch die Aufgaben.

³Übrigens wird das Zeichen # in allen Simulationsprogrammen dieses Kapitels für [Kommentare](#) verwendet. Alles ab diesem Zeichen bis zum Ende der jeweiligen Zeile wird ignoriert.

Verbesserte Turingmaschinen

Wir werden nun die Turingmaschinen etwas „aufbohren“. Es wäre nett, wenn wir ein weiteres Symbol zur Verfügung hätten. Dann könnten wir nämlich Zahlen **binär** statt **unär** darstellen und Instruktionen wie diese schreiben:⁴

```
00R1 | 10S2 | --L1 | **H1
```

Solche Turingmaschinen können mit der vorherigen Minimalversion simuliert werden. Der „Trick“, mit dem wir das erreichen, besteht darin, je zwei Zellen des Bandes zu einem Symbol zusammenzufassen: Wir interpretieren I- als 0, II als 1 und -- als leere Zelle. Dadurch wird die Logik natürlich komplizierter, weil Lesen, Schreiben und Bewegungen nun häufig aus zwei Aktionen des Kopfes bestehen können. Hat er beispielsweise einen senkrechten Strich gelesen, so ist noch nicht klar, ob es sich um den Anfang einer Null oder einer Eins handelt. Dass das aber prinzipiell machbar ist, wird mit einer vollständigen Umsetzung im Abschnitt *Better Turing machines* [des PDFs](#) demonstriert.

Aufgabe 12.3. Offensichtlich „verschenken“ wir dadurch etwas, dass wir die Zeichenfolge -I gar nicht verwenden. Man könnte also (wenn man das Leerzeichen mitzählt) mit demselben „Trick“ auch Turingmaschinen mit vier statt drei Symbolen simulieren. Ist Ihnen klar, wie man Maschinen mit noch mehr Symbolen realisieren könnte?

Aufgabe 12.4. Stellen Sie sich einen (linearen) Computerspeicher vor, in dem man nur Nullen und Einsen speichern kann. In diesem Speicher steht eine (natürliche) Zahl, die Sie mit einem Programm (in irgendeiner „normalen“ Programmiersprache) auslesen sollen. Es liegt nahe, diese Zahl **binär** zu codieren, z. B. die Zahl 19 als 10011. Sie fangen also am Anfang des Speichers an und lesen Nullen und Einsen bis zum Ende der Zahl.

Das Problem: Wo endet die Zahl? Wenn Sie mit einer festen **Wortgröße** (etwa 64 Bit) arbeiten, ist das natürlich einfach, weil Sie nach entsprechend vielen Schritten aufhören können zu lesen. Wenn die Zahl aber beliebig groß sein darf, müssen Sie eine andere Strategie wählen. (Ist z. B. 100110 die Zahl 38 oder ist es die Zahl 19, hinter der eine Null folgt, die nicht mehr dazugehört?)

Wie könnte man eine Zahl nur mit Nullen und Einsen so in den Speicher schreiben, dass ein Programm, das den Speicher sukzessive ausliest, sie eindeutig identifizieren kann? (Wie erwähnt soll die Zahl dabei theoretisch beliebig groß sein können.⁵ Anderenfalls ist das Problem nicht besonders interessant.)

Grundsätzlich ist erst einmal jeder Lösungsansatz gut. Aber wenn Sie einen haben, dann überlegen Sie vielleicht zusätzlich noch, ob man diesen dahingehend verbessern kann, dass möglichst wenig Speicher „verschwendet“ wird.

Angesichts der Überlegung in Aufgabe 12.3 werden wir in Zukunft für Turingmaschinen beliebige Alphabete als Zeichenvorräte zulassen. Wir behalten aber im Kopf, dass man im Prinzip mit zwei Zeichen (von denen eines das Leerzeichen ist) auskommen könnte.

⁴Eine leere Zelle stellen wir auch in dieser Version durch - dar..

⁵Der Speicher ist also ebenfalls beliebig groß.

Formal lassen sich Turingmaschinen auf verschiedene Arten beschreiben. Wir beginnen mit einer, die erstens den bisherigen Automatenbeschreibungen ähnelt und die zweitens in dieser oder sehr ähnlicher Form in der Fachliteratur häufig anzutreffen ist.

Definition

Eine **partielle Funktion** f von A nach B ist eine **rechtseindeutige** Teilmenge von $A \times B$, also faktisch eine Funktion von einer Teilmenge von A nach B . So ein f wird typischerweise wie eine Funktion behandelt, bei der $f(x)$ nicht unbedingt für *alle* $x \in A$ definiert ist. Für $x \in A$ mit $x \notin \text{dom}(f)$ wird üblicherweise $f(x) = \perp$ geschrieben.⁶ Während man für Funktionen $f: A \rightarrow B$ schreibt, notiert man partielle Funktionen als $f: A \rightharpoonup B$. Eine „normale“ Funktion von A nach B nennt man zur Unterscheidung auch **total**.

In diesem Sinne könnte man etwa die Funktion g , die x auf $g(x) = 1/x$ abbildet, als partielle Funktion von $\mathbb{R}$ nach $\mathbb{R}$ betrachten. Für $x = 0$ hat sie keinen Funktionswert und wir würden $g(0) = \perp$ schreiben.

Aufgabe 12.5. Verständnisfrage: Ist jede totale Funktion eine partielle Funktion?

Definition

Eine **Turingmaschine** ist ein 8-Tupel $T = (S, I, \Gamma, \delta, s_0, \square, \vdash, F)$ mit den folgenden Eigenschaften:

- (i) S ist eine nichtleere endliche Menge von sogenannten **Zuständen**.
- (ii) I ist ein Alphabet, das **Eingabealphabet** genannt wird.
- (iii) $\Gamma \supseteq I$ ist ein Alphabet, das **Bandalphabet** genannt wird.
- (iv) Die **Übergangsfunktion** δ ist eine partielle Funktion von $S \times \Gamma$ nach $S \times \Gamma \times \{\leftarrow, \rightarrow, \diamond\}$ mit $\delta(s, \sigma) = \perp$ für alle $s \in F$ und alle $\sigma \in \Gamma$.
- (v) Der **Startzustand** s_0 ist ein Element von S .
- (vi) Das **Leerzeichen** $\square$ ist ein Element von $\Gamma \setminus I$.
- (vii) Die **Grenzmarkierung** $\vdash$ ist ein Element von $\Gamma \setminus I$ mit $\vdash \neq \square$.
- (viii) F ist eine Teilmenge von S , deren Elemente **akzeptierende Zustände** oder **Endzustände** genannt werden.

Beachten Sie, dass das Bandalphabet nach dieser Definition eine Obermenge des Eingabealphabets ist. Oft wird Γ jedoch außer $\square$ und $\vdash$ nur die Symbole von I enthalten.⁷ Das, was wir oben als den „Bandinhalt“ bezeichnet haben, ist in dieser Definition einfach ein Wort über Γ . Wenn wir beispielsweise eine Turingmaschine mit dem Eingabealphabet Σ_{bool} mit einem Band starten, auf dem die Zahl 42 binär dargestellt ist, dann ist der Bandinhalt beim Start der Maschine das Wort $\vdash 101010$. Wie eine Turingmaschine den Bandinhalt nach und nach ändert, wird wie bei **Kellerautomaten** durch *Konfigurationen* beschrieben.

⁶Wir setzen dabei implizit $\perp \notin B$ voraus.

⁷ $\square$ und $\vdash$ waren in den obigen Simulationsprogrammen die Zeichen – bzw. *.

Eine **Konfiguration** einer Turingmaschine $T = (S, I, \Gamma, \delta, s_0, \square, \vdash, F)$ ist ein Tripel (v, s, w) , wobei $v \in \Gamma^*$, $s \in S$ und $w \in \Gamma^+$ gilt. Wir definieren eine Relation $\Rightarrow_T$ auf der Menge dieser Konfigurationen durch

$$\begin{aligned} (v, s, \sigma v') \Rightarrow_T (v, s', \sigma' v') & \text{ genau dann, wenn } \delta(s, \sigma) = (s', \sigma', \diamond) \\ (v, s, \sigma w) \Rightarrow_T (v \sigma', s', w) & \text{ genau dann, wenn } \delta(s, \sigma) = (s', \sigma', \rightarrow) \\ (v, s, \sigma) \Rightarrow_T (v \sigma', s', \square) & \text{ genau dann, wenn } \delta(s, \sigma) = (s', \sigma', \rightarrow) \\ (v \rho, s, \sigma v') \Rightarrow_T (v, s', \rho \sigma' v') & \text{ genau dann, wenn } \delta(s, \sigma) = (s', \sigma', \leftarrow) \end{aligned}$$

für $v, v' \in \Gamma^*$, $w \in \Gamma^+$, $\rho \in \Gamma$, $s, s' \in S$ sowie $\sigma, \sigma' \in \Gamma \setminus \{\vdash\}$ oder $\sigma = \sigma' = \vdash$.

Definition

Informell ist mit einer Konfiguration (v, s, w) gemeint, dass vw der aktuelle Bandinhalt ist, die Maschine sich im Zustand s befindet und der Kopf sich das erste Symbol von w „anschaut“. Wir werden Konfigurationen statt als (v, s, w) in der Form $v[s]w$ schreiben. Der Teil $[s]$ zeigt an, wo der Kopf steht und in welchem Zustand sich die Maschine befindet.

$v[s]w$

Aufgabe 12.6. Durch welchen Teil der Definition wird geregelt, dass das Band immer „lang genug“ ist? Und inwieweit wird $\vdash$ in der Definition anders als alle anderen Zeichen behandelt?

Zunächst ein einfaches Beispiel. Die Turingmaschine ist

$$T_1 = (\{s_0, s_1, s_2\}, \Sigma_{\text{bool}}, \Sigma_{\text{bool}} \cup \{\square, \vdash\}, \delta_1, s_0, \square, \vdash, \{s_2\})$$

und δ_1 sei durch die folgende Tabelle definiert.

	s_0	s_1	s_2
0	$(s_0, 0, \rightarrow)$	$(s_1, 0, \leftarrow)$	$\perp$
1	$(s_0, 1, \rightarrow)$	$\perp$	$\perp$
$\square$	$(s_1, 0, \leftarrow)$	$(s_2, \square, \rightarrow)$	$\perp$
$\vdash$	$(s_0, \vdash, \diamond)$	$(s_2, \vdash, \rightarrow)$	$\perp$

In T_1 können beispielsweise die folgenden Konfigurationsübergänge stattfinden.

$$\begin{aligned} \vdash[s_0]000 &\Rightarrow_{T_1} \vdash 0[s_0]00 \Rightarrow_{T_1} \vdash 00[s_0]0 \Rightarrow_{T_1} \vdash 000[s_0]\square \\ &\Rightarrow_{T_1} \vdash 00[s_1]00 \Rightarrow_{T_1} \vdash 0[s_1]000 \Rightarrow_{T_1} \vdash [s_1]0000 \\ &\Rightarrow_{T_1} [s_1]\vdash 0000 \Rightarrow_{T_1} \vdash [s_2]0000 \end{aligned}$$

Beachten Sie, wie zwischendurch gemäß der obigen Definition ein Leerzeichen hinzugefügt wird. (Siehe Aufgabe 12.6.)

Wie inzwischen üblich sei $\Rightarrow_T^*$ die **reflexive und transitive Hülle** der Relation $\Rightarrow_T$. Im konkreten Beispiel würde also

$$\vdash[s_0]000 \Rightarrow_{T_1}^* \vdash[s_2]0000 \quad (12.1)$$

gelten.

Aufgabe 12.7. Machen Sie sich klar, dass für diese Maschine auch

$$\vdash [s_0]11 \Rightarrow_{T_1}^* \vdash 1[s_1]10$$

gilt, indem Sie die einzelnen $\Rightarrow_{T_1}$ -Übergänge notieren.

Aufgabe 12.8. Warum sind der obigen Tabelle für δ_1 wohl einige Einträge grau gefärbt?

Die Konfigurationen $\vdash [s_2]0000$ und $\vdash 1[s_1]10$, die in den beiden Beispielen auftreten, haben die Gemeinsamkeit, dass es jeweils „nicht mehr weitergeht“, weil δ_1 für die Kombination aus dem aktuellen Zustand und dem Zeichen, auf dem der Schreib-Lese-Kopf steht, nicht definiert ist. Wir sagen in so einer Situation, dass der Ablauf der Turingmaschine *terminiert*. Im ersten Fall ist das nach Punkt (iv) der Definition so gewollt, weil s_2 ein Endzustand ist. Der zweite Fall demonstriert, dass das auch bei anderen Zuständen möglich ist.

Ein zweites Beispiel wird zeigen, dass noch etwas anderes passieren kann. Die Turingmaschine ist

$$T_2 = (\{s_0, s_1\}, \{0\}, \{0, \square, \vdash\}, \delta_2, s_0, \square, \vdash, \emptyset)$$

und δ_2 sieht so aus:

	s_0	s_1
0	$(s_1, 0, \rightarrow)$	$(s_0, 0, \leftarrow)$
$\square$	$(s_1, \square, \rightarrow)$	$(s_0, \square, \leftarrow)$
$\vdash$	$(s_0, \vdash, \diamond)$	$(s_1, \vdash, \diamond)$

Damit ist das hier möglich:

$$\vdash [s_0]00 \Rightarrow_{T_2} \vdash 0[s_1]0 \Rightarrow_{T_2} \vdash [s_0]00 \Rightarrow_{T_2} \vdash 0[s_1]0 \Rightarrow_{T_2} \dots$$

Es ist offensichtlich, dass es immer so weitergehen kann. Beim Programmieren würde man das eine **Endlosschleife** nennen. Und im Prinzip ist es bei Turingmaschinen auch so zu verstehen. Wir würden in so einem Fall jedenfalls sagen, dass die Maschine *nicht* terminiert.

Anders als die bisherigen Automaten können Turingmaschinen nicht nur Wörter konsumieren, sondern sie können auch Wörter generieren. Im Beispiel (12.1) kann man das Wort 0000, das sich nach dem Erreichen des Endzustandes auf dem Band befindet, als „Ausgabe“ der Maschine interpretieren. Formal geht das so:

Definition

Eine Turingmaschine $T = (S, I, \Gamma, \delta, s_0, \square, \vdash, F)$ **berechnet** aus der **Eingabe** $w \in I^*$ die **Ausgabe** $w' \in I^*$, wenn es einen Zustand $s \in F$, Zahlen $m, n \in \mathbb{N}$ und Wörter $v, v' \in \Gamma^*$ mit $\vdash [s_0]w\square \Rightarrow_T^* \vdash v[s]v'$ und $vv' = \square^m w' \square^n$ gibt.

Das sieht kniffliger aus, als es ist. Informell bedeutet es: Das Wort w wird hinter der Grenzmarkierung auf das Band geschrieben, die Turingmaschine in den Startzustand s_0 versetzt und der Kopf auf dem Anfang des Wortes platziert. Die Maschine berechnet daraus w' , wenn sie diese Startkonfiguration in eine überführen kann, bei der sie in einem Endzustand terminiert und sich danach auf dem Band außer Leerzeichen und der Grenzmarkierung nur noch das Wort w' befindet. (v und v' werden nur benötigt, damit es egal ist, wo auf dem Band der Kopf stehenbleibt. m und n geben die Anzahl der Leerzeichen um w' herum an und sind irrelevant. Beachten Sie jedoch, dass vorsichtshalber hinter w ein Leerzeichen platziert wird, damit garantiert ist, dass der Kopf nicht „ins Leere schaut“ – denn w könnte ε sein.)

Durch diese Definition wird die von T berechnete partielle Funktion $f_T: I^* \rightarrow I^*$ determiniert, die für solche Wörter w den Funktionswert $f_T(w) = w'$ hat und für andere Wörter (bei deren Eingabe T entweder nicht terminiert oder in einem Zustand terminiert, der kein akzeptierender ist) nicht definiert ist.

berechnete Funktion
 f_T

Aufgabe 12.9. Geben Sie zu den beiden obigen Beispielen T_1 und T_2 jeweils die berechnete Funktion an.

Aufgabe 12.10. Was macht die Turingmaschine

$$T = (\{q_0, \dots, q_4\}, \Sigma_{\text{bool}}, \Sigma_{\text{bool}} \cup \{\square, \vdash\}, \delta, q_0, \square, \vdash, \{q_3\}),$$

deren Übergangsfunktion durch die folgende Tabelle definiert ist?

	q_0	q_1	q_2	q_4
0	$(q_0, 0, \rightarrow)$	$(q_1, 0, \rightarrow)$	$(q_2, 0, \leftarrow)$	$(q_4, 1, \leftarrow)$
1	$(q_1, 1, \rightarrow)$	$(q_1, 1, \rightarrow)$	$(q_2, 1, \leftarrow)$	$(q_2, 0, \leftarrow)$
$\square$	$(q_2, \square, \leftarrow)$	$(q_4, \square, \leftarrow)$	$\perp$	$\perp$
$\vdash$	$\perp$	$\perp$	$(q_3, \vdash, \rightarrow)$	$\perp$

Aufgabe 12.11. Geben Sie eine Turingmaschine an, die aus Eingaben über dem Alphabet Σ_{bool} alle Einsen entfernt und die Nullen entsprechend zusammenrückt. Aus der Eingabe 011001 soll also z. B. 000 werden. (Sie können zum Ausprobieren auch mit dem Simulator `tm2` arbeiten und Ihr „Programm“ dann in die mathematische Syntax übersetzen.)

Aufgabe 12.12. Konstruieren Sie eine Turingmaschine, die **Palindrome** über Σ_{bool} erkennen kann. Bei Eingaben wie 110011, 01110 oder auch 1 und ε soll die Ausgabe 1 (für „ja“) sein,⁸ bei Eingaben wie 1100 hingegen 0 (für „nein“).

Aufgabe 12.13. Schreiben Sie ein `tm2`-Programm, das Aufgabe 12.12 löst. (Zur Kontrolle finden Sie einen Lösungsvorschlag im [Docker-Image](#).)

Aufgabe 12.14. In der Literatur werden auch sogenannte *Mehrspur-Turingmaschinen* beschrieben, bei denen das Band aus mehreren „Spuren“ besteht, die mit fest verbundenen Schreib-Lese-Köpfen bearbeitet werden. Die Köpfe bewegen sich also immer gleichzeitig über denselben Positionen wie bei klassischen [Festplattenlaufwerken](#) und

Mehrspur-
Turingmaschine

⁸Es sollen also Palindrome ungerader und auch gerader Länge erkannt werden.

Mehrband-
Turingmaschine

die Maschine befindet sich immer nur in einem Zustand zur Zeit. Ein- und Ausgabe würden typischerweise über die erste Spur realisiert werden, während die anderen Spuren für „Notizen“ oder „Zwischenrechnungen“ zur Verfügung stünden. Wie könnte man so eine Maschine mit einer simulieren, die wie unsere bisherigen nur eine „Spur“ hat?

Aufgabe 12.15. *Mehrband-Turingmaschinen* sind eine Erweiterung des Konzepts aus Aufgabe 12.14. Sie unterscheiden sich von den Mehrspur-Maschinen dadurch, dass jede Spur (die man nun Band nennt) einen eigenen Schreib-Lese-Kopf hat, der unabhängig von den anderen arbeiten kann. Nach wie vor gibt es aber eine gemeinsame „Steuerungslogik“. Auch hier sollten Sie sich überlegen, wie man so etwas mit den Mitteln simulieren könnte, die wir bereits haben, z. B. mit einer Mehrspur-Maschine. Wenn Ihnen auch nach längerem Überlegen nichts einfällt, schauen Sie sich die Lösung an. (Aber geben Sie nicht nach zehn Minuten schon auf!)

Aufgabe 12.16. Unter der URL <https://aturingmachine.com/> finden Sie die Beschreibung (inkl. Video) einer „realen“ Turingmaschine, die der Amerikaner Mike Davey vor einigen Jahren gebaut hat. Abgesehen davon, dass das Resultat ein kleines Meisterwerk ist, hilft es vielleicht auch dabei, sich die Arbeitsweise einer Turingmaschine vorzustellen.

Aufgabe 12.17. Die Maschine aus dem Video in Aufgabe 12.16 arbeitet mit einem beidseitig unbeschränkten Band, also einem, das keine Grenzmarkierung hat, sondern auch nach links durch Anfügen von Leerzeichen bei Bedarf wächst. Sie ahnen schon, was jetzt kommt: Wie könnte man so etwas mit „unseren“ Turingmaschinen simulieren?

★**Aufgabe 12.18.** Bereits die Originalkonstruktion von Turing arbeitete mit einem Band, auf dem die Kästchen eindimensional in einer Reihe angeordnet sind. Menschen, die mit Kästchenpapier arbeiten, haben jedoch zwei Dimensionen zur Verfügung: außer links/rechts auch noch oben/unten. Überlegen Sie sich, wie die Befehle für eine zweidimensionale Turingmaschine aussehen müssten. Kann man so etwas mit „unseren“ Turingmaschinen simulieren?

Aufgabe 12.19. Bei den vorherigen Aufgaben ging es immer darum, scheinbar „mächtigere“ Turingmaschinen durch einfachere zu simulieren (und damit zu zeigen, dass Erstere gar nicht wirklich mehr können). Nun gehen wir einen umgekehrten Weg. Was wäre, wenn man die „Bewegung“ $\diamond$ nicht erlauben⁹ würde? Würde man dadurch wirklich etwas verlieren oder könnte man das auch durch Simulation auffangen?

Aufgabe 12.20. Konstruieren Sie eine Turingmaschine zum Eingabealphabet Σ_{bool} , die ein Wort, das anfangs direkt hinter der Grenzmarkierung steht, auf dem Band um eine Stelle nach rechts verschiebt. Konkret soll aus einem initialen Bandinhalt der Form $\vdash w \sqcap$ der Bandinhalt $\vdash \sqcap w \sqcap$ werden.

Aufgabe 12.21. Analog zu Aufgabe 6.9 könnte man sich fragen, ob die am Anfang des Abschnitts definierten Turingmaschinen nicht auch mit *einem* akzeptierenden Zustand auskommen könnten. Was meinen Sie?

⁹ δ wäre also eine partielle Funktion von $S \times \Gamma$ nach $S \times \Gamma \times \{\leftarrow, \rightarrow\}$. Anders ausgedrückt würde in den Simulationsprogrammen an der dritten Stelle nur L, R oder H, aber nicht mehr S stehen dürfen.

Aufgabe 12.22. Natürlich kann [FLACI](#) auch Turingmaschinen simulieren und man findet online noch weitere Tools, die das erledigen können. Schauen Sie sich aber jeweils genau an, inwieweit deren Spezifikationen von denen in diesem Skript abweichen.

★**Aufgabe 12.23.** Erstaunlicherweise war die Idee [vom Anfang des Kapitels](#) gar nicht so abwegig. Kellerautomaten mit zwei Stacks können Turingmaschinen simulieren! Haben Sie eine Idee, wie das ungefähr ablaufen könnte?

Aufgabe 12.24. Schreiben Sie ein Programm für `tm2`, das aus der Eingabe n das Ergebnis $n - 2$ berechnet. Die Eingaben $n = 0$ und $n = 1$ sollen nicht verändert werden. (Wenn bei `tm2`-Programmen nicht explizit etwas anderes verlangt wird, ist immer gemeint, dass Zahlen [binär](#) dargestellt werden.)

Aufgabe 12.25. Schreiben Sie ein `tm2`-Programm, das zwei Zahlen (die durch ein Leerzeichen getrennt eingegeben werden) wie folgt addiert: In einer Schleife wird abwechselnd die erste Zahl um eins verringert und die zweite um eins erhöht. (Sie können dafür die Instruktionen aus `Code/inc . tm2` und `Code/dec . tm2` verwenden.) Die Schleife endet, wenn die erste Zahl null ist. Dann wird die zweite Zahl (die nun die Summe ist) an den Anfang des Bandes geschoben.

★**Aufgabe 12.26.** Die Lösung von Aufgabe [12.25](#) braucht m Schleifendurchläufe zur Berechnung von $m + n$. Schreiben Sie ein `tm2`-Programm, das dieselbe Aufgabe effizienter löst, indem wie bei der schriftlichen Addition gerechnet wird.

13. Die Church-Turing-These

Im vorherigen Abschnitt konnte man sehen, wie aufwendig selbst simple Aufgaben wie die Addition oder Subtraktion zweier Zahlen für Turingmaschinen sind. Von Multiplikation oder Division wollen wir gar nicht erst reden! Diese extrem primitiven Vorrichtungen wurden für mathematische Beweise erdacht und waren nie dafür vorgesehen, physisch realisiert zu werden. Wir wollen uns aber im Folgenden davon überzeugen, dass Turingmaschinen in gewissem Sinne genauso leistungsfähig wie moderne Computer sind. Dafür müssen wir zunächst klarstellen, wie das gemeint ist.

Was ist überhaupt ein „Programm“?

Wenn Sie den ["Taschenrechner" von Windows](#) (oder ein vergleichbares Programm eines anderen Betriebssystems) verwenden, können Sie „ $2 + 5 =$ “ tippen und erhalten die Antwort 7. Die Turingmaschine `add . tm`, die Sie im letzten Abschnitt kennengelernt haben, kann aus dem Eingabeband `II-IIIIII` das Ausgabeband `IIIIIIII` machen. Aber ist das wirklich dasselbe? Man könnte argumentieren, dass `add . tm` nur Striche hin- und herschiebt, die wir lediglich als Zahlen *interpretieren*. Diese Sichtweise ignoriert jedoch, was im Falle des Taschenrechners „unter der Haube“ passiert.

Wenn wir etwas tippen, ist unsere Eingabe ein physischer Vorgang (das Berühren der Tasten), der durch die Tastatur in elektronische Signale konvertiert wird, die

wiederum in einer komplizierten Abfolge von Ereignissen vom Betriebssystem verarbeitet, umgewandelt und an das Rechnerprogramm weitergeleitet werden. Ebenso besteht die Ausgabe 7, die wir sehen, aus Pixeln auf dem Bildschirm, die das Betriebssystem auf Ersuchen des „Taschenrechners“ auf eine bestimmte Art arrangiert hat. Wir *interpretieren* diese nur als die Zahl 7, weil wir das Symbol dafür wiedererkennen. Was der Rechner intern gemacht hat, sehen wir nicht; doch eigentlich hat er auch nur „Striche hin- und hergeschoben“. (Mit dem Unterschied, dass die „Striche“ **Bits** sind, die durch die An- oder Abwesenheit von **Elektronen** im **RAM** des Computers repräsentiert werden.)

Der entscheidende Punkt ist, dass wir in der Theoretischen Informatik nur an Programmen interessiert sind, die aus irgendeiner Art von Eingabe irgendeine Art von Ausgabe machen.¹⁰ Wir **abstrahieren** von den technischen Details von Ein- und Ausgabe, solange wir eine konsistente *Interpretation* derselben haben. In diesem Sinne macht die Turingmaschine `add.tm` bei der Addition von 2 und 5 dasselbe wie das Windows-Programm.

Das wird die Sichtweise auf den folgenden Seiten sein. Dazu gehört auch, dass wir uns vorerst nicht um Geschwindigkeit oder Speicherverbrauch kümmern. Ein moderner **Prozessor** kann selbstverständlich zwei Zahlen um mehrere **Größenordnungen** schneller addieren als eine Turingmaschine, die von einem **Julia**-Programm simuliert wird. Zunächst wird es jedoch nur darum gehen, ob etwas überhaupt geht. (Die Frage, wie lange bestimmte Vorgänge dauern, behandeln wir **später**.)

RISC-V 32I

Es gibt heutzutage eine Unmenge von Programmiersprachen.¹¹ Aber im Endeffekt wird jedes Programm von einem **Compiler** oder **Interpreter** in die „Muttersprache“ der **CPU** übersetzt: die **Maschinensprache**. Wenn wir also darüber reden, was Computer können oder nicht können, dann reicht es, über die Fähigkeiten der CPU nachzudenken.

Moderne CPUs wie beispielsweise solche mit der Architektur **x86-64**, die in den meisten Linux- und Windows-Rechnern (und auch in „alten“ **Macs**) zu finden sind, können sehr viel. Neben einfacher Arithmetik (Addition, Multiplikation, Subtraktion und Division) mit **ganzen Zahlen** und **Fließkommazahlen** können sie auch komplexere Aufgaben durchführen wie das Berechnen von **Logarithmen** oder **trigonometrischen Funktionen**. Aber was davon benötigt man wirklich?

Man könnte das mit einem Blick auf alte Rechner beantworten. Die hatten wesentlich primitivere CPUs wie den **Intel 8088** im ersten **IBM PC** oder den Chip **MOS Technology 6502** im **Apple II**. Der 6502 konnte z. B. nicht mit Fließkommazahlen rechnen und weder multiplizieren noch dividieren. Aber für diese Computer gab

¹⁰ Insbesondere kümmern wir uns nicht um interaktive Programme wie **Webbrowser** oder **Entwicklungsumgebungen**, die die meiste Zeit nur auf Eingaben warten.

¹¹ Nach manchen Schätzungen wurden in den letzten Jahrzehnten mehr als 10 000 Sprachen entwickelt. Viele von denen werden allerdings Nischensprachen für sehr spezielle Anwendungen sein.

es trotzdem schon [Tabellenkalkulationsprogramme](#), [Textverarbeitungsprogramme](#) und – noch wichtiger – Compiler und Interpreter für diverse Programmiersprachen.

Wir brauchen jedoch nicht ins letzte Jahrtausend zu schauen. Selbst heutzutage werden sehr einfache CPUs in vielfältiger Weise verwendet. Einige bekannte Exemplare verwenden die Architektur [RISC-V](#), die an der [University of California, Berkeley](#) entwickelt wurde. Sie hat einerseits den Vorteil eines sehr sauberen, gewissermaßen „akademischen“ Designs, während andererseits diverse Firmen tatsächlich Prozessoren herstellen und verkaufen, die auf ihr beruhen. Zudem wird die Architektur von einer Reihe von [quelloffenen](#) Tools unterstützt. Deshalb werden wir sie im Folgenden als Grundlage wählen, und zwar in ihrer einfachsten Variante, die [RV32I](#) genannt wird.¹²

Die Beispiele im [Begleitdokument](#) demonstrieren, wie diverse Programme, die in der [Sprache C](#) geschrieben sind, in Maschinensprache für eine RV32I-CPU übersetzt werden und wie diese auf einem mit der [Virtualisierungssoftware QEMU](#) simulierten Prozessor dieses Typs ausgeführt werden. Gezeigt werden unter anderem das klassische [Hallo-Welt-Programm](#), die Multiplikation [ganzer Zahlen](#), die Berechnung des [Logarithmus](#) von [Fließkommazahlen](#) und – gewissermaßen als „Krönung“ – eine Simulation von Turingmaschinen, die dasselbe macht wie das [Julia-Programm tm](#).¹³

Das scheint auf den ersten Blick nicht besonders beeindruckend zu sein, aber man beachte dabei, dass RV32I wie der 6502 von 1975 weder Fließkommazahlen kennt noch multiplizieren oder dividieren kann! Es gibt lediglich [etwa 40 Instruktionen](#) für Addition, Subtraktion, [bitweise Operationen](#), [Speicherzugriffe](#) und einfache [Sprunganweisungen](#). Trotzdem kann man prinzipiell jedes C-Programm in RV32I übersetzen. Das sollte Sie hoffentlich davon überzeugen, dass man – zumindest theoretisch – auch jedes Programm in irgendeiner anderen Programmiersprache in RV32I übersetzen kann. (Eine praktische Komplikation ist, dass man Funktionalitäten wie z. B. [dynamische Speicherverwaltung](#), die typischerweise Aufgabe des [Betriebssystems](#) sind, selbst implementieren müsste. Dass es sich dabei um kein unüberwindbares Problem handelt, kann man damit begründen, dass sowohl der [Kernel](#) von Linux als auch die meisten gängigen Linux-Programme in C geschrieben sind.¹⁴)

Aufgabe 13.1. Einige der in den letzten Absätzen genannten Beispiele werden in der Vorlesung vorgeführt. Sie sollten sich diese aber trotzdem im [Docker-Image](#) näher anschauen und ein paar Dinge selbst ausprobieren.

¹²Unter https://www.robertwinkler.com/projects/riscv_book/riscv_book.html finden Sie ein kostenloses Buch, das sich gut als Ausgangspunkt eignet, wenn Sie sich eingehender mit der Programmierung von RISC-V beschäftigen wollen.

¹³Aus didaktischen Gründen wurde der „Kern“ von `tm` nicht in Julia, sondern in C geschrieben, so dass der entsprechende Code, den man in `TM/src/sim.c` findet, auch von RV32I aus verwendet werden kann.

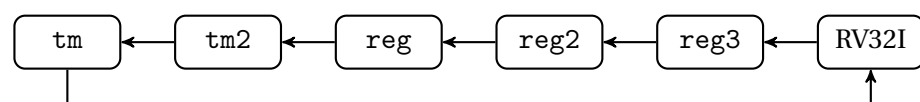
¹⁴Ich weiß, dass in der jüngeren Vergangenheit teilweise auch andere Sprachen wie [Rust](#) verwendet wurden. Das macht jedoch aktuell (2026) weniger als ein Prozent des Codes aus und ändert am generellen Argument natürlich nichts.

Von RV32I zur Turingmaschine

Das [Docker-Image](#) enthält Programme, die nach und nach immer komplexere Rechnermodelle implementieren, gleichzeitig jedoch in jedem Schritt zeigen, wie man das neue Modell mit dem alten simulieren kann. Diese Schritte werden in der Vorlesung vorgeführt, im zugehörigen PDF ausführlich erklärt und hier daher nur kurz zusammengefasst:

- Das erste Modell (reg) ist eine primitive [Registermaschine](#),¹⁵ die lediglich drei Befehle beherrscht. Es wird gezeigt, wie man Programme für solche Maschinen mit Turingmaschinen vom Typ `tm2` simulieren kann. Dabei wird ein einfacher [Makroprozessor](#) eingesetzt, der auch in den folgenden Schritten verwendet wird.
- Das zweite Modell (reg2) ist eine Registermaschine, die normale Ganzzahlarithmetik beherrscht sowie komfortable Sprunganweisungen. Zusätzlich implementiert sie eine bestimmte Form der *Gödelisierung*, die [weiter unten](#) erläutert wird. Es wird demonstriert, wie man Programme für reg2 in Programme für reg übersetzen kann, die – wenn auch mit erheblich größerem Aufwand – dasselbe tun.
- Die dritte und letzte Registermaschine (reg3) verwendet die erwähnten Gödelisierungsfunktionen, um ein typisches [RAM](#) und [Prozessorregister](#) zu implementieren. Zudem beherrscht sie diverse Funktionen, die auf 32-Bit-Wörter und [Zweierkomplement](#)-Arithmetik zugeschnitten sind. Selbstverständlich wird auch gezeigt, wie man Programme für das Modell reg3 in reg2-Programme übersetzen kann.
- Schließlich wird ein Programm für das Modell reg3 gezeigt, das einen [RV32I](#)-Rechner simulieren kann. Insbesondere kann die im letzten Abschnitt erwähnte Simulation des ersten Turingmaschinen-Typs (die dasselbe macht wie `tm`) in diesem Programm ausgeführt werden.

Damit haben wir die folgende Situation, wobei ein Pfeil von *A* nach *B* für „*A* kann durch *B* simuliert werden“ steht:



RV32I ganz rechts steht dabei gemäß der Überlegungen weiter oben stellvertretend für „irgendeine Programmiersprache“. Der unten verlaufende Pfeil von links nach rechts ist nicht weiter überraschend. Natürlich kann man so simple Vorrichtungen wie Turingmaschinen mit einer modernen Programmiersprache simulieren. Erstaunlich ist hingegen die Folge der Pfeile von rechts nach links: Man kann umgekehrt mit Turingmaschinen alle Programme simulieren, die auf einem modernen Computer ablaufen! Das führt zu der folgenden Aussage, von der Sie hoffentlich durch die vorherigen Seiten überzeugt wurden:

¹⁵Die [englische Wikipedia-Seite](#) zu Registernmaschinen ist wesentlich ausführlicher als die deutsche. Das Modell, das wir anfangs betrachten, ist eigentlich eine *counter machine*, basierend auf einem Design von [Marvin Minsky](#).

Church-Turing-These

Jede intuitiv berechenbare Funktion kann von einer Turingmaschine berechnet werden.

Der amerikanische Mathematiker [Alonzo Church](#), der [Doktorvater](#) von Turing, formulierte diese These (in etwas anderer Form) 1936 und Turing und andere sahen das damals ebenso. Der ursprüngliche Anlass war, dass in den 1930er Jahren neben den Turingmaschinen noch weitere Modelle dafür entwickelt worden waren, wie Berechnungsvorgänge ablaufen, wobei die wichtigsten die *μ -rekursiven Funktionen* (von Gödel und dem Franzosen [Jacques Herbrand](#)) und Churchs *λ -Kalkül* sind. Es stellte sich heraus, dass diese Modelle alle äquivalent waren: mit ihnen konnten jeweils dieselben Klassen von Funktionen berechnet werden. In den kommenden Jahrzehnten wurden noch diverse weitere Modelle vorgeschlagen, z. B. die erwähnten Registermaschinen, aber alle stellten sich als äquivalent zu den bereits vorhandenen heraus.

Trotzdem ist diese Aussage eine *These* und kein mathematisches [Theorem](#). Allein schon deshalb, weil es keine klare Definition dafür gibt, was „intuitiv berechenbar“ bedeuten soll, kann man sie nicht beweisen. Man könnte sie höchstens widerlegen, indem man ein mächtigeres Modell entwickelt. Das ist aber in den letzten 90 Jahren niemandem gelungen. Und inzwischen sind wohl so ziemlich alle Experten der Meinung, dass die These wahr ist. Man könnte fast von einem [Axiom](#) der Informatik sprechen.

Aufgabe 13.2. An dieser Stelle kommt oft die Frage, ob die Church-Turing-These nicht durch [Quantencomputer](#) oder [Künstliche Intelligenz](#) (KI) überholt ist. Kann man mit diesen Werkzeugen nicht mehr erreichen als mit herkömmlichen Computerprogrammen?

Aufgabe 13.3. In gewissem Sinne sind theoretische Modelle wie Turing- oder Registermaschinen sogar „mächtiger“ als echte Computer. Haben Sie eine Idee, wie das gemeint sein könnte?

★**Aufgabe 13.4.** Eine andere Herangehensweise, in der im Endeffekt auch Programme einer typischen Programmiersprache in Turingmaschinen übersetzt werden, finden Sie nebst Begleitsoftware im „alten“ [Skript](#). Die Methode ist in gewissen Sinne direkter, allerdings ohne den zugehörigen Quellcode und nicht auf der Basis einer real existierenden CPU. Trotzdem könnten Sie sich das bei Interesse ja einmal anschauen.

In der Fachsprache dreht man ürigens den Spieß um und nennt eine Programmiersprache oder ein abstraktes Modell, das alles berechnen kann, was eine Turingmaschine berechnen kann, *turingvollständig*. In diesem Sinne sind alle gängigen Programmiersprachen wie [C](#), [Java](#) oder [Lisp](#) turingvollständig. Es gibt aber auch sehr überraschende Beispiele für Turingvollständigkeit wie etwa das *Spiel des Lebens* des englischen Mathematikers [John Conway](#) oder *Minecraft*.

turingvollständig

Fractran

Ein weiteres Beispiel stammt ebenfalls von Conway und lässt sich leicht erklären. Es handelt sich um die „Programmiersprache“ **Fractran** (ein Kofferwort aus *fraction* und *Fortran*). Fractran-Programme sind einfach Listen von Brüchen wie diese:

$$\frac{17}{91} \quad \frac{78}{85} \quad \frac{19}{51} \quad \frac{23}{38} \quad \frac{29}{33} \quad \frac{77}{29} \quad \frac{95}{23} \quad \frac{77}{19} \quad \frac{1}{7} \quad \frac{11}{13} \quad \frac{13}{11} \quad \frac{15}{14} \quad \frac{15}{2} \quad \frac{55}{1} \quad (13.1)$$

Das Wort *Programmiersprache* steht oben zwischen **Tüddelchen**, weil es bei solchen Modellen (wie auch bei den oben erwähnten Spielen) auf die Interpretation ankommt, was man als Ein- und Ausgabe betrachtet. (Darüber **sprachen wir schon**.) Unter dem richtigen Blickwinkel betrachtet ist (13.1) jedoch ein (erstaunlich kurzes) Programm, das in der Lage ist, eine sortierte Liste aller Primzahlen auszuwerfen. Details darüber, wie Fractran funktioniert und warum das Modell turingvollständig ist, werden in dem Video *Die einfachste Programmiersprache der Welt* erläutert.

Wir können nun zwei grundlegende Begriffe definieren und sie angesichts der Vorüberlegungen gleich mit Leben füllen.

Definition

Sei Σ ein Alphabet. Eine partielle Funktion $f: \Sigma^* \rightarrow \Sigma^*$ heißt **berechenbar**, wenn es eine Turingmaschine T_1 zum Alphabet Σ gibt, für die $f_{T_1} = f$ gilt. Eine Sprache L über Σ heißt **entscheidbar** oder **rekursiv**, wenn es Wörter $w_1, w_2 \in \Sigma^*$ mit $w_1 \neq w_2$ und eine Turingmaschine T_2 zum Alphabet Σ gibt, für die für alle $w \in \Sigma^*$ das Folgende gilt:

$$f_{T_2}(w) = \begin{cases} w_1 & w \in L \\ w_2 & w \notin L \end{cases}$$

Formulierungen wie „wenn es eine Turingmaschine gibt“ kann man aufgrund der Church-Turing-These im Geiste immer durch „wenn man es irgendwie programmieren könnte“ übersetzen. Denn wenn Sie sich beispielsweise vorstellen können, eine Funktion in **Python** zu programmieren, dann wissen Sie nun, dass es auch eine Turingmaschine geben muss, die dasselbe leistet.

