Edmund Weitz

THEORETISCHE INFORMATIK

(Skript für den Studiengang *Media Systems* an der HAW Hamburg)

14. Mai 2026, 21:17

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-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 Andrin Azzola, Jörg Balzer, Marvin Behrmann, Justus Boos, Ulrich Dorobeck, Alexander Farwer, Tom Kalberg, Lionel Kaplanski, Oskar Kotte, Lucy Mäge, Marco Ramón, Thomas Schneider, Konrad Smyk, Francesco Vinciguerra, Zanna Wahedi, Helmut Wollmersdorfer und Jonathan Wübbe.

Inhaltsverzeichnis

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 in meinem Buch *Konkrete Mathematik (nicht nur) für Informatiker* nach.[1] Zu dem 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

Aufgaben

Im Skript finden Sie einige Aufgaben und zusätzlich erhalten Sie regelmäßig Hausaufgaben, die in der folgenden Woche in den Vorlesungen besprochen werden. Die 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.

[1] Alternativ werden Sie die meisten Inhalte auch in meinem aktuellen Mathe-Skript finden.

(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. 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 von der Vorlesung zu diesem Skript, die im Wintersemester 2023/2024 aufgenommen wurden. 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.

Warum der Titel „Theoretische Informatik“?

Im Modulhandbuch heißt diese Veranstaltung offiziell *Mathematische Methoden der Informatik*. Es ist allerdings das Modul, das nach der vorherigen Studienordnung *Theoretische Informatik* hieß. Und auch in der kommenden Studienordnung wird es wieder unter dieser Bezeichnung laufen, weil Media Systems in Zukunft Medieninformatik heißen wird und Theoretische Informatik zum Standardstoff jedes Informatikstudiums gehört. Im Vorgriff auf diesen Wechsel übernehme ich jetzt bereits den zukünftigen Namen.

I

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 Rechenvorgang?“ 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

Alphabet
Symbol Zeichen

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

Σ_{bool}

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.

Σ_{lat}

Ein weiteres Beispiel ist die Menge $\Sigma_{\text{lat}} = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$, die aus den 26 lateinischen Buchstaben $\mathtt{a}$ bis $\mathtt{z}$ besteht und die wohl am ehesten der klassischen Vorstellung eines Alphabets entspricht.

Aufgabe 1.1. Warum ist die Menge $\mathbb{N}$ der natürlichen Zahlen[2] kein Alphabet?

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

[2] In 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 $(\mathtt{z},\mathtt{a},\mathtt{p},\mathtt{p},\mathtt{a})$ oder $(\mathtt{x},\mathtt{x})$ 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 *leere Wort* und schreibt dafür ε.[4] Beachten Sie, dass ε in diesem Sinne *kein* Symbol ist! (Und sinnvollerweise verwenden wir auch keine Alphabete, in denen ein Symbol ε vorkommt …)

Länge $|w|$

$|w|_x$

Ist ein Wort w ein n-Tupel, so bezeichnen wir n als seine *Länge* und schreiben dafür $|w|$. Die Länge von `zappa` ist also 5 und wir würden $|\mathtt{zappa}| = 5$ 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. $|\mathtt{00101}|_{\mathtt{0}} = 3$ und $|\mathtt{00101}|_{\mathtt{1}} = 2$.

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

Definition

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

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

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 $\mathtt{0} \circ \mathtt{0}$ 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

[3] Diese 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 $(\mathtt{1},\mathtt{11})$ und $(\mathtt{11},\mathtt{1})$ zwei *verschiedene* Wörter über diesem Alphabet, die man in der Schreibweise `111` nicht unterschieden könnte. Wir werden jedoch immer mit *einstelligen* Symbolen arbeiten, bei denen so etwas nicht passieren kann.

[4] In manchen Büchern wird es auch mit λ bezeichnet.

[5] Die Länge ist also *nicht* 3, denn es ist nicht die Mächtigkeit der Menge $\{\mathtt{z},\mathtt{a},\mathtt{p},\mathtt{p},\mathtt{a}\}$ 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.

[6] Ein 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 $\{\mathtt{a}, \mathtt{b}\}$ und v das Wort $\mathtt{ab}$ über Σ. Geben Sie zur Übung $\mathtt{a}^4$, v^3, $\mathtt{a}^2 v^2$, $\mathtt{ab}^3$, $(\mathtt{ab})^3$ 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?

Definition

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 $X^{k+1} = X^k \circ X$.

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

Aufgabe 1.7. Sei $\Sigma = \{\mathtt{a}, \mathtt{b}\}$. Geben Sie Σ^3 an.

Aufgabe 1.8. Wie kann man die Menge Σ^k mit Worten[9] beschreiben, wenn Σ ein Alphabet ist?

[7] Das ist dieselbe Regel wie bei der normalen Multiplikation.

[8] Rein 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.

[9] Damit 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 1.9. Wie viele Elemente enthält Σ^k, wenn Σ ein Alphabet ist?

Aufgabe 1.10. Wie kann man X^1 kürzer schreiben?

Aufgabe 1.11. 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 1.12. Wie kann man die Menge Σ^* mit Worten beschreiben, wenn Σ ein Alphabet ist?

Aufgabe 1.13. Sei $X = \{\mathtt{a}\}$ und $Y = X \cup \{\varepsilon\}$. Wie sehen X^k und Y^k für $k \in \mathbb{N}$ aus und wie X^* und Y^*?

Aufgabe 1.14. Aus Aufgabe 1.8 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 1.15. Wie lang ist das längste Wort, das in 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}.

[10] Benannt 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.

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. $n_{101010} = 42$. n_w

Im ersten Beispiel geht es um die Sprache

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

Die simple Frage, ob diese Sprache endlich oder unendlich ist, konnte bis heute (Stand Februar 2024) 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

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

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

Aufgabe 1.17. Warum steht in (1.1) n_{1w} und nicht einfach n_w?

Aufgabe 1.18. 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 1.19. 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.

2. Grammatiken

Von dem amerikanischen Linguisten Noam Chomsky stammt die Idee, Sprachen durch formale Grammatiken zu beschreiben. Seine erste Veröffentlichung dazu

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

stammt aus dem Jahr 1956. Chomsky ging es eigentlich um natürliche Sprachen, aber seine Theorie wird heutzutage hauptsächlich in der Informatik eingesetzt.

Definition

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.

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

$$P = \{(X, \varepsilon), (X, \mathtt{a}), (X\mathtt{a}, \mathtt{b}Y), (Y, X\mathtt{a})\}.$$

Aufgabe 2.1. Wieso wäre das kein Beispiel für eine Grammatik mehr, wenn man in P das Paar $(X\mathtt{a}, \mathtt{b}Y)$ durch $(\mathtt{a}, \mathtt{b}Y)$ ersetzen würde?