Die Wörter w_1 und w_2 in der Definition spielen einfach die Rolle von *ja* und *nein*. Wichtig ist, dass die Funktion f_{T_2} , die für die Entscheidbarkeit sorgt, **total** ist. Sie ist also auf ganz Σ^* definiert, d. h., T_2 muss bei *jeder* Eingabe terminieren.

Wir werden diese Konzepte übrigens auch häufig im Zusammenhang mit Zahlen verwenden und dann beispielsweise sagen, dass eine Funktion *berechenbar* ist, deren Definitions- oder Wertebereich aus natürlichen Zahlen besteht, oder dass eine Teilmenge von $\mathbb{N}$ *entscheidbar* ist. Das passt rein formal nicht zur Definition der Begriffe, ist dann aber natürlich so gemeint, dass die Zahlen entsprechend codiert sind – typischerweise über dem Alphabet Σ_{bool} im **Dualsystem**. (In Abschnitt 15 wird das noch einmal präzisiert.)

Aufgabe 13.5. Begründen Sie anhand der Definition (und ggf. mithilfe von Programmen aus dem [Docker-Image](#)), dass die durch $g(n) = n + 1$ definierte Funktion g von $\mathbb{N}$ nach $\mathbb{N}$ berechenbar ist.

Aufgabe 13.6. Begründen Sie anhand der Definition (und ggf. mithilfe von Programmen aus dem [Docker-Image](#)), dass die Palindrom-Sprache $\{ww^R : w \in \Sigma_{\text{bool}}^*\}$ entscheidbar ist.

Aufgabe 13.7. Ist $\mathbb{P}$ entscheidbar?

Aufgabe 13.8. Wieso sind alle endlichen Mengen entscheidbar?

Aufgabe 13.9. Begründen Sie, warum es zu jeder entscheidbaren Menge von natürlichen Zahlen unendlich viele Programme für `tm2` gibt, die sie entscheiden.

Gödelisierung

Ein Detail der Simulationskaskade aus dem letzten Abschnitt wollen wir uns etwas genauer anschauen. Um einen „richtigen“ Rechner mit einem [RAM](#) zu simulieren, wurde der Arbeitsspeicher des simulierten Computers in der Registermaschine durch eine einzige Zahl dargestellt. Dieses Konzept wird in der Theoretischen Informatik häufig verwendet. Es beruht auf einer Idee des österreichischen Mathematikers [Kurt Gödel](#), der diese zuerst 1931 im Beweis seines berühmten [Unvollständigkeitssatzes](#) einsetzte – auf dem übrigens [Turing's Arbeit von 1936](#) basierte.

Sei L eine formale Sprache. Eine Funktion $g: L \rightarrow \mathbb{N}$ wird **Gödelisierung** von L genannt, wenn sie die folgenden Bedingungen erfüllt:

- (i) g ist [injektiv](#)¹⁶ und berechenbar.
- (ii) $g[L]$ ist entscheidbar.
- (iii) Die Umkehrfunktion $g^{-1}: g[L] \rightarrow L$ ist berechenbar.

Den Funktionswert $g(w)$ nennt man die **Gödelnummer** des Wortes w .

Definition

Die Anforderungen an g kann man umgangssprachlich so beschreiben:

- (i) g soll verschiedenen Wörtern verschiedene Zahlen zuordnen. Und es muss ein Computerprogramm geben,¹⁷ das für jedes Wort w von L die Gödelnummer $g(w)$ berechnen kann.
- (ii) Es muss ein Computerprogramm geben, das erkennen kann, ob eine natürliche Zahl die Gödelnummer eines Wortes aus L ist.
- (iii) Es muss ein Computerprogramm geben, das aus der Gödelnummer eines L -Wortes das Wort rekonstruieren kann.

¹⁶Falls Sie vergessen haben, was Injektivität ist oder was die Schreibweise $g[L]$ bedeutet, schauen Sie im [Matheskript](#) nach.

¹⁷Die hier erwähnten Programme muss es nicht wirklich geben. Es reicht, wenn man sich überzeugt, dass man sie schreiben *könnte*. Es geht ja um *Theoretische* Informatik.

Aufgabe 13.10. Fällt Ihnen für die Sprache Σ_{lat}^* eine Gödelisierung ein, die in der Praxis tatsächlich verwendet wird?

kanonische
Gödelisierung

Gödels ursprüngliche Gödelisierung ist prinzipiell für *alle* formalen Sprachen geeignet.¹⁸ Wir werden sie in Zukunft verwenden und die *kanonische Gödelisierung* nennen. Sie ist für die Praxis nicht zu gebrauchen, weil die Gödelnummern sehr groß werden, für die Theorie ist sie aber ideal. Sie beruht auf dem **Fundamentalsatz der Arithmetik**,¹⁹ den ich als bekannt voraussetze:

- (i) Jedem Zeichen a des Alphabets Σ wird injektiv eine natürliche Zahl n_a zugeordnet. Der Einfachheit halber nimmt man typischerweise die Zahlen 1 bis m , wenn Σ aus m Symbolen besteht.
- (ii) Ist nun L eine Sprache über Σ und w das Wort $(\sigma_1, \dots, \sigma_k)$ aus L , so wird w die natürliche Zahl

$$g(w) = p_1^{n_{\sigma_1}} p_2^{n_{\sigma_2}} \cdots p_k^{n_{\sigma_k}}$$

zugeordnet, wobei p_i die i -te **Primzahl** sein soll.

Aufgabe 13.11. Sei Σ das Alphabet $\{a, b, c\}$ mit der naheliegenden Zuordnung $n_a = 1$, $n_b = 2$, $n_c = 3$. L sei die Sprache Σ^* und g die kanonische Gödelisierung. Geben Sie $g(\text{abba})$ an.

Aufgabe 13.12. Schreiben Sie für das Beispiel aus Aufgabe 13.11 in einer Programmiersprache Ihrer Wahl Funktionen, die den drei Computerprogrammen in der Definition des Begriffs Gödelisierung entsprechen. (Es ist empfehlenswert, eine Sprache zu wählen, in der es bereits Funktionen für **Primfaktorzerlegung** und die i -te Primzahl gibt.)

Aufgabe 13.13. Begründen Sie, wie man die Idee der kanonischen Gödelisierung auch auf die Menge

$$L = \bigcup_{k \in \mathbb{N}} \mathbb{N}^k = \mathbb{N}^0 \cup \mathbb{N}^1 \cup \mathbb{N}^2 \cup \dots$$

aller **Tupel** aus natürlichen Zahlen anwenden kann, obwohl L *keine* formale Sprache ist. Es geht also darum, jedem beliebigen Tupel natürlicher Zahlen eindeutig eine natürliche Zahl zuzuordnen.

Aufgabe 13.14. Begründen Sie, warum die Zahl 42 nicht als Funktionswert der Funktion g aus Aufgabe 13.13 vorkommen kann.

Aufgabe 13.15. Schreiben Sie in einer Programmiersprache Ihrer Wahl Funktionen die das Auslesen und Speichern von natürlichen Zahlen für ein „**Array**“ simulieren, das wie in Aufgabe 13.13 gödelisiert ist. (Der Inhalt des Arrays soll sich also die ganze Zeit in einer einzigen natürlichen Zahl befinden.)

¹⁸Genauer: für alle Sprachen, die man überhaupt gödelisieren kann. Erinnern Sie sich an das Ende von Kapitel I.

¹⁹Kapitel 8 in **KMFI**.

Aufgabe 13.16. Implementieren Sie analog zu Aufgabe 13.15 einen **Stack**, der mit den Operationen push und pop modifiziert werden kann, als eine einzige Zahl.

Die kanonische Gödelisierung ist mathematisch elegant, aber erfordert für die Umsetzung unter anderem das Programmieren einer **Primfaktorzerlegung**. Im **Docker-Image** wurde daher eine einfacher zu implementierende Variante gewählt, die folgendermaßen arbeitet:

- Paare (m, n) von natürlichen Zahlen werden durch die Zahl $2^m \cdot (2n + 1)$ codiert.
- Allgemeine **Tupel** von natürlichen Zahlen werden mithilfe von Paaren **rekursiv** als **verkettete Listen** codiert. Das (einzige) 0-Tupel ist die Zahl 1. Ist $n > 0$, so wird das n -Tupel $(a_1, a_2, \dots, a_n)$ als Paar mit der ersten Komponente a_1 und der zweiten Komponente $(a_2, \dots, a_n)$ codiert. (Diese Technik wird z. B. seit fast 70 Jahren in **Lisp** verwendet.)
- Das **RAM** eines Computers kann dann als Liste (Tupel) von Paaren der Form $((a_1, b_1), \dots, (a_n, b_n))$ codiert werden. In der Speicherstelle mit der Adresse a_i steht dabei der Wert b_i . (Einigt man sich darauf, dass in allen Speicherzellen initial die Zahl 0 steht, so muss man nur die Zellen in der Liste vorhalten, in denen ein anderer Wert steht.)

Aufgabe 13.17. Begründen Sie, dass die gerade beschriebene Codierung von Paaren natürlicher Zahlen (also von $\mathbb{N} \times \mathbb{N}$) die Anforderungen an eine Gödelisierung erfüllt.

Aufgabe 13.18. Implementieren Sie analog zu Aufgabe 13.15 auch für diese Form der Gödelisierung ein „Array“.

14. Akzeptierende Turingmaschinen und formale Sprachen

Es ist in der Theoretischen Informatik üblich, mit Turingmaschinen zu argumentieren, weil das mathematische Modell, auf dem sie beruhen, so einfach ist. Wir werden das auf den folgenden Seiten auch machen. Die Überlegungen aus den letzten Abschnitten, die zur **Church-Turing-These** führten, erlauben es uns jedoch, auf das mühsame „Programmieren“ derselben in der Regel zu verzichten, solange wir die entsprechenden Vorgänge präzise genug beschreiben und uns vorstellen können, sie in einer gängigen Programmiersprache zu codieren. Wir klären als Erstes die Frage, welche Klasse von Sprachen zu den Turingmaschinen passt.

Eine Turingmaschine $T = (S, I, \Gamma, \delta, s_0, \square, \vdash, F)$ **akzeptiert** ein Wort $w \in I^*$, wenn es einen Zustand $s \in F$ und Wörter $v, v' \in \Gamma^*$ mit $\vdash [s_0] w \square \Rightarrow_T^* \vdash [s] v' v'$ gibt. Die **von T akzeptierte Sprache** ist dann definiert als

$$L(T) = \{ w \in I^* : T \text{ akzeptiert } w \}.$$

Definition

Mit anderen Worten: w wird akzeptiert, wenn die Maschine bei dieser Eingabe in einem akzeptierenden Zustand (daher der Name ...) terminiert.

Aufgabe 14.1. Diese Definition der Akzeptanz ähnelt sehr stark der [Definition der Berechnung](#). Welcher Zusammenhang besteht zwischen beiden?

nichtdeterministische
Turingmaschine

Für das weitere Vorgehen wird das bereits aus den vorhergehenden Kapiteln bekannte Konzept des Nichtdeterminismus hilfreich sein. Man kann aus einer gewöhnlichen Turingmaschine $T = (S, I, \Gamma, \delta, s_0, \square, \vdash, F)$, die nach unserer Definition deterministisch agiert, eine *nichtdeterministische Turingmaschine* machen, indem man für δ partielle Funktionen von $S \times \Gamma$ nach $\mathcal{P}(S \times \Gamma \times \{\leftarrow, \rightarrow, \diamond\})$ zulässt. Die Funktionswerte sind dann also nicht einzelne Tripel, sondern Mengen solcher Tripel (und zwar offenbar endliche). Hat so eine Menge mehr als ein Element, dann hat die Maschine an dieser Stelle eine Wahlmöglichkeit. Akzeptanz eines Wortes wird für nichtdeterministische Turingmaschinen sinnvollerweise so definiert, dass es ausreicht, wenn mindestens ein möglicher Berechnungsweg bei der Eingabe des Wortes in einem akzeptierenden Zustand terminiert.

Aufgabe 14.2. Verständnisfrage: Bei nichtdeterministischen Turingmaschinen könnte man auf das adjektiv *partiell* für δ verzichten. Wieso?

Aufgabe 14.3. Wie muss man [die Definition von \$\Rightarrow_T\$](#) ändern, damit sie für nichtdeterministische Turingmaschinen korrekt ist?

Wie bei [endlichen Automaten](#) (und [anders als bei Kellerautomaten](#)) hat man durch den Nichtdeterminismus quasi „nichts gewonnen“.

Satz 14.1

Zu jeder nichtdeterministischen Turingmaschine gibt es eine deterministische Turingmaschine, die dieselbe Sprache akzeptiert.

Beweis. Sei $T = (S, I, \Gamma, \delta, s_0, \square, \vdash, F)$ eine nichtdeterministische Turingmaschine. Da sowohl der Definitionsbereich der Übergangsfunktion δ als auch deren Funktionswerte endlich sind, hat die Maschine nur endlich viele „Entscheidungspunkte“. Konkret zählen wir alle Paare (x, y) aus $(S \times \Gamma) \times (S \times \Gamma \times \{\leftarrow, \rightarrow, \diamond\})$ auf, bei denen $y \in \delta(x)$ gilt,²⁰ und nennen sie e_1 bis e_w . Jedes Mal, wenn T eine Entscheidung²¹ trifft, entspricht das der Auswahl eines dieser e_i . Anders ausgedrückt lässt sich jeder mögliche terminierende Durchlauf von T durch ein Tupel von Zahlen aus $\{1, \dots, w\}$ beschreiben; und zwar in dem Sinne, dass das Tupel $(k_1, k_2, \dots, k_n)$ bedeutet, dass die Turingmaschine nacheinander die „Entscheidungen“ e_{k_1}, e_{k_2} bis e_{k_n} getroffen hat.

Diese Tupel kann man alle der Reihe nach aufzählen. Für $w = 2$ würde das beispielsweise so aussehen:

$$(), (1), (2), (1, 1), (1, 2), (2, 1), (2, 2), (1, 1, 1), (1, 1, 2), (1, 2, 1), \dots \quad (14.1)$$

Für die deterministische Turingmaschine T' , die T simulieren soll, verwenden wir nun der Einfachheit halber eine [Mehrband-Maschine](#) mit drei Bändern. Auf dem

²⁰Das sind quasi alle Einträge in einer Tabelle wie auf Seite 69, wobei im nichtdeterministischen Fall in einer Tabellenzeile jedoch mehrere Einträge stehen können.

²¹Faktisch gibt es nur etwas zu entscheiden, wenn $\delta(x)$ mehr als ein Element hat.

ersten Band steht die ganze Zeit unverändert das Eingabewort. Auf dem zweiten Band generiert T' nacheinander die Tupel der Sequenz (14.1). Und für jedes dieser Tupel wird das Eingabewort auf das dritte Band kopiert und dann auf diesem Band der Ablauf von T gemäß des Tupels simuliert. Wird dabei auf dem dritten Band ein akzeptierender Zustand erreicht, so hält T' an. Wenn nicht, wird das nächste Tupel ausgewählt. (Die Folge (14.1) wird sehr viele „unmögliche“ Tupel enthalten, die gar keinen möglichen Ablauf von T beschreiben, da sie entweder widersprüchliche Angaben machen oder „zu kurz“ sind. Das macht aber nichts. T' bricht in so einem Fall ab und wendet sich dem nächsten Tupel zu.)

Dieser Vorgang wäre in der Praxis aufgrund des hohen Aufwands nahezu undurchführbar, in der Theorie beweist er jedoch, was bewiesen werden sollte. ■

Mithilfe der nichtdeterministischen Turingmaschinen können wir nun zur Klassifizierung von Sprachen gemäß der [Chomsky-Hierarchie](#) zurückkehren.

Eine Grammatik (N, T, P, S) wird **kontextsensitiv** oder **Typ-1-Grammatik** genannt, wenn für alle Produktionen $(\alpha, \beta) \in P$ die Bedingung $|\alpha| \leq |\beta|$ erfüllt ist.²² Weil damit das leere Wort nicht abgeleitet werden kann, wird noch die **ε -Sonderregel** eingeführt: Die Produktion $S \rightarrow \varepsilon$ ist erlaubt, wenn S nie auf der rechten Seite einer Produktion auftaucht.

Eine beliebige formale Grammatik (N, T, P, S) wird auch **Phrasenstrukturgrammatik** oder **Typ-0-Grammatik** genannt.

Definition

Aufgabe 14.4. Welche der folgenden Grammatiken sind kontextsensitiv?

- (i) $S \rightarrow aS \mid abT, bT \rightarrow cc$
- (ii) $S \rightarrow aS \mid abT, abT \rightarrow cc$
- (iii) $S \rightarrow \varepsilon \mid aS \mid abT, bT \rightarrow cc$
- (iv) $S \rightarrow aS \mid abT, bT \rightarrow cc, T \rightarrow \varepsilon$
- (v) $S \rightarrow \varepsilon \mid a \mid abT, bT \rightarrow cc$

Eine Sprache wird **kontextsensitiv** genannt, wenn es eine kontextsensitive Grammatik gibt, die sie erzeugt. Eine Sprache wird **rekursiv aufzählbar** genannt, wenn es eine Phrasenstrukturgrammatik gibt, die sie erzeugt.

Definition

Zum Begriff „kontextsensitiv“ siehe die Erläuterung zu „kontextfrei“ am Anfang von Abschnitt 10. Woher der Begriff „rekursiv aufzählbar“ kommt, wird [später](#) noch deutlich werden.

Aufgabe 14.5. Mithilfe des Pumping-Lemmas 10.2 hatten wir begründet, dass die Sprache $\{a^k b^k c^k : k \in \mathbb{N}\}$ nicht kontextfrei ist. Zeigen Sie, dass diese Sprache kontextsensitiv ist.

²²In manchen Büchern wird das etwas umständlicher definiert und das, was hier definiert wurde, als *monotone Grammatik* bezeichnet. In beiden Fällen ergibt sich jedoch dieselbe Klasse von Sprachen.

Aufgabe 14.6. In Aufgabe 10.7 wurde gezeigt, dass $L = \{ ww : w \in \Sigma_{\text{bool}}^* \}$ nicht kontextfrei ist. Geben Sie eine kontextsensitive Grammatik für L an. (Sie können eine ähnliche Idee wie in Aufgabe 14.5 verwenden.)

Chomsky-Hierarchie

Wir haben nun die bereits erwähnte Chomsky-Hierarchie für Sprachen bzw. Grammatiken vervollständigt und fassen sie hier noch einmal zusammen.

Grammatik	alternative Bezeichnung	Bezeichnung der Sprache
Typ-0	Phrasenstrukturgrammatik	rekursiv aufzählbar
Typ-1		kontextsensitiv
Typ-2		kontextfrei
Typ-3		regulär

Wir wissen bereits, dass jede reguläre Sprache kontextfrei ist. Außerdem ist es offensichtlich, dass jede kontextsensitive Sprache rekursiv aufzählbar ist. Dass wir es wirklich mit einer Hierarchie zu tun haben, wird durch die folgende Aufgabe geklärt.

Aufgabe 14.7. Begründen Sie, dass jede kontextfreie Sprache kontextsensitiv ist.

Aufgabe 14.8. In der obigen Tabelle könnte man eine Klasse von Sprachen einfügen, die noch wesentlich einfacher als die regulären sind. Welche ist wohl gemeint?

Wie die regulären und die kontextfreien Sprachen kann man auch die rekursiv aufzählbaren durch einen Automatentyp charakterisieren.

Satz 14.2

Eine Sprache ist genau dann rekursiv aufzählbar, wenn sie von einer (deterministischen) Turingmaschine akzeptiert wird.

Beweis. Wir konstruieren zunächst eine Turingmaschine, die die von einer Phrasenstrukturgrammatik $G = (N, T, P, S)$ erzeugte Sprache akzeptiert. Dabei gehen wir so vor, dass wir die Ableitungssequenz, die ein Wort w aus dem Startsymbol S ableitet, gleichsam „rückwärts“ generieren. Wie nutzen aus, dass so eine Sequenz aus endlich vielen Schritten der Form $\alpha \Rightarrow_G \beta$ besteht, wobei im ersten Schritt $\alpha = S$ und im letzten $\beta = w$ gilt. Ganz wesentlich wird dabei sein, dass wir im Beweis zunächst eine *nichtdeterministische* Turingmaschine konstruieren – aus der wir anschließend nach Satz 14.1 eine deterministische machen können, die dieselbe Sprache akzeptiert.

Am Anfang steht ein Wort w auf dem Band. Die Maschine wählt nun ein Teilwort β von w aus und sucht nach einer Produktion der Form $\alpha \rightarrow \beta$. Wenn sie die findet, ersetzt sie auf dem Band β durch α (dabei sind ggf. Verschiebungen diverser Symbole vonnöten) und wiederholt diesen Vorgang. Das macht sie so lange, bis auf dem Band nur noch das Zeichen S steht. In diesem Fall akzeptiert sie. Findet sie zwischendurch kein β , das man ersetzen kann, bricht sie ab. Sowohl die Suche nach dem Teilwort (das evtl. mehrfach vorkommen könnte) als auch die Auswahl

der Produktion erfolgen nichtdeterministisch. Damit ist sichergestellt, dass die Maschine alle möglichen „Rückwärtsableitungen“ durchprobiert.

Jetzt wollen wir eine Grammatik konstruieren, die dieselbe Sprache wie eine vorgegebene deterministische Turingmaschine akzeptiert. Wir machen das exemplarisch für eine Turingmaschine mit dem Eingabealphabet Σ_{bool} und dem Bandalphabet $\Sigma_{\text{bool}} \cup \{\square, \vdash\}$. Außerdem verzichten wir auf die „Bewegung“ $\diamond$. (Siehe Aufgabe 12.19.) Wenn Sie den Beweis verstanden haben, wird Ihnen auch klar sein, wie Sie ihn auf andere Turingmaschinen übertragen können.

Die konkrete Maschine, die wir konvertieren wollen, soll n Zustände haben, die wir s_1 bis s_n nennen, wobei s_1 der Startzustand sein soll. Die Übergangsfunktion heißt wie üblich δ .

Die Terminalsymbole der Grammatik, die wir basteln, sind natürlich 0 und 1. Als Nichtterminalsymbole verwenden wir 3-Tupel aus $\Gamma \times \Gamma \times \{0, \dots, n\}$. Sequenzen aus solchen Tripeln werden im Laufe der Ableitung den Bandinhalt repräsentieren, pro „Kästchen“ ein Tripel. Dabei steht die erste Komponente für den ursprünglichen Inhalt beim Start der Maschine, die zweite für das aktuelle Zeichen. Die dritte Komponente wird in der Regel null sein. Hat sie den Wert $k \neq 0$, dann bedeutet das, dass der Schreib-Lese-Kopf gerade über dem jeweiligen Kästchen steht und die Maschine sich im Zustand s_k befindet. Die Produktionen werden dafür sorgen, dass nur Wörter abgeleitet werden können, in denen maximal ein Tripel mit von null verschiedener dritter Komponente vorkommt.

Gilt nun $\delta(s_k, \sigma) = (s_m, \sigma', \rightarrow)$, dann fügen wir für alle $\varphi, \chi, \tau \in \Gamma$ die Produktion

$$(\varphi, \sigma, k)(\chi, \tau, 0) \rightarrow (\varphi, \sigma', 0)(\chi, \tau, m')$$

hinzu.²³ Dabei gilt $m' = 0$, wenn s_m ein akzeptierender Zustand ist, und anderenfalls $m' = m$. Analog wird es Produktionen der Form

$$(\varphi, \tau, 0)(\chi, \sigma, k) \rightarrow (\varphi, \tau, m')(\chi, \sigma', 0)$$

geben, wenn $\delta(s_k, \sigma) = (s_m, \sigma', \leftarrow)$ gilt.

Wir ergänzen noch drei weitere Nichtterminalsymbole S (das Startsymbol), S_2 und S_3 zusammen mit den folgenden Produktionen, die die von der Grammatik simulierte Turingmaschine in den Anfangszustand versetzen:

$$\begin{aligned} S &\rightarrow (\vdash, \vdash, 1)S_2 & S_2 &\rightarrow (0, 0, 0)S_2 \mid (1, 1, 0)S_2 \mid S_3 \\ S_3 &\rightarrow (\square, \square, 0)S_3 \mid (\square, \square, 0) \end{aligned}$$

Die Idee dabei ist, dass S_2 verwendet wird, um ein von der Maschine akzeptiertes Wort auf das Band zu schreiben. Mit S_3 fügt man dann so viele Leerzeichen hinzu, wie die Maschine bis zum Terminieren benötigt. Die Produktionen weiter oben beschreiben nun den Programmablauf bis zur Akzeptanz. Nur dadurch kann ein

²³Für den „illegalen“ Fall $\sigma \neq \sigma'$ mit $\vdash \in \{\sigma, \sigma'\}$ fügen wir allerdings keine Produktion hinzu.

Wort abgeleitet werden, in dem alle Tripel null als letzte Komponente haben. Ist man so weit gekommen, dann wird durch die Produktionen

$$\begin{array}{ll} (0, \sigma, 0) \rightarrow 0 & (1, \sigma, 0) \rightarrow 1 \\ (\square, \sigma, 0) \rightarrow \varepsilon & (\vdash, \sigma, 0) \rightarrow \varepsilon \end{array}$$

der ursprüngliche Bandinhalt, der die ganze Zeit konserviert wurde, in Form von Terminalsymbolen wiederhergestellt und die irrelevanten Bandzeichen werden entfernt. ■

Aufgabe 14.9. Wie viele Nichtterminalsymbole hat die in diesem Beweis konstruierte Grammatik?

Was noch fehlt, sind die kontextsensitiven Sprachen. Die spielen zwar in der Theoretischen Informatik keine so große Rolle,²⁴ aber wir sind ihrer Charakterisierung durch abstrakte Automaten so nahe, dass wir sie zumindest kurz skizzieren wollen.

„Unsere“ Turingmaschinen sind durch die Grenzmarkierung $\vdash$ *einseitig beschränkt*. (Siehe auch Aufgabe 12.17.) Man kann noch stärker reglementieren, indem man ein weiteres Zeichen $\dashv$ für das *Bandende* einführt, für das ähnlich strikte Regeln wie für den Bandanfang gelten, nur eben für Bewegungen nach rechts. Setzt man zudem fest, dass bei der Eingabe w der Bandinhalt $\vdash w \dashv$ ist, dann kann die Maschine nur die $|w|$ Kästchen des Bandes benutzen, die w ursprünglich einnahm. Solche Turingmaschinen nennt man *linear beschränkt*.²⁵ Obwohl das nach einer sehr drastischen Restriktion aussieht,²⁶ können solche Maschinen doch noch recht viel.

Satz 14.3

Eine Sprache ist genau dann kontextsensitiv, wenn sie von einer *nichtdeterministischen* linear beschränkten Turingmaschine akzeptiert wird.

Beweis. Wir orientieren uns am Beweis von Satz 14.2. Dessen ersten Teil können wir wortwörtlich übernehmen. Aufgrund der speziellen Eigenarten kontextsensitiver Grammatiken wird der Bandinhalt niemals länger werden.²⁷ Daher kann man die Aufgabe auch mit einer linear beschränkten Turingmaschine erledigen.

Wenden wir uns dem zweiten Teil zu. Die Grammatik, die dort konstruiert wurde, ist *fast* kontextsensitiv. Lediglich das Entfernen der überflüssigen Zeichen am Ende ist problematisch. Das kann man jedoch beheben, indem man von 3-Tupeln zu 4-Tupeln wechselt. Die vierte Komponente enthält die Information darüber, ob es sich um das Symbol direkt hinter dem Bandanfang bzw. das direkt vor dem Bandende handelt. Alle Produktionen müssen natürlich entsprechend angepasst

²⁴Im Standardwerk von Hopcroft et al., das in der Literaturliste aufgeführt ist, wurde das Kapitel über kontextsensitive Sprachen beispielsweise ab der zweiten Auflage entfernt.

²⁵Im Englischen *linear bounded automaton*, abgekürzt LBA.

²⁶Andererseits ist das in gewissem Sinne sogar „realistischer“ als eine Standard-Turingmaschine, weil echte Computer auch nur über begrenzten Speicherplatz verfügen.

²⁷Das gilt zwar nicht für die ε -Sonderregel, aber die kann die Maschine als Spezialfall zuerst abhandeln.

werden, aber das ist nur Fleißarbeit. Entscheidend ist, dass man am Anfang weder die Grenzmarkierung noch Leerzeichen auf das simulierte Band schreiben und sie daher am Ende auch nicht wieder entfernen muss. ■

Wichtig ist allerdings der in der Formulierung des Satzes hervorgehobene Nicht-determinismus. Kann man den Wortbestandteil „nicht“ streichen? Akzeptieren deterministische linear beschränkte Turingmaschinen dieselben Sprachen wie die nichtdeterministischen? (Den Beweis von Satz 14.1 kann man sicher nicht auf linear beschränkte Turingmaschinen übertragen, denn die Maschine, die dort zur Simulation herangezogen wurde, wird evtl. viel mehr Band benötigen als die simulierte.) Tatsächlich weiß aktuell niemand die Antwort auf diese Frage. Es handelt sich um ein ungelöstes Problem, das unter dem Namen *erstes LBA-Problem* bekannt ist.

erstes LBA-Problem

Wir können die Tabelle von vorhin nun noch durch die passenden Automaten ergänzen:

Grammatik	Bezeichnung der Sprache	Automatentyp
Typ-0	rekursiv aufzählbar	Turingmaschine
Typ-1	kontextsensitiv	LBA
Typ-2	kontextfrei	PDA
Typ-3	regulär	DFA oder NFA

Dabei ist zu beachten, dass es für die Typen 0 und 3 irrelevant ist, ob die Automaten deterministisch oder nichtdeterministisch sind, während die Kellerautomaten für Typ-2 nichtdeterministisch sein müssen. Bei Typ-1 haben wir die Äquivalenz für die nichtdeterministische Variante begründet, während es wie erwähnt ungeklärt ist, ob man auch mit deterministischen LBAs auskommen könnte.

Bei den regulären Sprachen haben wir als dritte Charakterisierung neben Grammatiken und Automaten noch die *regulären Ausdrücke*. Für die anderen Sprachklassen haben wir kein Pendant dafür.

★Aufgabe 14.10. Warum spricht man von *linearer* Beschränkung? Weil sich nichts Wesentliches ändern würde, wenn man der Maschine $k|w|$ statt $|w|$ Zeichen bei der Eingabe w zugestehen würde, wobei $k \in \mathbb{N}^+$ eine von der Eingabe unabhängige Konstante sein soll. Sehen Sie, warum das so ist?

Aufgabe 14.11. Geben Sie zwei kontextsensitive Grammatiken G_1 und G_2 über demselben Alphabet an, für die man mit der Methode aus dem Beweis von Satz 10.3 keine Grammatik für $L(G_1) \circ L(G_2)$ erhält. (Hinweis: Verwenden Sie Grammatiken mit $\varepsilon \notin L(G_1) \cup L(G_2)$. Man kommt mit insgesamt drei Produktionen aus.)

Aufgabe 14.12. Geben Sie eine kontextsensitive Grammatik $G_1 = (N_1, \Sigma, P_1, S_1)$ an, die nicht das leere Wort erzeugt und für die die Grammatik²⁸

$$G = (N_1 \cup \{S\}, \Sigma, P_1 \cup \{S \rightarrow S_1 S, S \rightarrow S_1\}, S)$$

²⁸Siehe die Lösung von Aufgabe 10.9.

mit $S \notin N_1$ nicht $L(G_1)^+$ erzeugt. (Hinweis: Man kommt mit zwei Produktionen aus. Auch hier sollte $\varepsilon \notin L(G_1)$ gelten.)

Aufgabe 14.13. In den Lösungen der Aufgaben 14.11 und 14.12 wurden der Einfachheit halber Grammatiken gezeigt, in denen bestimmte Produktionen gar nicht verwendet werden konnten. Erweitern Sie diese Lösungen so, dass alle Produktionen wirklich genutzt werden, es aber trotzdem Gegenbeispiele im Sinne der ursprünglichen Aufgabenstellung bleiben.

Aufgabe 14.14. Geben Sie ein Verfahren an, mit dem man jede kontextsensitive Grammatik G in eine kontextsensitive Grammatik G' umwandeln kann, so dass $L(G) = L(G')$ gilt und auf den linken Seiten der G' -Produktionen nur Nichtterminalsymbole stehen.

Lemma 14.4

Die Menge der kontextsensitiven Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen Vereinigung und Konkatenation.

Beweis. Wenn man die beiden Grammatiken zunächst wie in Aufgabe 14.14 umwandelt, kann man den Beweis von Satz 10.3 kopieren und sich überzeugen, dass so etwas wie in Aufgabe 14.11 nicht passieren kann. ■

Lemma 14.5

Die Menge der kontextsensitiven Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen die kleenesche Hülle.

Beweis. Sei $G_1 = (N_1, \Sigma, P_1, S_1)$ eine kontextsensitive Grammatik. O. B. d. A. sei ε kein Element von $L(G_1)$ (sonst muss man die Produktion $S_1 \rightarrow \varepsilon$ entfernen) und auf den linken Seiten der Produktionen seien keine Nichtterminalsymbole. (Siehe Aufgabe 14.14.) G'_1 sei die Grammatik $(N'_1, \Sigma, P'_1, S'_1)$, die aus G_1 entsteht, indem man jedes Symbol $A \in N_1$ durch ein neues Symbol A' ersetzt und in allen Produktionen von P_1 auch diese Substitutionen durchführt. G'_1 erzeugt natürlich dieselbe Sprache wie G_1 .

Man erhält dann eine Grammatik G mit $L(G) = L(G_1)^*$ durch

$$G = (N_1 \cup N'_1 \cup \{S, T_1, T_2\}, \Sigma, P_1 \cup P'_1 \cup P, S)$$

mit neuen Symbolen S und T , wobei P die sechs Produktionen

$$S \rightarrow T_1 \mid \varepsilon \quad T_1 \rightarrow S_1 T_2 \mid S_1 \quad T_2 \rightarrow S'_1 T_1 \mid S'_1$$

enthält. Durch das „Abwechseln“ von G_1 und G'_1 wird dafür gesorgt, dass so etwas wie in Aufgabe 14.12 nicht passieren kann. ■

Lemma 14.6

Die Menge der kontextsensitiven Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen die Bildung von Durchschnitten.

Beweis. Seien L_1 und L_2 zwei kontextsensitive Sprachen. Wir wählen dazu nach Satz 14.3 linear beschränkte Turingmaschinen T_1 und T_2 mit $L_i = L(T_i)$. Dann

konstruieren wir eine linear beschränkte Turingmaschine T mit zwei **Spuren**,²⁹ die zunächst ihr Eingabewort von der ersten auf die zweite Spur kopiert und dann auf der ersten Spur T_1 simuliert. Gerät diese Simulation in einen akzeptierenden Zustand, wechselt die Maschine in die zweite Spur, simuliert dort T_2 und akzeptiert, wenn die Simulation akzeptiert. Offenbar akzeptiert T die Sprache $L_1 \cap L_2$. ■

Satz von Immerman und Szelepcsényi

Die Menge der kontextsensitiven Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen die Bildung von Komplementen.

Satz 14.7

Für diesen Satz erhielten seine beiden Entdecker 1995 den **Gödel-Preis**.³⁰ Der Beweis ist zu aufwendig für dieses Skript, aber wenn Sie den obigen Link anklicken, werden Sie zur Wikipedia-Seite des Satzes geführt, wo er zumindest skizziert wird.

²⁹Oder entsprechend größerem Bandalphabet, siehe Aufgabe 14.10.

³⁰Der Slowake Róbert Szelepcsényi war ein 20 Jahre alter Student, als er – unabhängig von dem Amerikaner Neil Immerman, der zeitgleich dasselbe schaffte – den Beweis fand.



Berechenbarkeitstheorie

In der Berechenbarkeitstheorie, die man auch *Rekursionstheorie* nennt, geht es um die grundsätzliche Frage, was man (mit Computern) überhaupt berechnen kann. Wir werden weiterhin regen Gebrauch von der [Church-Turing-These](#) machen, indem wir zwar die ganze Zeit über Turingmaschinen sprechen, meistens aber deren Arbeitsweise nur grob skizzieren, wenn wir uns vorstellen können, die entsprechende Aufgabe auch mit einer „normalen“ Programmiersprache erledigen zu können.



Eine wesentliche Rolle werden in diesem Kapitel die Begriffe *entscheidbar* (bzw. *rekursiv*) und *berechenbar* spielen. Vielleicht schauen Sie sich vorsichtshalber noch einmal deren Definition auf Seite [78](#) an.

15. Berechenbarkeit und Entscheidbarkeit

Wir werden uns im Folgenden auf (partielle) Funktionen beschränken, die mit natürlichen Zahlen (statt mit beliebigen Wörtern) als Ein- und Ausgabe arbeiten. Das ist keine relevante Einschränkung, da wir ja mithilfe der [Gödelisierung](#) zwischen Zahlen und beliebigen Wörtern hin und her springen können.

Eine Turingmaschine T mit dem Eingabealphabet Σ_{bool} **berechnet** die partielle Funktion $g: \mathbb{N} \rightarrow \mathbb{N}$, wenn¹

$$f_T(w_1) = \begin{cases} w_2 & g(n_{w_1}) = n_{w_2} \\ \perp & g(n_{w_1}) = \perp \end{cases}$$

für alle $w_1, w_2 \in \Sigma_{\text{bool}}^+$ und $f_T(\varepsilon) = \perp$ gilt. Wir verwenden dann auch T als Synonym für g , d. h., wir schreiben $T(n_{w_1}) = n_{w_2}$ bzw. $T(n_{w_1}) = \perp$. Wenn es eine solche Maschine gibt, dann nennen wir g **berechenbar**.

Definition

¹Zur Schreibweise n_w siehe Seite [10](#). Wegen möglicher führender Nullen ist die Definition eigentlich nicht eindeutig. Man kann das aber eindeutig machen, indem man z. B. festlegt, dass w_2 die kürzestmögliche Antwort sein soll.

Definition

Die **charakteristische Funktion** oder **Indikatorfunktion** einer Menge $A \subseteq \mathbb{N}$ ist die durch

$$\chi_A(n) = \begin{cases} 1 & n \in A \\ 0 & n \notin A \end{cases}$$

definierte Funktion χ_A von $\mathbb{N}$ nach $\{0, 1\}$. A heißt **entscheidbar** oder **rekursiv**, wenn χ_A von einer Turingmaschine berechnet werden kann.

Analoge Begriffe und Schreibweisen definieren wir für partielle Funktionen von $\mathbb{N}^m$ nach $\mathbb{N}^n$ bzw. Teilmengen von $\mathbb{N}^m$, indem wir die Tupel wie in Aufgabe 13.13 gödelisieren. Für eine Maschine T , die aus der Eingabe $2^a 3^b$ die Ausgabe $a + b$ macht, würden wir also $T(a, b) = a + b$ schreiben.

Offensichtliche Beispiele für berechenbare Funktionen sind $n \mapsto n + 1$, $n \mapsto 2^n$ oder $(m, n) \mapsto \text{ggT}(m, n)$. Beispiele für entscheidbare Mengen sind $\{n \in \mathbb{N} : n \text{ ist gerade}\}$, $\mathbb{P}$ oder $\{(m, n) \in \mathbb{N}^2 : m \leq n\}$.

Aufgabe 15.1. Geben Sie noch weitere Beispiele an.

Aufgabe 15.2. Begründen Sie, warum $A \subseteq \mathbb{N}$ genau dann entscheidbar ist, wenn $\mathbb{N} \setminus A$ entscheidbar ist.

Definition

Eine Menge $A \subseteq \mathbb{N}$ heißt **semi-entscheidbar**, wenn es eine Turingmaschine T mit $L(T) = \{w \in \Sigma_{\text{bool}}^+ : n_w \in A\}$ gibt. Man sagt dann auch, dass T die Menge A **akzeptiert**.

Es sollte klar sein, wieso man von *halber* Entscheidbarkeit spricht, denn anders als bei Entscheidbarkeit wird die Maschine zwar im positiven Fall in einem Endzustand terminieren, aber ihr Verhalten im negativen Fall ist unbekannt: Sie könnte in eine **Endlosschleife** geraten.

Aufgabe 15.3. Warum ist jede entscheidbare Menge semi-entscheidbar?

Aufgabe 15.4. Warum gibt es zu jeder semi-entscheidbaren Menge $A \subseteq \mathbb{N}$ eine Turingmaschine, die nur bei der Eingabe von Zahlen aus A anhält?

Wir definieren nun einen weiteren Begriff, der sich als zum letzten äquivalent herausstellen wird.² Das geschieht nicht, um Sie zu verwirren, sondern um zwei unterschiedliche Herangehensweisen an dasselbe Konzept zu verdeutlichen und die Herkunft der Ausdrücke zu klären.

Definition

Eine Menge $A \subseteq \mathbb{N}$ heißt **rekursiv aufzählbar**, wenn es eine berechenbare Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$ gibt, deren Wertebereich A ist.³

²Das kann man wegen Satz 14.2 bereits ahnen.

Betrachtet man f als **Folge**, so kommen alle Elemente von A (und nur die) irgendwann als Folgenglieder vor. Das erklärt den Begriff der *Aufzählung*. Da f berechenbar ist, können wir uns, wenn unsere Programmiersprache so etwas wie einen `print`-Befehl hat, ein Programm vorstellen, das die Elemente von A nach und nach ausgibt. Wenn wir lange genug warten, werden wir jedes Element (mindestens) einmal sehen.⁴

Eine Menge $A \subseteq \mathbb{N}$ ist genau dann semi-entscheidbar, wenn sie rekursiv aufzählbar ist.

Satz 15.1

Beweis. Wir begründen zunächst, warum jede rekursiv aufzählbare Menge A semi-entscheidbar ist. Dafür sei T eine Turingmaschine, die eine Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$ berechnet, deren Wertebereich A ist. Wir konstruieren nun einfach auf der Basis von T eine Turingmaschine T' , die als Eingabe eine Zahl n bekommt und die dann sukzessive $f(0)$, $f(1)$ und so weiter berechnet. Wenn einer der berechneten Werte die Zahl n ist, dann hält T' an und gibt *ja* aus, ansonsten läuft die Maschine weiter – evtl. für immer. Das war's schon.

Etwas schwieriger ist es, auf der Basis einer Turingmaschine T , die A akzeptiert, eine Maschine T' zu konstruieren, die A aufzählen kann. Dazu erstellen wir eine Tabelle aller Paare (i, j) von natürlichen Zahlen und durchlaufen diese wie in **Cantors erstem Diagonalargument**.⁵ T' erhält nun die Eingabe n . Wir setzen einen Zähler c auf 0 und simulieren für jedes Paar (i, j) die ersten j Schritte der Turingmaschine T mit der Eingabe i . Wenn die Simulation bis dahin eine positive Antwort gegeben hat, terminiert T' mit der Ausgabe *i*, wenn $c = n$ gilt. Anderenfalls wird c um eins erhöht und das nächste Paar kommt dran. Ist die Simulation nach j Schritten nicht fertig, wird sie abgebrochen und es kommt ebenfalls das nächste Paar dran. Da jeder Simulationslauf nach endlich vielen Schritten beendet ist, sorgt das Cantorsche Verfahren dafür, dass jedes Paar von natürlichen Zahlen irgendwann einmal drankommt. Ist i ein Element von A , so muss T bei der Eingabe von i mit *ja* antworten und das muss nach einer endlichen Anzahl j von Schritten geschehen. Wenn (i, j) von T' bearbeitet wird, wird also i ausgegeben werden. ■

Sind $A \subseteq \mathbb{N}$ und $\mathbb{N} \setminus A$ rekursiv aufzählbar, dann ist A entscheidbar.

Satz 15.2

Beweis. Wir besorgen uns Turingmaschinen, T_1 und T_2 die A bzw. $\mathbb{N} \setminus A$ aufzählen. Um zu entscheiden, ob eine Zahl n zu A gehört, rufen wir erst T_1 mit der Eingabe 0 auf, dann T_2 mit derselben Eingabe, dann nacheinander T_1 und T_2 mit der Eingabe 1 und immer so weiter. Irgendwann muss eine der beiden n auswerfen. War es T_1 , antworten wir mit *ja*, anderenfalls mit *nein*. ■

³Anders ausgedrückt ist A das **Bild** $f[\mathbb{N}]$ von $\mathbb{N}$ unter f bzw. f bildet die Menge der natürlichen Zahlen **surjektiv** auf A ab.

⁴Ausführliche Beispiele dazu in Python findet man in Kapitel 18 von **KMFL**.

⁵Auch dazu mehr in Kapitel 18 von **KMFL**.

Wir werden jetzt Turingmaschinen Zahlen zuordnen. Wie genau wir das machen, spielt keine Rolle, solange aus der Zahl die Turingmaschine wieder rekonstruierbar ist. Am Beispiel von Maschinen für den Simulator `tm2` mit dem Eingabealphabet Σ_{bool} soll jedoch gezeigt werden, wie man vorgehen könnte. Zunächst machen wir uns klar, dass wir nur einen endlichen Vorrat von Zeichen brauchen. Denen teilen wir Zahlen zu, etwa so:⁶

0	1	-	*	L	R	S	H		↓
0	1	2	3	4	5	6	7	8	9

Die Beschreibung einer `tm2`-Maschine ist nun nichts weiter als eine Folge dieser Zahlen, die wir als Dezimalzahl interpretieren können, weil wir mit zehn Zeichen auskommen. Leerzeichen können ignoriert werden; die „Sprungadressen“ im Programm werden binär codiert. Am Beispiel von `Code/inc.tm2` aus dem [Docker-Image](#) (in ausführlicher Form) sieht das so aus:

11R1		00R1		--L2		**H1		1151	8	0051	8	22410	8	3371	9
10L2		01H2		--S2		**R3		10410	8	01710	8	22610	8	33511	9
11S3		01R4		--S3		**S3		11611	8	015100	8	22611	8	33611	9
11S4		00R4		-OH4		**S4		116100	8	005100	8	207100	8	336100	

Dieser Turingmaschine entspricht also die Zahl 115180...8336100.

Aufgabe 15.5. Warum wurden die Instruktionen für das Symbol 1 zuerst aufgeführt und nicht die für das Symbol 0?

Aufgabe 15.6. Man könnte sogar mit weniger als zehn Zeichen auskommen. Haben Sie dazu Ideen?

Man hätte auch die zehn Zeichen durch jeweils vier Bits binär codieren oder das Verfahren aus Aufgabe 13.13 verwenden können. Entscheidend ist lediglich, dass wir, nachdem wir uns für eine Notation entschieden haben, eine [Gödelisierung](#) erhalten: Unterschiedliche Maschinen erhalten unterschiedliche Zahlen und die Übersetzungsprozesse zwischen Maschinen und Zahlen müssen als Programme realisierbar sein. Das ist offenbar der Fall. Die berechnete Zahl nennen wir dann die *Maschinenbeschreibung* der Turingmaschine. Dabei ist der [bestimmte Artikel](#) natürlich nur dann angebracht, wenn man sich auf ein Verfahren geeinigt hat, weil man je nach Codierung unterschiedliche Maschinenbeschreibungen erhält.

Maschinen-
beschreibung

Betrachtet man die Menge aller Turingmaschinen zum selben Eingabe- und Bandalphabet,⁷ so kann man deren Maschinenbeschreibungen aufsteigend sortieren. Das würde man die *Standardaufzählung* dieser Menge nennen.

Standardaufzählung

⁶Das Symbol ↓ soll dabei für einen [Zeilenumbruch](#) stehen.

⁷Für die Leser, die mit den Fallstricken der Mengenlehre vertraut sind, sei darauf hingewiesen, dass das ganz allgemein keine Menge, sondern eine [echte Klasse](#) sein wird. Man kann aber offenbar o. B. d. A. eine Konvention einführen, in der die Zustände einer Turingmaschine nur Elemente einer vorgegebenen abzählbaren Menge sein dürfen. Dann hat man es mit einer abzählbaren Menge zu tun. (Und ohne so eine Konvention kann man ohnehin keine eindeutige Maschinenbeschreibung festlegen.)

Aufgabe 15.7. Begründen Sie, warum es zu jeder Turingmaschine M_i aus der Standardaufzählung eine Turingmaschine M_j mit $j > i$ gibt, die dieselbe Funktion wie M_i berechnet.

16. Das Halteproblem und der Satz von Rice

Wir betrachten in diesem Abschnitt nur noch Turingmaschinen mit Eingabealphabet Σ_{bool} und Bandalphabet $\Sigma_{\text{bool}} \cup \{\square, \vdash\}$, was keine wesentliche Einschränkung ist.⁸ Für diese Maschinen gibt es eine Standardaufzählung und für die i -te Maschine in dieser Aufzählung schreiben wir durchgehend M_i .

M_i

Wir interpretieren alle diese Turingmaschinen als partielle Funktionen, die natürliche Zahlen auf natürliche Zahlen abbilden. Gleichzeitig können wir mittels der kanonischen Gödelisierung wie in Aufgabe 13.13 solche Funktionen bei Bedarf auch immer als Funktionen von $\mathbb{N}^m$ nach $\mathbb{N}^n$ interpretieren und machen das ab und zu auch. (In diesem Sinne kann ein und dieselbe Turingmaschine je nach Interpretation unterschiedliche Aufgaben erledigen.)

Weil es **nur abzählbar viele** Turingmaschinen gibt, ist klar, dass es nur abzählbar viele semi-entscheidbare Mengen gibt. Die „meisten“ Mengen von natürlichen Zahlen sind *nicht* semi-entscheidbar (und nach Aufgabe 15.3 „erst recht“ nicht entscheidbar). Aber kann man solche Mengen konkret angeben? Und gibt es „interessante“ Mengen, die nicht (semi-)entscheidbar sind?

Wir nennen eine Turingmaschine T^U mit $T^U(i, n) = M_i(n)$ für alle $(i, n) \in \mathbb{N}^2$ eine **universelle Turingmaschine**.

Definition

Wie ist das gemeint? Wenn T^U die Eingabe (i, n) bekommt, dann gibt sie das aus, was die i -te Turingmaschine M_i bei der Eingabe n ausgeben würde. Eine *universelle* Turingmaschine kann also alle Turingmaschinen simulieren.

Es gibt eine universelle Turingmaschine.

Satz 16.1

Beweis. Als Turing 1936 diese Idee hatte, war deren Beschreibung, die die Leser überzeugen sollte, sehr aufwendig.⁹ Mit unserem heutigen Wissen über Computer und speziell nach der Lektüre von Abschnitt 13 entlockt uns das aber nur noch ein Achselzucken. Das **Docker-Image** enthält ja einen (in C geschriebenen) Simulator und wir haben uns überzeugt, dass wir aus so einem Programm im Prinzip eine Turingmaschine machen könnte. Das wäre im Wesentlichen eine universelle Turingmaschine.

Wir wollen aber trotzdem kurz eine universelle **Zweiband-Turingmaschine** skizzieren (die man dann noch in eine Maschine mit nur einem Band übersetzen müsste).

⁸Wir wissen ja aus **dem letzten Kapitel**, dass wir notfalls sogar auf eines der beiden Zeichen aus Σ_{bool} verzichten könnten.

⁹Das wird übrigens sehr gut in dem empfehlenswerten Buch *The Annotated Turing* erklärt.

Auf das erste Band wird die Eingabe (i, n) geschrieben. Im ersten Schritt berechnet die Maschine eine Beschreibung der Turingmaschine M_i und schreibt sie auf das zweite Band. Das ist erstens ein extrem rechenintensiver Prozess und zweitens müssen wir uns auch noch überlegen, wie wir die Beschreibung auf das Band schreiben, aber dass das grundsätzlich möglich ist, sollte wohl klar sein. Nachdem das getan ist, wird der Inhalt des ersten Bandes durch n ersetzt (i wird quasi „gelöscht“) und dann wird mithilfe der Informationen auf dem zweiten Band auf dem ersten Band der Ablauf von M_i bei der Eingabe n simuliert. In der Praxis wäre das ein Albtraum, aber in der Theorie ist es nicht schwer. ■

Definition

Die Menge $H = \{(i, n) \in \mathbb{N}^2 : M_i \text{ hält bei der Eingabe } n \text{ an}\}$ wird als **Halteproblem** bezeichnet.¹⁰

Die entscheidende Aussage aus Turings Arbeit von 1936 ist die folgende:¹¹

Satz 16.2

Das Halteproblem H ist semi-entscheidbar, aber nicht entscheidbar.

Beweis. Dass H semi-entscheidbar ist, ist klar: Man nehme sich eine universelle Turingmaschine T^U und konstruiere eine Turingmaschine T' , die bei der Eingabe (i, n) den Wert $T^U(i, n)$ berechnet. T' akzeptiert offenbar H .

Wir betrachten zunächst

$$H^* = \{n \in \mathbb{N} : M_n \text{ hält bei der Eingabe } n \text{ an}\} \quad (16.1)$$

und zeigen, dass diese Menge nicht entscheidbar ist. Wäre das nämlich der Fall, dann gäbe es eine Turingmaschine T , die $\mathbb{N} \setminus H^*$ akzeptiert und für Eingaben aus H^* nicht terminiert. (Wir nehmen eine Maschine, die H^* entscheidet, geben bei der Ausgabe 0 eine Eins aus und gehen bei der Ausgabe 1 in eine Endlosschleife. Siehe auch die Aufgaben 15.3 und 15.4.) Es gilt also für alle $n \in \mathbb{N}$, dass T bei der Eingabe n genau dann anhält, wenn M_n bei der Eingabe n *nicht* anhält. Aber T kommt in der Standardaufzählung als $T = M_k$ vor und nach unserer Überlegung eben hält T bei der Eingabe k genau dann an, wenn M_k bei der Eingabe k nicht anhält. Das ist ein offensichtlicher Widerspruch.

Dass H nicht entscheidbar ist, ist nun eine simple Folgerung. Gäbe es eine Turingmaschine T_H , die H entscheidet, dann könnten wir eine Maschine T' bauen, die $T'(n) = T_H(n, n)$ berechnet und damit H^* entscheidet. ■

Problem

¹⁰Allgemein bezeichnet man in der Theoretischen Informatik eine Eigenschaft P , die natürliche Zahlen haben oder nicht haben können, bzw. die zugehörige Menge $\{n \in \mathbb{N} : P(n)\}$ als *Problem*. Eine *Lösung* des Problems ist eine Turingmaschine, die die Menge entscheidet. (Siehe dazu auch die formale Definition auf Seite 105.) Für Tupel von Zahlen wie in der Definition von H greifen wir auf die *kanonische Gödelisierung* derselben zurück.

¹¹Die Bezeichnung *Halteproblem* sowie die heutige Beschreibung sind allerdings neueren Datums.

Die Menge $\{n \in \mathbb{N} : M_n \text{ hält bei jeder Eingabe an}\}$ wird als **gleichmäßiges Halteproblem** bezeichnet.

Definition

Das gleichmäßige Halteproblem ist nicht entscheidbar.

Satz 16.3

Beweis. Wir schreiben eine Turingmaschine T , die aus einer Turingmaschine eine andere Turingmaschine macht – wobei sie rein technisch natürlich nur mit den zugehörigen Indizes der Standardaufzählung arbeitet. Im Detail erhält T eine Zahl i als Eingabe und konstruiert nun eine Turingmaschine M'_i , die $M'_i(n) = M_i(i)$ für alle $n \in \mathbb{N}$ berechnet. Diese Konstruktion ist gar nicht so schwer: Die Eingabe n wird vom Band entfernt und durch (i, i) ersetzt, dann übernimmt eine universelle Turingmaschine. T berechnet anschließend die Maschinenbeschreibung von M'_i und daraus die Position von M'_i in der Standardaufzählung, die zurückgegeben wird. Der „Witz“ an T ist, dass $M_{T(i)}$ genau dann bei jeder Eingabe anhält, wenn M_i bei der Eingabe i anhält. Gäbe es nun eine Turingmaschine G , die das gleichmäßige Halteproblem entscheidet, dann würde die „zusammengesetzte“ Maschine, die $G(T(i))$ für alle $i \in \mathbb{N}$ berechnet, das Problem H^* aus (16.1) entscheiden. ■

Die obigen Aussagen über die Nichtentscheidbarkeit einzelner Probleme lassen sich dramatisch verallgemeinern.

Satz von Rice

Sei $\mathcal{P}$ die Menge aller partiellen berechenbaren Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ und $\mathcal{A}$ eine nichtleere echte Teilmenge von $\mathcal{P}$. Dann ist

$$A = \{i \in \mathbb{N} : M_i \text{ berechnet eine Funktion aus } \mathcal{A}\}$$

nicht entscheidbar.

Satz 16.4

Beweis. Wir können o.B.d.A. annehmen, dass die Funktion $\emptyset$ nicht zu $\mathcal{A}$ gehört.¹² (Sonst betrachten wir $\mathbb{N} \setminus A$, siehe Aufgabe 15.2.) Da $\mathcal{A}$ nicht leer ist, gibt es eine Zahl $a \in A$.

Analog zum Beweis von Satz 16.3 konstruieren wir eine Turingmaschine T mit der Eigenschaft, dass $M_{T(i)}$ dasselbe wie M_a macht, wenn M_i bei der Eingabe i anhält, und anderenfalls nie anhält: T konstruiert (bei Eingabe i) die Turingmaschine M'_i dadurch, dass zuerst eine universelle Turingmaschine mit der Eingabe (i, i) gestartet wird. Hält diese an, wird das Band geleert und M_a mit der eigentlichen Eingabe gestartet. Hält sie nicht an, dann hält logischerweise auch M'_i nicht an. Wie eben gibt T die Position von M'_i in der Standardaufzählung zurück.

Gäbe es nun eine Turingmaschine R , die A entscheidet, so könnten wir wieder H^* aus (16.1) entscheiden, indem wir R mit dem Ergebnis von T „füttern“: Gilt $i \in H^*$, so hält M_i bei der Eingabe i an und $M_{T(i)}$ „simuliert“ M_a , d. h. es gilt $T(i) \in A$.

¹² $\emptyset$ ist die Funktion, die immer den Funktionswert $\perp$ hat.

Gilt hingegen $i \notin H^*$, so hält $M_{T(i)}$ nie an und „berechnet“ damit $\emptyset$, d. h., es gilt $T(i) \notin A$. ■

Eine Konsequenz des Satzes von Rice¹³ ist, dass sich Fragen wie die folgenden niemals „vollautomatisiert“ klären lassen:

- Berechnet das Programm M eine bestimmte Funktion $f: \mathbb{N} \rightarrow \mathbb{N}$, d. h., tut es, was es tun soll?
- Berechnet das Programm M dasselbe wie ein vorgegebenes Programm N ?
- Berechnet das Programm M für die Eingabe 42 denselben Funktionswert wie für die Eingabe 43?

Und wegen der [Church-Turing-These](#) gelten sie nicht nur für Turingmaschinen. Ebenso wenig kann man beispielsweise ein Java-Programm schreiben, dass solche Fragen beantwortet, indem es den Quellcode von Java-Programmen analysiert.

Das ist allerdings immer so gemeint, dass ein „Prüfprogramm“ erstellt werden soll, das für *jede* Eingabe M die korrekte Antwort gibt. Natürlich lassen sich solche Fragen für einzelne Programme oft beantworten, aber darum geht es nicht.

Im Video [Die seltsamste Zahl](#) gehe ich noch ausführlicher auf die Konsequenzen der Unlösbarkeit des Halteproblems ein und begründe unter anderem, dass diese auch Auswirkungen auf wichtige mathematische Probleme wie die [Goldbachsche Vermutung](#) hat. Eine weitere Folgerung wird in dem Video [Computer sind zu dumm für Tetris](#) thematisiert.

Aufgabe 16.1. Warum gilt der Satz von Rice nicht für die Fälle $\mathcal{A} = \emptyset$ und $\mathcal{A} = \mathcal{P}$?

In den obigen Beweisen haben wir mehrfach auf eine bestimmte Art ausgenutzt, dass die Menge H^* aus (16.1) nicht entscheidbar ist. Daraus kann man ein allgemeines Prinzip machen.

Definition

Seien A und B Mengen von natürlichen Zahlen. Gibt es eine berechenbare Funktion g von $\mathbb{N}$ nach $\mathbb{N}$ mit $A = g^{-1}[B] = \{n \in \mathbb{N} : g(n) \in B\}$, dann sagt man, dass (das Problem) A auf B **reduzierbar** ist.

Intuitiv wird ein Problem durch Reduktion höchstens „schwieriger“:

Lemma 16.5

Seien $A, B \subseteq \mathbb{N}$ und sei A auf B reduzierbar. Ist B entscheidbar, dann auch A .

Beweis. Ist B entscheidbar, dann gibt es eine Turingmaschine T_B , die B entscheidet. Außerdem gibt es nach Definition eine Turingmaschine T_g , die eine Funktion $g: \mathbb{N} \rightarrow \mathbb{N}$ mit $A = g^{-1}[B]$ berechnet. Sei nun T die Turingmaschine, die durch „Hintereinanderschalten“ von T_g und T_B entsteht. Ist n ein Element von A , so ist $g(n)$ ein Element von B und T antwortet mit *ja*. Ist n kein Element von A , dann ist $g(n)$ kein Element von B und T antwortet mit *nein*. ■

¹³Der amerikanische Mathematiker [Henry Gordon Rice](#) bewies ihn 1953.

Aufgabe 16.2. Welches Problem wurde im Beweis des Satzes von Rice auf welches reduziert? Und wie war die entsprechende Rollenverteilung beim gleichmäßigen Halteproblem?

Aufgabe 16.3. A_0 bis A_2 seien Mengen von natürlichen Zahlen. A_0 sei endlich, A_1 entscheidbar und A_2 rekursiv aufzählbar. Welche der folgenden Aussagen sind mit Sicherheit wahr? Geben Sie bei den wahren Aussagen eine Begründung an und bei den anderen ein Gegenbeispiel.

- (i) $\mathbb{N} \setminus A_2$ ist semi-entscheidbar.
- (ii) $A_0 \cup A_2$ ist entscheidbar.
- (iii) $\mathbb{N} \setminus A_1$ ist semi-entscheidbar.
- (iv) A_0 ist entscheidbar.
- (v) $\mathbb{N} \setminus A_0$ ist entscheidbar.
- (vi) $A_0 \cap A_2$ ist entscheidbar.
- (vii) A_1 ist semi-entscheidbar.

17. Sprachen und Entscheidbarkeit

Es fehlen noch ein paar Angaben zu den Eigenschaften kontextsensitiver und rekursiv aufzählbarer Sprachen, die wir jetzt nachholen. Obwohl es nun wieder um ganz allgemeine Sprachen geht, kann man ohne Probleme die entsprechenden Argumente über Zahlenmengen aus dem letzten Abschnitt übertragen.



Aufgabe 17.1. Wie genau ist das mit dem Übertragen von Argumenten gemeint?

Alle kontextsensitiven Sprachen sind entscheidbar.

Satz 17.1

Beweis. Ist eine Sprache L kontextsensitiv, dann gibt es eine kontextsensitive Grammatik $G = (N, T, P, S)$, die sie erzeugt. Gehört ein Wort $w \neq \varepsilon$ zu L , so können in der Ableitung von w aus S nur Wörter mit einer Länge von maximal $|w|$ Zeichen vorgekommen sein, weil keine Produktion – außer evtl. $S \rightarrow \varepsilon$ – Zeichenketten verkürzt. Will man nun testen, ob ein Wort w zu L gehört, so kann man einen gerichteten **Graphen** konstruieren, dessen Knotenmenge aus allen Wörtern über $N \cup T$ besteht, deren Länge höchstens $|w|$ ist, und bei dem eine Kante vom Knoten α zum Knoten β verläuft, wenn $\alpha \Rightarrow_G \beta$ gilt. Gilt $w \in L$, so gibt es in dem Graphen einen **Weg** von S nach w und aus den Informatikvorlesungen wissen Sie, dass es Algorithmen wie z. B. **den von Dijkstra** gibt, die in endlicher Zeit feststellen können, ob es so einen Weg gibt. (Siehe auch Seite 114.) ■

Damit wissen wir natürlich insbesondere, dass auch alle kontextfreien und alle regulären Sprachen entscheidbar sind. (Man spricht in diesem Zusammenhang übrigens auch vom **Wortproblem**.)

Wortproblem

Außerdem haben wir eine Lücke aus dem letzten Kapitel geschlossen, denn bisher kannten wir keine Sprache, die rekursiv aufzählbar, aber nicht kontextsensitiv ist.

Durch das Halteproblem kennen wir nun so eine Sprache. Wie die nächsten beiden Resultate zeigen, gibt es sogar *entscheidbare* Sprachen, die nicht kontextsensitiv sind.

Lemma 17.2

Sei $(N_i)_{i \in \mathbb{N}}$ eine Folge von Turingmaschinen mit dem Eingabealphabet Σ_{bool} , die bei jeder Eingabe anhalten. Außerdem soll die zugehörige Folge der Maschinenbeschreibungen berechenbar sein. Dann gibt es eine entscheidbare Sprache über Σ_{bool} , die von keiner dieser Maschinen akzeptiert wird.

Beweis. Wir definieren $L = \{ w \in \Sigma_{\text{bool}}^+ : w \notin L(N_{n(w)}) \}$, wobei wir wieder die Schreibweise n_w von Seite 10 verwenden. Dass L entscheidbar ist, ist offensichtlich, denn um zu prüfen, ob ein Wort w zu L gehört, muss man nur $N_{n(w)}$ mit dieser Eingabe laufen lassen. Ist jedoch N_i eine der Maschinen aus der Folge und $w \in \Sigma_{\text{bool}}^+$ ein Wort mit $n_w = i$, dann gilt $w \in L$ nach Definition genau dann, wenn $w \notin L(N_i)$ gilt, d. h., $L \neq L(N_i)$. ■

Satz 17.3

Es gibt entscheidbare Sprachen, die nicht kontextsensitiv sind.

Beweis. Nach Satz 3.1 gibt es nur abzählbar viele kontextsensitive Sprachen über Σ_{bool} , die wir deshalb in der Form $(L_i)_{i \in \mathbb{N}}$ aufzählen können. Nach Satz 17.1 gibt es zu jedem $i \in \mathbb{N}$ eine Turingmaschine N_i , die immer anhält und für die $L(N_i) = L_i$ gilt.¹⁴ Man überlege sich, dass die Folge der zugehörigen Maschinenbeschreibungen berechenbar ist. Nach Lemma 17.2 gibt es eine entscheidbare Sprache L , die von keiner der Maschinen aus dieser Aufzählung akzeptiert wird und die daher nicht kontextsensitiv sein kann. ■

Aufgabe 17.2. Die Chomsky-Hierarchie wird *Hierarchie* genannt, weil Typ-3-Sprachen Typ-2-Sprachen sind, Typ-2-Sprachen Typ-1-Sprachen und Typ-1-Sprachen Typ-0-Sprachen. Können Sie sich noch an die Begründungen für diese Teilmengenbeziehungen erinnern? Und können Sie anhand von Beispielen aus dem Skript belegen, warum es sich jeweils um echte Teilmengen handelt?

Aufgabe 17.3. Wenn man als fünfte Klasse die entscheidbaren Sprachen hinzufügen würde, wie würden sich die Antworten auf Aufgabe 17.2 dann ändern?

Wie steht es mit den Abschlusseigenschaften?

Satz 17.4

Die Menge der rekursiv aufzählbaren Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen Vereinigung, Durchschnitt, Konkatenation und kleevesche Hülle.

¹⁴Um der formalen Definition der Akzeptanz zu entsprechen, nehmen wir die Maschine, die der Satz liefert, und bauen sie so um, dass sie bei einer negativen Antwort in einem nicht akzeptierenden Zustand terminiert.

Beweis. Formal führen wir den Beweis nur für $\Sigma = \Sigma_{\text{bool}}$. Man kann ihn jedoch mit einem gewissen technischen Aufwand auf beliebige Alphabete übertragen.

Zunächst zur Konkatenation. Dazu wählen wir zu zwei vorgegebenen Sprachen L_1 und L_2 Turingmaschine T_1 und T_2 , die sie aufzählen. Wir verwenden wieder [Cantors erstes Diagonalargument](#), um alle Paare (i, j) aus $\mathbb{N} \times \mathbb{N}$ zu durchlaufen. Für jedes dieser Paare rufen wir T_1 mit der Eingabe i und danach T_2 mit der Eingabe j auf und hängen die beiden erhaltenen Wörter hintereinander. Damit erzeugen wir nach und nach sicher alle Wörter aus $L_1 \circ L_2$.¹⁵

Für die kleenesche Hülle wählen wir zur Sprache L eine Turingmaschine T , die sie aufzählt, und gehen im Prinzip ähnlich vor. Nur müssen wir jetzt *alle* Tupel aus natürlichen Zahlen durchlaufen. $(2, 3)$ würde beispielsweise bedeuten, dass wir T erst mit der Eingabe 2 und dann mit 3 aufrufen, während $(42, 100, 13, 7)$ vier Aufrufe von T nach sich ziehen würde. Eine solche Aufzählung kann man erreichen, indem man nacheinander für $n = 0, 1, 2, \dots$ alle Tupel aufzählt, die höchstens n Komponenten haben und deren Komponenten kleiner als n sind (wobei man sich Wiederholungen natürlich sparen kann). Anfangen würde das so:

0,
 (0),
 (1), (0, 0), (0, 1), (1, 0), (1, 1),
 (2), (0, 2), (1, 2), (2, 0), (2, 1), (2, 2), (0, 0, 0), (0, 0, 1), ...

Der Rest sollte dann klar sein.

Den Teil des Beweises für Vereinigung und Durchschnitt können Sie sich selbst überlegen – siehe Aufgabe 17.4. ■

Aufgabe 17.4. Begründen Sie, warum sowohl der Durchschnitt als auch die Vereinigung von zwei rekursiv aufzählbaren Mengen (bzw. Sprachen) wieder rekursiv aufzählbar sind. (Tipp: In diesen Fällen ist es ggf. leichter, mit akzeptierenden statt mit aufzählenden Turingmaschinen zu arbeiten.)

Aufgabe 17.5. In Satz 17.4 wird das [Komplement](#) nicht erwähnt. Das liegt natürlich daran, dass es rekursiv aufzählbare Sprachen L über einem Alphabet Σ gibt, für die $\Sigma^* \setminus L$ nicht rekursiv aufzählbar ist. Fällt Ihnen ein Beispiel ein, das bereits erwähnt wurde?

Aufgabe 17.6. Wir haben nun endlich alle Abschlusseigenschaften für die Chomsky-Hierarchie beisammen. Wir können also mit anderen Worten immer die folgende Frage beantworten: „Sind L_1 und L_2 Sprachen über demselben Alphabet Σ und beide vom Typ x , ist dann auch $L_1 \cup L_2$ bzw. $L_1 \cap L_2$ bzw. $\Sigma^* \setminus L_1$ bzw. $L_1 \circ L_2$ bzw. L_1^* mit Sicherheit von diesem Typ?“ Es empfiehlt sich, dafür eine Tabelle zu erstellen. Ich will Ihnen die Arbeit nicht komplett abnehmen, habe aber zu Ihrer Orientierung in der

¹⁵Das ist nur eine Skizze, die noch „ausgemalt“ werden muss. Die so konstruierte Maschine soll $L_1 \circ L_2$ ja ebenfalls aufzählen. Wir brauchen daher noch einen Zähler und müssen je nach Eingabe ein anderes Wort ausgeben. Es sollte aber aus vorherigen Beweisen klar sein, wie man das machen würde.

folgenden Übersicht eingetragen, wo Sie nachschauen sollten, falls Sie es vergessen haben.

Sprachenklasse	$L_1 \cup L_2$	$L_1 \cap L_2$	$\Sigma^* \setminus L_1$	$L_1 \circ L_2$	L_1^*
regulär	Satz 6.6				
kontextfrei	Satz 10.3	Aufg. 10.10	Aufg. 10.11	Satz 10.3	
kontextsensitiv	Lemma 14.4	Lemma 14.6	Satz 14.7	Lemma 14.4	Lemma 14.5
rek. aufzählbar	Satz 17.4		Aufg. 17.5	Satz 17.4	

Aufgabe 17.7. Welche der fünf Abschlusseigenschaften hat die Klasse der endlichen Sprachen aus Aufgabe 14.8?

Aufgabe 17.8. In der Übersicht zu Aufgabe 17.6 fehlt die Operation $L_1 \setminus L_2$. Was kann man darüber sagen?

Aufgabe 17.9. Sei L_3 und L_2 Sprachen über dem Alphabet Σ . L_3 sei regulär und L_2 kontextfrei. Ist dann $L_2 \cup L_3$ mit Sicherheit kontextfrei? Und $\Sigma^* \setminus L_3$?

★Aufgabe 17.10. Zum Knobeln: Unter den Voraussetzungen von Aufgabe 17.9 kann man mit den bisherigen Mitteln *nicht* schließen, dass $L_2 \cap L_3$ kontextfrei ist. Die Sprache *ist* jedoch kontextfrei. Das kann man durch die Konstruktion eines geeigneten Kellerautomaten begründen. Haben Sie eine Idee, wie man das machen könnte?

VI

Komplexitätstheorie

Im den letzten beiden Kapiteln sind wir immer davon ausgegangen, dass wir „genug“ Zeit und Speicherplatz zur Verfügung haben, weil es uns um die grundsätzliche Frage ging, ob gewisse Probleme überhaupt gelöst werden können. In diesem Kapitel wird es nun um prinzipiell lösbare Probleme gehen und um die Frage, mit welchem Aufwand man für deren Lösung rechnen muss. Man spricht dann von *Komplexität*, womit gemeint ist, wie viele Ressourcen benötigt werden.



Dabei ist die Rolle der Komplexitätstheorie gleichsam die des Überbringers der schlechten Nachrichten. In der *Praktischen Informatik* beschäftigt man sich (unter anderem) mit dem Entwurf und der Analyse von *Algorithmen*. Hat man für ein Problem ein Verfahren entwickelt und dieses untersucht, so hat man damit automatisch auch eine Obergrenze für den Ressourcenverbrauch: So geht es und vielleicht geht es sogar (mit einer anderen Idee) noch schneller oder mit noch weniger Speicher. In der Theoretischen Informatik ist man hingegen hauptsächlich an *unteren* Schranken interessiert: Wie viel Zeit bzw. Platz wird ein Algorithmus für ein bestimmtes Problem auf jeden Fall benötigen? Das ist im Allgemeinen eine wesentlich schwierigere Frage, weil man nicht nur *ein* Lösungsverfahren betrachten muss, sondern alle denkbaren – auch die, die noch gar nicht existieren. Anders ausgedrückt: Man untersucht nicht den Algorithmus, sondern versucht die Schwere des Problems zu bewerten. Viele Fragen in diesem Bereich sind trotz jahrzehntelanger Forschung noch offen, aber es gibt auch schon eine umfangreiche Theorie.

Wir werden uns nur mit der *Zeit*komplexität beschäftigen. Die Methoden bei der Untersuchung der *Platz*komplexität sind ähnlich. Zudem werden wir dieses umfangreiche Thema nur kurz anschnitten können. Ausführlicher wird es in den *Aufzeichnungen der Vorlesung aus dem Sommersemester 2014* behandelt.

18. Polynomiale Laufzeit

Als einführendes Beispiel schauen wir uns zwei *Python*-Funktionen zur Berechnung der Summe $\sum_{k=0}^n 1.01^k$ an, die beide nicht den Operator `**` für das Potenzieren verwenden.

```
def aop1(n):
    s = 0
    for k in range(n+1):
        p = 1
        for i in range(k):
            p *= 1.01
        s += p
    return s

def aop2(n):
    s = 1
    for i in range(n):
        s *= 1.01
        s += 1
    return s
```

Landau-Symbole

Beide Funktionen berechnen (bis auf eventuelle Rundungsfehler) dasselbe Ergebnis, aop2 ist jedoch schneller als aop1. Aber wie genau ist das gemeint? In der Informatik verwendet man für die Präzisierung solcher Aussagen die sogenannten *Landau-Symbole*. Die sollten Sie aus der Mathevorlesung kennen, aber der Vollständigkeit halber folgt hier eine Definition.

Definition

Für die Menge aller Funktionen von einer Menge A in eine Menge B verwenden wir die übliche Schreibweise B^A . Ist f eine Funktion von $\mathbb{N}$ nach $\mathbb{R}_{\geq 0}$, so schreiben wir

$$\mathcal{O}(f) = \{g \in \mathbb{R}_{\geq 0}^{\mathbb{N}} : \text{Es gibt ein } C > 0 \text{ mit } g(n) \leq C f(n) \text{ für alle } n \in \mathbb{N}\}.$$

Wenn wir Schreibweisen wie $\mathcal{O}(n^2)$ verwenden, dann ist mit n^2 die Funktion gemeint, die n auf n^2 abbildet.

Aufgabe 18.1. Lesen Sie sich noch einmal durch, worum es bei der Landau-Notation geht, falls Sie das vergessen haben. (Siehe z. B. Kapitel 39 von [KMFL](#).)

In diesem Sinne hat aop2 ein Laufzeitverhalten von $\mathcal{O}(n)$, während das von aop1 $\mathcal{O}(n^2)$ ist. Diese Darstellung impliziert das Folgende:

- Es geht immer um ganze Klassen von Problemen, die mit einem Parameter (der meistens n heißt) versehen sind, der gewissermaßen die Schwere einer Probleminstanz quantifiziert. Die entsprechende Funktion, z. B. $n \mapsto n^2$, beschreibt, wie der Zeitverbrauch von diesem Parameter abhängt.
- Man fokussiert sich auf den wesentlichen Einfluss dieses Parameters. So würden beispielsweise $3n^2 + 5n$ und $n^2 + 8$ über einen Kamm geschert werden. Insbesondere spielen konstante Faktoren keine Rolle. Natürlich ist es in der Praxis nicht egal, ob ein Programm denselben Job doppelt so schnell wie

ein anderes erledigt. Aber solche Unterschiede spielen in der Regel für die Beurteilung der Güte eines Algorithmus keine Rolle.¹

- Man ist in erster Linie daran interessiert, wie sich die Algorithmen für große n schlagen, denn die Landau-Notation sagt nur etwas über das *asymptotische* Laufzeitverhalten aus. Insbesondere sind Probleme, die nur endlich viele Instanzen haben, relativ uninteressant.²

Für spezifische Algorithmen kann ein Unterschied wie oben zwischen $\mathcal{O}(n)$ und $\mathcal{O}(n^2)$ bereits sehr viel ausmachen. Für die Komplexitätstheorie hat sich jedoch ein wesentlich größeres Maß als zielführend herauskristallisiert. Bei Ordnungen der Form $\mathcal{O}(n^k)$ für $k \in \mathbb{N}^+$ spricht man von *polynomialem* Laufzeitverhalten. Das wird im Allgemeinen als „gut“ bzw. tolerierbar angesehen. Dabei handelt es sich um eine heuristische Beurteilung: Natürlich wäre ein Algorithmus mit einem Laufzeitverhalten von $\mathcal{O}(n^{1000})$ nicht zu gebrauchen. Die Praxis zeigt jedoch, dass bei so ziemlich allen Lösungen mit polynomialem Laufzeitverhalten für *realistische* Probleme nur vergleichsweise kleine Exponenten vorkommen. *Nicht* tolerierbar sind hingegen z. B. Funktionen von der Form b^n für $b > 1$. Das nennt man *exponentielles* Wachstum.³

polynomial

exponentiell

★**Aufgabe 18.2.** Es gibt Funktionen, die zwischen den beiden eben beschriebenen Klassen liegen: Begründen Sie, dass $n^{\ln n}$ schneller als jedes Polynom, aber langsamer als jede Exponentialfunktion⁴ wächst.

Aufgabe 18.3. Fallen Ihnen noch andere Funktionen ein, die wie die aus Aufgabe 18.2 *superpolynomial*, aber *subexponentiell* sind?

Wie wir schon ganz am Anfang – am Ende von Abschnitt 1 – gesehen hatten, eignen sich formale Sprachen dafür, beliebige Fragestellungen zu formalisieren. Das machen wir nun.

Ein **Entscheidungsproblem** ist eine formale Sprache L . Und eine Turingmaschine, die L entscheidet, wird als **Lösung** des Problems bezeichnet.

Definition

$E = \{w \in \Sigma_{\text{bool}}^+ : n_w \equiv 0 \pmod{2}\}$ ist z. B. ein (ganz einfaches) Entscheidungsproblem. Es geht lediglich darum, festzustellen, ob man die Binärdarstellung einer geraden Zahl vor sich hat.

¹Man könnte aop1 etwa dadurch beschleunigen, dass man eine kompilierte Sprache wie C oder einen schnelleren Prozessor verwendet. Das ändert aber nichts daran, dass das in aop2 benutzte Verfahren besser ist.

²Zumindest theoretisch könnte man sämtliche Lösungen tabellieren. So etwas ist in der Informatik *durchaus üblich*.

³Rein formal gilt beispielsweise $n^2 \in \mathcal{O}(2^n)$. So gesehen bedeutet auch $\mathcal{O}(n^2)$ „exponentielles Wachstum“. Im üblichen Sprachgebrauch ist „exponentiell“ aber immer im Sinne von „exponentiell und *nicht* polynomial“ gemeint. Bei der Angabe von Landau-Symbolen gehen wir also implizit immer davon aus, dass eine scharfe untere Schranke gemeint ist.

⁴Natürlich sind damit nur Funktionen mit Basen gemeint, die größer als 1 sind.

Konvention

Wir betrachten in diesem Kapitel nur entscheidbare Sprachen. Daher gehen wir im Folgenden auch davon aus, dass wir es nur mit Turingmaschinen zu tun haben, die immer anhalten, und interpretieren die Formulierung, dass eine Turingmaschine eine Sprache *entscheidet*, entsprechend liberal. Es reicht, wenn man nach dem Terminieren zweifelsfrei erkennen kann, ob das Eingabewort zur Sprache gehört oder nicht. Das kann z. B. durch Akzeptanz geschehen (Abschnitt 14) oder wie in der Definition von *entscheidbar* auf Seite 78.

Definition

Ist T eine deterministische Turingmaschine, die bei Eingabe des Wortes w nach k Übergängen der Form $\Rightarrow_T$ terminiert, so schreiben wir $\text{time}_T(w)$ für k . Im Falle einer nichtdeterministischen Turingmaschine T schreiben wir $\text{ntime}_T(w)$ für die *minimale* Anzahl von Übergängen, die bei der Eingabe w bis zum Terminieren benötigt werden.