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

Konvention

Wir verwenden im Zusammenhang mit Grammatiken immer kursive Großbuchstaben für Nichtterminalsymbole und Kleinbuchstaben (sowie evtl. Ziffern und Sonderzeichen) in Schreibmaschinenschrift für Terminalsymbole. Ferner schreiben wir Produktionen (α, β) in der Form $\alpha \to \beta$ auf und fassen mehrere Produktionen $\alpha \to \beta_1$ bis $\alpha \to \beta_n$ mit derselben linken Seite abkürzend als $\alpha \to \beta_1 \mid \cdots \mid \beta_n$ zusammen. Die Mengen der Terminal- bzw. Nichtterminalsymbole werden sich implizit durch das Aufschreiben der Produktionen ergeben.[12] Wir legen außerdem fest, dass S das Startsymbol ist, wenn nicht explizit ein anderes Zeichen als Startsymbol deklariert wird.

Die Beispielgrammatik G von eben würden wir also in der Form

$$\begin{aligned} X &\to \varepsilon \mid \mathtt{a} \\ X\mathtt{a} &\to \mathtt{b}Y \\ Y &\to X\mathtt{a} \end{aligned} \tag{2.1}$$

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

Wofür Grammatiken verwendet werden, beschreibt die nächste Definition.

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

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 $\gamma_1, \ldots, \gamma_n \in (N \cup T)^*$ mit $\gamma_1 = \alpha$, $\gamma_n = \beta$ und $\gamma_i \Rightarrow_G \gamma_{i+1}$ für $i = 1, \ldots, n-1$ gibt.[13] 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 $\gamma \to \delta$ eine der Produktionen ist. Im Beispiel (2.1) würde unter anderem $\mathtt{b}Y\mathtt{b} \Rightarrow_G \mathtt{b}X\mathtt{ab}$ gelten, weil wir die Produktion $Y \to X\mathtt{a}$ haben. Und auch $\mathtt{b}X\mathtt{ab} \Rightarrow_G \mathtt{bb}Y\mathtt{b}$ wegen der Produktion $X\mathtt{a} \to \mathtt{b}Y$. 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.[14] In unserem Beispiel haben wir uns gerade überlegt, dass $\mathtt{b}Y\mathtt{b} \Rightarrow_G^* \mathtt{bb}Y\mathtt{b}$ gilt, weil wir in zwei Schritten vom einen Wort zum anderen kommen. Wir sagen in so einem Fall, dass $\mathtt{bb}Y\mathtt{b}$ in G aus $\mathtt{b}Y\mathtt{b}$ *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 werden. Die Terminalsymbole sind hier die beiden Klammerzeichen `[` und `]`, das einzige Nichtterminalsymbol ist S und es gibt drei Produktionen:

Dyck-Sprache

$$S \to \varepsilon \mid SS \mid \mathtt{[}S\mathtt{]} \tag{2.2}$$

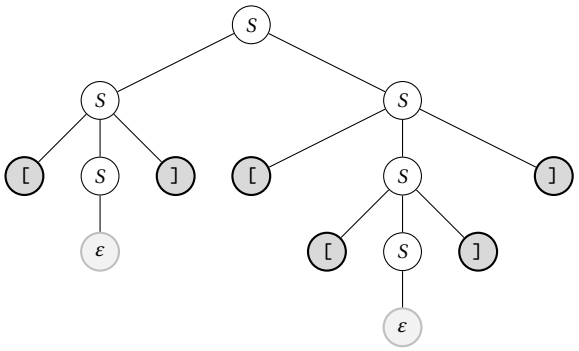
Ein Wort, das man herleiten kann, ist `[][[]]`, und zwar z. B. folgendermaßen:

$$S \Rightarrow_G SS \Rightarrow_G \mathtt{[}S\mathtt{]}S \Rightarrow_G \mathtt{[]}S \Rightarrow_G \mathtt{[]}\mathtt{[}S\mathtt{]} \Rightarrow_G \mathtt{[]}\mathtt{[[}S\mathtt{]]} \Rightarrow_G \mathtt{[]}\mathtt{[[]]} \tag{2.3}$$

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

[13] Wir setzen dabei $n \geq 1$ voraus, erlauben also auch $\alpha = \beta$ ohne Anwendung einer Produktion.
[14] $\Rightarrow_G$ ist also eine Relation auf $(N \cup T)^*$ und $\Rightarrow_G^*$ deren reflexive und transitive Hülle.
[15] Benannt nach dem Mathematiker Walther von Dyck.



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

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

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?

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 eignen 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 $\{\texttt{a}\}^*$ erzeugt.

Aufgabe 2.9. Erzeugt die Grammatik $S \to \varepsilon \mid \texttt{ab}S$ die Sprache $\{\texttt{a},\texttt{b}\}^*$?

Aufgabe 2.10. Geben Sie für die Grammatik (2.4) möglichst viele verschiedene Herleitungen für das Wort `x=y+(z*x)` an.

Aufgabe 2.11. Mit der Lernsoftware FLACI (siehe Aufgabe 1.19) kann man auch mit Grammatiken rumexperimentieren. Ich empfehle Ihnen, das auch zu tun, um eine

[16] Beachten 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*).

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 2.12**. Beschäftigen Sie sich intensiver mit der Grammatik (2.4). Leiten Sie einige Ausdrücke ab und zeichnen Sie dazu die Ableitungbäume. 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.)

Aufgabe 2.13. 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.

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.[18]

Satz 3.1

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

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

[17] Allerdings 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.

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

offenbar keine Rolle für die resultierende Sprache. Wir können also o. B. d. A.[19] 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:[20]

S_1 ◇ ◇ S_1 ◇ a ◇ S_1 a ◇ b S_2 ◇ S_2 ◇ S_1 a

Dabei fungiert das Zeichen ◇ 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 ◇ 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:[21] Σ enthält mindestens ein Zeichen a und damit bereits die Wörter a, a^2, a^3 und so weiter. Daher ist die Menge $\mathscr{P}(\Sigma^*)$ 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.

[19] Diese 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.

[20] Vereinbarungsgemäß „heißt“ X nun S_1 und Y heißt S_2.

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

III

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

Definition

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.

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:

$$\begin{aligned} S &\to \varepsilon \mid \mathtt{a}B \\ B &\to \mathtt{b}C \\ C &\to \mathtt{b}S \end{aligned} \tag{4.1}$$

[1] Manchmal 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 $(\mathtt{abb})^3$. 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

$$S \to \varepsilon \mid \mathtt{a}S \mid \mathtt{b}T$$
$$T \to \mathtt{c}$$

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

Aufgabe 4.5. Die Grammatik

$$S \to \varepsilon \mid \mathtt{a}T$$
$$T \to \mathtt{ab}S$$

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

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

Beachten Sie den feinen Unterschied zur Sprache $\{(\mathtt{ab})^k : k \in \mathbb{N}\}$. Die ist nach den Aufgaben 2.9 und 4.5 offensichtlich regulär!

Korollar 4.2

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.

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.

[3] Dass es sich nicht um ein hinreichendes Kriterium handelt, wird in Aufgabe 8.10 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. 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.

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**[5] (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.

(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.