$\text{time}_T(w)$ ist also die Anzahl der „Rechenschritte“, die T braucht, um die Eingabe w zu verarbeiten. Für unser einfaches Beispiel E von oben könnten wir eine Turingmaschine T_E verwenden, die einfach über das Eingabewort wandert, sich jeweils die letzte gesehene Binärziffer als Zustand merkt und beim ersten Leerzeichen entsprechend entscheidet, ob sie in einem akzeptierenden oder einem anderen Zustand terminiert. Dann würde etwa $\text{time}_{T_E}(010) = 4$ gelten.

Definition

Für eine Funktion f von $\mathbb{N}$ nach $\mathbb{R}_{\geq 0}$ sei $\text{TIME}(f)$ die Klasse aller Sprachen L , für die es eine deterministische Turingmaschine T gibt, die L entscheidet und für die $\text{time}_T(w) \leq f(|w|)$ für alle Wörter w über ihrem Eingabealphabet gilt. Analog wird $\text{NTIME}(f)$ für nichtdeterministische Turingmaschinen definiert. Schließlich setzen wir

$$\mathbf{P} = \bigcup_{p \text{ Polynom}} \text{TIME}(p) \quad \text{und} \quad \mathbf{NP} = \bigcup_{p \text{ Polynom}} \text{NTIME}(p).$$

Die simple Maschine T_E zeigt offenbar, dass das Problem E ein Element von $\text{TIME}(n+1)$ und damit von $\mathbf{P}$ ist.

$\mathbf{P}$ steht für *polynomiales* Laufzeitverhalten und $\mathbf{NP}$ für *nichtdeterministisches polynomiales* Laufzeitverhalten.⁵ Weil sicher jede deterministische Turingmaschine durch eine nichtdeterministische mit identischem Laufzeitverhalten simuliert werden kann, ist klar, dass $\mathbf{P} \subseteq \mathbf{NP}$ gelten muss.

Man beachte, dass nach dieser Definition $\text{time}_T(w) \leq f(n)$ für *alle* Wörter mit $|w| = n$ gelten muss. $f(n)$ ist also eine obere Schranke für das Laufzeitverhalten von Eingaben der Länge n und wird daher vom langsamsten Durchlauf für so ein Wort determiniert. Wir machen damit eine Aussage über den sogenannten *worst case*. Es ist auch möglich, z. B. die *durchschnittliche* Laufzeit für Eingaben einer

worst case

⁵Und nicht etwa, wie es manchmal missverstanden wird, für „nicht polynomial“.

bestimmten Länge zu untersuchen, aber das ist in der Regel komplizierter und wir werden es hier nicht machen.⁶

Für die Definitionen dieser wichtigen Klassen werden typischerweise Turingmaschinen verwendet, damit man eine einfache und gleichzeitig präzise Grundlage hat. Aber wir wissen auch schon, dass Turingmaschinen im Vergleich zu „richtigen“ Computern sehr langsam sind. Die folgenden Theoreme zeigen, dass das in diesem Fall irrelevant ist.

Wenn eine **Mehrband-Turingmaschine** ein Entscheidungsproblem in polynomialer Zeit⁷ lösen kann, dann gibt es auch eine Maschine mit *einem* Band, die dieses Problem in polynomialer Zeit lösen kann. Man kann das sogar mit einer Maschine erreichen, die nur eine Spur und einen eingeschränkten Zeichenvorrat (z. B. Σ_{bool}) verwendet.

Lemma 18.1

Beweis. In Aufgabe 12.15 wurde skizziert, wie eine Mehrband-Turingmaschine T mit einer Mehrspur-Turingmaschine T' simuliert werden kann, die für die Simulation eines Schritts von T ihr eigenes Band zweimal abfährt. Sei p ein Polynom mit $\text{time}_T(w) \leq p(|w|)$ für alle Wörter w über dem Eingabealphabet. Der beschriebene Teil der Bänder von T kann bei einer Eingabe der Länge n maximal die Länge $2p(n)$ erreichen, weil pro Rechenschritt nur ein Feld geschrieben werden kann. (Die Schreib-Lese-Köpfe können sich allerdings in unterschiedliche Richtungen bewegen.) T' benötigt zur Simulation eines Schritts von T also höchstens $4p(n)$ Schritte und damit zur Simulation von $p(n)$ Schritten höchstens $4p(n)^2$ Schritte. Das ist ebenfalls ein Polynom in n .

Will man nun – wie in Aufgabe 12.14 beschrieben – die mehrspurige Maschine T' durch eine mit nur einer Spur simulieren, so muss man lediglich das Bandalphabet vergrößern, aber das ändert nichts an der Anzahl der Rechenschritte.

Schließlich muss man sich noch überlegen, welche Auswirkungen das Einschränken des Zeichenvorrats (siehe dazu Aufgabe 12.3) auf die Laufzeit hat. In diesem Fall geht es jedoch nur um einen konstanten Faktor. Hat die simulierte Maschine beispielsweise ein Bandalphabet mit 16 Zeichen, so kann man jedes von denen durch jeweils vier Bit (also eine Sequenz aus vier Σ_{bool} -Symbolen) darstellen. Um einen Schritt der simulierten Maschine nachzustellen, braucht die simulierende Maschine dann bis zu zwölf Schritte: vier für das Lesen, vier für das Schreiben und vier für das Positionieren des Kopfes. Insgesamt wird die Anzahl der Schritte also lediglich mit dem Faktor 12 multipliziert und bleibt damit polynomial. Nicht einmal **der Grad des Polynoms** wird dabei größer.⁸ ■

⁶Beachten Sie, dass es schon schwierig ist, überhaupt zu definieren, was mit der „durchschnittlichen“ Laufzeit überhaupt gemeint ist.

⁷Bei solchen und ähnlichen Formulierungen sollte in Zukunft hoffentlich immer klar sein, wie es gemeint ist: Ist T so eine Maschine, dann soll es ein Polynom p mit $\text{time}_T(w) \leq p(|w|)$ für alle möglichen Eingabewörter w geben. Sinngemäß übertragen wir diese Bezeichnungen auch auf andere Rechnermodelle, wobei ein sinnvolles Maß für die Anzahl der Rechenschritte vorausgesetzt wird.

⁸Nebenbei bemerkt wird das Eingabewort natürlich auch entsprechend länger.

Satz 18.2

Wenn ein Programm auf einem herkömmlichen Computer mit **Von-Neumann-Architektur** ein Entscheidungsproblem in polynomialer Zeit löst, dann kann dieser Algorithmus auf einer Turingmaschine simuliert werden, die ebenfalls in polynomialer Zeit arbeitet.

Beweis. Eigentlich ist das viel zu vage formuliert, um von einem Satz zu sprechen, den man beweisen kann. Wir werden jedoch im Folgenden eine Reihe von realistischen Annahmen machen und einen Beweis unter diesen Voraussetzungen skizzieren.⁹ Das wird Sie hoffentlich überzeugen, dass die Aussage von der Grundidee her korrekt ist.

Wir gehen von einem Computer mit einer **Wortbreite** von k Bits aus, der 2^k Speicherzellen adressieren kann.¹⁰ Dass das Problem in polynomialer Zeit gelöst wird, soll bedeuten, dass es ein **Polynom** p mit der Eigenschaft gibt, dass für den Rechenvorgang bei einer Eingabe, die aus n Datenworten besteht, maximal $p(n)$ Taktzyklen vergehen. (Die Eingabe wird dabei am Anfang in einem dafür vorgesehenen Teil des Speichers stehen.) Unser Computer hat diverse **Register**, in denen Berechnungen vorgenommen werden. Da wir als Zeiteinheit Taktzyklen verwenden, gehen wir von einem Programm aus, das bereits kompiliert in **Maschinensprache** vorliegt. Je nach Prozessor kann dessen Maschinensprache sehr unterschiedliche Befehle zur Verfügung stellen. Man kann diese jedoch immer grob in einige wenige Klassen einteilen:

- Befehle, die Daten aus dem Speicher in ein Register laden,
- Befehle, die Daten von einem Register in den Speicher schreiben,
- Befehle, die Berechnungen mit den Registerinhalten durchführen, und
- (evtl. bedingte) Sprungbefehle.

Wir gehen der Einfachheit halber davon aus, dass alle Befehle nur einen Taktzyklus benötigen und pro Befehl maximal ein Datenwort zwischen den Registern und dem Speicher ausgetauscht werden kann. Komplexere Befehle, über die manche Prozessoren verfügen, lassen sich als Kombinationen solcher Basisbefehle realisieren – und benötigen dann entsprechend mehr Taktzyklen.

Die zum Simulieren verwendete Turingmaschine T ist eine **Mehrband-Maschine**, die die folgenden Bänder verwendet:

- Ein Band für den Speicher des Computers, in dem für alle vom Programm verwendeten Speicherzellen auf dem Band $2k + 2$ Symbole stehen: k Bits für den Inhalt, k Bits für die Adresse und zwei Trennzeichen. Das könnte beispielsweise so aussehen, dass für $k = 8$ auf dem Band $\$10000110\#00101010$

⁹Beachten Sie, dass wir *nicht* die „Übersetzung“ von herkömmlichen Programmen in Turingmaschinen verwenden, die in Kapitel IV vorgeführt wurde. Die war zwar vergleichsweise einfach zu verstehen und zu implementieren, ist aber wegen des „Umwegs“ über die Registermaschinen extrem ineffizient.

¹⁰Natürlich muss es nicht so sein, dass die Wortbreite und die Länge der Adressen übereinstimmen. In diesem Fall wählen wir als k einfach den größeren der beiden Werte.

steht und das so interpretiert wird, dass sich in der Speicherzelle mit der Adresse 134 der Inhalt 42 befindet.

- Ein Band für die Adresse des aktuellen Befehls. Dieses Band simuliert also den **Befehlszähler**.
- Ein Band für den aktuellen Befehl.
- Ein Band für die zum aktuellen Befehl gehörende Speicheradresse.
- Ein Band pro Register.
- Ein oder mehrere „Arbeitsbänder“ für Zwischenrechnungen.

Analog zum Beweis von Lemma 18.1 können wir uns zunächst überlegen, dass der Computer bei einer Eingabe der Länge n maximal $p(n) + n + C$ Datenworte des Speichers verwendet haben kann, wobei C die konstante Länge des Programms ist. Wenn T also etwas auf dem Band, das den Speicher simuliert, sucht oder auf dieses Band etwas schreiben muss, werden dafür jeweils höchstens $(2k + 2)(p(n) + n + C)$ Rechenschritte verbraucht. Das ist ein Polynom in n (vom selben Grad wie p , weil k konstant ist), das wir q nennen.

Für jeden Befehl des Computers muss T zunächst den Befehl vom Speicherband laden, auf den der Befehlszähler zeigt. Dieser Befehl wird analysiert und evtl. wird eine Speicheradresse berechnet. Dann muss evtl. ein Datenwort vom Speicherband gelesen werden. Anschließend finden ggf. Berechnungen statt und vielleicht wird danach ein Datenwort von einem Register in den Speicher transferiert. Zum Schluss wird der Befehlszähler aktualisiert, falls er nicht bereits durch einen Sprungbefehl gesetzt wurde. Pro Zyklus gibt es zwei Zugriffe auf den Speicher, die maximal $2q(n)$ Rechenschritte von T erfordern. Alle Operationen, die auf anderen Bändern stattfinden, z. B. arithmetische Operationen mit den Registerinhalten, sind zwar evtl. aufwendig, hängen aber nicht von n , sondern nur von k ab. Für jeden simulierten Befehl braucht T jedenfalls maximal $r(n)$ Rechenschritte, wobei r ein Polynom in n ist. Damit ergeben sich für die Simulation des kompletten Programms $r(n)p(n)$ Schritte, und wir haben es wieder mit einem Polynom zu tun. ■

Aufgabe 18.4. Vergleichen Sie den im Beweis von Satz 18.2 skizzierten Computer mit einem tatsächlichen Rechner und überlegen Sie sich, was man für eine realistischere Modellierung ändern müsste und dass das keine Auswirkungen auf die Aussage des Satzes haben würde.

Aufgabe 18.5. Müssen im Beweis von Satz 18.2 die Zellen auf dem Band, das den Speicher simuliert, nach Adressen sortiert sein?

Aufgabe 18.6. Turingmaschinen haben (auf ihren Bändern) theoretisch unbegrenzten Speicherplatz zur Verfügung. Für den simulierten Computer aus Satz 18.2 gilt das jedoch wegen der festen Länge für die Adressen nicht. Überlegen Sie sich, wie man das ändern könnte. Welche Auswirkungen hat das auf die simulierende Turingmaschine? Was muss man sinnvollerweise voraussetzen, damit die Simulation weiterhin polynomiales Laufzeitverhalten hat?

Satz 18.2 und Lemma 18.1 zeigen, dass die Wahl der Turingmaschinen als Modell für die Definition von Klassen wie **P** keine Einschränkung ist, wie man zunächst

vermuten könnte. Man hätte die Begriffe stattdessen auch mittels [Registermaschinen](#) oder anderer Modelle definieren können und wäre auf dasselbe Ergebnis gekommen. Das kann man folgendermaßen in sehr allgemeiner Form ausdrücken:

Erweiterte Church-Turing-These

Wenn ein Problem für ein Rechnermodell zur Klasse **P** gehört, dann gehört es für jedes Rechnermodell zur Klasse **P**.

Anders als bei der „klassischen“ Church-Turing-These ist hier allerdings Vorsicht geboten. Nicht alle Experten stimmen der erweiterten These vollumfänglich zu. So ist beispielsweise nicht klar, ob sie auch für [Quantencomputer](#) zutrifft. Für konventionelle Rechner sind Turingmaschinen jedoch auch dann als Modell geeignet, wenn es nicht nur um die prinzipielle Berechenbarkeit, sondern um *effiziente* Berechenbarkeit geht. In diesem Sinne ist polynomiales Laufzeitverhalten ein robustes und rechnerunabhängiges Maß für die Komplexität von Problemen.

An dieser Stelle sei noch einmal der am Anfang des Kapitels bereits erwähnte wesentliche Unterschied zwischen den Untersuchungsgegenständen der Praktischen und der Theoretischen Informatik im Bezug auf die Komplexität hervorgehoben. Während man in der Praktischen Informatik die Güte von spezifischen Algorithmen für ein bestimmtes Problem quantifiziert, möchte man in der Theoretischen Informatik etwas über das gesamte Problem (und damit über *alle möglichen* Algorithmen für dieses Problem) aussagen. Gehört ein Problem beweisbar *nicht* zur Klasse **P**, dann ist es wirklich *schwer* in dem Sinne, dass man jegliche Hoffnung aufgeben kann, jemals einen Lösungsalgorithmus zu finden, der eine in der Praxis tolerierbare Laufzeit hat. Welche Bedeutung die Klasse **NP** hat, bei der es um fiktive Maschinen geht, die in der Praxis nicht zu realisieren sind, wird sich noch herausstellen.

19. Realistische Probleme



Unsere Definition des Begriffs *Entscheidungsproblem* ist zwar einerseits sehr präzise, legt uns andererseits aber ein sehr enges Korsett an. Eigentlich sind wir es gewohnt, Probleme umgangssprachlich zu formulieren, zum Beispiel: Man entscheide für eine vorgegebene Zahl $n \in \mathbb{N}$, ob sie eine [Primzahl](#) ist. Damit daraus jedoch ein Entscheidungsproblem im engeren Sinne wird, muss man die Frage in eine formale Sprache einkleiden. Das könnte in der Form

$$P = \{ w \in \Sigma_{\text{bool}}^+ : n_w \in \mathbb{P} \}$$

geschehen. Aber man könnte es auch ganz anders machen, indem man die Zahlen etwa dezimal, [unär](#) oder gar in [römischer Zahlschrift](#) darstellt. Und rein technisch hätte man bereits ein anderes Problem, wenn man in P die Rollen von 0 und 1 vertauschen würde. Man hätte jeweils dieselbe Fragestellung unterschiedlich *codiert*. Wir wollen uns einerseits keine unnötigen Gedanken über technische Feinheiten machen, andererseits wollen wir aber auch keine unsinnigen Codierungen zulassen.

Codierung

Betrachten wir beispielsweise die Menge D aus (1.2) vom Anfang des Skripts und davon die Teilmenge E der Wörter, die nach der dortigen Interpretation erfüllbare Formeln darstellen, so handelt es sich um ein Entscheidungsproblem, für das bisher nur Lösungsalgorithmen bekannt sind, die exponentielle Laufzeitverhalten wie 2^n haben. Dabei ist jedoch wesentlich, dass in unsere Definition der Laufzeit mithilfe von Klassen der Form $\text{TIME}(f)$ die Länge der Eingabe eingeht. Wir könnten das Problem E nun folgendermaßen ändern: Wir fügen zum Alphabet ein weiteres Zeichen $\#$ hinzu und betrachten statt E die Sprache

$$E' = \{w\#^{2^{|w|}} : w \in E\},$$

d. h., wir fügen an jedes Wort w aus E einfach eine bedeutungslose Zeichenkette der Länge $2^{|w|}$ an. E' beschreibt im Prinzip dieselbe Fragestellung, wir hätten aber für E' einen polynomialen Algorithmus, denn bei der Ermittlung des Laufzeitverhaltens würden ja die überflüssigen Zeichen einbezogen werden! Das wäre natürlich Quatsch.

Solche Erwägungen sind nicht so skurril, wie Sie vielleicht denken. Wenn wir darüber sprechen, dass ein Algorithmus ein Laufzeitverhalten wie $\mathcal{O}(n^3)$ oder $\mathcal{O}(2^n)$ hat, dann müssen wir uns vorher gut überlegt haben, welche Größe die Rolle von n einnimmt, d. h., welcher Parameter die Schwere des Problems korrekt wiedergibt. Das spielt häufig dann eine Rolle, wenn es um die Codierung von Zahlen geht. Die Frage, ob eine vorgegebene Zahl $n \in \mathbb{N}$ durch 3 teilbar ist, lässt sich beispielsweise mit linearem Laufzeitverhalten lösen: der Aufwand ist proportional zur Eingabegröße. Aber diese Eingabegröße ist *nicht* $n!$ Ein entsprechender Algorithmus braucht nicht doppelt so lange für $2n$ wie für n . Die Eingabegröße ist die Anzahl der Stellen von n im Binärsystem (oder einem anderen [Stellenwertsystem](#)). Ausschlaggebend für die Laufzeit ist also der Logarithmus von n .¹¹

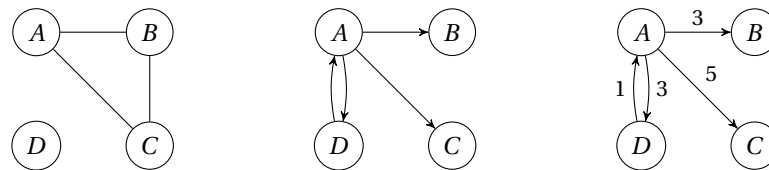
Wir gehen jedenfalls im Folgenden von *sinnvollen* – und insbesondere möglichst kurzen – Codierungen aus. Da es immer sehr viele verschiedene Darstellungsmöglichkeiten gibt, sei hier im Vorgriff auf Lemma 19.1 schon einmal gesagt, dass wir zwei Codierungen derselben Frage als gleichwertig behandeln können, wenn die Umwandlung von jeder von beiden in die jeweils andere mit polynomialem Zeitaufwand zu bewältigen ist.

Ein (**ungerichteter**) **Graph** ist ein Paar (V, E) , wobei V eine Menge und E eine Menge von zweielementigen Teilmengen von V ist. Ein **gerichteter Graph** ist ein Paar (V, E) , wobei V eine Menge und E eine Teilmenge von $V \times V$ ist. Die Elemente von V nennt man die **Ecken** oder **Knoten** (engl. *vertices*) des Graphen, die von E die **Kanten** (engl. *edges*). Aus einem (ungerichteten oder gerichteten) Graphen macht man einen **gewichteten** Graphen, indem man zu V und E noch eine Funktion $f: E \rightarrow \mathbb{R}$ hinzufügt, die jeder Kante eine Zahl (ihr *Gewicht*) zuordnet.

Definition

¹¹Wäre n die entscheidende Größe, dann würde das nach unserer Definition bedeuten, dass wir die Eingabe unär auf das Band der Turingmaschine schreiben.

Wir betrachten nur Graphen, bei denen V (und damit auch E) endlich ist. Graphen werden in der Regel dadurch visualisiert, dass man die Knoten als Punkte in der Ebene zeichnet und die Kanten als Verbindungslinien zwischen den Punkten. Bei gerichteten Graphen erhalten diese Linien zusätzlich Pfeile, bei gewichteten Graphen fügt man den Linien die Gewichte hinzu. Die folgende Skizze zeigt links einen ungerichteten Graphen mit $V = \{A, B, C, D\}$ und $E = \{\{A, B\}, \{A, C\}, \{B, C\}\}$, in der Mitte einen gerichteten Graphen mit derselben Eckenmenge und den Kanten $E = \{(A, B), (A, C), (A, D), (D, A)\}$ sowie rechts denselben gerichteten Graphen mit der Gewichtsfunktion $f = \{((A, B), 3), ((A, C), 5), ((A, D), 3), ((D, A), 1)\}$.



Aufgabe 19.1. Wie viele Kanten kann ein ungerichteter Graph mit n Ecken maximal haben? Und wie ist das bei gerichteten Graphen?

Definition

Ein **Weg** in einem Graphen ist eine Folge von Knoten des Graphen, in der jeweils zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind.¹² In einem gewichteten Graphen bezeichnet man die Summe der Kantengewichte als **Länge** des Weges. Gibt es keine Gewichte, so werden einfach die Kanten gezählt.¹³

Im obigen Beispiel links ist (C, A, B) ein Weg von C nach B der Länge 2. Rechts ist (D, A, B) ein Weg von D nach B der Länge 4. Es gibt dort aber keinen Weg von B nach D . Ein typisches Problem aus der Praxis besteht darin, in einem vorgegebenen Graphen den kürzesten Weg zwischen zwei Knoten zu finden.

Aufgabe 19.2. Inwiefern passt dieses Problem nicht zu denen, über die wir in diesem Kapitel bisher gesprochen haben?

Aufgabe 19.2 weist auf eine formale Schwierigkeit hin. In der Praxis geht es oft darum, bestimmte Ergebnisse zu ermitteln oder – wie im Fall der kürzesten Wege in Graphen – aus einer Menge möglicher Ergebnisse das auszuwählen, das nach bestimmten Kriterien das beste ist. Man spricht dann beispielsweise von *Funktions-*, *Optimierungs-* oder *Suchproblemen*. In der Komplexitätstheorie beschäftigt man sich trotzdem hauptsächlich mit Entscheidungsproblemen, weil die am einfachsten zu handhaben sind. Wir werden auf den folgenden Seiten für Beispiele auch Probleme betrachten, bei denen *ja* oder *nein* als Antwort nicht ausreichen, während wir uns bei mathematischen Definitionen und Aussagen weiterhin auf Entscheidungsprobleme beschränken werden.

¹²In einem ungerichteten Graphen *verbindet* die Kante $\{v_1, v_2\}$ die Ecken v_1 und v_2 . In einem gerichteten Graphen *verbindet* die Kante (v_1, v_2) die Ecken v_1 und v_2 (aber *nicht* v_2 und v_1).

¹³Das entspricht einer Funktion, die jeder Kante das Gewicht 1 zuordnet.

Häufig kann man auch einen Zusammenhang zwischen der Komplexität allgemeinerer Probleme und der von verwandten Entscheidungsproblemen herstellen. Nehmen wir beispielsweise die Frage nach der Länge eines kürzesten Weges von A nach B durch einen Graphen. Hat man einen Algorithmus, der diese Frage beantworten kann, dann kann man mit dessen Hilfe offenbar auch ganz einfach das folgende Entscheidungsproblem $W(G, A, B, k)$ lösen: „Gibt es im Graphen G einen Weg von A nach B , der höchstens die Länge k hat?“ Umgekehrt kann man jedoch mit einem Programm, das dieses Entscheidungsproblem lösen kann, auch die Frage nach der Länge eines kürzesten Weges beantworten. Man muss nur nacheinander $W(G, A, B, 1)$, $W(G, A, B, 2)$, $W(G, A, B, 3)$ und so weiter fragen, bis zum ersten Mal die Antwort *ja* kommt. Das ist zwar aufwendiger, aber man weiß von vornherein, dass man höchstens so oft fragen muss, wie der Graph Knoten hat. Ist der Algorithmus für das Entscheidungsproblem polynomial in der Anzahl der Knoten, so hat man damit auch eine Lösung für das Optimierungsproblem, die in polynomialer Zeit arbeitet.

Aufgabe 19.3. Fallen Ihnen mathematische Probleme ein, für die ein Entscheidungsproblem signifikant einfacher ist als ein eng verwandtes Problem, bei dem es nicht nur um *ja* oder *nein* geht?

Doch zurück zur Graphentheorie. Der folgende, absichtlich „naiv“ gestaltete Algorithmus findet für einen gerichteten Graphen (V, E) mit einer Gewichtsfunktion $f: E \rightarrow \mathbb{R}_{\geq 0}$ einen kürzesten Weg vom Knoten A zum Knoten B .

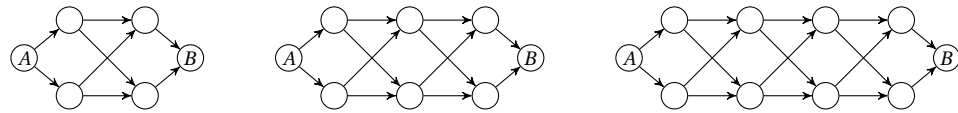
```

Q ← {A}, P ← {(A, 0)}
while Q ≠ ∅ do
  s ← some element of Q
  Q ← Q \ {s}
  for each n ∈ V with (s, n) ∈ E do
    for each (p, w) ∈ P do
      if p ends with s and doesn't contain n then
        P ← P ∪ {(pn, w + f((s, n)))}
        Q ← Q ∪ {n}
      end if
    end for
  end for
end while
if P' = {(p, w) ∈ P : p ends with B} ≠ ∅ then
  return (p, w) ∈ P' with w ≤ w' for all (p', w') ∈ P'
else
  return "no path from A to B"
end if

```

Algorithmus 19.1

Das Problem dieses Algorithmus ist, dass er *alle* von A ausgehenden Wege durchprobiert. Die folgende Grafik (bzw. Aufgabe 19.4) zeigt jedoch, dass das zu exponentiellem Wachstum der Laufzeit führen würde.



Aufgabe 19.4. Geben Sie für Graphen, die nach dem obigen Muster erstellt wurden, eine von n abhängende Formel für die Anzahl der Wege von A nach B an, wobei n die Anzahl der Knoten des Graphen sein soll.

Aber Sie kennen sicher aus den Informatikvorlesungen den **Dijkstra-Algorithmus**. Er findet ebenfalls einen kürzesten Weg von A nach B , hat allerdings ein wesentlich günstigeres Laufzeitverhalten von $\mathcal{O}(|V|^2)$:

Algorithmus 19.2

```

 $Q \leftarrow V, d[A] \leftarrow 0, d[v] \leftarrow \infty$  for all  $v \in V \setminus \{A\}, p[v] = \perp$  for all  $v \in V$ 
while  $Q \neq \emptyset$  do
   $u \leftarrow$  some element of  $\{v \in Q : d[v] \leq d[w] \text{ for every } w \in Q\}$ 
  if  $u = B$  then
     $r \leftarrow \varepsilon$ 
    while  $u \neq A$  do
       $r \leftarrow u \circ r$ 
       $u \leftarrow p[u]$ 
    end while
    return  $A \circ r$  and  $d[B]$ 
  end if
   $Q \leftarrow Q \setminus \{u\}$ 
  for each  $n \in Q$  with  $(u, n) \in E$  do
     $d' \leftarrow d[u] + f((u, n))$ 
    if  $d' < d[n]$  then
       $d[n] \leftarrow d', p[n] \leftarrow u$ 
    end if
  end for
end while
return "no path from A to B"

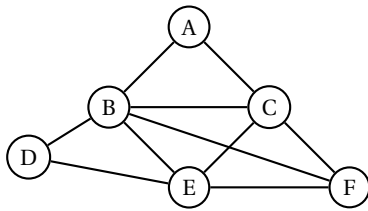
```

Dieses Beispiel demonstriert zwei wesentliche Punkte: Dass ein „schlechter“ Algorithmus für ein Problem existiert, sagt nichts über die Schwere des Problems aus. Es kann deutlich bessere Algorithmen geben (die vielleicht bisher noch nicht gefunden wurden). Und dass es exponentiell viele „Kandidaten“ gibt, die man untersuchen könnte, impliziert nicht notwendig, dass eine Lösung exponentielle Laufzeit haben muss.

Definition

Ist (V, E) ein Graph, dann nennt man eine Knotenmenge V' eine **Clique**, wenn $\{v, w\} \in E$ für alle $v, w \in V'$ mit $v \neq w$ gilt. Das **Cliquenproblem (CLIQUE)** ist das folgende Entscheidungsproblem: Gegeben sind ein Graph $G = (V, E)$ und eine natürliche Zahl k . Gibt es in G eine Clique mit k Knoten?

In der folgenden Skizze ist beispielsweise $\{A, B, C\}$ eine Clique mit drei Knoten und $\{B, C, E, F\}$ eine mit vier. Cliques mit mehr als vier Knoten gibt es nicht.



Die Frage, wie man graphentheoretische Probleme sinnvoll codiert, wird in Aufgabe 19.5 behandelt. Auf einem herkömmlichen Rechner lässt sich **CLIQUE** z. B. folgendermaßen lösen.

```

for each  $V' \subseteq V$  with  $|V'| = k$  do
   $s \leftarrow \text{true}$ 
  for each  $(v, w) \in V' \times V'$  with  $v \neq w$  do
    if  $\{v, w\} \notin E$  then
       $s \leftarrow \text{false}$ 
    end if
  end for
  if  $s$  then
    return "yes"
  end if
end for
return "no"

```

Algorithmus 19.3

Aufgabe 19.5. Geben Sie das Cliques-Problem als formale Sprache an, so dass es ein Entscheidungsproblem im Sinne der Definition von Seite 105 ist. (Mit dem Alphabet muss man also Paare der Form (G, k) darstellen können, wobei G ein Graph und k eine natürliche Zahl ist.)

Für einen Graphen mit n Knoten ist die Anzahl der äußeren Schleifendurchläufe im schlechtesten Fall $\binom{n}{k}$. Für festes k ist dieser **Binomialkoeffizient** ein Polynom¹⁴ k -ten Grades in n und damit nicht von der Ordnung $\mathcal{O}(n^{k-1})$. Daher ist dieser Algorithmus *nicht* polynomial in der Anzahl der Knoten. In diesem Fall liegt es jedoch nicht daran, dass ich ein besonders „naives“ Verfahren gewählt habe. Man kann es zwar sicher noch im Detail verbessern, aber für **CLIQUE** ist bisher keine Lösung mit polynomialer Laufzeit bekannt, d. h., man weiß nicht, ob **CLIQUE** zur Klasse **P** gehört. Wir werden sehen, dass **CLIQUE** nur ein Beispiel für viele ähnliche Probleme ist.

Aufgabe 19.6. Begründen Sie, dass der obige Algorithmus 19.3 für **CLIQUE** für festes k eine Laufzeit hat, die polynomial von der Anzahl der Ecken des Graphen abhängt.

¹⁴Zumindest dann, wenn n groß genug ist.

Aufgabe 19.7. Begründen Sie anhand der formalen Sprache aus Aufgabe 19.5 die folgende Aussage: Gibt es eine Turingmaschine T , die das Cliques-Problem entscheidet, und ein Polynom p mit $\text{time}_T(w) \leq p(|w|)$ für alle Wörter w über dem Eingabealphabet von T , dann gibt es ein Polynom q mit $\text{time}_T(w) \leq q(|V|)$ für alle Wörter w , die syntaktisch korrekt einen Graphen (V, E) darstellen. Begründen Sie außerdem, warum man nicht $|V|$ durch $|E|$ ersetzen kann.

Mit anderen Worten: Die Zugehörigkeit des Cliques-Problems zu **P** bzw. **NP** hängt per definitionem von der Codierung ab (siehe Seite 110), aber bei einer sinnvollen Codierung ist die Anzahl der Ecken die relevante Größe.

★**Aufgabe 19.8.** Überlegen Sie sich, warum es im Rahmen der Komplexitätstheorie bei graphentheoretischen Fragen sinnvoll ist, die Anzahl der Ecken als Maß zu verwenden. (Siehe dazu auch Aufgabe 19.7.)

★**Aufgabe 19.9.** Neben der Frage, wie man Graphen sinnvoll codiert, um sie platzsparend als Eingabe für einen Algorithmus präsentieren zu können, ist auch die Frage interessant, mit welchen Datenstrukturen man Graphen in Programmen darstellen sollte. Das hängt natürlich von den Anforderungen des jeweiligen Algorithmus ab und es gibt unterschiedliche Ansätze. Denken Sie darüber mal nach.

Bevor wir uns weitere Probleme ansehen, überzeugen wir uns jedoch, dass **CLIQUE** in der Klasse **NP** liegt. Das machen wir mit dem folgenden Algorithmus.

Algorithmus 19.4

```

 $V' \leftarrow \emptyset$ 
for each  $v \in V$  do
  select
    1:  $V' \leftarrow V' \cup \{v\}$ 
    2:  $V' \leftarrow V'$ 
  end select
end for
if  $|V'| = k$  and  $V'$  is clique then
  return "yes"
else
  return "no"
end if

```

Der Nichtdeterminismus äußert sich hier durch den farblich hervorgehobenen **select**-Block, der so gemeint ist, dass das Programm an dieser Stelle eine Wahlmöglichkeit hat. Damit hat, bei n Knoten, die erste **for**-Schleife eine Laufzeit von $\mathcal{O}(n)$. Der Test, ob es sich wirklich um eine Clique handelt, ist sicher auch mit einem Aufwand von maximal $\mathcal{O}(n^2)$ machbar. (Siehe Aufgabe 19.11.) Damit ist klar, dass dieser Algorithmus polynomielle Laufzeit hat.

Aufgabe 19.10. Warum so kompliziert? Könnte man in den Pseudocode von Algorithmus 19.4 statt des **select**-Blocks nicht einfach einfügen, dass das Programm sich eine k -elementige Teilmenge von V aussuchen soll?

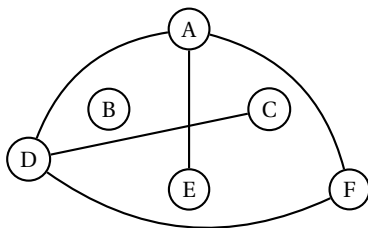
Aufgabe 19.11. Begründen Sie etwas ausführlicher, dass der Test der Cliqueneigenschaft in Algorithmus 19.4 von der Ordnung $\mathcal{O}(n^2)$ ist.

Aufgabe 19.12. Satz 14.1 demonstriert, dass man nichtdeterministische Algorithmen durch deterministische simulieren kann. Wie würde das Laufzeitverhalten eines deterministischen Algorithmus aussehen, der den nichtdeterministischen Algorithmus 19.4 für **CLIQUE** simuliert?

Ist (V, E) ein Graph, dann nennt man eine Knotenmenge V' **stabil** oder **unabhängig**, wenn $\{v, w\} \notin E$ für alle $v, w \in V'$ gilt. Das **Stabilitätsproblem (INDSET)** ist das folgende Entscheidungsproblem: Gegeben sind ein Graph $G = (V, E)$ und eine natürliche Zahl k . Gibt es in G eine stabile Menge mit k Knoten?

Definition

In der folgenden Skizze ist beispielsweise $\{A, B, C\}$ eine unabhängige Menge mit drei Knoten und $\{B, C, E, F\}$ eine mit vier. Unabhängige Mengen mit mehr als vier Knoten gibt es nicht.



Aufgabe 19.13. Welcher Zusammenhang besteht zwischen Cliques und stabilen Mengen? Wenn Sie es nicht direkt der Definition entnehmen können, dann vergleichen Sie die beiden Beispielskizzen.

Aufgabe 19.14. Was müsste man gemäß Aufgabe 19.13 im Algorithmus 19.3 ändern, damit er **INDSET** statt **CLIQUE** löst?

Aufgabe 19.15. Unter der URL <https://weitz.de/indset/> finden Sie eine interaktive Anwendung, mit der man sich mit **INDSET** vertraut machen kann.

Aufgabe 19.14 zeigt, wie ein Algorithmus zum Lösen von **INDSET** aussehen könnte. Wir wollen aber auf etwas anderes hinaus. Dafür betrachten wir diesen Algorithmus, der als Eingabe einen Graphen (V, E) erhält und dessen Komplementgraphen berechnet:

```

 $E' = \emptyset$ 
for each  $(v, w) \in V \times V$  do
  if  $v \neq w$  and  $\{v, w\} \notin E$  then
     $E' = E' \cup \{\{v, w\}\}$ 
  end if
end for

```

Algorithmus 19.5

return (V, E')

Offensichtlich erhalten wir einen deterministischen Algorithmus für **INDSET**, indem wir Algorithmus 19.5 ausführen und dessen Ausgabe als Eingabe für Algorithmus 19.3 verwenden. Ebenso erhalten wir einen nichtdeterministischen Algorithmus für **INDSET**, indem wir Algorithmus 19.5 ausführen und dessen Ausgabe an Algorithmus 19.4 weitergeben. Diesen „Trick“ wollen wir nun formalisieren.¹⁵

Definition

Seien L_1 und L_2 Entscheidungsprobleme über den Alphabeten Σ_1 und Σ_2 . T sei eine (deterministische) Turingmaschine, deren Funktion f_T auf ganz Σ_1^* definiert ist und für die es ein Polynom p mit $\text{time}_T(w) \leq p(|w|)$ für alle $w \in \Sigma_1^*$ gibt. Gilt $L_1 = f_T^{-1}[L_2] = \{w \in \Sigma_1^* : f_T(w) \in L_2\}$, so sagt man, dass L_1 **polynomial reduzierbar** auf L_2 ist, und schreibt dafür $L_1 \leq_p L_2$.

Lemma 19.1

Seien L_1 und L_2 Entscheidungsprobleme mit $L_1 \leq_p L_2$. Gilt $L_2 \in \mathbf{P}$, dann gilt auch $L_1 \in \mathbf{P}$. Und dieselbe Aussage gilt für **NP**.

Beweis. Seien T_r die Turingmaschine und p_r das Polynom, die für $L_1 \leq_p L_2$ sorgen. T und p seien Turingmaschine und Polynom, die für $L_2 \in \mathbf{P}$ sorgen. Das Hintereinanderschalten von T_r und T entscheidet dann offenbar L_1 . Für die Laufzeit der so entstandenen Turingmaschine T^* gilt

$$\begin{aligned} \text{time}_{T^*}(w) &= \text{time}_{T_r}(w) + \text{time}_T(f_{T_r}(w)) \leq p_r(|w|) + p(|f_{T_r}(w)|) \\ &\leq p_r(|w|) + p(p_r(|w|)) \end{aligned}$$

und der Ausdruck am Ende der Ungleichungskette ist ein Polynom in $|w|$. Für **NP** verläuft der Beweis ganz analog. ■

Durch Lemma 19.1 wird noch einmal bestätigt, was durch Aufgabe 19.14 eigentlich schon klar war: Auch **INDSET** gehört zur Klasse **NP**. Bevor wir uns mit der Frage beschäftigen, welche Bedeutung diese ominöse Klasse hat, die Berechnungen auf nichtexistenten Rechnern beschreibt, schauen wir uns noch eine hilfreiche alternative Charakterisierung von **NP** an.

Definition

Eine Sprache L über dem Alphabet Σ heißt **in Polynomialzeit verifizierbar**, wenn es ein Polynom p und eine in polynomialer Zeit arbeitende Turingmaschine T mit Eingabealphabet $I \supseteq \Sigma$ gibt, die eine Sprache L' entscheidet, für die gilt:

- (i) $L' \subseteq \{w\#z : w \in L \text{ und } z \in I^* \text{ und } |z| \leq p(|w|)\}$
- (ii) $L = \{w : \text{Es gibt ein } z \text{ mit } w\#z \in L'\}$

Dabei soll $\#$ ein Zeichen aus $I \setminus \Sigma$ sein.

¹⁵Man erinnere sich an die Definition des Begriffs *reduzierbar* von Seite 98.

Wie ist das zu verstehen? Wegen Bedingung (ii) gibt es für jedes Wort w aus L ein „kurzes“ Wort z (das man ein *Zertifikat* für w nennt), so dass T mittels der Eingabe $w\#z$ „schnell“ entscheiden kann, ob w zu L gehört. Dabei sorgen die polynomiale Zeitbeschränkung von T und die polynomiale Längenbeschränkung von z (in Relation zur Länge von w) dafür, dass die Entscheidung in polynomialer Zeit in *Abhängigkeit von $|w|$* durchgeführt werden kann. Außerdem kann T auch jede Eingabe der Form $w\#z$ mit $w \notin L$ „schnell“ erkennen, solange z irgendeine nicht zu lange Zeichenkette ist, weil solche Eingaben wegen (ii) nicht zu L' gehören. (In diesem Fall ist z kein Zertifikat für w .)

Das mag immer noch etwas kryptisch klingen, daher hilft vielleicht ein Beispiel.

Beim **Erfüllbarkeitsproblem der Aussagenlogik (SAT)** geht es darum, für eine aussagenlogische Formel zu entscheiden, ob sie erfüllbar ist.

Die Formel $x_1 \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_1) \wedge \neg x_3$ ist beispielsweise erfüllbar, indem man x_1 den Wert 1 (für *wahr*) und x_2 sowie x_3 den Wert 0 (für *falsch*) zuordnet. Die nur leicht modifizierte Formel $x_1 \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge \neg x_3$ ist hingegen nicht erfüllbar. Eines der Beispiele am Ende von Abschnitt 1 hat demonstriert, wie man dieses Entscheidungsproblem als formale Sprache codieren könnte.¹⁶

Dieses Problem ist in Polynomialzeit verifizierbar. Als Zertifikat kann man einfach die Belegung der Variablen nehmen. Hängt man an die erste Formel die Zeichenkette 100 als Zertifikat an,¹⁷ so kann eine Turingmaschine leicht überprüfen, dass die Formel mit dieser Belegung erfüllt wird. Für die zweite Formel gibt es kein solches Zertifikat.

Aufgabe 19.16. Wie würde ein Zertifikat für **CLIQUE** aussehen?

Eine Sprache gehört genau dann zur Klasse **NP**, wenn sie in Polynomialzeit verifizierbar ist.

Beweis. Wenn eine Sprache L ein Element von **NP** ist, dann gibt es eine nichtdeterministische Turingmaschine T , die L in Polynomialzeit entscheidet. Im Beweis von Satz 14.1 haben wir gesehen, dass wir einen Durchlauf von T durch eine endliche Folge von „Entscheidungen“ darstellen können. Die Länge dieser Folge entspricht der Anzahl der Rechenschritte und hängt nach Voraussetzung polynomial von der Länge der Eingabe ab. Also kann die Folge der Entscheidungen als Zertifikat für die Eingabe verwendet werden: Eine deterministische Turingmaschine kann mithilfe der Folge den entsprechenden Ablauf von T nachvollziehen.

¹⁶Dabei ist zu beachten, dass man in der Regel die Anzahl der Variablen als Maß für die Komplexität dieser Frage verwendet und daher zusätzlich fordert, dass die Länge der Formel polynomial beschränkt bezüglich dieser Zahl ist. Ansonsten hätte man es – siehe oben – mit einer unsinnigen Codierung zu tun.

¹⁷100 steht natürlich für $x_1 = 1, x_2 = 0, x_3 = 0$.

Zertifikat



Definition

Satz 19.2

Ist umgekehrt eine Sprache L in Polynomialzeit verifizierbar, dann gibt es ein Polynom p und eine verifizierende Turingmaschine T wie in der Definition des Begriffs. Dazu konstruiert man eine nichtdeterministische Turingmaschine, die bei der Eingabe w alle möglichen Zeichenketten durchprobiert, deren Länge höchstens $p(|w|)$ ist, und diese zusammen mit w eine Simulation von T durchlaufen lässt. ■

Diese Aussage ist für das Verständnis sehr wichtig, weil sie die Sprachen aus **NP** beschreibt, ohne das Konzept des Nichtdeterminismus zu verwenden. Die Probleme in **NP** sind vereinfacht ausgedrückt die, die evtl. schwer zu lösen sind, bei denen man die Lösungen aber vergleichsweise leicht überprüfen kann. Außerdem kann man diese Charakterisierung auch verwenden, um zu beweisen, dass man es mit einem **NP**-Problem zu tun hat: Wenn man zeigen kann, dass ein Problem in Polynomialzeit verifizierbar ist, muss man keinen nichtdeterministischen Algorithmus mehr angeben, der es in Polynomialzeit löst.

20. NP-vollständige Probleme

In der Komplexitätstheorie hat sich ein Merkmal dafür herauskristallisiert, wann ein Problem als schwer anzusehen ist. Hier erkennt man endlich den Sinn des Nichtdeterminismus in diesem Zusammenhang.

Definition

Ein Problem L wird **NP-schwer** genannt,¹⁸ wenn $L' \leq_p L$ für alle $L' \in \mathbf{NP}$ gilt. L wird **NP-vollständig** genannt, wenn L **NP-schwer** ist und zur Klasse **NP** gehört.

Diese Definition klingt zunächst so, als wäre es unmöglich, ein **NP-vollständiges** Problem zu finden. Es muss sich um eine Art „Superproblem“ handeln, weil man *alle* Probleme aus **NP** auf dieses eine polynomial reduzieren kann. Es ist damit mindestens so schwer wie die restlichen Probleme in **NP**. Wie soll das gehen? Es geht aber. Das erste Problem dieser Art wurde Anfang der 1970er Jahre von [Stephen Cook](#) und [Leonid Levin](#) nachgewiesen.

Satz 20.1

Satz von Cook-Levin
SAT ist **NP-vollständig**.

Beweis. Dass **SAT** zu **NP** gehört, wissen wir schon, weil wir uns im letzten Abschnitt überlegt haben, dass dieses Problem in Polynomialzeit verifizierbar ist. Wir gehen nun von einem beliebigen Entscheidungsproblem $L \in \mathbf{NP}$ aus und müssen zeigen, dass und wie man es auf **SAT** polynomial reduzieren kann. Die Begründung dafür wird im Folgenden grob skizziert.

Nach Voraussetzung gibt es eine (nichtdeterministische) Turingmaschine T , die L in Polynomialzeit entscheidet. Wir werden ein Verfahren angeben, dass zu jeder möglichen Eingabe w eine aussagenlogische Formel φ_w konstruiert, die genau

¹⁸Im Englischen **NP-hard**, was manchmal falsch als „**NP**-hart“ übersetzt wird.

dann erfüllbar ist, wenn w zu L gehört, wenn w also zu einer positiven Antwort von T führt. Das werden wir dadurch erreichen, dass die Formel den entsprechenden Programmablauf von T widerspiegelt. „Nebenbei“ müssen wir darauf achten, dass die Formel nicht zu groß wird, dass ihre Länge also in Relation zur Länge der Eingabe polynomial beschränkt ist.

Als Vorbereitung überlegen wir uns, dass es zu m aussagenlogischen Variablen x_1 bis x_m immer eine Formel $Q(x_1, \dots, x_m)$ gibt, die dann und nur dann erfüllt ist, wenn *genau eine* der Variablen wahr ist. Das ist leicht. Für $m = 4$ sieht es beispielsweise so aus:

$$\begin{aligned} & (x_1 \vee x_2 \vee x_3 \vee x_4) \\ & \wedge (x_1 \Rightarrow \neg(x_2 \vee x_3 \vee x_4)) \wedge (x_2 \Rightarrow \neg(x_1 \vee x_3 \vee x_4)) \\ & \wedge (x_3 \Rightarrow \neg(x_1 \vee x_2 \vee x_4)) \wedge (x_4 \Rightarrow \neg(x_1 \vee x_2 \vee x_3)) \end{aligned} \quad (20.1)$$

Und offensichtlich ist die Länge von $Q(x_1, \dots, x_m)$ von der Ordnung $\mathcal{O}(m^2)$.

Nun kommt der steinige Weg zur besagten Formel. Für T gibt es ein Polynom p derart, dass es bei einer Eingabe der Länge n maximal $p(n)$ Rechenschritte gibt. Für so eine Eingabe kann ein kompletter Programmablauf mit einer Tabelle wie der folgenden beschrieben werden.

Schritt	Position	Zustand	Zelle 1	Zelle 2	...	Zelle $p(n)$
1	1	1	a_1	a_2	...	$\square$
2	?	?	?	?	...	?
3	?	?	?	?	...	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$p(n)$	?	h	1	$\square$	...	$\square$

Der Programmablauf muss dabei nicht exakt $p(n)$ Schritte benötigen; man kann z. B. vereinbaren, dass die Zeile, bei der der Ablauf terminiert, wiederholt wird, bis $p(n)$ Zeilen gefüllt sind.

Nun führen wir Aussagenvariablen $s_{k,m}$ ein, die genau dann wahr sein sollen, wenn der Schreib-Lese-Kopf im k -ten Rechenschritt über der m -ten Zelle steht. Das sind insgesamt $p(n)^2$ Variablen. Für jede Zeile k bilden wir die Formel $Q(s_{k,1}, \dots, s_{k,p(n)})$, die besagt, dass der Kopf pro Schritt nur an genau einer Stelle stehen kann.

Analog führen wir Variablen $z_{k,q}$ ein, die dafür stehen, dass T im k -ten Schritt im q -ten Zustand ist. Und Variablen $b_{k,m,r}$, deren Bedeutung sein soll, dass im k -ten Schritt in der m -ten Zelle das r -te Zeichen des Bandalphabets steht.¹⁹ Auch für diese Variablen bilden wir wie oben Formeln, die aussagen, dass sich T in jedem Schritt in genau einem Zustand befinden und dass in jedem Schritt in jeder Zelle genau ein Zeichen stehen kann.

¹⁹Die endlich vielen Zustände sowie die Symbole des Bandalphabets werden entsprechend durchnummeriert. Deren Anzahl hängt nur von T und nicht von der Eingabe ab.

Außerdem brauchen wir noch Formeln, die den legalen Ablauf eines Programms gemäß der Übergangsfunktion der Turingmaschine beschreiben. Das könnte so etwas sein wie:

Wenn im 42. Schritt der Schreib-Lese-Kopf auf der 17. Zelle steht, sich dort das 23. Zeichen auf dem Band befindet und die Maschine im fünften Zustand ist, dann muss sich der Kopf im 43. Schritt auf der 18. Zelle befinden, in der 17. Zelle muss das 12. Zeichen stehen und der Zustand ist nun der dritte.

In eine Formel übersetzt würde das folgendermaßen aussehen:

$$s_{42,17} \wedge b_{42,17,23} \wedge z_{42,5} \Rightarrow s_{43,18} \wedge b_{43,17,12} \wedge z_{43,3} \quad (20.2)$$

Das ist offensichtlich möglich, wobei allerdings zu bedenken ist, dass wir solche Formeln für *jede* Zeile der Tabelle, *jede* Zelle und *jeden* Übergang benötigen und dass die Formeln für nichtdeterministische Übergänge noch etwas länger werden. Nichtsdestotrotz kann man zeigen, dass alles „im polynomialen Rahmen“ bleibt.

Schließlich brauchen wir eine Formel, die für die erste Zeile codiert, welches Eingabewort sich auf dem Band befindet, und eine, die für die letzte Zeile codiert, dass die Maschine mit einer positiven Antwort terminierte. (Letzteres wird in der Tabelle oben exemplarisch durch den Zustand h wie *halt* und den Bandinhalt $1 \square \dots \square$ symbolisiert.) φ_w ist nun die **Konjunktion** all der Formeln, die in den letzten Absätzen vorkamen.

Im Prinzip war's das, aber weil der Beweis recht lang war, sei der wesentliche Punkt noch einmal wiederholt. Die polynomiale Reduktion besteht darin, für ein vorgegebenes Eingabewort w mit der eben beschriebenen Methode eine aussagenlogische Formel φ_w zu konstruieren, die genau dann erfüllbar ist, wenn w zu L gehört. Die Konstruktion von φ_w kann von einer deterministischen Turingmaschine in polynomialer Zeit erledigt werden, wenn der „Bauplan“ von T bekannt ist. φ_w kann anschließend an eine Turingmaschine übergeben werden, die **SAT** löst. ■

Aufgabe 20.1. Überzeugen Sie sich, dass die Länge der im Beweis von Satz 20.1 konstruierten Formel φ_w wirklich polynomial in $|w|$ ist.

Definition

In der Aussagenlogik bezeichnet man eine Aussagenvariable oder deren **Negation** als **Literal**. Eine **Disjunktion** von Literalen nennt man eine **Klausel**. Und eine Formel ist in **konjunktiver Normalform**, wenn sie eine **Konjunktion** von Klauseln ist. Beim Problem **CNF-SAT** geht es darum, für Formeln in konjunktiver Normalform zu entscheiden, ob sie erfüllbar sind.

Man könnte denken, **CNF-SAT** sei leichter als **SAT**. Dem ist jedoch nicht so.

Korollar 20.2

CNF-SAT ist NP-vollständig.

Beweis. Man kann die im Beweis des Satzes von Cook und Levin vorkommenden Formeln gleich in konjunktiver Normalform aufschreiben. Siehe Aufgabe 20.2. ■

★**Aufgabe 20.2.** Geben Sie zu (20.1) bzw. zu (20.2) äquivalente Formeln in konjunktiver Normalform an.

Nachdem man erst einmal ein **NP**-vollständiges Problem hat, wird es leichter, weitere zu finden. Wenn nämlich L_1 **NP**-vollständig ist und man für ein Problem $L_2 \in \mathbf{NP}$ zeigen kann, dass $L_1 \leq_p L_2$ gilt, dann gilt offenbar $L' \leq_p L_1 \leq_p L_2$ für jedes Problem L' aus **NP**.

CLIQUE ist **NP**-vollständig.

Satz 20.3

Beweis. Wir werden **CNF-SAT** polynomial auf **CLIQUE** reduzieren. Wir brauchen also eine Methode, eine Formel in konjunktiver Normalform so in einen Graphen umzuwandeln, dass ein Zusammenhang zwischen Cliquen in dem Graphen und der Erfüllbarkeit der Formel besteht. Dafür nummerieren wir die Klauseln der Formel durch und gehen o. B. d. A. davon aus, dass die beteiligten Aussagenvariablen in der Form x_1, x_2 und so weiter vorliegen. Als Knotenmenge definieren wir

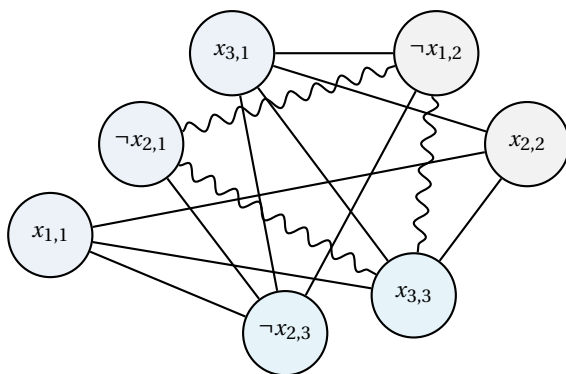
$$V = \{x_{k,m} : x_k \text{ kommt in der } m\text{-ten Klausel vor}\} \\ \cup \{\neg x_{k,m} : \neg x_k \text{ kommt in der } m\text{-ten Klausel vor}\}.$$

Zwei Ecken werden dann mit einer Kante verbunden, wenn sie aus verschiedenen Klauseln sind und nicht eine von beiden die Negation der anderen repräsentiert.

Die Konstruktion wird hier am Beispiel der Formel

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_3)$$

vorgeführt. Sie führt zu dem folgenden Graphen:



Verbunden sind z. B. $x_{3,1}$ und $x_{3,3}$ sowie $x_{3,1}$ und $\neg x_{1,2}$, weil es sich in beiden Fällen um verschiedene Klauseln handelt. Nicht verbunden sind u. a. $x_{1,1}$ und $x_{3,1}$ (dieselbe Klausel) sowie $\neg x_{2,1}$ und $x_{2,2}$ (verschiedene Klauseln, aber die Knoten repräsentieren $\neg x_2$ und x_2).

Man kann sich nun bei einer Formel mit n Klauseln leicht überlegen:

- Wenn es eine Clique mit n Knoten gibt, dann muss aus jeder Klausel genau ein Knoten beteiligt sein, weil Knoten aus derselben Klausel nicht verbunden sind. (Ein Beispiel für so eine Clique ist in der obigen Skizze hervorgehoben.)
- Weil kein Knoten mit „seiner“ Negation verbunden ist, erhält man eine erfüllende Belegung für die Formel, wenn die Literale, die zu den Knoten der Clique gehören, wahr sind.
- Umgekehrt liefert jede erfüllende Belegung eine Clique mit n Knoten.

Man kann den Graphen also an eine Turingmaschine übergeben, die **CLIQUE** löst. Damit hat man ein Verfahren, um **CNF-SAT** zu lösen. ■

Korollar 20.4

INDSET ist **NP**-vollständig.

Beweis. Algorithmus 19.5 eignet sich offenbar nicht nur, um $\text{INDSET} \leq_p \text{CLIQUE}$ zu zeigen, sondern man kann ihn auch für $\text{CLIQUE} \leq_p \text{INDSET}$ verwenden. ■



★**Aufgabe 20.3.** Dass **CNF-SAT** **NP**-vollständig ist, kann man auch anders als im Beweis von Korollar 20.2 begründen: indem man nämlich eine polynomiale Reduktion von **SAT** auf **CNF-SAT** angibt. Man kann dafür jedoch nicht die evtl. aus den Informatikvorlesungen [bekannte Methode](#) zur Konstruktion einer konjunktiven Normalform benutzen, weil die im *worst case* zu exponentiellem Wachstum der Formel führen könnte. Da man aufgrund der Formulierung der beiden Probleme nicht an [Äquivalenz](#), sondern nur an [Erfüllbarkeitsäquivalenz](#) interessiert ist, kann man jedoch die sogenannte [Zeitin-Transformation](#) verwenden, die in polynomialer Zeit arbeitet und die in dem am Rande verlinkten Video erklärt wird. Führen Sie für (20.1) und (20.2) Zeitin-Transformationen durch.

★**Aufgabe 20.4.** **3-SAT** ist eine Variante von **CNF-SAT**, in der jede Klausel aus maximal drei Literalen besteht. Zeigen Sie, dass **3-SAT** **NP**-vollständig ist. Eine mögliche polynomiale Reduktion wird im [Wikipedia-Artikel über Erfüllbarkeitsäquivalenz](#) beschrieben.

Und so kann man weitermachen. **SAT** war das Problem, das den Stein quasi ins Rollen brachte. Inzwischen sind außer **CNF-SAT**, **CLIQUE** und **INDSET** noch mehrere Tausend weitere **NP**-vollständige Probleme bekannt! [Eine unvollständige Liste findet man auf Wikipedia](#). Was bedeutet das?

- Die Klasse der **NP**-vollständigen Probleme bildet eine Teilklasse von **NP**. Es handelt sich um die in Bezug auf die Laufzeitkomplexität schwierigsten Probleme in **NP**.
- Würde man für ein einziges **NP**-vollständiges Problem eine deterministische Lösung in polynomialer Laufzeit finden, dann hätte man auf einen Schlag für *alle* Probleme in **NP** eine deterministische Lösung in polynomialer Laufzeit. Und das würde bedeuten, dass **P** und **NP** identisch sind.
- Die große Mehrheit der Experten glaubt das jedoch nicht. Sie glaubt, dass die beiden Klassen nicht gleich sind und dass deswegen die Probleme in $\text{NP} \setminus \text{P}$ wirklich schwieriger sind als die in **P**.

Es gibt aber bisher (nach mehr als einem halben Jahrhundert Forschung) immer noch keinen Beweis für diese Hypothese. Es ist nach wie vor möglich, dass morgen jemand für **CLIQUE** oder ein anderes NP-vollständiges Problem einen deterministischen polynomialen Lösungsalgorithmus präsentiert. Die Frage, ob

$$\mathbf{P} = \mathbf{NP}$$

eine wahre oder falsche Aussage ist, ist dadurch inzwischen eines der bekanntesten ungelösten Probleme der Mathematik geworden. Seit dem Jahr 2000 ist sie eines der sieben **Millennium-Probleme**, für deren Lösung jeweils ein Preisgeld von einer Million Dollar ausgesetzt ist.

Da diese Frage inzwischen auch außerhalb der Fachkreise eine gewisse Berühmtheit erlangt hat, seien hier kurz einige Missverständnisse ausgeräumt:

- *NP-vollständige Probleme sind die schwierigsten bekannten Probleme.*

Das stimmt nicht. Alle Probleme aus **NP** haben deterministisch maximal exponentielle Laufzeit,²⁰ aber es gibt Probleme wie das Entscheidungsproblem für die **Presburger-Arithmetik**, von denen man weiß, dass sie *nicht* in exponentieller Laufzeit deterministisch lösbar sind. Außerdem gibt es ja auch Probleme, die beweisbar **überhaupt nicht lösbar** sind.

- *NP-vollständige Probleme sind deshalb schwierig, weil es so viele mögliche Lösungen gibt.*

Das stimmt nicht. Siehe dazu das Beispiel mit dem **Dijkstra-Algorithmus**. Umgekehrt kennt man auch Probleme, **die nur eine Lösung haben und trotzdem NP-vollständig sind**.

- *Eine deterministische Lösung eines NP-vollständigen Problems hat immer exponentielles Laufzeitverhalten.*

Das stimmt nicht. Erstens könnte ja nach aktuellem Stand **P = NP** wahr sein und zweitens gibt es NP-vollständige Probleme, für die bereits subexponentielle (aber immer noch superpolynomiale) Algorithmen bekannt sind. Ein Beispiel ist **eine Variante von INDSET für planare Graphen**.

- *Jede Instanz eines NP-vollständigen Problems ist schwierig.*

Das stimmt nicht. Beispielsweise sind alle Instanzen von **CNF-SAT**, bei denen **in jeder Klausel nur zwei Literale stehen**, in polynomialer Zeit lösbar.²¹ Diese Problem wird auch **2-SAT** genannt.

- *Quantencomputer können NP-vollständige Probleme effizient lösen.*

Das stimmt wahrscheinlich nicht, jedenfalls gehen die meisten Forscher momentan davon aus. Aber es gibt noch keine definitive Antwort. Auf jeden Fall gibt es aktuell und wohl auch auf absehbare Zeit noch keine Quantencomputer, die genügend **Qubits** haben und **fehlertolerant** genug sind, um in der Praxis eingesetzt zu werden.

²⁰Das kann man sich überlegen, indem man Aufgabe 19.12 verallgemeinert.

²¹Ersetzt man 2 durch 3, dann stimmt das nicht mehr. Siehe Aufgabe 20.4.

Sollte man bei **NP**-vollständigen Problemen also alle Hoffnung auf eine effiziente Lösung aufgeben, wenn man sich der Meinung der Experten anschließt und $P \neq NP$ glaubt? Nein. Erstens haben wir bereits gelernt, dass manchmal zumindest subexponentielle Lösungen existieren. Und zweitens haben wir die ganze Zeit über den *worst case* gesprochen und über Programme, die *immer korrekt antworten*. Es gibt viele Beispiele für Algorithmen, die entweder nicht immer, aber „oft genug“ in polynomialer Zeit richtig antworten oder die immer in polynomialer Zeit Antworten geben, die aber evtl. nur „fast“ richtig sind.²² Diverse Beispiele für solche Ansätze findet man in der am Anfang des Kapitels erwähnten [Playlist](#).

²²Das könnte bei einem Suchproblem z. B. ein Weg sein, der nicht der kürzeste ist, dessen Länge aber die der besten Lösung um höchstens 5 % überschreitet.

Lösungen zu ausgewählten Aufgaben

Lösung 1.1. Weil wir nur *endliche* Mengen als Alphabete zulassen.

Lösung 1.2. Die Antwort ist $|\Sigma|$, denn zu jedem Symbol $x \in \Sigma$ gibt es das Wort (x) der Länge 1.

Ich habe hier absichtlich noch einmal die „umständliche“ Schreibweise (x) verwendet. Rein technisch sind das Symbol x und das Wort (x) zwei verschiedene Objekte. Diesen Unterschied werden wir jedoch in der Regel ignorieren. Daher können wir uns auch erlauben, statt (x) einfach x zu schreiben.

Lösung 1.3. $a^4 = a \circ a^3 = a \circ a \circ a^2 = a \circ a \circ a \circ a = aaaa$. $v^3 = ababab$. $a^2 v^2 = aaabab$. $ab^3 = abbb$. $(ab)^3 = ababab$. (Beachten Sie, dass diese beiden Ausdrücke unterschiedlich sind.) $|v^{10}| = 10 \cdot |v| = 20$.

Lösung 1.4. $\circ$ ist nicht kommutativ. Beispielsweise sind $00 \circ 1 = 001$ und $1 \circ 00 = 100$ unterschiedliche Wörter. Das neutrale Element ist ε , d. h., es gilt offenbar $\varepsilon \circ w = w \circ \varepsilon = w$ für jedes Wort w .

Lösung 1.5. Nichts. $\circ$ ist zwar nicht kommutativ, aber offensichtlich gilt immer $w^k \circ w = w \circ w^k$.

Lösung 1.6. $X \circ Y = \{abbc, abd, abe, acbc, acd, ace\}$. In $Y \circ X$ ist aber z. B. das Wort $bcab$ enthalten, das nicht zu $X \circ Y$ gehört.

Lösung 1.7. $X \circ Y$ ist $\emptyset$, denn nach Definition besteht diese Menge aus allen Kombinationen der Form vw , wobei v ein Wort aus X und w eines aus Y ist. Da aber Y gar keine Wörter enthält, kann $X \circ Y$ auch keine enthalten.

In $X \circ Z$ sind nach Definition alle Wörter der Form $v\varepsilon$ mit $v \in X$ enthalten. Weil $v\varepsilon$ dasselbe wie v ist (siehe Aufgabe 1.4, ist $X \circ Z$ einfach X).

Lösung 1.8. Der Reihe nach:

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

$$\Sigma^1 = \{a, b\}$$

$$\Sigma^2 = \{aa, ab, ba, bb\}$$

$$\Sigma^3 = \{aaa, aba, baa, bba, aab, abb, bab, bbb\}$$

Lösung 1.9. Es ist die Menge aller Wörter der Länge k , die man mit Symbolen aus Σ bilden kann.

Lösung 1.10. Das ist eine Kombinatorik-Aufgabe, erstes Semester Mathematik. Die Antwort ist $|\Sigma|^k$.

Lösung 1.11. $X^1 = X^0 \circ X = \{\varepsilon\} \circ X = \{\varepsilon w : w \in X\} = \{w : w \in X\} = X$.

Lösung 1.12. Es gilt auf jeden Fall immer $|X \circ Y| \leq |X| \cdot |Y|$, denn bekanntlich ist $|X| \cdot |Y|$ die Mächtigkeit von $X \times Y$ und es gilt offenbar

$$X \circ Y = \{vw : (v, w) \in X \times Y\}.$$

Aber mit $X = Y = \{a, aa\}$ gilt $X \circ Y = \{aa, aaa, aaaa\}$, also $3 = |X \circ Y| < |X| \cdot |Y| = 4$.

Lösung 1.13. Es ist die Menge aller Wörter, die man mit Symbolen aus Σ bilden kann. Siehe Aufgabe 1.9.

Lösung 1.14. So geht es los:

$X^0 = \{\varepsilon\}$	$Y^0 = \{\varepsilon\}$
$X^1 = \{a\}$	$Y^1 = \{\varepsilon, a\}$
$X^2 = \{aa\}$	$Y^2 = \{\varepsilon, a, aa\}$
$X^3 = \{aaa\}$	$Y^3 = \{\varepsilon, a, aa, aaa\}$

Allgemein gilt offensichtlich $Y^k = X^0 \cup \dots \cup X^k$. X^* und Y^* sind dann wieder identisch $X^* = Y^* = \{a^k : k \in \mathbb{N}\}$. Aber: $Y^+ = Y^* \neq \{a^k : k \in \mathbb{N}^+\} = X^+$.

Lösung 1.15. Es gilt grundsätzlich nicht, wenn X das leere Wort enthält. Das hätte Ihnen beim Lösen von Aufgabe 1.14 auffallen können. Das einfachste Beispiel ist $X = \{\varepsilon\}$. Dann gilt nämlich $X^k = X$ für alle $k \in \mathbb{N}$ und damit $X^* = X^+ = X$, aber $X^* \setminus \{\varepsilon\} = \emptyset$.

Lösung 1.16. Das war eine Fangfrage, die auf einem häufigen Fehler basiert. Sowohl die kleenesche als auch die positive Hülle sind nur für *Mengen* von Wörtern (oder Zeichen) definiert und nicht für einzelne Wörter. Ist w ein Wort, dann sind die Ausdrücke w^* und w^+ weder definiert noch ergeben sie einen Sinn.

Lösung 1.17. Die Frage steht nur deswegen im Skript, weil ich weiß, dass es an dieser Stelle oft Missverständnisse gibt. Es gibt in Σ^* kein längstes Wort! Zu jedem Wort $w \in \Sigma^*$ ist wx ein noch längeres Wort, wenn x irgendein Symbol aus X ist. Beachten Sie außerdem, dass Wörter nach Definition (weil sie Tupel sind) immer *endliche* Länge haben.

Lösung 1.18. Wenn es für eine Sprache L über dem Alphabet Σ ein k gibt, so dass alle Wörter aus L maximal die Länge k haben, dann enthält L insgesamt höchstens $\sum_{i=0}^k |\Sigma|^i$ Wörter (siehe Aufgabe 1.10) und ist damit endlich.

Lösung 1.19. Offensichtlich kann man mit solchen Aufgaben Schwierigkeiten bekommen, wenn man die Mengenlehre aus dem ersten Semester vergessen hat ...

Die Paare sind $A_1 = B_3$, $A_2 = B_5$, $A_3 = B_2$, $A_4 = B_4$ und $A_5 = B_1$. Als Hilfe zum Verständnis hier noch einmal die fünf Mengen in informeller (!) Schreibweise:

$$\begin{aligned} A_1 &= \{\epsilon, 00, 0000, 000000, \dots\} \\ A_2 &= \{0, 000, 00000, 0000000, \dots\} \\ A_3 &= \{00, 000, 0000, 00000, \dots\} \\ A_4 &= \{0, 00, 000, 0000, \dots\} \\ A_5 &= \{0, 00, 000, \dots\} \cup \{1, 11, 111, \dots\} \end{aligned}$$

Wenn Ihnen nicht klar ist, warum das stimmt, fragen Sie nach!

Lösung 1.20. Man könnte es so machen:

$$\begin{aligned} A_1 &= \{w \in \Sigma_{\text{bool}}^* : |w| \bmod 3 = 2\} = \{0^{3n+2} : n \in \mathbb{N}\} \\ A_2 &= \{w \in \Sigma_{\text{bool}}^* : |w| \leq 3\} \\ A_3 &= \{w \in \Sigma_{\text{bool}}^* : |w| = 3\} \\ A_4 &= \{0^m 1^n : m, n \in \mathbb{N}^+\} \end{aligned}$$

Es kann aber natürlich auch anders aussehen. Wichtig ist, dass Sie die richtige Schreibweise verwenden, die Sie im ersten Semester gelernt haben.

Beachten Sie den Unterschied zwischen A_2 und A_3 .

Lösung 1.21. Die Paare sind L_1 und L_7 , L_2 und L_8 , L_3 und L_5 sowie L_4 und L_6 .

Beachten Sie, dass aus $|w|_1 = |w| - 1$ folgt, dass w genau eine Null enthält.

Lösung 1.22. $\{0\}^+ \circ \{1\}^+$ ist eine Menge von Wörtern, also eine Sprache. $\{\{0\}^+ \circ \{1\}^+\}$ ist daher eine Menge von Sprachen und keine Sprache. Daher ist der Ausdruck aus der Aufgabenstellung gar nicht definiert. Beachten Sie, dass A und $\{A\}$ immer unterschiedliche Objekte sind – unabhängig davon, was A ist. Das sollte seit dem ersten Semester klar sein.²³

Gemeint war wahrscheinlich $(\{0\}^+ \circ \{1\}^+)^3$. Das ergibt Sinn und beschreibt kurz und knapp die Sprache

$$\{0^{n_1} 1^{n_2} 0^{n_3} 1^{n_4} 0^{n_5} 1^{n_6} : n_1, n_2, \dots, n_6 \in \mathbb{N}^+\}.$$

Lösung 1.23. Grundsätzlich wird es immer mehr als eine richtige Antwort geben. Meine Lösungen sind nur als Beispiele zu verstehen. Wenn Sie eine andere Lösung haben, kann die also trotzdem richtig sein. (Falls Sie sich nicht sicher sind, ob Ihre Lösung korrekt ist, fragen Sie! Und stellen Sie solche Fragen nicht erst zwei Wochen vor der Klausur ...)

²³Falls Sie so etwas übertrieben kleinlich finden, dann erinnern Sie sich bitte daran, dass Sie Informatik studieren und welchen Unterschied scheinbar „unbedeutende Kleinigkeiten“ wie beispielsweise ein fehlendes Semikolon beim Programmieren ausmachen können.

- (i) Offensichtlich ist wichtig, dass die Sprachen verschieden sein müssen, weil die Aufgabe sonst **trivial** wäre. Und ohne die Forderung der Unendlichkeit wäre es auch sehr einfach; man könnte dann z. B. $L_1 = \{\varepsilon\}$ und $L_2 = \{0\}$ nehmen oder $L_1 = \emptyset$ mit irgendeiner anderen Sprache L_2 (siehe dazu auch Aufgabe 1.7) oder $L_1 = \{1\}$ und $L_2 = \{11\}$.
Eine mögliche Lösung wäre $L_1 = \Sigma_{\text{bool}}^*$ und $L_2 = \Sigma_{\text{bool}}^+$. Dann erhält man $L_1 \circ L_2 = L_2 \circ L_1 = \Sigma_{\text{bool}}^+$. Ein anderes Beispiel wäre $L_1 = \{00\}^*$ und $L_2 = \{0\} \circ L_1$. Damit ergibt sich $L_1 \circ L_2 = L_2 \circ L_1 = L_2$.
- (ii) Die wohl einfachste Idee ist, zu L_2 etwas hinzuzufügen, das L_2^+ nicht „größer“ macht. Wählt man z. B. $L_2 = \{0\}$, dann gehört 00 zu L_2^+ . Man kann dann $L_1 = \{0, 00\}$ nehmen und erhält $L_1^+ = L_2^+$.
Der „Witz“ an dieser Aufgabe ist, dass umgekehrt aus $L_1 \subseteq L_2$ immer $L_1^+ \subseteq L_2^+$ folgt. (Ist Ihnen das klar?) Die Aufgabe zeigt, dass es sich um eine **hinreichende, aber keine notwendige Bedingung** handelt.
- (iii) Hier ist es die letzte Bedingung, die die Sache etwas erschwert, denn sonst könnte man einfach $L_1 = \Sigma_{\text{bool}}^*$ nehmen. Eine mögliche Lösung könnte so aussehen, dass man für L_2 die „neutrale“ Menge $\{\varepsilon\}$ nimmt und für L_1 eine Sprache mit $L_1 = L_1^*$, z. B. $L_1 = \{00\}^*$. „Umgekehrt“ geht es aber z. B. auch: Man nehme $L_1 = \{\varepsilon, 0\}$ und $L_2 = L_1^*$.
- (iv) Auch verwehrt die Unendlichkeit uns die einfachsten Lösungen. Man hätte sonst so etwas wie $L_1 = \{\varepsilon\}$ und $L_2 = \{0\}$ nehmen können. Aber richtig schwer ist es trotzdem nicht. Beispielsweise funktioniert es mit $L_1 = \{00\}^*$ und $L_2 = \{0\} \circ L_1$.
- (v) Man kann zeigen, dass $L_1 \circ (L_2 \cap L_3) \subseteq (L_1 \circ L_2) \cap (L_1 \circ L_3)$ für beliebige Sprachen L_1, L_2 und L_3 gilt. (Und $L_1 \circ (L_2 \cup L_3) = (L_1 \circ L_2) \cup (L_1 \circ L_3)$ gilt ebenfalls immer.) Die umgekehrte Inklusion gilt aber im Allgemeinen nicht. Ist z. B. $L_2 = \{0\}$ und $L_3 = \{\varepsilon\}$, so ist $L_2 \cap L_3 = \emptyset$, und damit ist $L_1 \circ (L_2 \cap L_3)$ unabhängig von L_1 auf jeden Fall die leere Menge. Wählt man aber z. B. $L_1 = \{1, 10\}$, so ist $L_1 \circ L_2 = \{10, 100\}$ und $L_1 \circ L_3 = \{1, 10\}$, der Schnitt der beiden letzten Mengen also $\{10\}$.

Lösung 1.24. Weil man mit führenden Nullen jede Zahl auf unendlich viele Arten binär darstellen kann. Die Sprache würde sonst unter anderem die Wörter 11, 011, 0011, 00011 und so weiter enthalten und damit trivialerweise unendlich sein.

Lösung 1.25. 1 und 10 sind Elemente von B_1 , 01 und 010 sind Elemente von B_0 , alle vier Wörter sind demnach Elemente von B und also auch von D_0 . ($1 \vee 10$) gehört damit auch zu D_0 und deshalb ist das gesamte Wort ein Element von D_2 und von D . Interpretiert handelt es sich um die Formel $\neg x_1 \wedge \neg x_2 \wedge (x_1 \vee x_2)$, die **nicht** erfüllbar ist (wovon man sich mithilfe einer **Wahrheitstafel** leicht überzeugt).

Lösung 2.1. Bedingung (iv) der Definition verbietet Produktionen, deren erster Komponente aus T^* ist. Anders ausgedrückt muss die erste Komponente mindestens ein Nichtterminalsymbol enthalten. Abgesehen davon können aber in beiden Komponenten beliebige Wörter stehen, die man mit den Zeichen des Vokabulars bilden kann.

Lösung 2.2. Nein, natürlich nicht. Das leere Wort gehört zu T^* , weil das leere Wort immer zu X^* gehört, wenn X eine Menge von Wörtern oder Symbolen ist. Und die erste Komponente darf kein Element von T^* sein.

Lösung 2.3. Man kann direkt aus X die Wörter ε und a herleiten, es gilt also $X \Rightarrow_G \varepsilon$ und damit $X \Rightarrow_G^* \varepsilon$ und dasselbe für a . Damit sind diese beiden Wörter schon einmal Elemente von $L(G)$. Nach einigem Nachdenken wird dann klar, dass die beiden anderen Produktionen überflüssig sind, weil man aus X kein Wort herleiten kann, in dem Xa oder Y vorkommt. Also folgt $L(G) = \{\varepsilon, a\}$.

Lösung 2.4. Eine mögliche Lösung wäre eine Grammatik, in der das Startsymbol niemals auf der linken Seite einer Produktion auftaucht (obwohl das unserer Konvention widersprechen würde). Man könnte aber auch – ähnlich wie im Beispiel (2.1) – Produktionen aufschreiben, an denen sich sofort erkennen lässt, dass man aus dem Startsymbol kein nur aus Terminalsymbolen bestehendes Wort herleiten kann, z. B. eine Grammatik, die aus den beiden Produktionen $S \rightarrow S$ und $T \rightarrow a$ besteht.

Lösung 2.5. Das könnte so aussehen:

$$S \Rightarrow_G SS \Rightarrow_G S[S] \Rightarrow_G S[[S]] \Rightarrow_G S[[]] \Rightarrow_G [S][[]] \Rightarrow_G [][[]]$$

Der Ableitungsbaum würde sich dadurch nicht ändern.

Lösung 2.6. Sie werden festgestellt haben, dass es gar nicht so einfach ist, das zu beschreiben, während man es mit einer Grammatik kurz und unmissverständlich hinkommt. Man könnte es vielleicht so ausdrücken: Ein korrekt geklammertes Wort hat die Eigenschaft, dass man so lange sukzessive Teilworte der Form $[]$ entfernen kann, bis nichts mehr übrig bleibt.

Lösung 2.7. Man könnte $[] [] []$ nehmen. Der Anfang der Herleitung wäre vielleicht $S \Rightarrow_G SS \Rightarrow_G SSS$, aber anschließend kann man sich aussuchen, ob man das erste oder das zweite S durch SS ersetzt.

Lösung 2.8. Die wohl einfachste Möglichkeit wäre eine Grammatik, die aus den beiden Produktionen $S \rightarrow \varepsilon \mid aS$ besteht.

Lösung 2.9. Es handelt sich um $\Sigma_{\text{bool}}^* \circ \{010\}$.

Lösung 2.10. Nein. Man kann unter anderem das Wort aa nicht herleiten. Die von dieser Grammatik erzeugte Sprache ist $\{(ab)^n : n \in \mathbb{N}\}$.

Lösung 2.13. Für die erste $S \rightarrow 1A \mid 0A$, $A \rightarrow \varepsilon \mid 1B$, $B \rightarrow 0C$, $C \rightarrow 1D$, $D \rightarrow 0A$, für die zweite $S \rightarrow \varepsilon \mid 0S \mid 1A$, $A \rightarrow 0A \mid 1B$, $B \rightarrow 0B \mid 1S$ und für die dritte klappt es mit $S \rightarrow \varepsilon \mid S0S0S1S \mid S0S1S0S \mid S1S0S0S$.

Um zu beweisen, dass die dritte Grammatik wirklich alle Wörter der Sprache erzeugt, greife man sich ein beliebiges Wort w heraus. Offenbar muss $|w|$ durch 3 teilbar sein, man kann das Wort also in Abschnitte à drei Zeichen unterteilen. Befindet sich in jedem dieser Abschnitte mindestens eine Eins, so können es nicht in jedem zwei Einsen sein. Mindestens einer hat also die Form 001, 010 oder 100.

Anderenfalls gibt es Abschnitte der Form 000. Da das Wort aber nicht nur aus solchen Teilworten bestehen kann, ergibt sich an der Grenze einer Sequenz aus Nullen eines der Muster 100 oder 001. In beiden Fällen kann man das ausgewählte Muster für die letzte Produktion nehmen, die nicht von der Form $S \rightarrow \varepsilon$ ist. Der Rest ergibt sich **induktiv**.