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

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

DFA FSM

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

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

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

akzeptierte Sprache

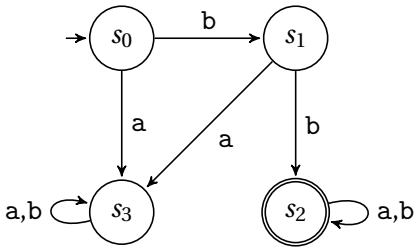
$$L(A) = \{\, w \in I^* : \delta^*(s_0,w) \in F \,\}$$

$L(A)$

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 $a = a\varepsilon$ 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:[6]



	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 $\{\mathtt{a}, \mathtt{b}\}$, 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.

$$\begin{aligned} \delta^*(s_0,\mathtt{bb}) &= \delta^*(\delta(s_0,\mathtt{b}),\mathtt{b}) = \delta^*(s_1,\mathtt{b}) = s_2, \\ \delta^*(s_1,\mathtt{ba}) &= \delta^*(\delta(s_1,\mathtt{b}),\mathtt{a}) = \delta^*(s_2,\mathtt{a}) = s_2 \quad \text{und} \\ \delta^*(s_0,\mathtt{bba}) &= \delta^*(\delta(s_0,\mathtt{b}),\mathtt{ba}) = \delta^*(s_1,\mathtt{ba}) = s_2. \end{aligned}$$

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“.

[6] Und 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.

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

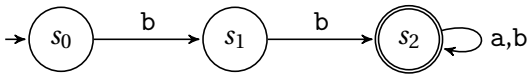
Da der Definitionsbereich der Übergangsfunktion δ die Menge $S \times I$ ist, muss im Prinzip in der grafischen Darstellung von *jedem* Zustand aus S für *jedes* Zeichen aus I ein Pfeil ausgehen. Die Darstellung von umfangreicheren Automaten wird dadurch jedoch oft unnötig unübersichtlich. Viele Pfeile 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

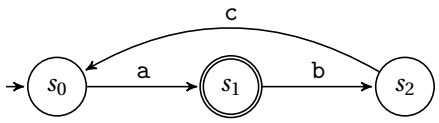
Fehlerzustand

Wenn in der grafischen Darstellung[7] 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?

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

Aufgabe 5.6. 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.

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

[7] Man 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.

Lemma 5.1 Zu jedem endlichen Automaten A gibt es eine reguläre Grammatik G mit $L(G) = L(A)$.

Beweis. Der Beweis ist naheliegend und ergibt sich quasi von selbst. Zum Automaten[8] $A = (Z, I, \delta, s_0, F)$ 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 $B \to aC$ hinzu. Für jeden Zustand $B \in F$ fügen wir außerdem $B \to \varepsilon$ hinzu. ■

Aufgabe 5.7. 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 5.8**. 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:

Lemma 5.2 Zu jedem endlichen Automaten A mit Eingabealphabet I gibt es einen Automaten, der die Sprache $I^* \setminus L(A)$ akzeptiert.

Aufgabe 5.9. 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 $\{\mathtt{a}, \mathtt{b}\}^*$ *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 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

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

Lemma 5.4

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.

Aufgabe 5.10. 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 5.11. 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 aus Kapitel 15 von KMFI kennen.)

6. Nichtdeterministische endliche Automaten

Nichtdeterminismus

In diesem Abschnitt begegnen wir zum ersten Mal einem wichtigen Konzept der Theoretischen Informatik,[9] 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.

Definition

NFA

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.

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

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

Und die von einem NFA A akzeptierte Sprache ist jetzt

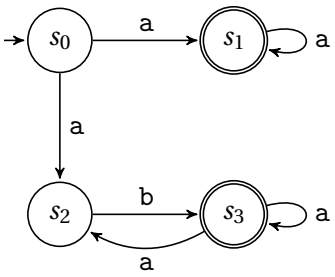
$$L(A) = \{ w \in I^* : \delta^*(\{s_0\}, w) \cap F \neq \emptyset \}.$$

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!

Wie ist diese neue Definition gemeint? Die Übergangsfunktion gibt für einen bestimmten Zustand und ein in diesem Zustand konsumiertes Zeichen nun nicht

[9] Für 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.

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.[10] 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 $\mathtt{a}^3$ wird im oberen Teil akzeptiert, wobei der Weg durch den Automaten im Zustand s_1 endet. Ein Wort wie $(\mathtt{ab})^2$ 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.)

Man mag zunächst den Eindruck haben, dass die nichtdeterministischen Automaten „mächtiger“ sind als die deterministischen: dass sie mehr Sprachen akzeptieren

[10] Vielleicht stellen Sie sich das wie einen Parallelrechner vor, der mehrere Wege gleichzeitig ausprobieren kann.

können. Das ist aber nicht der Fall, denn es gilt auch die Umkehrung von Aufgabe 6.4:

Satz 6.1

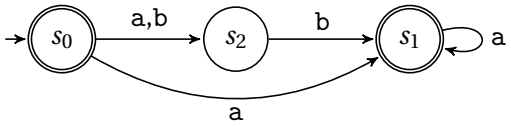
Satz von Rabin und Scott
Zu jedem nichtdeterministischen endlichen Automaten $A = (S, I, \delta, s_0, F)$ gibt es einen deterministischen Automaten A' mit $L(A) = L(A')$.

Beweis. Der „Trick“ ist, als Menge der Zustände von A' die Menge alle Zustands*mengen* von A zu nehmen: $S' = \mathscr{P}(S)$. Der neue Startzustand ist $\{s_0\}$, die zugehörige Übergangsfunktion wird definiert durch[11]

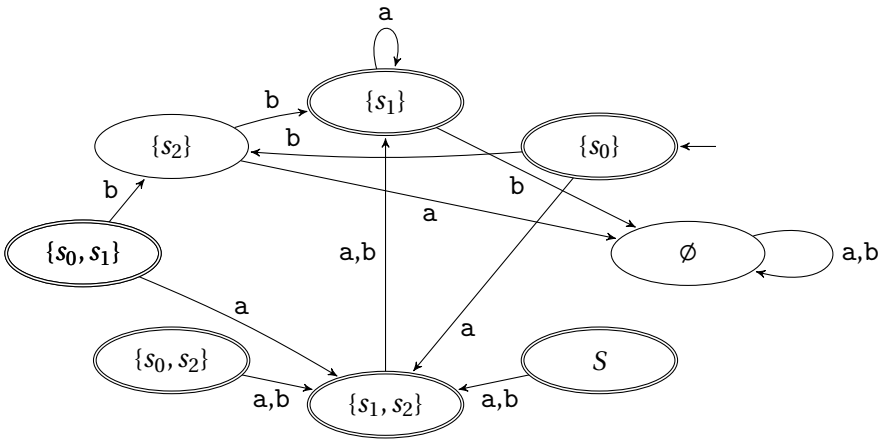
$$\delta'(s,a) = \bigcup_{t \in s} \delta(t,a) = \{ r \in S : r \in \delta(t,a) \text{ für ein } t \in s \}$$

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. ■

Im diesem Beispiel sieht A so aus:



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 $\{s_1\}$.

Beachten Sie jedoch, dass wir für die Aussage dieses Satzes einen ggf. hohen Preis zahlen müssen, weil die Anzahl der Zustände des deterministischen Automaten

[11] Hier 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.