(Die ε -Produktionen werden ganz am Ende durchgeführt, so dass man vorher immer Wörter hat, bei denen zwischen zwei Terminalsymbolen jeweils ein S steht.)

Alternative: $S \rightarrow \varepsilon \mid S0SES \mid SES0S, E \rightarrow 0S1 \mid 1S0$

Lösung 2.14. Ich gebe hier keine vollständige Lösung an, biete aber noch einen Hinweis zum letzten Aufgabenteil. Wenn Sie in der ursprünglichen Grammatik die Zeichenkette $z=x+y*y$ ableiten, haben Sie verschiedene Möglichkeiten. Wie müsste man die Grammatik ändern, damit eine bestimmte Reihenfolge erzwungen wird, die im Prinzip der entspricht, die man auch bei der Herleitung von $z=x+(y*y)$ einhalten müsste?

Lösung 2.16. $S \rightarrow 0TS \mid A, T0 \rightarrow 0T, TA \rightarrow 1A0, T1 \rightarrow 1T, A \rightarrow \varepsilon$. Eine Alternativlösung ist $S \rightarrow \varepsilon \mid T, T \rightarrow 0TAB \mid 0CB, BA \rightarrow AB, CA \rightarrow 1C, CB \rightarrow 10, 0B \rightarrow 00$.

Lösung 2.17. Es gibt $3 \cdot 3 \cdot 4 \cdot 4 \cdot 4 = 576$ Möglichkeiten. Da das Substantiv den Artikel festlegt, sind davon nur $576/3 = 192$ grammatisch korrekt.

Lösung 4.1. Es handelt sich um $\{abb\}^* = \{(abb)^n : n \in \mathbb{N}\}$.

Lösung 4.3. Das war eine Fangfrage! Haben Sie mit *nein* geantwortet? Die Antwort ist zwar richtig, aber Sie können das eigentlich noch nicht wissen, wenn der Stoff neu für Sie ist. Darum ist vermutlich Ihre Begründung falsch. Es ist richtig, dass die *Grammatik* (2.2) nicht regulär ist, aber das heißt noch lange nicht, dass die von ihr erzeugte Sprache nicht regulär ist. Es könnte ja sein, dass es eine *andere* Grammatik gibt, die regulär ist und die auch D_1 erzeugt. (Die gibt es nicht, aber das muss man erst einmal beweisen.)

Lösung 4.4. Man kann ein neues, bisher nicht genutztes Nichtterminalsymbol U einführen und die Produktion $T \rightarrow c$ durch die beiden Produktionen $T \rightarrow cU$ und $U \rightarrow \varepsilon$ ersetzen.

Lösung 4.5. Man kann ein neues, bisher nicht genutztes Nichtterminalsymbol U einführen und die Produktion $T \rightarrow abS$ durch die beiden Produktionen $T \rightarrow aU$ und $U \rightarrow bS$ ersetzen.

Lösung 4.6. Wenn die Sprache aus den Wörtern w_1 bis w_n besteht, dann verwenden wir die Produktionen $T_i \rightarrow w_i U$ für $i = 1, \dots, n$ sowie zusätzlich $U \rightarrow \varepsilon$ und $S \rightarrow T_1 \mid \dots \mid T_n$. Offensichtlich erzeugt diese Grammatik die Sprache. Sie ist nicht regulär, aber das lässt sich – siehe Aufgabe 4.5 – leicht beheben.

Lösung 4.8. Wenn L regulär wäre, dann würde das Pumping-Lemma eine Zahl n mit den dort genannten Eigenschaften liefern. Für das Wort $1^n 0 1^n$ aus L müsste dabei der „aufpumpbare“ Teil v komplett in der linken Hälfte des Wortes liegen und damit ein Element von $\{1\}^*$ sein, also nur aus Einsen bestehen. Ersetzt man nun

v z. B. durch v^2 , so ist das dadurch entstehende Wort w von der Form $1^m 0 1^n$ mit $m > n$. Es gilt dann offenbar nicht mehr, dass das mittlere Zeichen von w eine Null ist, d. h., w gehört nicht zu L . Das ist ein Widerspruch, also kann L nicht regulär sein.

Lösung 4.10. Sei n der vom Lemma garantierte Wert und $p \in \mathbb{P}$ mit $p > n + 1$. Das Teilstück v , das man wiederholen darf, habe die Länge $k > 0$. Der Rest hat dann die Länge $p - k$ und wegen $k \leq |uv| \leq n < p - 1$ gilt $p - k > 1$. Man ersetze v durch v^{p-k} . Das so entstehende Wort hat die Länge $s = (p - k) + k \cdot (p - k) = (1 + k)(p - k)$. Als Produkt zweier Zahlen ist s keine Primzahl. (Der Faktor $p - k$ ist weder 1 noch p , also ist das Produkt nicht von der Form $1 \cdot p$.)

Lösung 5.1. Man muss nur in die Definition einsetzen und erhält

$$\delta^*(s, a) = \delta^*(s, a\varepsilon) = \delta^*(\delta(s, a), \varepsilon) = \delta(s, a).$$

Lösung 5.2. Zum einzigen akzeptierenden Zustand s_2 gelangt man nur über s_1 mit einem b und dorthin kommt man nur von s_0 mit einem b . Danach kann der Automat aber beliebig viele weitere Zeichen konsumieren, ohne s_2 zu verlassen. Es ergibt sich somit $\{b^2\} \circ \{a, b\}^*$ als die gesuchte Sprache.

Lösung 5.3. Die Übergangsfunktion sieht folgendermaßen aus:

	s_0	s_1	s_2	s_e
a	s_1	s_e	s_e	s_e
b	s_e	s_2	s_e	s_e
c	s_e	s_e	s_0	s_e

Die akzeptierte Sprache ist²⁴ $\{a\} \circ \{bca\}^*$.

Lösung 5.4. Mit $\{s_0, s_1\}$ ergibt sich

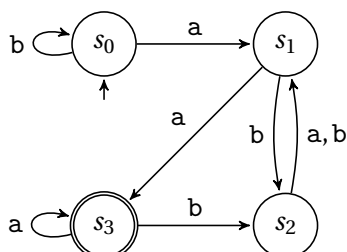
$$\{\varepsilon\} \cup \{a(bca)^m(bc)^n : m \in \mathbb{N} \wedge n \in \{0, 1\}\},$$

mit $\{s_0\}$ erhält man $\{abc\}^*$.

Lösung 5.5. Dafür muss man in der Grafik aus Aufgabe 5.3 nur c durch b ersetzen und $\{s_0\}$ zur Menge der akzeptierenden Zustände machen.

Lösung 5.6. Es handelt sich um die Sprache $\{w0 : w \in \Sigma_{\text{bool}}^*\}$.

Lösung 5.9. Man könnte es so machen:



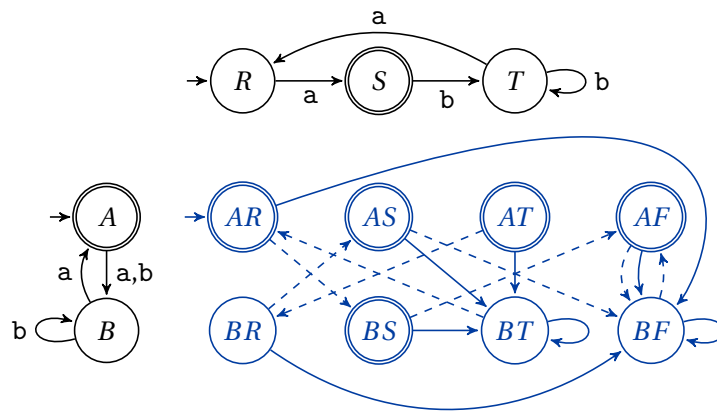
²⁴Natürlich gibt es immer mehr als eine Möglichkeit, so eine Menge aufzuschreiben. Ihre Antwort kann ruhig anders aussehen, sie muss nur dieselbe Menge beschreiben.

Lösung 5.11. Sei $A = (S, I, \delta, s_0, F)$. Da S nur ein Element hat, folgt $S = \{s_0\}$. δ ist auch eindeutig festgelegt – es muss $\delta(s_0, a) = s_0$ für alle $a \in I$ gelten. Für F gibt es genau zwei Möglichkeiten: $F = \emptyset$ oder $F = S$. Im ersten Fall ist $L(A) = \emptyset$, im zweiten gilt $L(A) = I^*$. Andere Sprachen kann so ein Automat nicht akzeptieren.

Lösung 5.13. Wenn man aus den entsprechenden Produktionen Übergänge macht, dann kann es passieren, dass von einem Zustand mehrere Pfeile mit demselben Symbol ausgehen, was nach unserer Definition nicht erlaubt ist. Wir werden es später doch so machen, aber dafür brauchen wir *nichtdeterministische Automaten*, die wir erst noch einführen müssen.

Lösung 5.14. Man muss lediglich die Menge der akzeptierenden Zustände ändern, indem man F durch das Komplement $S \setminus F$ ersetzt.

Lösung 5.15. Hier wird nur eine Skizze eines Produktautomaten gezeigt. Man spricht natürlich deshalb von *Produktautomaten*, weil es um das aus der Mathevorlesung bekannte *Mengenprodukt* geht, das man z. B. in Tabellenform darstellen kann. Im Folgenden ist der Produktautomat blau eingefärbt.



Namen wie AS stehen abkürzend für Tupel wie (A, S) . F steht für den im oberen Automaten nicht eingezeichneten „Fehlerzustand“. Der Übersichtlichkeit halber wurden die Beschriftungen der Pfeile weggelassen. Gestrichelte Pfeile stehen für a-Übergänge, durchgezogene für b-Übergänge.

Lösung 5.16. Mit simpler Mengenlehre hat man

$$I^* \setminus (L_1 \cap L_2) = (I^* \setminus L_1) \cup (I^* \setminus L_2)$$

und nach den Lemmata 5.2 und 5.3 gibt es einen Automaten für diese Sprache. Erneute Anwendung von Lemma 5.2 liefert einen Automaten für $L_1 \cap L_2$. (Alternativ kann man aber auch den Beweis von Lemma 5.3 modifizieren, indem man das Wörtchen *oder* durch *und* ersetzt.)

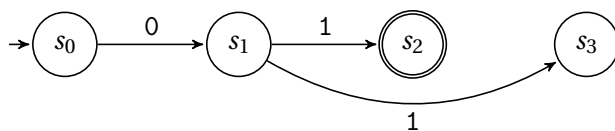
Auch für die Differenz kann man einfach mengentheoretisch argumentieren, nämlich mit $L_1 \setminus L_2 = L_1 \cap (I^* \setminus L_2)$.

Lösung 6.1. Das sieht so aus:

$$\delta^*({s}, a) = \delta^*({s}, a\epsilon) = \bigcup_{s' \in {s}} \delta^*(\delta(s', a), \epsilon) = \delta^*(\delta(s, a), \epsilon) = \delta(s, a)$$

Lösung 6.2. Es handelt sich um $\{a\}^+ \cup (\{ab\} \circ \{a\}^*)^+$.

Lösung 6.3. Das sollte wohl ungefähr so aussehen:



Lösung 6.4. Man definiert δ' durch $\delta'(s, a) = \{\delta(s, a)\}$. Das war's schon.

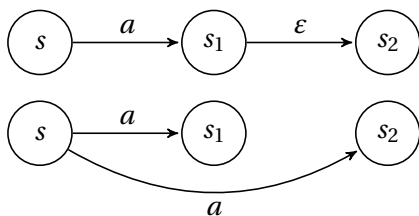
Lösung 6.5. Die wesentliche Idee ist: Ein Wort der Länge m muss vom Startzustand zu einem akzeptierenden Zustand m Pfeile und damit $m + 1$ Zustände passieren. Ist m mindestens so groß wie $|S|$, dann wird mindestens ein Zustand s auf diesem Weg mehrfach besucht. Die „Schleife“ von s nach s kann man sich sparen oder man kann sie auch mehrfach befahren.

Lösung 6.7. Der Algorithmus für den Umbau könnte wie folgt aus zwei Teilen bestehen:

Im ersten Teil werden alle Ketten von mindestens zwei verschiedenen Zuständen identifiziert, die durch ϵ -Übergänge direkt ineinander übergehen. Wir suchen also jeweils Zustände $s_1, s_2, \dots, s_n$ mit $n \geq 2$ und $s_{i+1} \in \delta(s_i, \epsilon)$ für alle $i < n$, so dass alle s_i paarweise verschieden sind. Für jede Kette dieser Art fügen wir – falls noch nicht vorhanden – einen neuen ϵ -Übergang von s_1 zu s_n hinzu.

(Beachten Sie, dass wir dies *nicht* für geschlossene Ketten – also für den Fall $s_1 = s_n$ – machen. Das ist auch nicht notwendig, weil sich durch so eine Folge von ϵ -Übergängen weder der Zustand des Automaten ändert noch ein Symbol konsumiert wird.)

Im zweiten Teil werden nun sukzessive alle ϵ -Übergänge entfernt und ggf. durch andere ersetzt: Gibt es Zustände s_1 und s_2 mit $s_2 \in \delta(s_1, \epsilon)$ und $s_1 \neq s_2$, so entferne s_2 aus $\delta(s_1, \epsilon)$ und füge für alle $(s, a) \in S \times I$ mit $s_1 \in \delta(s, a)$ den Zustand s_2 zu $\delta(s, a)$ hinzu. Falls s_1 ein Startzustand ist, so füge außerdem s_2 zur Menge der Startzustände hinzu.

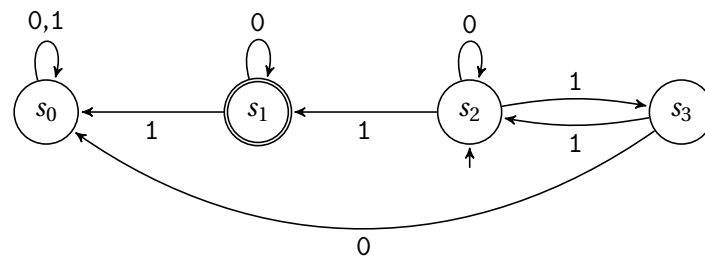


In jedem Schritt wird ein ϵ -Übergang entfernt und kein neuer hinzugefügt. Der Algorithmus **muss also abbrechen** und hinterlässt dann einen Automaten ohne

ε -Übergänge. Es sollte klar sein, dass der neue Automat dieselbe Sprache wie der alte akzeptiert.

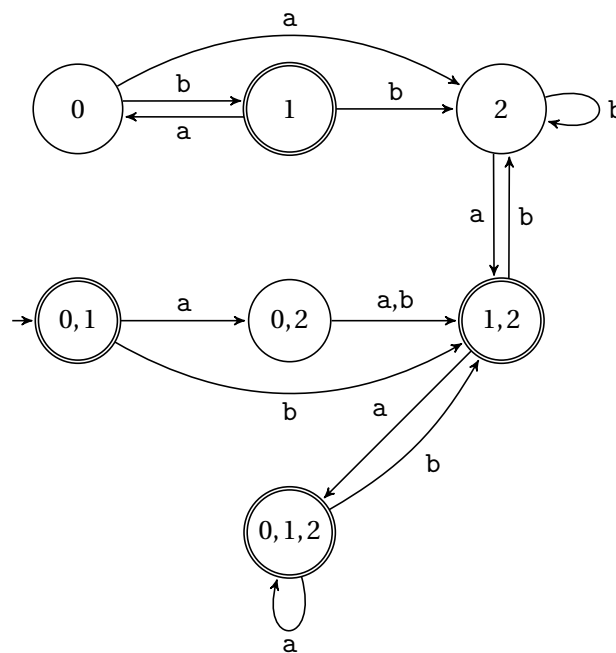
Lösung 6.9. Sie haben hoffentlich kein Problem damit, Automaten oder reguläre Grammatiken für $\{w \in \{0\}^* : 3 \mid |w|\}$ und $\{w \in \{0\}^* : 5 \mid |w|\}$ anzugeben. (Siehe z. B. Aufgabe 2.13.) Die Sprache L aus der Aufgabenstellung ist nach Lemma 5.3 als Vereinigung dieser beiden Sprachen dann auch regulär. Wenn es einen deterministischen endlichen Automaten A mit nur einem Endzustand s_F und $L(A) = L$ gäbe, dann müsste s_F offensichtlich gleichzeitig der Startzustand sein, weil ε zu L gehört. Weil 0^3 zu L gehört, muss es einen Weg von s_F zu s_F geben, der über drei 0-Pfeile führt. Ebenso muss es wegen $0^5 \in L$ einen solchen Weg über fünf 0-Pfeile geben. Führt man beide Wege nacheinander ab, erhält man jedoch einen Weg über acht 0-Pfeile von s_F nach s_F , obwohl 0^8 nicht zur Sprache gehört.

Lösung 6.11. Grafisch sieht A so aus:



Sei $L_1 = \{0^n 1^{2m} : m, n \in \mathbb{N}\}$ und $L_2 = \{10^n : n \in \mathbb{N}\}$. $L(A)$ ist dann $L_1^* \circ L_2$.

Lösung 6.12. Es sollte ungefähr so aussehen:



Lösung 6.13. Ist $A = (S, I, \delta, s_0, F)$ der (deterministische) Automat aus dem Lemma, dann konstruieren wir einen (nichtdeterministischen) Automaten A' für $L(A)^*$ folgendermaßen: Zur Zustandsmenge S fügen wir einen neuen Zustand s' hinzu, der der Startzustand von A' und gleichzeitig der einzige Endzustand wird. Das Eingabealphabet ändert sich natürlich nicht. Für jeden in A akzeptierenden Zustand fügen wir einen ε -Übergang von diesem zu s' hinzu. Und natürlich brauchen wir einen ε -Übergang von s' nach s_0 . Die Übergänge von A bleiben erhalten. Das war's bereits.

Falls Sie übrigens die naheliegende und einfacher klingende Idee hatten, von allen akzeptierenden Zuständen einen ε -Übergang zu s_0 hinzuzufügen und s_0 ebenfalls zum Endzustand zu machen: Das klappt oft, aber leider nicht immer. Die folgende Skizze zeigt ein Gegenbeispiel.



Wenn der Automat links A ist, dann ist $L(A)$ offenbar $\{1\}^* \circ \{0\}$. Der rechte Automat akzeptiert jedoch unter anderem das Wort 1, das nicht zu $L(A)^*$ gehört.

Lösung 6.14. Wie im Originalbeweis sei jeweils $A_i = (S_i, I, \delta_i, s_i, F_i)$ der Automat mit $L_i = L(A_i)$ und o. B. d. A. seien S_1 und S_2 disjunkt. Wir definieren einen nichtdeterministischen Automaten A für $L(A_1) \cup L(A_2)$, indem wir als Menge der Zustände $S_1 \cup S_2$ nehmen, als Menge der akzeptierenden Zustände $F_1 \cup F_2$ und als Menge (siehe Aufgabe 6.6) der Startzustände $\{s_1, s_2\}$. Für $s \in S_i$ und $a \in I$ setzen wir $\delta(s, a) = \{\delta_i(s, a)\}$. (Anders ausgedrückt zeichnen wir einfach A_1 und A_2 nebeneinander und deklarieren das Ergebnis als grafische Darstellung von A .)

Lösung 7.1. Nein, keiner von beiden. Die Aufgabe existiert nur, weil ich solche „regulären Ausdrücke“ oft als Lösungen in Klausuren sehe. Wenn Ihnen nicht klar ist, warum es sich nicht um reguläre Ausdrücke handelt, dann lesen Sie sich die Definition noch einmal genau durch.

Lösung 7.2. Die Grammatik sollte ungefähr so aussehen:

$$S \rightarrow 0 \mid 1 \mid \varepsilon \mid \emptyset \mid SS \mid (S + S) \mid (S)^*$$

Das einzige nichtterminale Symbol ist S . Die terminalen Symbole sind $0, 1, \varepsilon, \emptyset, +, *$ und die Klammern. Zu beachten ist, dass ε in dieser Grammatik *nicht* für das leere Wort steht, sondern ein Symbol des Alphabets ist. Zum Glück wird das leere Wort nicht benötigt; ansonsten müsste man dafür in diesem Fall ein anderes Symbol verwenden.²⁵

Reguläre Ausdrücke über Σ_{bool} sind also genau die Wörter, die sich mit dieser Grammatik herleiten lassen.

²⁵Zum Beispiel λ , siehe Fußnote 4 auf Seite 6.

Lösung 7.3. Es handelt sich um $\{11\}$, $\{11, 0\}$ und $\{11, 0\}^*$.

Lösung 7.4. Die erste Aussage ist wahr. Die zweite Aussage ist falsch, denn die Sprache $\mathcal{L}(0(0)^*) = \{0\} \circ \{0\}^*$ enthält z. B. das Wort 0^3 , das in der Sprache $\{00\}^*$ nicht enthalten ist. Da die dritte Aussage wahr ist, sind die vierte und fünfte falsch. Die letzte Aussage ist auch wahr, weil die regulären Ausdrücke $0 + 0^*$ und 0^* dieselbe Sprache beschreiben und weil in diesem Fall die Reihenfolge der Konkatination keine Rolle spielt, da alle Wörter nur aus Nullen bestehen.

Lösung 7.5. Die zweite Aussage ist falsch, weil die linke Sprache das Wort a enthält, die rechte aber nicht. Die dritte Aussage ist falsch, weil die rechte Sprache das Wort aa enthält, die linke aber nicht. Die fünfte Aussage ist falsch, weil die linke Sprache das Wort 0101 enthält, die rechte aber nicht. Die sechste Aussage ist falsch, weil die rechte Sprache das Wort 0 enthält, die linke aber nicht. Die siebte Aussage ist falsch, weil die linke Sprache das Wort 01 enthält, die rechte aber nicht. Die anderen Aussagen sind wahr.

Lösung 7.6. Hier drei Vorschläge:

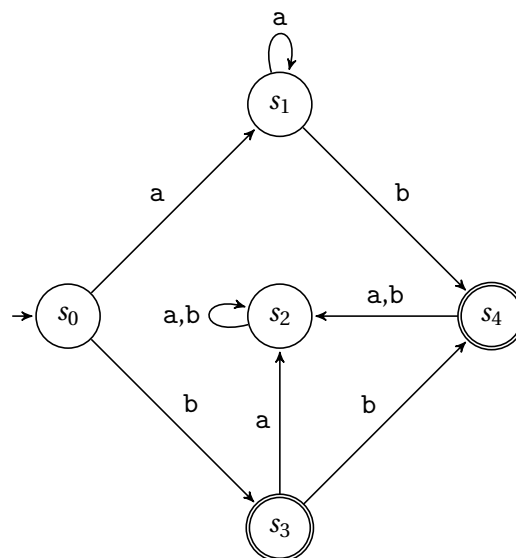
$$L_1 = \mathcal{L}((0 + 1)(0 + 1)(0 + 1))$$

$$L_2 = \mathcal{L}(0^*(10^*10^*)^*)$$

$$L_3 = \mathcal{L}(1 + (0 + 1)(0 + 1)(0 + 1)^*)$$

Es gibt natürlich jeweils noch andere Möglichkeiten, eine korrekte Lösung zu formulieren.

Lösung 7.7. Der Automat könnte z.B. so aussehen:



Beachten Sie, dass auch das Wort b akzeptiert werden muss!

Lösung 7.8. Da die regulären Ausdrücke $(\alpha + (\beta + \gamma))$ und $((\alpha + \beta) + \gamma)$ offenbar äquivalent sind, schreiben wir der Einfachheit halber $(\alpha + \beta + \gamma)$ (und ebenso für

„Summen“ von mehr als drei Ausdrücken). Wir benutzen die folgenden regulären Ausdrücke als Abkürzungen:

$$\kappa = (A + B + C + \dots + Z)$$

$$\lambda = (1 + 2 + \dots + 9)$$

$$\lambda_0 = (0 + \lambda)$$

$$\Lambda_3 = (\lambda + \lambda\lambda_0 + \lambda\lambda_0\lambda_0)$$

$$\Lambda_4 = (\Lambda_3 + \lambda\lambda_0\lambda_0\lambda_0)$$

Der reguläre Ausdruck für Kfz-Kennzeichen aus Regen könnte dann so aussehen:

$$\text{REG}-(\kappa\kappa\Lambda_3 + \kappa\Lambda_4)$$

Lösung 7.9. Der in Perl erlaubte reguläre Ausdruck $([01]^*) \setminus 1$ beschreibt z. B. die Sprache $\{ww : w \in \Sigma_{\text{bool}}^*\}$, von der man sich mittels des Pumping-Lemmas 4.1 leicht überzeugt, dass sie nicht regulär im Sinne unserer Definition ist.

Lösung 7.10. In PERL-Syntax sieht es etwas ökonomischer aus:

$$\text{REG}-([A-Z]\{2\} [1-9] \setminus \{0, 2\} | [A-Z] [1-9] \setminus \{0, 3\})$$

Lösung 8.1. 42 und 378 sind ziffernfremd, 378 und 26 auch. 42 und 26 sind aber nicht ziffernfremd.

Lösung 8.2. R_1 ist reflexiv auf Σ^* (man nehme $v = \varepsilon$), aber nicht symmetrisch, denn z. B. gilt aR_1ab , aber nicht abR_1a . R_1 ist transitiv, denn $w_1R_1w_2$ bedeutet offensichtlich, dass w_1 ein „Anfangsstück“ des Wortes w_2 ist. Und wenn w_1 ein Anfang von w_2 und w_2 ein Anfang von w_3 ist, dann ist natürlich w_1 ein Anfang von w_3 .

R_2 ist aus denselben Gründen transitiv, aber nicht symmetrisch. Allerdings ist R_2 nicht reflexiv auf Σ^* , denn offenbar kann nur $w_1R_2w_2$ gelten, wenn w_2 länger als w_1 ist.

R_3 ist symmetrisch, weil doppeltes Umkehren der Reihenfolge wieder das ursprüngliche Wort ergibt. R_3 ist jedoch nicht reflexiv auf Σ^* , denn z. B. gilt nicht abR_3ab . Transitiv ist R_3 auch nicht, denn wir haben beispielsweise abR_3ba und baR_3ab , woraus abR_3ab folgen müsste.

R_4 ist offensichtlich symmetrisch und transitiv. R_4 ist zudem reflexiv auf Σ^+ . Das war aber nicht die Frage! R_4 ist *nicht* reflexiv auf Σ^* , weil $\varepsilon R_4\varepsilon$ nicht gilt. (Bei diesem Beispiel ging es in erster Linie darum, dass der Begriff *reflexiv* nur Sinn ergibt, wenn klar ist, welche Grundmenge gemeint ist. Für Symmetrie und Transitivität ist das nicht nötig.)

Lösung 8.5. Keine. R_4 ist allerdings eine Äquivalenzrelation auf Σ^+ .

Lösung 8.6. Nein. Die Relation ist zwar reflexiv auf Σ_{bool}^* und symmetrisch, aber nicht transitiv: Es gilt z. B. $01R011$ und $011R0011$, aber nicht $01R0011$.

Lösung 8.7. Nein. R ist zwar reflexiv auf Σ_{bool}^* und symmetrisch, aber nicht transitiv: Zum Beispiel gilt $\varepsilon R 0$ und $0 R 00$, aber nicht $\varepsilon R 00$.

Lösung 8.8. Zu $A \times A$ gibt es nur eine Äquivalenzklasse, nämlich die Menge A . Die Äquivalenzklassen der Identität haben hingegen alle nur ein Element. Es gibt also zu jedem Element von A eine eigene Äquivalenzklasse, die nur aus diesem Element besteht, d. h., für alle $a \in A$ gilt $[a]_{\text{id}_A} = \{a\}$.

Lösung 8.9. Es gibt drei Äquivalenzklassen, nämlich $[\varepsilon]_R = \{\varepsilon\}$, $[0]_R = [1]_R = \{0, 1\}$ und $\{00, 01, 10, 11\}$.

Lösung 8.10. Für den ersten Teil der Aufgabe kann man R aus (8.2) mit der unendlichen Grundmenge $\mathbb{Z}$ nehmen. Für den zweiten Fall eignet sich jede unendliche Menge A mit der Äquivalenzrelation id_A .

Lösung 9.2. Für die drei Wörter ε , a und ab gibt es jeweils genau eine Möglichkeit, aus ihnen durch Konkatenation ein L -Wort zu machen, und diese drei Möglichkeiten sind paarweise verschieden. Alle anderen Wörter sind nicht der Anfang eines L -Wortes. Es gibt somit vier verschiedene Klassen: $[\varepsilon]_L$, $[a]_L$ und $[ab]_L$ und $[b]_L$. (Die letzte hätte man natürlich auch als $[aa]_L$ oder $[b^7]_L$ oder auf beliebig viele andere Arten schreiben können.)

Lösung 9.3. Alle Wörter aus L sind äquivalent zueinander, denn man kann an sie Wörter aus L (aber keine anderen) anhängen und bleibt in der Sprache. Ferner sind alle Wörter aus $L \circ \{a\}$ äquivalent zueinander, denn an sie kann man genau die Wörter aus $\{b\} \circ L$ anhängen, um wieder Wörter aus L zu erhalten. Alle anderen Wörter, die nicht in L liegen, kann man auch durch Konkatenation nicht zu L -Wörtern machen. Man erhält drei Klassen: $[\varepsilon]_L$, $[a]_L$ und $[b]_L$.

Lösung 9.4. Es sind die acht Klassen $[\varepsilon]_L$, $[1]_L$, $[0]_L$, $[00]_L$, $[001]_L$, $[0011]_L$, $[00110]_L$ sowie $[001101]_L$.

Lösung 9.5. Die Wörter $a^m b$ und $a^n b$ sind nicht äquivalent, wenn $m \neq n$ gilt, denn man kann das erste Wort mit ba^m zu einem Wort der Sprache ergänzen, das zweite jedoch nicht. Allein damit hat man schon unendlich viele Äquivalenzklassen.

Lösung 9.6. Die Wörter 0^m und 0^n sind nicht äquivalent, wenn $m \neq n$ gilt, denn man kann das erste Wort mit 10^{m-1} zu einem Wort der Sprache ergänzen, das zweite jedoch nicht. Allein damit hat man schon unendlich viele Äquivalenzklassen.

Lösung 9.7. Die Klasse $[a]_L$ enthält außer a z. B. noch das Wort ab^3 . Die Klasse $[ab]_L$ enthält außer ab z. B. noch das Wort ab^4 . Die Klasse $[ab^2]_L$ enthält außer ab^2 z. B. noch das Wort ab^5 . Die Klasse $[b]_L$ enthält außer b z. B. noch das Wort b^2 . Die Klasse $[\varepsilon]_L$ enthält außer ε keine weiteren Wörter.

Lösung 9.8. Die erste Aussage ist falsch. Das Wort 11 ist nicht äquivalent zu einem Wort der Form $0^n 1$: So ein Wort lässt sich nämlich durch Konkatenieren von n Nullen zu einem Wort von L machen, während das bei 11 nicht möglich ist.

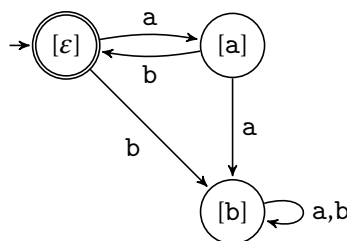
Die zweite Aussage ist falsch. ε und 1 enthalten beide keine Null, sind aber nicht äquivalent, weil man das erste Wort durch Konkatenieren von 1 zu einem Wort der Sprache machen kann, das zweite jedoch nicht.

Die dritte Aussage ist wahr. Äquivalent zu 01 sind alle Wörter der Form $0^n 10^{n-1}$ mit $n \in \mathbb{N}^+$, weil es für alle diese Wörter nur genau eine Möglichkeit (Konkatenation einer Null) gibt, sie zu einem Wort der Sprache zu machen.

Die vierte Aussage ist falsch. Alle Wörter dieser Form liegen in derselben Äquivalenzklasse. Sie haben alle die Eigenschaft, dass sie zur Sprache gehören, dass dies aber nicht mehr der Fall ist, wenn ein nichtleeres Wort konkateniert wird.

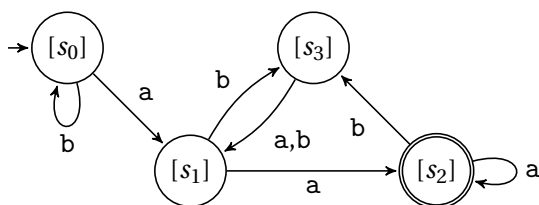
Lösung 9.9. Die Zustände sind $[\varepsilon]_L$, $[a]_L$ und $[b]_L$, wobei $[\varepsilon]_L$ sowohl der Startzustand als auch der einzige Endzustand ist. Die Übergangsfunktion sieht so aus:

	$[\varepsilon]$	$[a]$	$[b]$
a	$[\varepsilon a] = [a]$	$[aa] = [b]$	$[ba] = [b]$
b	$[\varepsilon b] = [b]$	$[ab] = [\varepsilon]$	$[bb] = [b]$



Lösung 9.10. Die Wörter a^m und a^n sind nicht äquivalent, wenn $m \neq n$ gilt. Durch Konkatenation von b^m lässt sich nämlich das erste Wort zu einem Wort der Sprache machen, das zweite aber nicht. Damit haben wir bereits unendlich viele Äquivalenzklassen und nach dem Satz von Myhill-Nerode kann L nicht regulär sein.

Lösung 9.12. Alle Zustände können von s_0 aus erreicht werden. s_1 und s_4 sind äquivalent. Es ergibt sich dieser Automat:



Ein passender regulärer Ausdruck ist $b^* a(b(a+b))^* a(a+b(a+b)(b(a+b))^* a)^*$.

Lösung 9.13. Die Äquivalenzklassen der Nerode-Relation sind $[\varepsilon]_L$, $[b]_L$, $[bb]_L$, $[a]_L$ und $[ab]_L$. Daher muss der Minimalautomat fünf Zustände haben.

Lösung 9.14. Ein möglicher Ausdruck ist $0(011011)^*$. Die Anzahl der Zustände ist nach Aufgabe 9.4 acht.

Lösung 9.15. Die folgende Tabelle ist so gemeint: Die Zahlen in der Spalte mit der Überschrift k stehen für Wörter dieser Länge, also 3 z. B. für 0^3 . Die sieben Zahlen jeweils dahinter stehen für die Längen der sieben kürzesten Wörter, mit denen man das entsprechende Wort zu einem Wort aus L fortsetzen kann.

k	$\in L$	k	$\in L$	k	$\in L$
0	0,3,5,6,9,10,12	5	0,1,4,5,7,10,13	10	0,2,5,8,10,11,14
1	2,4,5,8,9,11,14	6	0,3,4,6,9,12,14	11	1,4,7,9,10,13,14
2	1,3,4,7,8,10,13	7	2,3,5,8,11,13,14	12	0,3,6,8,9,12,13
3	0,2,3,6,7,9,12	8	1,2,4,7,10,12,13	13	2,5,7,8,11,12,14
4	1,2,5,6,8,11,14	9	0,1,3,6,9,11,12	14	1,4,6,7,10,11,13

Hinter 1 stehen also beispielsweise 2 und 4, weil 0^10^2 und 0^10^4 Wörter aus L sind, 0^10^0 und 0^10^3 aber nicht. Da alle 15 Tabellenzeilen verschieden sind,²⁶ hat die Nerode-Relation für L mindestens 15 verschiedene Äquivalenzklassen. Da andererseits zwei Wörter aus $\{0\}^*$ bezüglich dieser Relation sicher äquivalent sind, wenn ihre Längen **kongruent modulo** 15 sind, gibt es auch nicht mehr als diese 15 Klassen. Ein Minimalautomat für L hat somit 15 Zustände.

Lösung 9.16. s_2 ist nicht erreichbar. s_1 und s_4 sind äquivalent und bilden zusammen den Fehlerzustand. Es verbleiben vier Zustände für die Sprache, die durch den regulären Ausdruck $\varepsilon + 001^*$ beschrieben wird.

Lösung 9.17. Ist $n \in \mathbb{N}^+$, so wird ein Wort der Form ab^n durch Konkatenieren von c^n ein Wort von L , durch Konkatenieren von c^{n+1} jedoch nicht. Damit hat man bereits unendlich viele Äquivalenzklassen der Nerode-Relation und der erste Teil der Aufgabe ist gelöst.

Seien andererseits $n \in \mathbb{N}^+$ und $x \in L$ mit $|x| \geq n$ vorgegeben. Dann sei $u = \varepsilon$, v der erste Buchstabe von x und w so gewählt, dass $x = uvw$ gilt. Offenbar erfüllt diese Zerlegung von x die Bedingungen des Pumping-Lemmas.

Lösung 10.1. Ist $n \in \mathbb{N}^+$, so wird ein Wort der Form $[^n$ durch Konkatenieren von $]^n$ ein Wort der Dyck-Sprache, durch Konkatenieren von $]^{n+1}$ jedoch nicht. Damit hat man schon unendlich viele Äquivalenzklassen der Nerode-Relation.

Lösung 10.2. Die Grammatik könnte aus den Produktionen $S \rightarrow XC$, $X \rightarrow aXb \mid \varepsilon$ und $C \rightarrow cC \mid \varepsilon$ bestehen.

Lösung 10.3. Es ergeben sich immer **Binärbäume**. Das Thema dürfte in einer Ihrer Informatik-Vorlesungen bereits behandelt worden sein. Schlagen Sie die Grundbegriffe nach, falls Sie sie vergessen haben, denn wir werden das noch brauchen.

Lösung 10.4. So ungefähr sollte es aussehen:

$$\begin{aligned} S &\rightarrow \varepsilon \mid XC \mid X_1 V_b \mid V_a V_b \mid V_c C \mid c & C &\rightarrow V_c C \mid c & X_1 &\rightarrow V_a X \\ X &\rightarrow X_1 V_b \mid V_a V_b & V_a &\rightarrow a & V_b &\rightarrow b & V_c &\rightarrow c \end{aligned}$$

Lösung 10.5. Das Ergebnis sollte so aussehen:

$$\begin{aligned} S &\rightarrow XB \mid XS \mid SB \mid SS \mid BS \mid V_a B \mid a \\ V_a &\rightarrow a \end{aligned}$$

²⁶Um das zu sehen hätten sogar drei statt sieben Wörter gereicht.

$$X \rightarrow BS \mid SS$$

$$B \rightarrow b$$

Lösung 10.7. Wäre L kontextfrei, dann würde uns das Pumping-Lemma ein entsprechendes n liefern. Wir betrachten nun das Wort $z = (0^n 1^n)^2$. Da das „Mittelstück“ vwx höchstens die Länge n hat, liegt es entweder komplett in einer Hälfte von z oder es besteht aus Einsen der ersten Hälfte und Nullen der zweiten, wobei v mindestens eine Eins enthält und x mindestens eine Null. Man überlegt sich leicht, dass das Ersetzen von vwx durch v^2wx^2 in jedem Fall zu einem Wort führt, das nicht zu L gehört.

Lösung 10.8. Man kann im Prinzip dieselbe Idee wie in der Lösung von Aufgabe 4.10 verwenden.

Lösung 10.9. Eine kontextfreie Grammatik für L_1^* erhält man durch

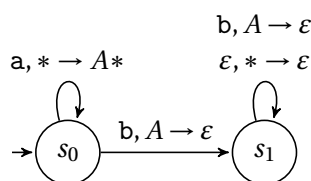
$$(N_1 \cup \{S\}, \Sigma, P_1 \cup \{(S, S_1 S), (S, \varepsilon)\}, S).$$

Lösung 10.10. Als Schnittmenge ergibt sich die Sprache, von der direkt nach dem Pumping-Lemma 10.2 gezeigt wurde, dass sie nicht kontextfrei ist. Damit ist gezeigt, dass der Durchschnitt zweier kontextfreier Mengen im Allgemeinen nicht kontextfrei ist.²⁷

Lösung 10.11. Wäre $\Sigma^* \setminus L$ kontextfrei für alle kontextfreien Sprachen L , dann wäre nach Satz 10.3 $\Sigma^* \setminus (L_1 \cap L_2) = (\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2)$ und damit auch $L_1 \cap L_2$ kontextfrei, wenn L_1 und L_2 kontextfrei wären. Das widerspricht jedoch Aufgabe 10.10.

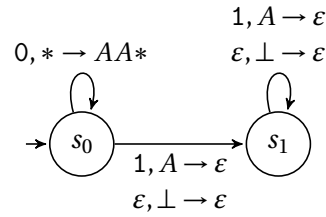
Lösung 10.12. Da gibt es viele Möglichkeiten. Zum Beispiel $\Sigma^* \setminus \{a\}$. (Warum ist diese Sprache regulär? Fällt Ihnen eine einfache Begründung ein?) Alternativ könnte man $\{a^k b^m : k, m \in \mathbb{N}\}$ nehmen.

Lösung 11.1. Das lässt sich durch eine kleine Modifikation des Automaten für L_∞ erreichen. Er darf nun auch das Symbol A vom Stack entfernen, ohne ein Zeichen zu konsumieren.



Lösung 11.2. Auch das kann man durch Modifikation des Automaten für L_∞ hinbekommen. Wir nutzen dabei aus, dass auch mehrere Zeichen auf einmal auf den Stack gelegt werden dürfen. Zu beachten ist allerdings, dass zu dieser Sprache anders als bei den vorherigen Beispielen auch das leere Wort gehört.

²⁷Die Formulierung „im Allgemeinen nicht“ ist Mathematikerschnack dafür, dass es mindestens eine Ausnahme gibt. Es ist also mitnichten so, dass der Durchschnitt zweier kontextfreier Sprachen *nie* kontextfrei ist. Er ist aber eben nicht *immer* kontextfrei.