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 5.8 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 $B \to aC$ fügen wir einen a-Pfeil von B nach C hinzu. Für jede Produktion $B \to \varepsilon$ 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.[12] 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.

⋆**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.[13] Überlegen Sie sich, dass man auch dadurch nichts gewonnen hat. Mit anderen Worten: Geben sie an, wie man einen 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.)

ε-Übergang

[12] Und 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.

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

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 5.6 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.[14]

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.[15] 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 6.10. 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.

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

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

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

Aufgabe 6.11. 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 sie 1973 erstmals in dem Unix-Tool `grep` verwendet und später (ab 1987) durch die Programmiersprache PERL popularisiert wurden.[16]

Wir definieren zunächst rekursiv die Syntax solcher Ausdrücke, ohne über ihre Bedeutung zu sprechen:

Definition

Sei Σ ein Alphabet. Alle Zeichen von Σ sowie die Symbole[17] 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.[18]

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}.

Konvention

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 $\mathtt{ab}^*$ für $\mathtt{a(b)}^*$ steht und *nicht* für $\mathtt{(ab)}^*$. (Das ist also wie bei Potenzen von Zahlen.)

Nun kommen wir zur Semantik – also zur intendierten Bedeutung – der regulären Ausdrücke.

[16] Die 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 7.4.

[17] Wir 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 5) gleich sein sollten.

[18] In manchen Büchern wird auch das Zeichen | statt + benutzt.

Definition

Jedem regulären Ausdruck α über Σ ordnen wir folgendermaßen eine Sprache $\mathscr{L}(\alpha)$ über Σ zu: Zunächst setzen wir $\mathscr{L}(\emptyset) = \emptyset$ und $\mathscr{L}(\varepsilon) = \{\varepsilon\}$ sowie $\mathscr{L}(a) = \{a\}$ für alle $a \in \Sigma$. Sind α und β beliebige reguläre Ausdrücke über Σ, so definieren wir ferner rekursiv

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

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 1.15.) Der reguläre Ausdruck 0^* beschreibt eine *Sprache*.

Aufgabe 7.1. Geben Sie die Sprachen zu den folgenden drei regulären Ausdrücken über Σ_{bool} an: 11, $11 + 0$, $(11 + 0)^*$.

Aufgabe 7.2. Welche der folgenden Aussagen sind wahr?

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

$\alpha \equiv \beta$

Für $\mathscr{L}(\alpha) = \mathscr{L}(\beta)$ 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 7.3. 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 $\{a, b\}$ ebenfalls richtig sind, und geben Sie ggf. Gegenbeispiele an.

(i) $a(a + a^*) \equiv a^*a$

(ii) $a(b + a^*) \equiv a^*b$

(iii) $(\mathtt{ab}^* + \mathtt{ba}^*) \equiv (\mathtt{a}+\mathtt{b})(\mathtt{a}+\mathtt{b})^*$
(iv) $(\mathtt{a}+\mathtt{b})^* \equiv (\mathtt{a}^* + \mathtt{b}^*)^*$

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

Lemma 7.2

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

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

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

Gleichzeitig sind reguläre Ausdrücke aber auch expressiv genug, um *alle* reguläre Sprachen zu beschreiben.

Satz 7.3

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

Beweis. Eine Hälfte dieser Aussage wurde in Lemma 7.2 bereits bewiesen. Für die andere Hälfte geben wir uns einen Automaten $A = (\{s_1, \ldots, s_n\}, \Sigma, \delta, s_1, F)$ vor und definieren für $i, j \in \{1, \ldots, n\}$ und $k \in \{0, \ldots, n\}$ 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:

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

$$R(i,j,k+1) = R(i,j,k) \cup \big(R(i,k+1,k) \circ R(k+1,k+1,k)^* \circ R(k+1,j,k)\big) \tag{7.2}$$

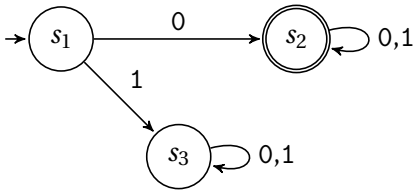
$$L(A) = \bigcup_{\substack{j \in \{1,\ldots,n\} \\ s_j \in F}} R(1,j,n) \tag{7.3}$$

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, $R(1,2,3)$. Diesen 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:

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

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

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

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

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

Der letzte Schritt liefert $L(A) = \mathscr{L}(0(0+1)^*)$ 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 7.4**. Die meisten Programmiersprachen, die reguläre Ausdrücke unterstützen, verwenden die Syntax von PERL. (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 7.5. FLACI (siehe Aufgabe 5.6) kann reguläre Ausdrücke in Automaten umwandeln und umgekehrt. (Beachten Sie allerdings, dass dort eine andere Syntax verwendet wird. Siehe Aufgabe 7.4.) 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 7.6**. 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. 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

$w_1 \sim_L w_2$

$$w_1 v \in L \text{ genau dann, wenn } w_2 v \in L \text{ für alle } v \in \Sigma^*$$

äquivalent

und sagen, dass w_1 und w_2 *äquivalent*[19] sind. Es handelt sich nämlich offensichtlich um eine Äquivalenzrelation,[20] die die Menge Σ^* in Äquivalenzklassen der Form

$[w]_L$

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

partitioniert, d. h., Σ^* ist die Vereinigung $\dot{\bigcup}_{w \in \Sigma^*} [w]_L$ solcher Klassen und diese sind paarweise disjunkt.[21]

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 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:

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

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

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

[20] Siehe dazu Abschnitt 5 im aktuellen Mathe-Skript.

[21] Der Punkt über dem Vereinigungszeichen ist kein Fliegendreck, sondern er steht für die disjunkte Vereinigung.

durch Konkatenieren von $\mathtt{1}^3$ oder $\mathtt{01}^4$ zu einem Wort von L_2 machen, aber für $\mathtt{0}^5$ oder $\mathtt{0}^2$ würde das nicht funktionieren. Ebenso können Wörter der Form $\mathtt{0}^n\mathtt{1}^m$ mit $0 < m \leq n$ zu Wörtern von L_2 gemacht werden, aber es gibt jeweils nur *eine* Möglichkeit: man muss $\mathtt{1}^{n-m}$ 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:

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

Aufgabe 8.1. Wie viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben sich für $L = \{\mathtt{ab}\}$ und wie sehen sie aus?

Aufgabe 8.2. Wie viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben sich für $L = \{\mathtt{ab}\}^*$ und wie sehen sie aus?

Aufgabe 8.3. Begründen Sie, dass sich für $L = \{ww^R : w \in \{\mathtt{a},\mathtt{b}\}^*\}$ unendlich viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben. Dabei soll w^R das Wort sein, das sich ergibt, wenn man die Reihenfolge der Symbole in w umkehrt. $(\mathtt{abb})^R$ ist also $\mathtt{bba}$ und $(\mathtt{aba})^R$ ist $\mathtt{aba}$. w^R

Das versprochene Kriterium liefert nun das folgende Theorem:

Satz von Myhill-Nerode **Satz 8.1**
Eine Sprache ist genau dann regulär, wenn sie bezüglich der Nerode-Relation nur endlich viele Äquivalenzklassen hat.