Lösung 11.4. Zunächst konstruieren wir zu $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ den Automaten $K' = (S', I, \Gamma', \delta', s'_0, \perp', \emptyset)$ mit $L(K') = L^*(K)$. Wir fügen zwei neue Zustände s'_0 und s'_e hinzu, so dass für die neue Zustandsmenge $S' = S \cup \{s'_0, s'_e\}$ gilt. s'_e wird der „Endzustand“, dessen Job es ist, den Stack zu leeren. Daher setzen wir

$$\delta'(s'_e, \varepsilon, \sigma) = \{(s'_e, \varepsilon)\}$$

für alle $\sigma \in \Gamma$. Damit s'_e auch erreicht wird, setzen wir

$$\delta'(s, \varepsilon, \sigma) = \delta(s, \varepsilon, \sigma) \cup \{(s'_e, \varepsilon)\}$$

für alle $s \in F$ und alle $\sigma \in \Gamma$, d. h., von den Endzuständen von K gibt es dann jeweils einen ε -Übergang zu s'_e . Schließlich müssen wir noch vermeiden, dass ein Wort durch leeren Stack akzeptiert wird, das in K nur zu einem Abbruch geführt hätte. Dafür sorgen s'_0 und $\perp'$: Indem wir mit $\delta'(s'_0, \varepsilon, \perp') = \{(s_0, \perp, \perp')\}$ anfangen, haben wir immer einen „Sicherheitspuffer“ ganz unten auf dem Stack. (Für alle Argumente, die bisher nicht erwähnt wurden, soll δ' sich wie δ verhalten. Und mit Γ' ist natürlich $\Gamma \cup \{\perp'\}$ mit $\perp' \notin \Gamma$ gemeint.)

Und jetzt konstruieren wir zu vorgegebenem $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ einen Kellerautomaten $K' = (S', I, \Gamma', \delta', s'_0, \perp', F')$ mit $L^*(K') = L(K)$. Wir fügen wieder zwei neue Zustände s'_0 und s'_e hinzu, so dass für die neue Zustandsmenge $S' = S \cup \{s'_0, s'_e\}$ gilt. Und ebenfalls soll hier $\Gamma' = \Gamma \cup \{\perp'\}$ mit $\perp' \notin \Gamma$ gelten. Für alle Argumente aus dem Definitionsbereich von δ soll sich δ' genau wie δ verhalten. (K' „simuliert“ gleichsam K .) Hinzu kommt wie oben $\delta'(s'_0, \varepsilon, \perp') = \{(s_0, \perp, \perp')\}$, weil wir nun einen leeren Stack von K daran erkennen können, dass $\perp'$ das oberste Stacksymbol von K' ist. Die Menge F' der Endzustände soll $\{s'_e\}$ sein und dafür setzen wir

$$\delta'(s, \varepsilon, \perp') = \{(s'_e, \varepsilon)\}$$

für alle $s \in S$.

Lösung 11.6. Die Grundidee, die in der Theoretischen Informatik öfter angewendet wird, ist, dass man Wörter einer begrenzten Länge durch Symbole ersetzt.²⁸ Das wird im Folgenden skizziert, aber nicht bis ins Detail ausgearbeitet.

Sei $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ ein beliebiger vorgegebener Kellerautomat. Da sowohl der Definitionsbereich der Übergangsfunktion δ als auch deren Funktionswerte

²⁸Das kennen Sie im Prinzip schon. Man kann z. B. alle 16 Wörter aus Σ_{bool}^* , deren Länge 4 ist, durch die [Hexadezimalziffern](#) 0 bis F ersetzen.

endlich sind, gibt es eine maximale Anzahl n von Symbolen, um die der Stack in einem einzigen Übergang anwachsen kann. Wir definieren durch

$$\Gamma' = \Gamma^1 \cup \Gamma^2 \cup \dots \cup \Gamma^n$$

ein neues Alphabet.²⁹ Die Elemente dieser Menge, die aus 1-Tupeln, 2-Tupeln und so weiter besteht, betrachten wir als neue Symbole. So etwas wie (σ, τ) ist also ein „atomares“, nicht weiter zerlegbares Objekt, das sich von (σ) oder (τ) unterscheidet. Für $(\tau_1, \dots, \tau_k)$ verwenden wir die Schreibweise $[\tau_1 \dots \tau_k]$. Die Idee ist, dass so ein *Symbol* Stellvertreter für das *Wort* $\tau_1 \dots \tau_k$ sein soll.

Der neue Automat K' bekommt Γ' als Stackalphabet, unterscheidet sich aber ansonsten nur durch die Übergangsfunktion von K . Die wird folgendermaßen konstruiert: Wir gehen alle (s, a, σ) aus dem Definitionsbereich von δ durch und für jedes Element (s', β) von $\delta(s, a, \sigma)$ gehen wir dann alle Symbole von Γ' durch, die mit σ anfangen, und fügen $(s', [\beta\tau_1 \dots \tau_k])$ zu $\delta'(s, a, [\sigma\tau_1 \dots \tau_k])$ hinzu. Dabei ist mit $[\beta\tau_1 \dots \tau_k]$ einfach $[\beta\tau_1 \dots \tau_k]$ gemeint, falls $|\beta| + k \leq n$ gilt. Anderenfalls ist damit das Wort $[\beta\tau_1 \dots \tau_{n-|\beta|}][\tau_{n-|\beta|+1} \dots \tau_k]$ über Γ' gemeint.

Lösung 11.7. Das sollte so aussehen:

$$\begin{aligned} (s_0, aab, \perp) &\Rightarrow_K (s_0, aab, S) \Rightarrow_K (s_0, aab, aSb) \Rightarrow_K (s_0, ab, Sb) \\ &\Rightarrow_K (s_0, ab, aSb) \Rightarrow_K (s_0, b, Sb) \Rightarrow_K (s_0, b, b) \Rightarrow_K (s_0, \varepsilon, \varepsilon) \end{aligned}$$

Man beachte, dass hier ganz wesentlich der Nichtdeterminismus eingeht, weil es für jedes Wort viele Wege gibt, von denen die meisten nicht zur Akzeptanz führen werden. Der Automat „probiert“ gleichsam, bis er eine passende Ableitung gefunden hat.

Lösung 11.8. Die erste Aussage ist falsch. Das wurde bereits in Aufgabe 5.11 begründet. Die zweite Aussage ist wahr. Das folgt aus dem ersten Teil des Beweises von Satz 11.1.

Lösung 11.9. Die akzeptierte Sprache ist $\{ab^n a^{2n+1} : n \in \mathbb{N}\}$. Als Grammatik könnte man z. B. die mit den Produktionen $S \rightarrow aT$ und $T \rightarrow bTaa \mid a$ nehmen.

Lösung 11.11. Die Menge der Nichtterminalsymbole würde aus dem neuen Startsymbol S_0 sowie zwölf³⁰ weiteren Symbolen wie z. B. $[s_0 A s_1]$ oder $[s_1 \perp s_1]$ bestehen. Es wäre viel zu ermüdend, hier nun alle Produktionen aufzuschreiben. Es sollen aber zumindest so viele herausgegriffen werden, dass eine Ableitung demonstriert werden kann. Dafür käme zunächst $S_0 \rightarrow [s_0 \perp s_1]$ als eine der Produktionen mit der linken Seite S_0 hinzu. In der folgenden Tabelle stehen drei weitere Produktionen und dahinter jeweils der Grund, warum sie in die Grammatik aufgenommen wurden.

$[s_0 \perp s_1] \rightarrow a[s_0 A s_1][s_1 \perp s_1]$	$\{(s_0, A \perp)\} \in \delta(s_0, a, \perp)$
$[s_0 A s_1] \rightarrow b$	$\{(s_1, \varepsilon)\} \in \delta(s_0, b, A)$
$[s_1 \perp s_1] \rightarrow \varepsilon$	$\{(s_1, \varepsilon)\} \in \delta(s_1, \varepsilon, \perp)$

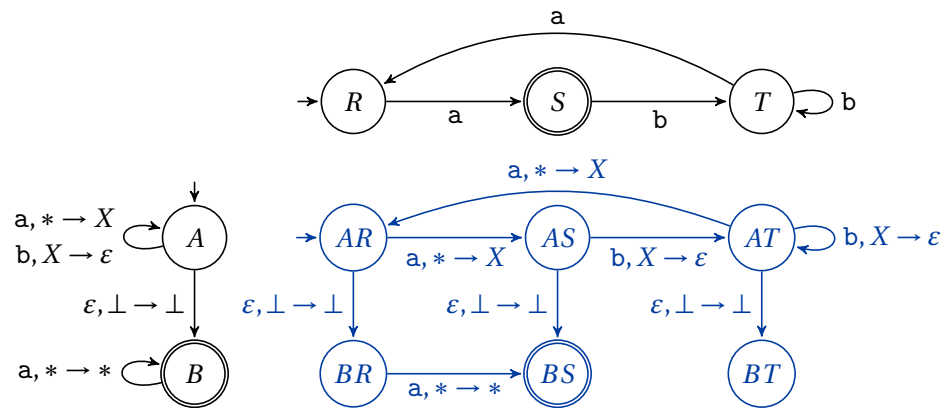
²⁹Erinnern Sie sich, dass wir ganz am Anfang Alphabete sehr allgemein definiert hatten.

³⁰Die vier Paare (s_0, s_0) , (s_0, s_1) , (s_1, s_0) und (s_1, s_1) werden mit den drei Stacksymbole $\perp$, A und B auf alle möglichen Arten kombiniert.

Mit diesen insgesamt vier Produktionen kann man nun aus S_0 exemplarisch das Wort ab ableiten.

Lösung 11.12. Für L_2 gibt es einen Kellerautomaten K mit $L_2 = L^*(K)$ (siehe Aufgabe 11.4) und für L_3 einen endlichen Automaten A mit $L_3 = L(A)$. Genau wie im Beweis von Lemma 5.3 baut man aus A und K nun einen Produktautomaten, allerdings (siehe Aufgabe 5.16) einen für den Durchschnitt der Sprachen. Dieser Produktautomat wird natürlich ein Kellerautomat sein, wobei nur die Übergänge von K für den Stack relevant sind.

Die Details müssten natürlich noch etwas genauer ausgearbeitet werden, aber die folgende Skizze (analog zur Lösung von Aufgabe 5.15) verdeutlicht hoffentlich das Prinzip.



Lösung 11.14. Wenn sich der Automat im Zustand s befindet, das nächste noch nicht konsumierte Zeichen a ist und das Zeichen oben auf dem Stack σ ist, dann gibt es drei Möglichkeiten, die sich gegenseitig ausschließen:

- (i) $\delta(s, a, \sigma)$ hat genau ein Element. Dadurch ist festgelegt, was passieren muss, weil nach der obigen Definition $\delta(s, \epsilon, \sigma)$ leer ist.
- (ii) $\delta(s, a, \sigma)$ ist leer und $\delta(s, \epsilon, \sigma)$ hat genau ein Element, wodurch auch wiederum festgelegt ist, wie es weitergeht.
- (iii) $\delta(s, a, \sigma)$ und $\delta(s, \epsilon, \sigma)$ sind beide leer. Dann muss der Automat abbrechen. (Siehe Fußnote 6.)

Ist der Stack leer, dann geht es ohnehin nicht weiter. Es gibt jedoch einen kleinen Rest Nichtdeterminismus, wenn das Wort bereits komplett gelesen wurde und noch Zeichen auf dem Stack sind. Geht beispielsweise das Wort w in einem deterministischen Kellerautomaten $K = (S, I, \Gamma, \delta, s_0, \perp, F)$ nach dem Konsumieren des letzten Zeichens in den Zustand s mit dem Wort $\sigma\alpha$ oben auf dem Stack über und ist $\delta(s, \epsilon, \sigma) = (s', \epsilon)$, dann gilt sowohl $(s_0, w, \perp) \Rightarrow_K^* (s, \epsilon, \sigma\alpha)$ als auch $(s_0, w, \perp) \Rightarrow_K^* (s', \epsilon, \alpha)$. Bezüglich der Akzeptanz (ob durch leeren Stack oder durch Endzustand) ist das aber offensichtlich kein Problem, weil so etwas nur ganz am Ende passieren kann.

Warum gibt es in deterministischen Kellerautomaten überhaupt ε -Übergänge? Die werden benötigt, um ggf. Stackzeichen abzubauen, weil pro Eingabesymbol mehrere Zeichen zum Stack hinzugefügt, aber immer nur eins entfernt werden kann.

Lösung 11.15. Nein, das können Sie nicht. Selbst an so einer simplen regulären Sprache scheitern deterministische Kellerautomaten, wenn durch leeren Stack akzeptiert werden soll. Damit 0 akzeptiert wird, muss der Stack nach dem Lesen der ersten Null leer sein. Dann kann danach aber keine Eins mehr gelesen werden – siehe Fußnote 6.

Deterministische Kellerautomaten können mit leerem Stack nur Sprachen akzeptieren, die *präfixfrei* sind. Damit ist gemeint, dass kein Wort der Sprache der Anfang eines anderen Wortes der Sprache ist.

präfixfrei

Lösung 11.16. Das lässt sich wohl am einfachsten mit einer Grammatik erledigen:

$$\begin{aligned} S &\rightarrow 0S_11 \mid 0S_211 \\ S_1 &\rightarrow 0S_11 \mid \varepsilon \\ S_2 &\rightarrow 0S_211 \mid \varepsilon \end{aligned}$$

Lösung 12.3. Man würde dafür mehr als zwei Zellen zu einem neuen Symbol zusammenfassen. Mit drei Zellen kann man z. B. bis zu acht verschiedene Symbole darstellen und allgemein mit n Zellen bis zu 2^n . Natürlich wird die Logik für die Umsetzung immer aufwendiger, aber das Verfahren wird nicht prinzipiell komplizierter.

Lösung 12.4. Wir haben in diesem Abschnitt bereits zwei Möglichkeiten gesehen, so etwas zu realisieren.

Die erste Möglichkeit ist das *unäre* System. Die Zahl 5 würde beispielsweise durch fünf Einsen gefolgt von Nullen dargestellt werden. Die Nullen stehen für „leeren“ (ungenutzten) Speicherplatz.

Die zweite Möglichkeit verwendet das Binärsystem und „codiert“ jede Ziffer durch zwei Bits, z. B. 10 für die Null, 11 für die Eins und 00 für ungenutzten Speicherplatz bzw. für „Ende“.

Die zweite Möglichkeit ist besser als die erste in dem Sinne, dass sie für Zahlen ab einer gewissen Größe weniger Bits benötigt. Man kann das aber noch verbessern: Man speichert die Länge der Binärdarstellung der Zahl im obigen Format und die Zahl selbst dann mit einem Bit pro Ziffer. Die Zahl 38 aus der Aufgabenstellung hat binär beispielsweise sechs Stellen und würde als

11111000100110

gespeichert werden.

Auch hier macht sich der Vorteil erst bei größeren Zahlen bemerkbar. Sie können zur Übung vielleicht mal für alle drei Varianten die Anzahl der benötigten Bits in *Landau-Notation* angeben.

Lösung 12.5. Ja. In der Formulierung der Definition ist von einer Teilmenge von A die Rede und A ist eine Teilmenge von A .

Lösung 12.6. Die zweite Zeile mit dem Rechtspfeil $\rightarrow$ fügt ein Leerzeichen hinzu, falls der Kopf vor dem letzten Zeichen stand. Das Band „wächst“ also nach rechts, falls das nötig sein sollte. Beachten Sie jedoch, dass das Band nicht nach links wachsen kann: Eine Bewegung nach links ist nur vorgesehen, wenn links vom Kopf noch mindestens ein Zeichen steht.

Am Ende der Definition steht außerdem, dass σ und σ' sich nur dann unterscheiden können, wenn beide Zeichen nicht die Grenzmarkierung $\vdash$ sind. Das sorgt dafür, dass weder Grenzmarkierungen hinzugefügt noch welche entfernt werden können. (Die Übergangsfunktion δ könnte so etwas zwar vorsehen, aber es würde sich nicht als Übergang in $\Rightarrow_T$ darstellen lassen.)

Lösung 12.7. Der Ablauf sieht so aus:

$$\vdash [s_0]11 \Rightarrow_{T_1} \vdash 1[s_0]1 \Rightarrow_{T_1} \vdash 11[s_0]\square \Rightarrow_{T_1} \vdash 1[s_1]10$$

Lösung 12.8. Weil s_2 ein Endzustand ist, muss dort nach Punkt (iv) der Definition $\perp$ stehen. Man hätte die Spalte auch weglassen können.

Lösung 12.9. f_{T_1} ist die partielle Funktion von Σ_{bool}^* nach Σ_{bool}^* , die durch

$$f_{T_1}(w) = \begin{cases} w0 & w \in \{0\}^* \\ \perp & w \notin \{0\}^* \end{cases}$$

definiert ist. f_{T_2} ist die partielle Funktion von $\{0\}^*$ nach $\{0\}^*$, die immer den „Funktionswert“ $\perp$ hat, also für kein Argument definiert ist.

Lösung 12.10. Die Maschine interpretiert ein Eingabewort aus Σ_{bool}^+ als Binärdarstellung einer Zahl und subtrahiert von dieser Zahl 1. Wenn die Eingabe die Zahl 0 darstellt, wird nicht subtrahiert. Siehe auch `Code/dec.tm2` im [Docker-Image](#).

Lösung 12.11. Mit dem Startzustand s_0 und dem einzigen Endzustand s_5 könnte die Übergangsfunktion so aussehen:

	s_0	s_1	s_2	s_3	s_4
0	$(s_0, 0, \rightarrow)$	$(s_1, 0, \rightarrow)$	$(s_4, \square, \leftarrow)$	$(s_3, 0, \leftarrow)$	$(s_4, 0, \leftarrow)$
1	$(s_1, 1, \rightarrow)$	$(s_1, 1, \rightarrow)$	$(s_3, \square, \leftarrow)$	$(s_3, 1, \leftarrow)$	$(s_3, 0, \leftarrow)$
$\square$	$(s_5, \square, \rightarrow)$	$(s_2, \square, \leftarrow)$	$\perp$	$\perp$	$\perp$
$\vdash$	$\perp$	$\perp$	$\perp$	$(s_0, \vdash, \rightarrow)$	$\perp$

Solange die Maschine im Zustand s_0 ist, geht sie einfach vorwärts bis zum Ende und kommt bei einem Leerzeichen in den Endzustand (und nur so). Sieht sie eine Eins, dann wechselt sie in den Zustand s_1 und bleibt, weiter vorwärts gehend, in diesem bis zum Ende des Wortes, um dann in den Zustand s_2 zu wechseln. Bis zu diesem Punkt wurde noch kein Zeichen geändert und die Maschine steht nun vor dem letzten Zeichen, das auf jeden Fall durch ein Leerzeichen ersetzt wird.

War es eine Eins, so geht die Maschine anschließend im Zustand s_3 wieder zurück zum Anfang des Wortes, von wo aus sie wieder von vorne anfängt. War das letzte Zeichen hingegen eine Null, dann läuft der Kopf (nun im Zustand s_4) ebenfalls rückwärts, ersetzt auf diesem Weg aber die erste Eins, der er begegnet, durch eine Null, um dann den Rest des Rückwegs ebenfalls im Zustand s_3 zu durchwandern.

Lösung 12.12. Hier nur eine grobe Idee: Die Maschine wandert von links nach rechts über das Wort, ihr Zustand hängt dabei vom ersten Zeichen ab, das nach dem „Betrachten“ gelöscht (also durch ein Leerzeichen ersetzt) wird. Das wird mit dem letzten Zeichen verglichen (das ebenfalls anschließend gelöscht wird). Sind die Zeichen nicht gleich, handelt es sich nicht um ein Palindrom. Ansonsten geht die Maschine zurück bis zum ersten nicht gelöschten Zeichen und fängt wieder von vorne an. Das wird wiederholt, bis maximal ein Zeichen übrig ist.

Lösung 12.14. Man kann so eine Maschine mit n Spuren ganz simpel durch eine normale „einspurige“ Turingmaschine simulieren, indem man immer n Zeichen zu einem zusammenfasst. (Siehe Aufgabe 12.3.) Hat die n -spurige Maschine das Bandalphabet Γ , so arbeitet die simulierende Maschine mit dem Alphabet Γ^n .

Lösung 12.15. Solche Maschinen mit n Bändern kann man (mit hohem Aufwand) durch Turingmaschinen mit $2n$ Spuren simulieren, wobei jeweils eine Spur ein Band des Originals simuliert, während die andere sich die Position des zugehörigen Schreib-Lese-Kopfs mit einem Markierungszeichen „merkt“. Die simulierende Maschine fährt zur Simulation eines Schritts der simulierten Maschine den kompletten Bandinhalt zweimal ab: im ersten Durchgang wird an den jeweiligen Positionen der simulierten Köpfe gelesen, im zweiten wird geschrieben und die Köpfe werden bewegt. Das „Gedächtnis“ für diesen komplizierten Prozess bilden die Zustände des simulierenden Automaten, von denen es deshalb sehr viele gibt.³¹

Lösung 12.17. Es gibt dafür mindestens zwei sinnvolle Ansätze. Der eine ist, dass man das Band im Geiste in zwei Hälften aufteilt, die sich abwechseln. Nummerieren wir die Zellen beginnend bei null (wo die Grenzmarkierung sitzt), so gehören die „ungeraden“ Zellen 1, 3, 5 und so weiter zum rechten Teil des Bandes, die Zellen 2, 4, 6 und so weiter zum neu hinzugekommenen linken Teil. Geht man beispielsweise vom Feld 3 aus vier Schritte nach links, so sorgt die „Übersetzungslogik“ dafür, dass man nacheinander auf den Feldern 1, 2, 4 und 6 landet. Man ist vom zweiten Feld „rechts von der Mitte“ zum dritten Feld „links von der Mitte“ gewandert.

Der zweite Ansatz ist wesentlich umständlicher, hat aber den Vorteil, dass man den Bandinhalt immer in einem Stück sehen kann: Wann immer der Kopf sich nach links auf die Grenzmarkierung bewegt, wird zunächst der gesamte Bandinhalt um ein Feld nach rechts verschoben. Anschließend wird der Kopf auf das nun leere Feld rechts von der Grenzmarkierung gesetzt.

Lösung 12.19. Auch das wäre kein wirklicher Verlust. Eine Maschine, die $\diamond$ nicht kennt, kann das immer durch $\rightarrow$ gefolgt von $\leftarrow$ ersetzen.

³¹ Siehe dazu auch Lemma 18.1 später.

Lösung 12.20. Mit dem Startzustand s_* und dem einzigen Endzustand s_e könnte die Übergangsfunktion so aussehen:

	s_0	s_1	s_*
0	$(s_0, 0, \rightarrow)$	$(s_0, 1, \rightarrow)$	$(s_0, \square, \rightarrow)$
1	$(s_1, 0, \rightarrow)$	$(s_1, 1, \rightarrow)$	$(s_1, \square, \rightarrow)$
$\square$	$(s_e, 0, \diamond)$	$(s_e, 1, \diamond)$	$(s_e, \square, \diamond)$
$\vdash$	$\perp$	$\perp$	$\perp$

Die Zustände s_0 bzw. s_1 stehen dafür, dass das letzte gelesene Zeichen eine Null bzw. Eins war.

Lösung 12.21. Da gemäß der in diesem Skript verwendeten Definition die Turingmaschine nach dem Übergang in einen Endzustand terminiert, würde man in der Tat mit einem solchen Zustand auskommen, da solche Zustände anders als bei endlichen Automaten nicht mehrfach „besucht“ werden können. Es ist daher irrelevant, in *welchen* Endzustand der Automat übergeht. (In den Simulationen tm und tm2 gibt es ja auch nur *einen* H-Befehl.)

Die Definition wurde jedoch so allgemein gehalten, weil man in der Literatur auch Varianten ohne die Bedingung $\delta(s, \sigma) = \perp$ für alle $s \in F$ und alle $\sigma \in \Gamma$ findet. Dann kann es sein, dass es von dem Zeichen, auf dem der Schreib-Lese-Kopf steht, abhängt, ob die Turingmaschine in einem Endzustand terminiert oder weiter fortfährt.

Lösung 12.23. Der Kellerautomat liest zunächst das ganze Eingabewort ein und schreibt es dabei Zeichen für Zeichen auf den ersten Stack. Da das Wort zu diesem Zeitpunkt bereits komplett konsumiert ist, wird der restliche Ablauf nur noch durch die Stacks gesteuert. Das Eingabewort wird vom ersten auf den zweiten Stack verschoben. Dabei wird seine Reihenfolge umgekehrt und es steht dort nun in der „richtigen“ Reihenfolge. Die Konfigurationen $\nu[s]w$ der simulierten Turingmaschine entsprechen nun den Konfigurationen des Kellerautomaten, bei denen w auf dem zweiten und ν auf dem ersten Stack steht. Zustände und Übergangsfunktion können mehr oder weniger direkt denen der Turingmaschine nachempfunden werden.

Lösung 13.2. Nein, jedenfalls nicht im Sinne der [uns interessierenden Fragestellung](#). KI-Programme laufen auf normaler Hardware und werden schlussendlich wie jedes andere Programm in Maschinensprache ausgeführt. Und Quantencomputer basieren zwar auf einer ganz anderen „Hardware“, aber sie lassen sich (zumindest theoretisch) mit herkömmlichen Computern simulieren.

Lösung 13.3. Das Band einer Turingmaschine ist immer so lang wie nötig und die Register einer Registermaschine sind groß genug für beliebige natürliche Zahlen. Echte Computer haben jedoch immer nur einen begrenzten Speicherplatz zur Verfügung. Wenn wir Vergleiche im Sinne der Church-Turing-These anstellen, ignorieren wir diesen Unterschied.

Lösung 13.5. Die Turingmaschine `Code/inc.tm2` aus dem Docker-Image realisiert genau diese Funktion. Nennt man diese Turingmaschine T , so gilt $g = f_T$.

Lösung 13.6. In diesem Fall kann man `Code/palindrome.tm2` aus dem Docker-Image verwenden.³² w_1 und w_2 sind hier die Wörter 1 und 0.

Lösung 13.7. Selbstverständlich. Es würde sicher keinen Spaß machen, eine Turingmaschine zu kreieren, die $\mathbb{P}$ entscheidet. Aber Sie können so etwas hoffentlich in einer Sprache Ihrer Wahl programmieren. (Können Sie doch, oder?) Wenn das geht, dann geht es nach der Church-Turing-These auch mit einer Turingmaschine.

Lösung 13.8. Weil man „nur“ ein Programm schreiben muss, dass die Eingabe der Reihe nach mit allen Elementen der Menge vergleicht. Ein **Python**-Programm, das die Menge $\{23, 42, 1024\}$ entscheidet, könnte z. B. so aussehen:

```
def decide(n):
    if n==23:
        return True
    if n==42:
        return True
    if n==1024:
        return True
    return False
```

Dabei wurde das Programm aus didaktischen Gründen absichtlich umständlich geschrieben. Zwei kürzere Alternativen sehen folgendermaßen aus:

```
decide = lambda n: n==23 or n==42 or n==1024
decide = lambda n: n in {23,42,1024}
```

Lösung 13.9. Wenn eine Menge entscheidbar ist, dann gibt es mindestens ein `tm2`-Programm, das sie entscheidet. Weitere Programme, die dasselbe leisten, erhält man, indem man einfach weitere Zeilen mit nicht erreichbaren Zuständen anhängt. Damit hat man schon unendlich viele „verschiedene“ Programme für dieselbe Aufgabe. Und natürlich ist das noch nicht alles. Es gibt eigentlich immer mehr als eine Möglichkeit, ein bestimmtes Ergebnis zu erzielen. Und abgesehen von ganz trivialen Fällen ist in der Regel nicht klar, welches der „beste“ Algorithmus für ein bestimmtes Problem ist. Jeder Mensch, der schon einmal programmiert hat, sollte das wissen.

Lösung 13.10. Für dieses „kleine“ Alphabet würde **ASCII** ausreichen. Jedem Zeichen wird eine siebenstellige Bitfolge zugeordnet. Wörter werden dadurch gödelisiert, dass man die Bitfolgen der Zeichen konkateniert und das Resultat als Binärzahl interpretiert. Aus dem Wort `haw` würde beispielsweise die Zeichenkette

1101000 1100001 1110111

und daraus die natürliche Zahl 1716471. Heutzutage verwendet man eher **Unicode**, aber das Prinzip ist dasselbe.

³²Nicht ganz! Diese Maschine terminiert nicht bei der Eingabe ϵ . Das sollten Sie reparieren.

Lösung 13.11. $g(\text{abba}) = p_1^1 p_2^2 p_3^2 p_4^1 = 2^1 \cdot 3^2 \cdot 5^2 \cdot 7^1 = 3150$

Lösung 13.12. Hier ein Lösungsvorschlag für Python:

```
from sympy import factorint, prime
from math import prod

N = lambda a: ord(a)-ord('a')+1 if a in "abc" else None
Ninv = lambda k: chr(k-1+ord('a')) if k in [1,2,3] else None
G = lambda w: prod(prime(i+1)**N(c) for i,c in enumerate(w))

def inL(n):
    g = factorint(n)
    if not set(g.values()) <= {1,2,3}:
        return None
    return list(g.keys()) == [prime(i+1) for i in range(len(g))]

def Ginv(n):
    if not inL(n):
        return None
    g, s = factorint(n), ""
    for i,p in enumerate(sorted(g.keys())):
        if p!=prime(i+1):
            return None
        s += Ninv(g[p])
    return s

Ginv(G("abba"))      # Beispiel
```

Lösung 13.13. Man bildet einfach das Tupel $t = (n_1, n_2, \dots, n_k)$ auf

$$g(t) = p_1^{n_1+1} p_2^{n_2+1} \dots p_k^{n_k+1}$$

ab, wobei wieder p_i die i -te **Primzahl** sein soll. Es muss lediglich darauf geachtet werden, dass der Exponent immer um eins höher als die zugehörige Komponente n_i ist, damit erkannt werden kann, wo das Tupel aufhört.

Relevant ist dabei natürlich, dass Tupel nach Definition nur endlich viele Komponenten haben. Sonst könnte man das obige Produkt nicht definieren.

Lösung 13.14. 42 hat die Primfaktorzerlegung $2^1 \cdot 3^1 \cdot 7^1$ bzw. $p_1^1 p_2^1 p_4^1$. Die dritte Primzahl $p_3 = 5$ fehlt.

Lösung 13.15. In Python könnte das so aussehen:

```
from sympy import factorint, prime

emptyarray = lambda: 1
def read(r,i):
    e = factorint(r).get(prime(i+1))
    return (e and e-1)
def store(r,i,v):
```

```

o = read(r,i)
if o:
    r //= prime(i+1)**(o+1)
return r*prime(i+1)**(v+1)

# Beispiel
r = emptyarray()
r = store(r,3,42)
read(r,3), read(r,2)

```

Lösung 13.16. Meine Python-Lösung sieht folgendermaßen aus:

```

from sympy import factorint,prime

emptystack = lambda: 1
def pop(s):
    if s == emptystack():
        return s, None
    f = factorint(s)
    p = max(f.keys())
    e = f[p]
    return s//p**e, e-1
push = lambda s,v: s*prime(len(factorint(s))+1)**(v+1)

# Beispiel
s = push(push(push(emptystack(),1),0),3)
s,v = pop(s)

```

Lösung 13.17. $a_1 = 2^{m_1} \cdot (2n_1 + 1)$ und $a_2 = 2^{m_2} \cdot (2n_2 + 1)$ sind auf jeden Fall verschiedene Zahlen, wenn m_1 und m_2 verschieden sind. Das liegt daran, dass die Faktoren $(2n_1 + 1)$ bzw. $(2n_2 + 1)$ ungerade sind. m_1 und m_2 sind deshalb die Potenzen der Zahl 2 in der Primfaktorzerlegung von a_1 und a_2 und nach dem [Fundamentalsatz der Arithmetik](#) können a_1 und a_2 daher nicht gleich sein. Sind m_1 und m_2 hingegen gleich und n_1 und n_2 verschieden, so sind a_1 und a_2 natürlich auch verschieden, weil dann offensichtlich $2n_1 + 1 \neq 2n_2 + 1$ gilt. Insgesamt ist die Abbildung $(m, n) \mapsto 2^m \cdot (2n + 1)$ also injektiv. Dass sie berechenbar ist, dürfte offensichtlich sein.

Die Umkehrfunktion erhält man, indem man eine Zahl so lange durch 2 dividiert, bis das nicht mehr ohne Rest möglich ist. Das liefert die erste Komponente des Paares. Von der verbleibenden Zahl subtrahiert man 1 und teilt dann durch 2. Dieser Vorgang ist offensichtlich auch berechenbar.

Und dass die Menge aller codierten Paare entscheidbar ist, ist besonders einfach: *Jede* natürliche Zahl außer 0 codiert ein Paar.

Lösung 13.18. Ein simpler Vorschlag in Python mit denselben Bezeichnungen wie in Aufgabe 13.15:

```

cons = lambda m,n: 2**m * (2*n+1)
emptyarray = lambda: 1
store = lambda r,i,v: cons(cons(i,v),r)
def carcdr(a):
    m = 0
    while a%2 == 0:
        m += 1
        a //= 2
    return m, (a-1)//2
def read(r,i):
    while r != 1:
        a,r = carcdr(r)
        m,n = carcdr(a)
        if m == i:
            return n
    return 0

```

Hier wird „leeren“ Speicherzellen immer der Wert 0 zugeordnet. Außerdem machen wir uns das Leben dadurch leicht, dass neue Inhalt einfach *vor* die alten gesetzt werden. Das verkürzt den Code, dafür werden die Zahlen aber größer. Wenn Sie Lust haben, dann „reparieren“ Sie das, indem die Funktion `store` den alten Wert in der Speicherstelle `i` überschreibt, falls es einen gibt.

Lösung 14.1. In beiden Fällen muss die Maschine in einem Zustand aus F terminieren. Bei der Definition der Berechnung muss der Bandinhalt dann aber eine bestimmte Form haben, während er bei der Definition der Akzeptanz irrelevant ist. Wenn also $f_T(w)$ definiert ist, dann wird w akzeptiert, während umgekehrt nicht $f_T(w) \neq \perp$ für jedes von T akzeptierte Wort w gilt.

Man kann sich jedoch leicht das Folgende überlegen: Gibt es eine Turingmaschine T , die die Sprache L akzeptiert, dann kann man eine Turingmaschine T' mit der Eigenschaft konstruieren, dass $f_{T'}$ genau für die Wörter aus w einen definierten Funktionswert hat. (Man baut T einfach so um, dass nach dem eigentlichen Akzeptieren noch das Band „aufgeräumt“ wird, indem alle Zellen geleert werden. Der definierte Funktionswert ist dann immer ε .)

Lösung 14.2. Weil nun der Funktionswert $\emptyset$ möglich ist, der die Rolle des „Funktionswertes“ $\perp$ übernehmen kann.

Lösung 14.3. Die Bedingung ist nun nicht mehr, dass ein bestimmtes Tripel der entsprechende Funktionswert von δ ist, sondern dass es ein *Element* des Funktionswertes ist. Man muss in der Definition lediglich alle Bedingungen der Form $\delta(s, \sigma) = (s', \sigma', x)$ durch solche der Form $(s', \sigma', x) \in \delta(s, \sigma)$ ersetzen, wobei x eines der Zeichen $\rightarrow$, $\leftarrow$ oder $\diamond$ ist.

Lösung 14.4. Die zweite Grammatik ist nicht kontextsensitiv, weil in der Produktion $abT \rightarrow cc$ die linke Seite länger als die rechte ist. Die dritte ist nicht kontextsensitiv, weil es die Produktion $S \rightarrow \varepsilon$ gibt und S auf der rechten Seite von $S \rightarrow aS$ auftritt. Die vierte ist wegen $T \rightarrow \varepsilon$ nicht kontextsensitiv. Die restlichen zwei sind kontextsensitiv.

(Sollten Sie übrigens auf die Idee kommen, anhand von alten Klausuren zu meinen Vorlesungen zu lernen, dann beachten Sie bitte, dass ich insbesondere die Definition kontextsensitiver Grammatiken früher laxer gehandhabt habe. Für zukünftige Prüfungen zählt aber natürlich dieses Skript und nicht der Stand von vor zehn Jahren.)

Lösung 14.5. Da wir noch kein anderes Mittel kennen, kann das nur durch die Angabe einer kontextsensitiven Grammatik für diese Sprache erreicht werden. Die könnte z. B. so aussehen:

$$S \rightarrow \varepsilon \mid T \quad T \rightarrow aTb \mid abc \quad cB \rightarrow Bc \quad bB \rightarrow bb$$

Die Idee ist in diesem Fall, dass erst ein Ausdruck wie $aaabcBcBc$ erzeugt wird. Dann „springt“ jedes B , das rechts von einem c steht, auf dessen linke Seite. Damit wird aus dem obigen Beispiel nacheinander $aaabcBBcc$ und $aaabBcBcc$ und schließlich $aaabBBccc$. Der Rest sollte klar sein.

Für eine ähnliche Idee siehe auch die zweite Lösung von Aufgabe 2.16.

Lösung 14.6. Man könnte es so machen:

$$\begin{aligned} S &\rightarrow \varepsilon \mid 0T_0 \mid 1T_1 \mid 00 \mid 11 \\ T_0 &\rightarrow 0NT_0 \mid 1ET_0 \mid S_0 \\ T_1 &\rightarrow 0NT_1 \mid 1ET_1 \mid S_1 \\ N0 &\rightarrow 0N & N1 &\rightarrow 1N \\ E0 &\rightarrow 0E & E1 &\rightarrow 1E \\ NS_0 &\rightarrow S_00 \mid 00 & ES_0 &\rightarrow S_01 \mid 01 \\ NS_1 &\rightarrow S_10 \mid 10 & ES_1 &\rightarrow S_11 \mid 11 \end{aligned}$$

Hier stehen N und E als Platzhalter für „zukünftige“ Nullen und Einsen. S_0 , S_1 , T_0 und T_1 sind ebenfalls Platzhalter, dienen aber gleichzeitig als Trennzeichen zwischen erstem und zweitem Teilwort.

Ein Beispiel: In den ersten drei Zeilen werden nacheinander die Wörter $0T_0$, $00NT_0$, $00N1ET_0$, $00N1E0NT_0$ sowie $00N1E0NS_0$ abgeleitet. Die folgenden vier Produktionen ermöglichen den Platzhaltern E und N das „Überspringen“ der terminalen Symbole. Man hätte dann $0010NENS_0$.

Durch die Produktionen am Ende „springen“ die Platzhalter über das Symbol S_0 und werden dabei gleichzeitig zu terminalen Symbolen. Das ergibt nacheinander $0010NES_0$, $0010NS_010$ und schließlich 00100010 , wodurch sich am Ende S_0 zum Pendant der Null verwandelt, die am Anfang der Wortes steht und im ersten Schritt abgeleitet wurde.

Lösung 14.7. Sie haben hoffentlich nicht einfach gedacht, dass jede kontextfreie Grammatik „offensichtlich“ kontextsensitiv ist. Das stimmt nämlich nicht. Selbst ganz einfache reguläre Grammatiken wie $S \rightarrow aS \mid \varepsilon$ sind nicht kontextsensitiv. (Warum?) Wir können jedoch zu jeder kontextfreien Sprache L nach Lemma 10.1

eine kontextfreie Grammatik G in Chomsky-Normalform angeben, die L erzeugt. Und G ist dann tatsächlich kontextsensitiv.

Lösung 14.8. Die Klasse der endlichen Sprachen. Die sind – siehe Aufgabe 4.6 – trivialerweise regulär, aber natürlich ist nicht jede reguläre Sprache endlich.

Lösung 14.9. Jedes Tripel aus $\Gamma \times \Gamma \times \{0, \dots, n\}$ ist ein Nichtterminalsymbol. Da Γ vier Elemente hat, ergeben sich damit $4 \cdot 4 \cdot (n+1) = 16n + 16$ Symbole. Da am Ende des Beweises noch drei weitere Symbole hinzukamen, ist die Gesamtzahl $16n + 19$.

Die Maschine aus der Lösung von Aufgabe 12.11 hat beispielsweise sechs Zustände. Die Grammatik dafür hätte 115 Nichtterminalsymbole.

Lösung 14.10. Mit einer linear beschränkten Turingmaschine T' kann man eine Turingmaschine T , deren (beidseitig beschränktes) Band k -mal so lang wie die Länge ihrer Eingabe sein darf, simulieren, wenn ihr Bandalphabet Γ^k ist (wobei Γ das Bandalphabet von T sein soll). Man arbeitet dann quasi mit einer (linear beschränkten) *k-Spur-Turingmaschine*, deren Spuren entsprechende Abschnitte des Bandes von T simulieren.

Lösung 14.11. Als Alphabet nehmen wir Σ_{bool} . Die beiden Grammatiken seien

$$G_1: S_1 \rightarrow 1 \quad \text{und} \quad G_2: S_2 \rightarrow 0 \quad 1S_2 \rightarrow 01.$$

Offenbar gilt $L(G_1) \circ L(G_2) = \{1\} \circ \{0\} = \{10\}$. Die Produktion $1S_2 \rightarrow 01$ kann in G_2 nicht verwendet werden. Weil die Mengen der Nichtterminalsymbole bereits disjunkt sind, würden wir nach der erwähnten Methode einfach die Produktion $S \rightarrow S_1 S_2$ hinzufügen und S zum neuen Startsymbol machen. Die so entstandene Grammatik G erlaubt jedoch die Ableitung $S \Rightarrow_G S_1 S_2 \Rightarrow_G 1S_2 \Rightarrow_G 01$.