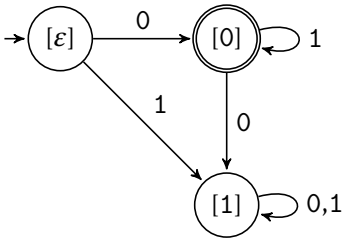
Beweis. Zunächst der einfache Teil der Aussage: Ist eine Sprache über Σ regulär, dann gibt es einen deterministischen endlichen Automaten $(S, \Sigma, \delta, s_0, F)$, der sie akzeptiert. Jedes Wort $w \in \Sigma^*$ landet beim Durchlaufen des Automaten bei einem bestimmten Zustand, nämlich bei $s = \delta^*(s_0, w)$. 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 $L_1 = \{\mathtt{0}\} \circ \{\mathtt{1}\}^*$ von oben. Deren Äquivalenzklassen $[\mathtt{0}]_{L_1}$,

$[\varepsilon]_{L_1}$ 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 8.4. Machen Sie dasselbe für die Sprache aus Aufgabe 8.2.

Aufgabe 8.5. Mit Satz 8.1 ist klar, dass die Sprache aus Aufgabe 8.3 nicht regulär ist. Begründen Sie auf die gleiche Art, dass $L = \{a^k b^k : k \in \mathbb{N}\}$ nicht regulär ist.[22]

Ganz nebenbei liefert uns Satz 8.1 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

Korollar 8.2

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

Beweis. Der im Beweis von Satz 8.1 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:[23]

(i) Erstelle eine Liste aller Zustände und streiche s_0 durch.

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

[23] Hierbei geht es immer um den *ganzen* Automaten inklusive Fehlerzustand. Siehe Fußnote 7 auf Seite 18.

(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.

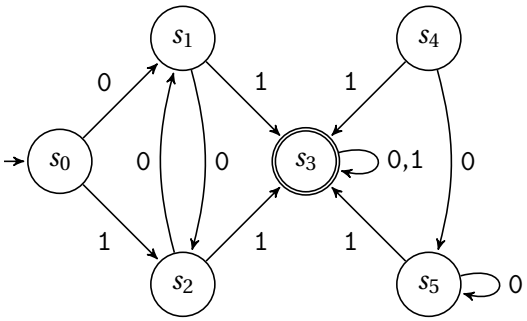
(vi) Markiere alle Paare (s, s') aus $S' \times S'$, für die $s \in F$ und $s' \notin F$ oder $s' \in F$ und $s \notin F$ gilt.

(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 äquivalent[24] 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.[25]

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



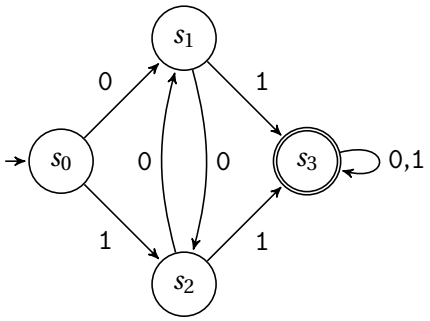
Im ersten Teil des Algorithmus sieht unsere Liste folgendermaßen aus:

- ~~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.

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

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



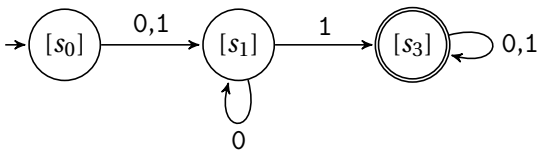
Jetzt erstellen wir die Tabelle, die uns die Nerode-Relation liefern wird. (Da die Relation symmetrisch ist, können wir uns etwas Arbeit sparen. Das sind die grauen Quadrate.) Da nur s_3 ein Endzustand ist, kann s_3 zu keinem der anderen Zustände äquivalent sein.

	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).

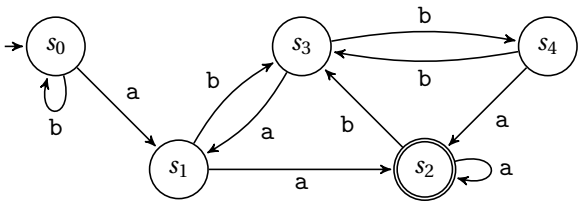
	s_0	s_1	s_2	s_3
s_0		×	×	×
s_1	■			×
s_2	■	■		×
s_3	■	■	■	

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 8.6. FLACI (siehe Aufgabe 5.6) kann für Sie Minimalautomaten konstruieren. Damit können Sie den obigen Algorithmus an eigenen Beispielen ausprobieren.

Aufgabe 8.7. 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 8.8. Die Sprache L wird durch den regulären Ausdruck $\mathtt{b} + \mathtt{ab}(\mathtt{bb})^*$ beschrieben. Wie viele Zustände hätte der Minimalautomat für diese Sprache? Beantworten Sie die Frage, ohne einen Automaten für L zu konstruieren.

Aufgabe 8.9. Haben Sie die Übungsaufgabe U 15 bearbeitet? War Ihre Lösung minimal? Begründen Sie Ihre Antwort.

⋆**Aufgabe 8.10**. Über dem Alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ sei mit

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

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.

III

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.

9. Kontextfreie Grammatiken

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

Definition

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.

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 $\mathtt{a}X \to \mathtt{ab}$, dann darf nach dieser Regel das nichtterminale Symbol X nur dann durch $\mathtt{b}$ ersetzt werden, wenn es rechts vom Symbol $\mathtt{a}$ 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 9.1.) Die regulären Sprachen bilden also sogar eine *echte* Teilmenge der kontextfreien.

Aufgabe 9.1. Begründen Sie die Korrektheit der obigen Behauptung, dass die Dyck-Sprache nicht regulär ist.

Aufgabe 9.2. Geben Sie für die Sprache $\{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^n : m, n \in \mathbb{N}\}$ 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 $S \to \varepsilon$, $A \to a$ oder $A \to BC$ sind, wobei $a \in T$, $A \in N$ und $B, C \in N \setminus \{S\}$ gelten muss.

Aufgabe 9.3. Überlegen Sie sich, wie Ableitungsbäume für Grammatiken in Chomsky-Normalform aussehen.

Lemma 9.1

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 $S' \to S$ hinzu.

(ii) Alle Produktionen der Form $A \to \varepsilon$ mit $A \neq S'$ werden entfernt. Ist $A \to \varepsilon$ 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 $S' \to S$, $S \to A \mid B$, $A \to \varepsilon$ und $B \to \mathtt{1}$ wird z. B. zunächst $A \to \varepsilon$ entfernt und aus $S \to A$ wird $S \to \varepsilon$. Dann muss aber die neue Produktion $S \to \varepsilon$ ebenfalls entfernt und die Produktion $S' \to \varepsilon$ 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 $V_x \to x$ hinzugefügt.

(iv) Alle Produktionen der Form $A \to B_1 B_2 \dots B_n$ mit $n > 2$ werden mithilfe neuer nichtterminale Symbole A_i in Produktionen $A \to A_{n-1}B_n$, $A_{n-1} \to A_{n-2}B_{n-1}$ und so weiter bis $A_2 \to B_1 B_2$ zerlegt.

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

(v) Produktionen der Form $A \to B$ mit $B \in N$ werden entfernt. Dafür wird für jede Produktion der Form $B \to \beta$ eine neue $A \to \beta$ 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 $A \to \beta$ 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 $S \to AB \mid ABA$, $A \to 0A \mid 0$ und $B \to 1B \mid \varepsilon$ erzeugt die Sprache $\{0\}^+\{1\}^*\{0\}^*$. Zuerst fügen wir $S' \to S$ hinzu und S' wird das neue Startsymbol.

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

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

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

Anschließend werden Variablen für 0 und 1 eingeführt.

$$S' \to S \quad S \to AB \mid A \mid ABA \mid AA \quad A \to V_0A \mid 0 \quad B \to V_1B \mid 1$$
$$V_0 \to 0 \quad V_1 \to 1$$

Dann wird $S \to ABA$ durch $S \to S_2A$ und $S_2 \to AB$ ersetzt.

$$S' \to S \quad S \to AB \mid A \mid S_2A \mid AA \quad A \to V_0A \mid 0 \quad B \to V_1B \mid 1$$
$$V_0 \to 0 \quad V_1 \to 1 \quad S_2 \to AB$$

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

$$S' \to S \quad S \to AB \mid S_2A \mid AA \mid V_0A \mid 0 \quad A \to V_0A \mid 0 \quad B \to V_1B \mid 1$$
$$V_0 \to 0 \quad V_1 \to 1 \quad S_2 \to AB$$

Wir müssten nun noch aufwendig dasselbe mit $S' \to S$ 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:

$$S \to AB \mid S_2A \mid AA \mid V_0A \mid 0 \quad A \to V_0A \mid 0 \quad B \to V_1B \mid 1$$
$$V_0 \to 0 \quad V_1 \to 1 \quad S_2 \to AB$$

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.

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

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

Aufgabe 9.4. Geben Sie für die Sprache aus Aufgabe 9.2 eine Grammatik in Chomsky-Normalform an.

Aufgabe 9.5. 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.

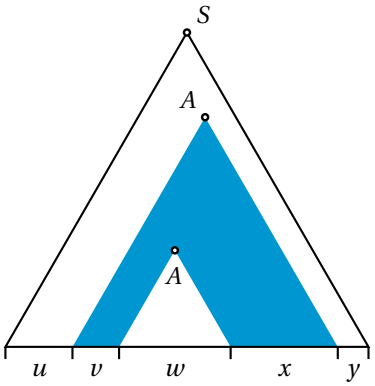
Die Chomsky-Normalform wird für Verfahren wie den *CYK-Algorithmus* benötigt, die feststellen können, ob ein Wort zu einer kontextfreien Sprache gehört. So etwas werden wir uns zwar nicht anschauen, aber wir benötigen die Normalform für das folgende Lemma, das eine analoge Aussage zu Lemma 4.1 macht.

Lemma 9.2

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) $uv^k wx^k y \in L$ 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 9.1 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 9.3.) 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).



[4] Wie im verlinkten Wikipedia-Artikel (Lösung von Aufgabe 9.3) soll ein Binärbaum, der nur aus der Wurzel besteht, die Höhe 1 haben.

Da die beiden Vorkommen von A an verschiedenen Stellen der Ableitung – also auf unterschiedlichen Ebenen des Ableitungsbaums – vorkommen, ist das innere weiße Dreieck kleiner als das blaue. Daher können nicht sowohl v als auch x das leere Wort sein. Das begründet (ii). Außerdem kann man die Situation so wählen, dass die beiden hervorgehobenen Vorkommen von A die letzte Dopplung eines Nichtterminalsymbols in der Herleitung sind, d. h., im blauen Dreieck gibt es keine weiteren Dopplungen und es kann daher nicht beliebig groß sein. Insbesondere folgt daraus (iii).

Dass die eigentliche Aussage des Lemmas, nämlich (iv), richtig ist, sollte hoffentlich durch die Skizze klar sein, weil man ja bei jedem Vorkommen von A dieselben Produktionen zur Verfügung hat. ■

Wir können nun z. B. begründen, dass die Sprache $L = \{\mathtt{a}^k\mathtt{b}^k\mathtt{c}^k : k \in \mathbb{N}\}$ *nicht* kontextfrei ist: Dazu wählen wir für einen Widerspruchsbeweis ein n, wie es das Pumping-Lemma liefert, und betrachten das Wort $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$. Da das „Mittelstück“ vwx höchstens die Länge n hat, kann es nicht sowohl das Symbol a als auch das Symbol c enthalten. Ersetzen wir es durch v^2wx^2 oder durch $w = v^0wx^0$, dann ist das resultierende Wort also keines aus L.

Beachten Sie, dass auch dieses Pumping-Lemma nur ein notwendiges Kriterium liefert. Wir können es nicht verwenden, um zu beweisen, dass eine Sprache kontextfrei ist. Analog zu Aufgabe 8.10 kann man auch in diesem Fall Sprachen konstruieren, die nicht kontextfrei sind, die sich aber mit Lemma 9.2 nicht „überführen“ lassen, z. B.

$$\{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n\mathtt{d}^n : m,n \in \mathbb{N}^+\} \cup \{\mathtt{b}^k\mathtt{c}^m\mathtt{d}^n : k,m,n \in \mathbb{N}^+\}.$$

Auf die Details soll hier aber nicht weiter eingegangen werden. Wenn Sie Lust haben, können Sie es ja selbst probieren.

⋆**Aufgabe 9.6**. Begründen Sie mithilfe des Pumping-Lemmas 9.2, dass die Sprache $L = \{ww : w \in \Sigma_{\text{bool}}^*\}$ nicht kontextfrei ist.

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

Satz 9.3

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 15 auf Seite 24.)

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 9.7. ■

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

Aufgabe 9.7. Ergänzen Sie den fehlenden Teil des Beweises von Satz 9.3.

Aufgabe 9.8. Sei $L_1 = \{\mathtt{a}^m\mathtt{b}^m\mathtt{c}^n : m, n \in \mathbb{N}\}$ und $L_2 = \{\mathtt{a}^m\mathtt{b}^n\mathtt{c}^n : m, n \in \mathbb{N}\}$. Diese Sprachen sind kontextfrei. (Siehe Aufgabe 9.2.) Wie sieht $L_1 \cap L_2$ aus? Was schließen Sie daraus?

Aufgabe 9.9. Begründen Sie mithilfe von Aufgabe 9.8, dass $\Sigma^* \setminus L$ nicht notwendig kontextfrei ist, wenn L eine kontextfreie Sprache über dem Alphabet Σ ist. (Hinweis: Erinnern Sie sich an Aufgabe 5.11.)