Der Hinweis diente dazu, Beispiele wie das folgende auszuschließen: Man wähle für $i = 1, 2$ die Grammatik G_i , die nur die eine Produktion $S_i \rightarrow \varepsilon$ hat. Die besagte Methode liefert dann eine Grammatik, die *nicht* kontextsensitiv ist (warum?), aber darum ging es in der Frage nicht.

Lösung 14.12. Als Alphabet nehmen wir wieder Σ_{bool} . G_1 habe die beiden Produktionen $S_1 \rightarrow 0$ und $0S_1 \rightarrow 11$. Offenbar gilt $L(G_1)^+ = \{0\}^+$. Die Produktion $0S_1 \rightarrow 11$ kann nicht verwendet werden. Die Grammatik G erlaubt jedoch die Ableitung $S \Rightarrow_G S_1 S \Rightarrow_G S_1 S_1 \Rightarrow_G 0S_1 \Rightarrow_G 11$.

Lösung 14.13. Das erste Beispiel könnte erweitert so aussehen:

$$G_1: S_1 \rightarrow 1 \quad \text{und} \quad G_2: S_2 \rightarrow 0 \mid 11S_2 \quad 1S_2 \rightarrow 01.$$

Nach wie vor gilt $01 \notin L(G_1) \circ L(G_2) = \{1\} \circ \{0, 101\} = \{10, 1101\}$.

Für das zweite Beispiel könnte man für G_1

$$S_1 \rightarrow 0 \mid 00S_1 \quad \text{und} \quad 0S_1 \rightarrow 11$$

nehmen. $L(G_1)^+$ enthielte dann weiterhin nicht das Wort 11.

Lösung 14.14. Man führt für alle Terminalsymbole a ein neues Nichtterminalsymbol V_a sowie die Produktion $V_a \rightarrow a$ ein. In allen Produktionen von G werden dann alle Terminalsymbole a durch ihre „Stellvertreter“ V_a ersetzt. Aus G_2 in der Lösung von Aufgabe 14.11 würde z. B. $S_2 \rightarrow 0$, $V_1 S_2 \rightarrow 01$, $V_1 \rightarrow 1$ werden.

Lösung 15.2. Offenbar gilt $\chi_{\mathbb{N} \setminus A}(n) = 1 - \chi_A(n)$ für alle $n \in \mathbb{N}$. Kann man die eine der beiden Funktionen berechnen, dann auch die andere.

Lösung 15.3. Das ist einfach. Wenn wir eine Turingmaschine haben, die die charakteristische Funktion einer entscheidbaren Menge berechnet, manipulieren wir sie einfach so, dass sie immer dann in eine Endlosschleife gerät, wenn sie eigentlich null ausgeben würde.

Lösung 15.4. Das ist ähnlich einfach wie Aufgabe 15.3. Wenn wir eine Maschine haben, die A akzeptiert, dann ersetzen wir sie durch eine, die immer dann in eine Endlosschleife gerät, wenn die Originalmaschine in einem nicht akzeptierenden Zustand terminieren würde.

Lösung 15.5. Würde man mit den 0-Instruktionen anfangen, dann finge die zugeordnete Zahl mit mindestens einer Null an. Das wäre mehrdeutig und man könnte daraus die Turingmaschine evtl. nicht wieder rekonstruieren.

Lösung 15.6. Da beim Decodieren der Zahl klar ist, wann man sich beim jeweils dritten Zeichen einer Instruktion befindet, kann man für L, R, S und H auch die Zeichen 0, 1, – und * wiederverwenden. Damit hat man schon vier Zeichen gespart. Und wenn man die Instruktionen konsequent so ausführlich aufschreibt wie im Beispiel, kann man statt $\Downarrow$ auch $|$ schreiben, weil dann jeder vierte senkrechte Strich für einen Zeilenumbruch stehen muss.

Lösung 15.7. Die Begründung liefert Aufgabe 13.9.

Lösung 16.1. Jede Turingmaschine berechnet eine Funktion aus $\mathcal{P}$, weil diese Menge so definiert ist. Im Fall $\mathcal{A} = \emptyset$ wäre die Menge A also die leere Menge und im anderen Fall ganz $\mathbb{N}$. Dass diese beiden Mengen entscheidbar sind, ist offensichtlich.

Lösung 16.2. In beiden Fällen war die Menge H^* aus (16.1) das Problem, das auf das jeweilige Problem aus der Formulierung des Satzes reduziert wurde. Wir haben die Implikation von Lemma 16.5 in den besagten Beweisen immer umgekehrt verwendet: Ist A nicht entscheidbar, dann ist B „erst recht“ nicht entscheidbar.

Lösung 16.3. Die ersten beiden Aussagen sind nicht immer wahr, alle anderen schon.

- (i) Die Aussage gilt z. B. nicht, wenn man als A_2 die Menge H^* aus (16.1) wählt. Sie ist rekursiv aufzählbar, aber nicht entscheidbar. Wäre $\mathbb{N} \setminus H^*$ ebenfalls semi-entscheidbar (also rekursiv aufzählbar nach Satz 15.1), so wäre H^* nach Satz 15.2 entscheidbar.
- (ii) Die Aussage gilt auch nicht allgemein. Man kann für A_2 wieder H^* nehmen und für A_0 z. B. die leere Menge.

- (iii) Die Aussage ist korrekt. Nach Aufgabe 15.2 ist $\mathbb{N} \setminus A_1$ entscheidbar und damit nach Aufgabe 15.3 „erst recht“ semi-entscheidbar.
- (iv) Die Aussage ist korrekt nach Aufgabe 13.8.
- (v) Weil A_0 entscheidbar ist, ist nach Aufgabe 15.2 auch $\mathbb{N} \setminus A_0$ entscheidbar.
- (vi) Weil A_0 endlich ist, ist $A_0 \cap A_2$ natürlich auch endlich und damit nach Aufgabe 13.8 entscheidbar.
- (vii) A_1 ist nach Aufgabe 15.3 semi-entscheidbar.

Lösung 17.1. Einerseits kann man sich eine Menge von (natürlichen) Zahlen als eine Sprache vorstellen; etwa als Menge der Binärdarstellungen dieser Zahlen über dem Alphabet Σ_{bool} . Wenn also beispielsweise eine Turingmaschine eine Menge $A \subseteq \mathbb{N}$ (nicht) entscheiden kann, dann kann sie auch die entsprechende Sprache (nicht) entscheiden.

Andererseits kann man offenbar für jedes Alphabet Σ die Menge Σ^* gödelisieren. Dadurch wird aus einer Sprache L über Σ eine Menge von natürlichen Zahlen. Kann man diese Menge entscheiden oder zumindest rekursiv aufzählen, dann hat man damit auch L entschieden bzw. rekursiv aufgezählt.

Lösung 17.2. Typ-3-Sprachen sind Typ-2-Sprachen, weil direkt aus der Definition folgt, dass reguläre Grammatiken kontextfreie Grammatiken sind. Analog ist sofort klar, dass Typ-1-Sprachen Typ-0-Sprachen sind. Dass jede Typ-2-Sprache eine Typ-1-Sprache ist, wurde in Aufgabe 14.7 begründet.

Für $L_3 = \{a^k b^k : k \in \mathbb{N}\}$ hatten wir mithilfe des Pumping-Lemmas 4.1 gezeigt, dass diese Sprache nicht regulär ist. Die Grammatik $S \rightarrow aSb \mid \varepsilon$ zeigt jedoch, dass es sich um eine kontextfreie Sprache handelt.

Ebenso hatten wir mit dem Pumping-Lemma 10.2 begründet, dass die Sprache $L_2 = \{a^k b^k c^k : k \in \mathbb{N}\}$ nicht kontextfrei ist. Dass L_2 kontextsensitiv ist, wurde in Aufgabe 14.5 gezeigt.

Warum nicht jede rekursiv aufzählbare Sprache kontextsensitiv ist, wurde hinter Satz 17.1 erwähnt.

Lösung 17.3. Die entscheidbaren Sprachen würden zwischen den Typ-0- und den Typ-1-Sprachen liegen. Nach Satz 17.1 ist jede kontextsensitive Sprache entscheidbar und nach Aufgabe 15.3 ist jede entscheidbare Sprache semi-entscheidbar und damit nach Satz 15.1 rekursiv aufzählbar.

In beiden Fällen handelt es sich um echte Inklusionen. Erstens ist z. B. die Menge H^* aus (16.1) rekursiv aufzählbar, aber nicht entscheidbar. Und zweitens zeigt Satz 17.3, dass es entscheidbare Sprachen gibt, die nicht kontextsensitiv sind.

Lösung 17.4. Seien L_1 und L_2 zwei rekursiv aufzählbare Sprachen und T_1 bzw. T_2 Turingmaschinen, die diese Sprachen akzeptieren. O. B. d. A. können wir davon ausgehen, dass diese Maschinen nicht anhalten, wenn ihre Eingabe kein Wort aus L_1 bzw. L_2 ist. (Siehe Aufgabe 15.4.)

Eine akzeptierende Maschine für $L_1 \cap L_2$ verwendet zwei Bänder wie in Aufgabe 12.15. Sie erhält als Eingabe ein Wort w auf dem ersten Band und kopiert dieses auf das zweite. Dann simuliert sie zunächst T_1 auf dem ersten Band. Hält diese Simulation an, so simuliert sie anschließend T_2 auf dem zweiten Band. Hält auch diese Simulation an, so akzeptiert sie. (In allen anderen Fällen hält sie nicht an und das soll sie auch nicht.)

Eine akzeptierende Maschine für $L_1 \cup L_2$ arbeitet sehr ähnlich. Allerdings führt sie abwechselnd auf beiden Bändern immer nur einen Schritt der Simulation von T_1 bzw. T_2 durch. Hält eine von beiden Maschinen an, so akzeptiert die „Vereinigungsmaschine“, anderenfalls nicht.

Lösung 17.5. Siehe Aufgabe 16.3.

Lösung 17.7. $L_1 \cup L_2$, $L_1 \cap L_2$ und $L_1 \circ L_2$ sind offensichtlich endlich, wenn L_1 und L_2 endlich sind. Das Komplement von L_1 ist nicht endlich, weil Σ^* unendlich ist. L_1^* ist nur dann endlich, wenn L_1 die leere Menge ist.

Lösung 17.8. $L_1 \setminus L_2$ ist dasselbe wie $L_1 \cap (\Sigma^* \setminus L_2)$. Wenn eine Sprachklasse also gegen Durchschnitt und Komplement abgeschlossen ist, dann ist sie auch gegen die mengentheoretische Differenz abgeschlossen. Das gilt somit für reguläre und für kontextsensitive Sprachen.

In den anderen beiden Fällen gehört $L_1 \setminus L_2$ jedoch nicht immer zur selben Klasse. Man kann dafür als L_1 jeweils Σ^* nehmen und für L_2 die Beispiele aus Aufgabe 10.11 bzw. Aufgabe 17.5.

Lösung 17.9. L_3 ist als reguläre Sprache insbesondere kontextfrei. Daher ist $L_2 \cup L_3$ nach Satz 10.3 kontextfrei. Außerdem ist $\Sigma^* \setminus L_3$ nach Satz 6.6 regulär und daher auch kontextfrei.

Lösung 17.10. Eine Skizze der Vorgehensweise sieht so aus: $A_3 = (S_3, \Sigma, \delta_3, s_3, F_3)$ sei ein DFA für L_3 . $K_2 = (S_2, \Sigma, \Gamma, \delta_2, s_2, \perp, \emptyset)$ sei ein PDA für L_2 . (Wir akzeptieren wie üblich mit leerem Stack und brauchen daher keine Endzustände.)

Die wesentliche Idee ist, dass der zu konstruierende Kellerautomat ähnlich wie im Beweis von Lemma 5.3 zwei Automaten parallel simuliert, dabei aber nur für einen von beiden einen Stack verwalten muss.

Wir wählen daher als Menge der Zustände für unseren Kellerautomaten $S_2 \times S_3$ zusammen mit zwei neuen Zuständen s_0 und e , wobei s_0 der Startzustand sein soll. Das Kelleralphabet ist $\Gamma \cup \{\perp^*\}$, wobei $\perp^*$ ein neues Symbol ist, das der initiale Stackinhalt wird.

Von s_0 führt ein ε -Übergang zu (s_2, s_3) , der den Stack $\perp^*$ in den Stack $\perp \perp^*$ überführt. Von jedem Zustand (s, t) gibt es nun einen Übergang mit dem Zeichen a zu (s', t') , wenn es entsprechende Übergänge von s zu s' und von t zu t' in K_2 bzw. A_3 gibt. Der Stack wird dabei wie in K_2 gehandhabt. Für ε -Übergänge von s nach s' in K_2 gibt es ε -Übergänge im neuen Automaten von (s, t) nach (s', t) für alle Zustände $t \in S_3$.

Schließlich gibt es für alle akzeptierenden Zustände t aus F_3 und alle Zustände s aus S_2 einen ε -Übergang von (s, t) nach e , der $\perp^*$ vom Stack entfernt.

Lösung 18.2. Zunächst einmal wächst der Logarithmus zwar sehr langsam, ist jedoch streng monoton wachsend und unbeschränkt. Daher gibt es für jedes $k \in \mathbb{N}$ ein $n_0 \in \mathbb{N}$ mit $\ln n \geq k + 1$ und damit $n^{\ln n} \geq n^{k+1}$ für alle $n \geq n_0$. Es gilt also $n^{\ln n} \notin \mathcal{O}(n^k)$ für alle $k \in \mathbb{N}$.

Andererseits erhält man durch zweimalige Anwendung der [Regel von de L'Hospital](#)

$$\lim_{x \rightarrow \infty} \frac{(\ln x)^2}{x} = \lim_{x \rightarrow \infty} \frac{2 \ln x}{x} = \lim_{x \rightarrow \infty} \frac{2}{x} = 0,$$

d. h., für alle $\varepsilon > 0$ gibt es ein $n_0 \in \mathbb{N}$ mit $(\ln n)^2 < n\varepsilon$ und damit

$$n \ln 2 - (\ln n)^2 > n \ln 2 - n\varepsilon = n(\ln 2 - \varepsilon) \quad (\text{L-1})$$

für alle $n \geq n_0$. Insbesondere ist damit der Term auf der linken Seite von (L-1) unbeschränkt und daher ist

$$\frac{2^n}{n^{\ln n}} = \frac{e^{n \ln 2}}{e^{(\ln n)^2}} = e^{n \ln 2 - (\ln n)^2}$$

ebenfalls unbeschränkt und es folgt $2^n \notin \mathcal{O}(n^{\ln n})$. Offenbar gilt dieselbe Aussage auch für andere Basen als 2.

Lösung 18.3. Man könnte statt $\ln$ einen anderen Logarithmus nehmen oder z. B. $n^{\sqrt{n}}$ betrachten.

Lösung 18.5. Nicht unbedingt. Für jede Zelle speichern wir ja ihre Adresse, so dass die Turingmaschine den Speicher ganz „dumm“ durchsuchen kann, indem sie sich alle Zellen nacheinander anschaut. Das haben wir bei unserer groben Abschätzung schon einkalkuliert.

Es wäre allerdings für umfangreiche Programme wohl besser, wenn man sortieren würde, damit die Maschine nicht unnötig viel Zeit damit verbringt, auf dem Speicherband herumzuirren. Dann muss man jedoch beim Einfügen neuer Speicherzellen Teile des Speicherbandes evtl. verschieben, was auch wiederum Aufwand impliziert.³³ Man kann sich jedoch überlegen, dass auch dieser Aufwand nichts am polynomialen Zeitverhalten ändert.

Lösung 18.6. In dieser Lösung wird nur eine Möglichkeit kurz diskutiert. Es geht nicht darum, die perfekte Implementation zu finden. Wichtig sind in diesem Kontext nur die Implikationen für das Laufzeitverhalten.

Bei unbegrenzt großem Speicher (wobei die einzelnen Zellen nach wie vor alle dieselbe Wortbreite haben sollen), können die Befehle der Maschinensprache keine feste Länge mehr haben. Vorstellbar ist etwa, dass man ein Bit „opfert“, das

³³Da man Arbeitsbänder als „Schmierpapier“ zur Verfügung hat, kann man den zu verschiebenden Teil dort zwischenspeichern, wodurch das Verschieben effizienter ist als bei einer Maschine, die nur ein Band hat.

signalisieren kann, dass der Befehl noch nicht vollständig eingelesen wurde und ein weiteres Wort aus dem Speicher geholt werden muss. (Das kann selbstverständlich mehrfach wiederholt werden, wodurch die Befehle beliebig lang werden können.)

Auf die Implementation der Simulation hat das keine wesentlichen Auswirkungen, weil das Speicherband ohnehin schon so angelegt ist, dass keine feste Adressbreite vorausgesetzt wird. Für das Laufzeitverhalten ist allerdings relevant, dass der simulierte Computer diese Erweiterung nicht „umsonst“ bekommen kann. Wir können nicht mehr pauschal einen Taktzyklus pro Befehl rechnen, sondern lange Befehle müssen proportional zu ihrer Länge mehr Zeit verbrauchen.³⁴ Mit etwas Aufwand lässt sich dann zeigen, dass Satz 18.2 nach wie vor gilt.

Lösung 19.1. Das sollten Sie mit Ihren Kenntnissen aus dem ersten Semester beantworten können. Im ungerichteten Fall sind es höchstens $n(n-1)/2$ Kanten, im gerichteten maximal n^2 .

Lösung 19.2. Bisher ging es nur um Entscheidungsprobleme im Sinne der Definition von Seite 105, also um Fragen, die man mit *ja* oder *nein* beantworten kann.

Lösung 19.3. Ein ganz einfaches Beispiel ist die Frage, ob eine Zahl durch 3 oder 9 teilbar ist, die mithilfe der **Quersumme** einfach zu beantworten ist, wenn die Zahl im Dezimalsystem angegeben ist. Um die Frage, was bei der Division durch 3 bzw. 9 herauskommt, zu beantworten, muss man einen höheren Aufwand treiben.

Ein für die Praxis relevantes Beispiel ist die Frage, ob eine Zahl eine Primzahl ist. Dafür gibt es inzwischen **effiziente Testverfahren**. Das **Zerlegen in Primfaktoren** ist jedoch nach aktuellem Forschungsstand deutlich aufwendiger.

Lösung 19.4. Im ersten Graphen hat man 6 Knoten und 4 Wege, im zweiten 8 Knoten und 8 Wege, im dritten 10 Knoten und 16 Wege. Für je zwei Knoten, die hinzukommen, verdoppelt sich die Anzahl der Wege von A nach B . Damit ergibt sich $2^{n/2-1}$ bzw. $1/2 \cdot \sqrt{2}^n$ als Anzahl dieser Wege.

Lösung 19.5. Man kann o. B. d. A. die Eckenmenge eines Graphen mit n Ecken als Menge der natürlichen Zahlen von 0 bis (inklusive) $n-1$ darstellen. Nimmt man nun als Alphabet $\{0, 1, -\}$, wobei $-$ ein Trennzeichen sein soll, dann kann man ein Paar (G, k) wie in der Aufgabenstellung als

$$b(k-1)-b(n-1)-b(v_1)-b(w_1)-\dots-b(v_m)-b(w_m)$$

darstellen. Dabei soll die Funktion b eine natürlich Zahl auf ihre Binärdarstellung abbilden. n soll die Anzahl der Ecken sein und $\{v_1, w_1\}$ bis $\{v_m, w_m\}$ sollen die m Kanten des Graphen sein.

Konkretes Beispiel: Nennen wir in dem Graphen auf Seite 115 die Ecken 0 bis 5 statt A bis F, so würde die Frage, ob dieser Graph eine Clique mit 4 Ecken hat, durch das Wort

11|101|0|1|0|10|1|10|1|11|1|100|1|101|10|100|10|5|11|100|100|101

³⁴Das ist ja auch realistisch, weil der Prozessor die Befehle aus dem Speicher holen muss.

codiert werden. Das Entscheidungsproblem wäre dann die Sprache, die aus allen Wörtern dieser Art besteht.

Es sei angemerkt, dass es im Allgemeinen nicht reicht, nur die Kanten eines Graphen zu speichern, weil man dann die Information über die Ecken verlieren würde, die nicht mit anderen verbunden sind. Beim Cliques-Problem braucht man die zwar nicht, aber bei anderen Problemen kann diese fehlende Information relevant sein.

Lösung 19.6. Der wesentlich Teil der Antwort steht bereits im Text vor der Aufgabe. $\binom{n}{k}$ ist für festes k ein Polynom k -ten Grades in n . Beispielsweise gilt

$$\binom{n}{2} = 1/2 \cdot (n^2 - n) \quad \text{und} \quad \binom{n}{3} = 1/6 \cdot (n^3 - 3n^2 + 2n).$$

Dieses Polynom determiniert die maximale Anzahl der äußeren Schleifendurchläufe. Die innere **for**-Schleife wird maximal n^2 -mal durchlaufen, denn es gilt $|V' \times V'| = k^2 \leq n^2$. Und auch der Test, ob v und w durch eine Kante verbunden sind, kann selbst bei ungünstiger Programmierung höchstens $n \times n$ Durchläufe durch eine Schleife verbrauchen.

Lösung 19.7. Codiert man so wie in Aufgabe 19.5, so benötigt man bei n Ecken für jede Zahl maximal $\lceil \log_2 n \rceil$ Bits für ihre Binärdarstellung. Das kann man ganz großzügig durch n nach oben abschätzen. Die Anzahl der Kanten (siehe Aufgabe 19.1) ist $n(n-1)/2$ und damit auf jeden Fall nicht größer als n^2 . Inklusive Trennzeichen braucht man also (deutlich) weniger als $(n^2 + 2)(n + 1)$ Zeichen für ein Wort w , wenn es um einen Graphen mit n Ecken geht. Ist p das Polynom aus der Aufgabenstellung, so ergibt sich $\text{time}_T(w) \leq p((n^2 + 2)(n + 1))$, wobei der rechte Term ein Polynom in n ist.³⁵

Für die zweite Frage stelle man sich den Extremfall eines Graphen mit n Ecken vor, bei dem $\{0, 1\}$ die einzige Kante ist. In Abhängigkeit von n kann $|w|$ dann beliebig groß werden, während $|E|$ konstant den Wert 1 hat. Offenbar kann es also kein Polynom wie in der Aufgabenstellung geben.

Lösung 19.10. Nein. Im Prinzip macht das Programm das zwar, aber man muss sich an dieser Stelle daran erinnern, wie nichtdeterministische Turingmaschinen in Abschnitt 14 definiert wurden: Es gibt Wahlmöglichkeiten, aber jeweils nur endlich viele. Insbesondere stehen die Wahlmöglichkeiten *vor* dem Start der Maschine fest und hängen damit *nicht* von der Eingabe ab. Im konkreten Algorithmus gibt es immer nur zwei Möglichkeiten und das ist OK. Würde man jedoch einen Algorithmus aufschreiben, in dem das Programm sich eine k -elementige Teilmenge von V aussuchen darf, dann würde die Anzahl der Wahlmöglichkeiten von n und k abhängen. Der **select**-Block hätte je nach Eingabe eine andere Laufzeit.

Lösung 19.11. Das Programm muss für alle zweielementigen Teilmengen von V' testen, ob sie zu E gehören. Wenn wir davon ausgehen, dass ein entsprechender

³⁵Wir setzen dabei voraus, dass p **monoton steigt**, was zumindest für größere Argumente zu erwarten ist.

Zugriff auf E eine Laufzeit von $\mathcal{O}(1)$ hat (wenn E beispielsweise als zweidimensionales **Array** abgespeichert ist), ist nur die Anzahl solcher Teilmengen relevant. Und die ist $(k^2 - k)/2$, also auf jeden Fall nicht größer als n^2 .

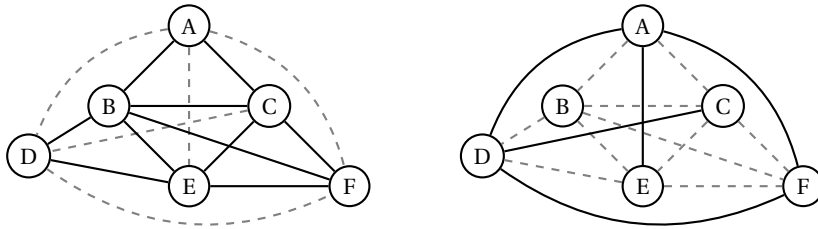
Lösung 19.12. Man muss es für dieses konkrete Programm nicht so kompliziert wie im Beweis des Satzes machen, weil es nur eine einzige nichtdeterministische Stelle im Algorithmus gibt. Aber in jedem **select**-Block werden aus einer Simulation zwei, so dass das deterministische Programm im Endeffekt 2^n Simulationen durchspielen muss. Wenn jede von denen ein Laufzeitverhalten von $\mathcal{O}(n^2)$ hat, ergibt sich insgesamt $\mathcal{O}(n^2 2^n)$.

Lösung 19.13. Ist $G = (V, E)$ ein Graph, dann bezeichnet man den Graphen

$$(V, \{\{v, w\} : v, w \in V \text{ und } v \neq w \text{ und } \{v, w\} \notin E\})$$

als den *Komplementgraphen* von G . Das ist also der Graph mit denselben Knoten, der genau dort Kanten hat, wo G keine hat.

Komplementgraph



Es ist hoffentlich offensichtlich, dass V' genau dann eine Clique in G ist, wenn V' eine stabile Menge im Komplementgraphen von G ist. Und das gilt auch umgekehrt, weil der Komplementgraph des Komplementgraphen von G natürlich wieder G ist.

Lösung 19.14. Im ersten **if**-Block muss $\{v, w\} \notin E$ durch $\{v, w\} \in E$ ersetzt werden. Alles andere kann so bleiben.

Lösung 19.16. Das ist ganz einfach. Ein Zertifikat wäre einfach eine Clique der Größe k .

Lösung 20.2. Das sieht so aus:

$$\begin{aligned} & (x_1 \vee x_2 \vee x_3 \vee x_4) \\ & \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \\ & \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4) \\ & (\neg s_{42,17} \vee \neg b_{42,17,23} \vee \neg z_{42,5} \vee s_{43,18}) \\ & \wedge (\neg s_{42,17} \vee \neg b_{42,17,23} \vee \neg z_{42,5} \vee b_{43,17,12}) \\ & \wedge (\neg s_{42,17} \vee \neg b_{42,17,23} \vee \neg z_{42,5} \vee z_{43,3}) \end{aligned}$$

Literatur

Im Folgenden führe ich ergänzend zum Skript diverse Lehrbücher zur Theoretischen Informatik auf. Es ist sicher eine gute Idee, sich ein paar dieser Bücher einmal anzuschauen. Vielleicht verstehen Sie die Erklärungen dort besser, vielleicht finden Sie weitere Aufgaben oder vielleicht werden Themen, die hier zu kurz kommen, dort ausführlicher behandelt. Die Links führen jeweils zu den Seiten der Verlage und die Bücher sind danach ausgewählt, dass man Sie als e-Books aus dem Netz der HAW Hamburg (und wohl auch aus den Netzen vieler anderer deutscher Hochschulen) kostenlos herunterladen oder lesen kann.

Das Buch von Hopcroft et al. ist seit Jahren eine Art Standard für Lehrbücher in diesem Bereich und besonders umfangreich. Mir persönlich gefällt zum Lesen am besten das Buch von Hoffmann. Das Buch von Wagenknecht und Hielscher behandelt nur einen Teilbereich des in diesem Skript behandelten Stoffs, geht dafür aber ausführlicher auf Anwendungen ein und basiert durchgehend auf der für das Buch entwickelten Lernsoftware [FLACI](#).

- Alexander Asteroth, Christel Baier: [Theoretische Informatik – Einführung in Berechenbarkeit, Komplexität und formale Sprachen](#), Pearson Deutschland, 2002.
- Norbert Blum: [Theoretische Informatik – Eine anwendungsorientierte Einführung](#), De Gruyter Oldenbourg, 2014.
- Ulrike Hedtstück: [Einführung in die Theoretische Informatik – Formale Sprachen und Automatentheorie](#), Oldenbourg, 2012.
- Dirk W. Hoffmann: [Theoretische Informatik](#), Hanser, 2011.
- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: [Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit](#), Pearson Studium, 2011.
- Juraj Hromkovič: [Theoretische Informatik – Formale Sprachen, Berechenbarkeit, Komplexitätstheorie, Algorithmik, Kommunikation und Kryptographie](#), Springer Vieweg, 2014.
- Lukas König, Friederike Pfeiffer-Bohnen, Hartmur Schmeck: [Theoretische Informatik – ganz praktisch](#), De Gruyter Oldenbourg, 2016.

- Lutz Prieße, Katrin Erk: [Theoretische Informatik – Eine umfassende Einführung](#), Springer Vieweg, 2018.
- Uwe Schöning: [Ideen der Informatik – Grundlegende Modelle und Konzepte der Theoretischen Informatik](#), Oldenbourg, 2009.
- Gottfried Vossen, Kurt-Ulrich Witt: [Grundkurs Theoretische Informatik – Eine anwendungsbezogene Einführung](#), Springer Vieweg, 2016.
- Christian Wagenknecht, Michael Hielscher: [Formale Sprachen, abstrakte Automaten und Compiler](#), Springer Vieweg, 2022.
- Renate Winter: [Theoretische Informatik – Grundlagen mit Übungsaufgaben und Lösungen](#), Oldenbourg, 2009.

Unter der URL <https://weitz.de/haw-videos/> finden Sie unter anderem Aufzeichnungen dreier kompletter Vorlesungen zur Theoretischen Informatik aus den Jahren 2013 bis 2015 – insgesamt 286 Videos. Dort werden eventuell manche Dinge anders erklärt als in der aktuellen Veranstaltung oder Sie finden Zusatzinformationen für Themen, die Sie besonders interessieren. (Falls Sie sich intensiver mit Komplexitätstheorie auseinandersetzen wollen, empfehle ich insbesondere die Vorlesung aus dem Sommersemester 2014.)

Und schließlich noch das Buch, in dem Sie nachschlagen sollten, wenn Ihnen hier verwendete mathematische Begriffe nicht geläufig sind. Mehr dazu in der [Gebrauchsanweisung](#).

- Edmund Weitz: [Konkrete Mathematik \(nicht nur\) für Informatiker](#), Springer Spektrum, 2021.

Alternativ können Sie sich auch das aktuelle [Skript zur Mathevorlesung](#) herunterladen, aber das Buch ist wesentlich umfangreicher und geht spezifischer auf Informatik-Anwendungen ein.



Index

- 2-SAT**, 125
- 3-SAT**, 124
- abgeschlossen, 24
- Ableitungsbaum, 13
- akzeptierender Zustand, *siehe*
Zustand, akzeptierender
- akzeptierte Sprache, *siehe* Sprache,
akzeptierte
- Alphabet, 5
- äquivalent, 40
- Äquivalenzklasse, 38
- Äquivalenzrelation, 38
- Ausdruck, regulärer, 31
- Ausgabe, 70
- Automat
 - endlicher, 20
 - nichtdeterministischer
 - endlicher, 25
- Band einer Turingmaschine, 64
- Bandalphabet, 68
- Bandende, 86
- berechenbar, 78, 91
- Berechenbarkeitstheorie, 91
- berechnen, 70, 91
- berechnete Funktion, 71
- beschränkter Kellerautomat, 57
- Binärbaum, 142
- BOOLE, GEORGE, 5
- CANTOR, GEORG, 16
- charakteristische Funktion, 92
- CHOMSKY, NOAM, 11
- Chomsky-Hierarchie, 17, 84
- Chomsky-Normalform, 48
- CHURCH, ALONZO, 77
- Church-Turing-These, 77
 - erweiterte, 110
- CLIQUE**, 114
- Cliquenproblem, 114
- CNF-SAT**, 122
- Codierung, 110
- CONWAY, JOHN, 77
- COOK, STEPHEN, 120
- CYK-Algorithmus, 50
- DAVEY, MIKE, 72
- DE MORGAN, AUGUSTUS, *siehe*
Morgan, Augustus De
- DFA, 21
- DIJKSTRA, EDSGER, 99, 114
- Dijkstra-Algorithmus, 99
- DYCK, WALTHER VON, 12
- Dyck-Sprache, 12
- Ecke, 111
- Eingabe, 70
- Eingabealphabet, 20, 53, 68
- endlicher Automat, 20
- Endzustand, 21, 54, 68
- entscheidbar, 78, 92
- Entscheidungsproblem, 105
- ϵ -Sonderregel, 83
- ϵ -Übergang, 28

- Erfüllbarkeitsproblem der
 - Aussagenlogik, *siehe* **SAT**
- erstes LBA-Problem, 87
- Erweiterte Church-Turing-These, 110
- erzeugte Sprache, 12
- exponentiell, 105
- Fehlerzustand, 22
- FLACI, 10
- formale
 - Grammatik, 11
 - Sprache, 8
- Fraclan, 78
- FSM, 21
- Funktion
 - berechenbare, 78, 91
 - charakteristische, 92
 - partielle, 68
 - totale, 68
- Funktionsproblem, 112
- Game of Life*, *siehe* Spiel des Lebens
- gerichteter Graph, 111
- Gewicht, 111
- gewichteter Graph, 111
- gleichmäßiges Halteproblem, 97
- GÖDEL, KURT, 77, 79
- Gödelisierung, 79
 - kanonische, 80
- Gödelnummer, 79
- Grammatik, 11
 - kontextfreie, 47
 - kontextsensitive, 83
 - monotone, 83
 - reguläre, 17
- Graph, 111
 - gerichteter, 111
 - gewichteter, 111
- Grenzmarkierung, 68
- Halteproblem, 96
 - gleichmäßiges, 97
- HERBRAND, JACQUES, 77
- herleiten, 12
- Hülle
 - kleenesche, 8
 - positive, 8
 - reflexive und transitive, 38
- „im Allgemeinen nicht“, 143
- IMMERMAN, NEIL, 89
- in Polynomialzeit verifizierbar, 118
- Indikatorfunktion, 92
- INDSET**, 117
- initialer Stackinhalt, 54
- kanonische Gödelisierung, 80
- Kante, 111
- Kelleralphabet, 53
- Kellerautomat, 53
 - beschränkter, 57
 - deterministischer, 59
 - mit zwei Stacks, 73
- Kellerspeicher, 53
- Klausel, 122
- KLEENE, STEPHEN COLE, 8, 64
- kleenesche Hülle, 8
- KMFI, 1
- Knoten, 111
- Komplementgraph, 163
- Komplexitätstheorie, 103
- Konfiguration, 54, 69
- konjunktive Normalform, 122
- Konkatenation, 6
- kontextfreie
 - Grammatik, 47
 - Sprache, 47
- kontextsensitive
 - Grammatik, 83
 - Sprache, 83
- Künstliche Intelligenz, 77
- LANDAU, EDMUND, 104
- Landau-Symbole, 104
- Länge, 6, 112
- LBA, 86
- LBA-Problem, 87
- leeres Wort, 6
- Leerzeichen, 68
- LEVIN, LEONID, 120
- linear beschränkt, 86
- Literal, 122
- Lösung, 96, 105

- Maschinenbeschreibung, 94
- Mehrband-Turingmaschine, 72
- Mehrspur-Turingmaschine, 71
- Menge
 - akzeptierte, 92
 - entscheidbare, 92
 - rekursiv aufzählbare, 92
 - rekursive, 92
 - semi-entscheidbare, 92
- Miller-Rabin-Test, 25
- Minecraft*, 77
- Minimalautomat, 43
- MINSKY, MARVIN, 76
- monotone Grammatik, 83
- MORGAN, AUGUSTUS DE, 25
- MYHILL, JOHN, 40
- NERODE, ANIL, 40
- Nerode-Relation, 40
- NFA, 25
- Nichtdeterminismus, 25
- nichtdeterministische
 - Turingmaschine, 82
- nichtdeterministischer
 - endlicher Automat, 25
 - Kellerautomat, *siehe*
 - Kellerautomat
- Nichtterminalsymbole, 11
- Normalform, konjunktive, 122
- NP-schwer, 120
- NP-vollständig, 120
- o. B. d. A., 16
- Optimierungsproblem, 112
- partielle Funktion, 68
- PDA, 53
- Phrasenstrukturgrammatik, 83
- polynomial, 105
- polynomial reduzierbar, 118
- positive Hülle, 8
- präfixfrei, 147
- PRESBURGER, MOJŻESZ, 125
- Presburger-Arithmetik, 125
- Problem, 96
- Produktautomat, 25
- Produktion, 11
- Pumping-Lemma
 - für kontextfreie Sprachen, 50
 - für reguläre Sprachen, 18
- Quantencomputer, 77, 110, 125
- RABIN, MICHAEL, 25
- reduzierbar, 98
 - polynomial, 118
- reflexiv, 37
- reflexive und transitive Hüllen, 38
- reflexive Hülle, *siehe* Hülle, reflexive und transitive
- reguläre
 - Grammatik, 17
 - Sprache, 17
- regulärer Ausdruck, 31
- Rekursionstheorie, *siehe*
 - Berechenbarkeitstheorie
- rekursiv, 78, 92
 - aufzählbare Menge, 92
 - aufzählbare Sprache, 83
- Relation, 37
- RICE, HENRY GORDON, 98
- SAT**, 119
- Satz
 - von Cantor, 16
 - von Cook und Levin, 120
 - von Immerman und Szelepcsényi, 89
 - von Myhill-Nerode, 42
 - von Rabin und Scott, 27
 - von Rice, 97
- Schreib-Lese-Kopf, 64
- SCHÜTZENBERGER, MARCEL, 17
- SCOTT, DANA, 25
- semi-entscheidbar, 92
- Spiel des Lebens*, 77
- Sprache, 8
 - akzeptierte, 21, 54, 57, 81
 - entscheidbare, 78
 - erzeugte, 12
 - kontextfreie, 47
 - kontextsensitive, 83
 - reguläre, 17
 - rekursiv aufzählbare, 83

- rekursive, 78
- stabil, 117
- Stabilitätsproblem, 117
- Stack, 53
- Stackalphabet, 53
- Stackinhalt
 - initialer, 54
- Standardaufzählung, 94
- Stapelspeicher, 53
- Startsymbol, 11
- Startzustand, 20, 54, 68
- subexponentiell, 105
- Suchproblem, 112
- superpolynomial, 105
- Symbol, 5
- symmetrisch, 37
- SZELEPCSÉNYI, RÓBERT, 89
- Terminalsymbole, 11
- terminieren, 70
- totale Funktion, 68
- transitiv, 37
- transitive Hülle, *siehe* Hülle, reflexive
 - und transitive
- TURING, ALAN, 63
- Turingmaschine, 68
 - linear beschränkte, 86
 - nichtdeterministische, 82
 - universelle, 95
 - turingvollständig, 77
- Typ-0-Grammatik, 83
- Typ-1-Grammatik, 83
- Typ-2-Grammatik, 47
- Typ-3-Grammatik, 17
- Übergangsfunktion, 20, 53, 68
- unabhängig, 117
- ungerichteter Graph, *siehe* Graph
- universelle Turingmaschine, 95
- verifizierbar, in Polynomialzeit, 118
- Vokabular, 11
- VON DYCK, WALTHER, *siehe* Dyck, Walter von
- Weg, 112
- worst case*, 106
- Wort, 6
 - leeres, 6
- Wortproblem, 99
- ZAPPA, FRANK, 6
- Zeichen, 5
- ZEITIN, GRIGORI, 124
- Zeitin-Transformation, 124
- Zertifikat, 119
- Zustand, 20, 53, 68
 - akzeptierender, 21, 54, 68

Mathematische Symbole

Es ergibt wenig Sinn, mathematische Symbole alphabetisch zu sortieren. Daher werden sie in der folgenden Liste einfach in der Reihenfolge aufgeführt, in der sie definiert bzw. eingeführt wurden.

Σ_{bool} , 5	$[a]_R$, 38
Σ_{lat} , 5	$\sim_L$, 40
$\mathbb{N}$, 5	$[w]_L$, 40
$\mathbb{N}^+$, 5	$\mathcal{P}_{<\infty}(A)$, 54
ε , 6	$\Rightarrow_K$, 54
$ w $, 6	$\Rightarrow_K^*$, 54
$ w _x$, 6	$L(K)$, 54
$v \circ m$, 6	$L^*(K)$, 57
w^k , 6	$\perp$, 68
$X \circ Y$, 7	$f: A \rightarrow B$, 68
X^k , 7	$\Rightarrow_T$, 69
X^* , 8	$v[s]w$, 69
X^+ , 8	$\Rightarrow_T^*$, 69
n_w , 10	f_T , 71
$\Rightarrow_G$, 12	$L(T)$, 81
$\Rightarrow_G^*$, 12	$T(n)$, 91
$L(G)$, 12	χ_A , 92
D_1 , 14	M_i , 95
δ^* , 21	H , 96
$L(A)$, 21	B^A , 104
$\alpha\beta$, 31	$\mathcal{O}(f)$, 104
$(\alpha + \beta)$, 31	$\text{time}_T(w)$, 106
$(\alpha)^*$, 31	$\text{ntime}_T(w)$, 106
$\mathcal{L}(\alpha)$, 31	$\text{TIME}(f)$, 106
$\alpha \equiv \beta$, 32	$\text{NTIME}(f)$, 106
xRy , 37	$\leq_p$, 118
w^R , 37	