10. Kellerautomaten

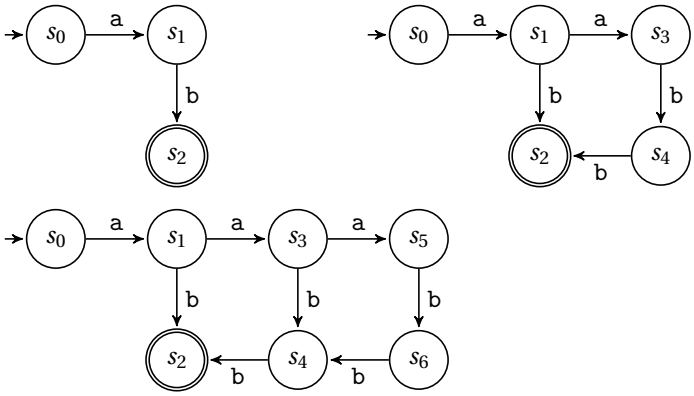
Wir wissen bereits, dass endliche Automaten nicht alle kontextfreien Sprachen akzeptieren können, z. B.

$$L_\infty = \{\mathtt{a}^k\mathtt{b}^k : k \in \mathbb{N}^+\}, \tag{10.1}$$

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

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

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 $\mathtt{a}^2$ und s_5 für $\mathtt{a}^3$. 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.

Definition

Ein **(nichtdeterministischer) Kellerautomat** (engl. *pushdown automaton*, abgekürzt PDA) ist ein 7-Tupel $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ 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) $\delta : S \times (I \cup \{\varepsilon\}) \times \Gamma \to \mathscr{P}_{<\infty}(S \times \Gamma^*)$ ist die **Übergangsfunktion**.[5]
(v) Der **Startzustand** s_0 ist ein Element von S.
(vi) $\bot \in \Gamma$ 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 $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ 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

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

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

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

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.

akzeptierte Sprache

Wie bei den Grammatiken (siehe Fußnote 14 auf Seite 9) sei $\Rightarrow_K^*$ die reflexive und transitive Hülle von $\Rightarrow_K$. Die von einem Kellerautomaten K *akzeptierte Sprache* wird dann definiert als

$L(K)$

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

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

Konvention

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

Und nun folgt unser erster Kellerautomat, der formal die Form

$$K = (\{s_0, s_1\}, \{\mathtt{a}, \mathtt{b}\}, \{A, \bot\}, \delta, s_0, \bot, \emptyset) \tag{10.2}$$

mit der folgenden Übergangsfunktion δ hat:[8]

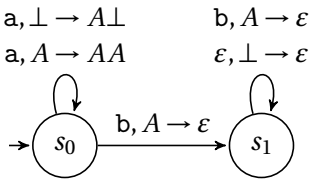
$$\delta(s_0, \mathtt{a}, \bot) = \{(s_0, A\bot)\} \qquad \delta(s_0, \mathtt{a}, A) = \{(s_0, AA)\}$$
$$\delta(s_0, \mathtt{b}, A) = \{(s_1, \varepsilon)\} \qquad \delta(s_1, \mathtt{b}, A) = \{(s_1, \varepsilon)\}$$
$$\delta(s_1, \varepsilon, \bot) = \{(s_1, \varepsilon)\}$$

[6] Man 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.

[7] Ja, Sie haben das richtig gelesen: die Menge F der Endzustände spielt gar keine Rolle. Auf die kommen wir später noch zu sprechen.

[8] Wir 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“.

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:



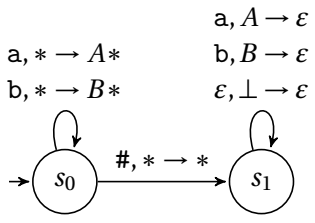
Exemplarisch schauen wir uns zwei Wege durch den Automaten an.

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

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.)

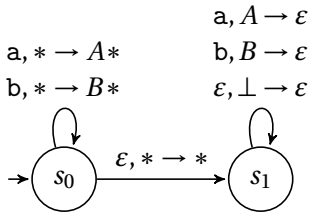
Es ist hoffentlich klar geworden, dass dieser Automat die Sprache L_∞ aus (10.1) akzeptiert. Für jedes `a` wird ein A auf den Stack gelegt und jedes `b` entfernt ein A vom Stack.

Noch ein weiteres Beispiel, bei dem das Eingabealphabet die Menge $\{\texttt{a}, \texttt{b}, \texttt{\#}\}$ ist und das Stackalphabet aus den Zeichen $\bot$, 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 $\to$ für dasselbe Symbol.[9]



Machen Sie sich klar, dass dieser Automat die Sprache $\{w\#w^R : w \in \{\texttt{a}, \texttt{b}\}^*\}$ 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 $\{ww^R : w \in \{\texttt{a}, \texttt{b}\}^*\}$:

[9] Die Zeile $\texttt{a}, * \to A*$ ersetzt z. B. drei Zeilen, weil $*$ jedes der drei Stacksymbole sein kann.



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

Aufgabe 10.2. Konstruieren Sie einen Kellerautomaten für die Sprache $\{\mathtt{0}^n\mathtt{1}^{2n} : n \in \mathbb{N}\}$.

⋆**Aufgabe 10.3**. Man kann Akzeptanz durch Kellerautomaten auch anders definieren, indem man für $K = (S, I, \Gamma, \delta, s_0, \bot, F)$

$L^*(K)$

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

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 10.4. FLACI kann auch Kellerautomaten simulieren, allerdings nur solche, die wie in Aufgabe 10.3 akzeptieren.[10] Trotzdem sollten Sie damit vielleicht ein bisschen herumspielen.

beschränkter Kellerautomat

⋆**Aufgabe 10.5**. 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.

Nun stellen wir den Zusammenhang zu den kontextfreien Sprachen her.

Satz 10.1

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

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

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 $\bot$ zum Stackalphabet $N \cup T \cup \{\bot\}$. 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 $A \to \beta$ 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 $\delta(s_0, \varepsilon, \bot) = \{(s_0, S)\}$ 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.

Für die andere Richtung des Beweises gehen wir von einem Kellerautomaten $K = (S, I, \Gamma, \delta, s_0, \bot, \emptyset)$ aus und wollen eine kontextfreie Grammatik G konstruieren, die $L(K)$ erzeugt. Dafür definieren wir die Menge[11]

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

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 $S_0 \to [s_0 \bot s]$ 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 $s_1, \ldots, s_k \in S$ die folgende Produktion zur Grammatik hinzu:

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

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 10.8. ■

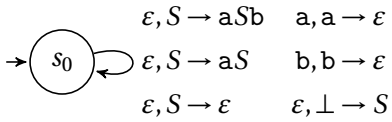
[11] $[s\sigma t]$ wählen wir hier als Abkürzung für (s, σ, t). Damit sind „atomare“ Symbole wie in der Lösung von Aufgabe 10.3 gemeint.

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

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

$$S \to \mathtt{a}S\mathtt{b} \mid \mathtt{a}S \mid \varepsilon$$

für die Sprache $\{\mathtt{a}^m\mathtt{b}^n : m, n \in \mathbb{N} \text{ und } m \geq n\}$. 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 10.6. Geben Sie einen Weg durch diesen Automaten an, durch den das Wort `aab` der Sprache akzeptiert wird.

Aufgabe 10.7. 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 10.8**. Überlegen Sie sich anhand des Beispielautomaten K aus (10.2), wie die zugehörige Grammatik aus dem Beweis von Satz 10.1 aussehen würde.

⋆**Aufgabe 10.9**. 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 9.8 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.)

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 genau mit *deterministisch* gemeint ist.

Definition

Ein Kellerautomat $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ 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.

Intuitiv heißt das, dass es für jedes Wort nur einen möglichen Weg durch den Automaten gibt. (Details dazu in Aufgabe 10.11.)

Aufgabe 10.10. Überzeugen Sie sich, dass die ersten beiden Beispiele für Kellerautomaten in diesem Skript deterministisch waren.

Aufgabe 10.11. In deterministischen Kellerautomaten sind nach wie vor ε-Übergänge erlaubt, die wir im Zusammenhang mit *nicht*deterministischen 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 10.3). 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 10.12. 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

$$L = L_1 \cup L_2 \text{ mit } L_1 = \{0^n 1^n : n \in \mathbb{N}^+\} \text{ und } L_2 = \{0^n 1^{2n} : n \in \mathbb{N}^+\} \tag{10.3}$$

verdeutlichen. L ist kontextfrei (siehe Aufgabe 10.13), 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 10.13. Begründen Sie, dass die Sprache aus (10.3) 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

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

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 $\{ww^R : w \in \{\mathtt{a},\mathtt{b}\}^*\}$ 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.

11. 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 noch keine Computer. 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 Rechenvorgangs mit einem Bleistift (und ggf. einem Radiergummi) auf einem Blatt Papier festhält. Diesen Vorgang vereinfachte er nach und nach, indem er

[1] Dabei 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.

Formal lässt sich das auf verschiedene Arten beschreiben und 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 $x \notin \mathrm{dom}(f)$ wird üblicherweise $f(x) = \perp$ geschrieben.[2] Während man für Funktionen $f\colon A \to B$ schreibt, notiert man partielle Funktionen als $f\colon A \rightharpoonup B$. 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.

Definition

Eine **Turingmaschine** ist ein 7-Tupel $T = (S, I, \Gamma, \delta, s_0, \square, F)$ 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 $S \times \Gamma \times \{\leftarrow, \rightarrow\}$ 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) 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 $\square$ nur die Symbole von I enthalten.

Band

Als *Band* bezeichnet man bei Turingmaschinen die oben beschriebene Sequenz von „Kästchen“, die man sich zunächst als in beide Richtungen unbeschränkt

[2] Wir setzen dabei implizit $\perp \notin B$ voraus.

vorstellen sollte.[3] Der „aktuelle Bandinhalt“ ist in diesem Sinne ein Wort über Γ. Wie eine Turingmaschine den Bandinhalt nach und nach ändert, wird wieder durch Konfigurationen beschrieben.

Definition

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

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

für $v \in \Gamma^*$, $w \in \Gamma^+$, $\rho, \sigma, \sigma' \in \Gamma$ und $s, s' \in S$.

Informell ist mit einer Konfiguration (v, s, w) gemeint, dass vw der aktuelle Bandinhalt ist, die Maschine sich im Zustand s befindet und sich das erste Symbol von w „anschaut“. Die Maschine hat also neben einem momentanen Zustand auch immer eine momentane Position. Wir übernehmen die übliche Sprechweise und stellen uns einen *Schreib-Lese-Kopf* vor – eine Vorrichtung, die jeweils über einem bestimmten Kästchen des Bandes steht und dessen Inhalt lesen und modifizieren kann. Wir werden Konfigurationen statt als (v, s, w) in der Form $v[s]w$ schreiben. Der Teil $[s]$ zeigt an, wo der Schreib-Lese-Kopf steht und in welchem Zustand sich die Maschine befindet.

Schreib-Lese-Kopf

$v[s]w$

Zunächst ein einfaches Beispiel. Die Turingmaschine ist

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

und δ_1 sei durch die folgende Tabelle definiert.

	s_0	s_1	s_2
0	$(s_0, 0, \rightarrow)$	$(s_1, 0, \leftarrow)$	$\bot$
1	$(s_0, 1, \rightarrow)$	$\bot$	$\bot$
$\square$	$(s_1, 0, \leftarrow)$	$(s_2, \square, \rightarrow)$	$\bot$

In T_1 können beispielsweise die folgenden Konfigurationsübergänge stattfinden.

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

[3] Damit ist nicht gemeint, dass das Band unendlich lang ist, sondern lediglich, dass es „lang genug“ ist. Eine Berechnung soll sozusagen nicht am mangelnden Speicherplatz scheitern.

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

$$\square[s_0]000\square \Rightarrow_{T_1}^* \square\square[s_2]0000 \tag{11.1}$$

gelten.

Aufgabe 11.1. Machen Sie sich klar, dass für diese Maschine auch

$$\square[s_0]11\square \Rightarrow_{T_1}^* \square1[s_1]10$$

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

Aufgabe 11.2. Warum sind der obigen Tabelle für δ_1 wohl einige Einträge grau gefärbt?

terminieren

Die Konfigurationen $\square\square[s_2]0000$ und $\square1[s_1]10$, 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

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

und δ_2 sieht so aus:

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

Damit ist das hier möglich:

$$\square[s_0]00\square \Rightarrow_{T_2} \square0[s_1]0\square \Rightarrow_{T_2} \square[s_0]00\square \Rightarrow_{T_2} \square0[s_1]0\square \Rightarrow_{T_2} \cdots$$

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 (11.1) 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 $T = (S, I, \Gamma, \delta, s_0, \square, F)$ **berechnet** aus der **Eingabe** $w \in I^*$ die **Ausgabe** $w' \in I^*$, wenn $\square[s_0]w\square \Rightarrow_T^* \square^*[s]w'\square^*$ für ein $s \in F$ gilt.

(Wir verwenden hier und im Folgenden die Konvention, dass mit dem formal nicht korrekten Ausdruck $\square^*$ eine beliebige Anzahl von Leerzeichen gemeint ist.)

Informell: Das Wort w wird auf das Band geschrieben, die Turingmaschine in den Startzustand s_0 versetzt und der Schreib-Lese-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 der Kopf auf dem Anfang von w' platziert ist. Leerzeichen werden ignoriert. Sie „verlängern“ lediglich das Band, wenn das nötig sein sollte.

berechnete Funktion f_T

Dadurch wird die *von T berechnete* partielle Funktion $f_T\colon I^* \rightharpoonup I^*$ 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. Man sagt auch, dass f_T *turingberechenbar* ist.

turingberechenbar

Aufgabe 11.3. Geben Sie zu den beiden obigen Beispielen T_1 und T_2 jeweils die berechnete Funktion an.

Aufgabe 11.4. 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.

Aufgabe 11.5. 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.

In der Literatur findet man viele Definitionen von Turingmaschinen, die von der obigen in einigen Details abweichen. Auf den ersten Blick hat man oft den Eindruck, die entsprechenden Maschinen könnten evtl. mehr oder weniger als „unser“ Modell. Es ist jedoch so, dass alle im Folgenden vorgestellten Varianten im Endeffekt dieselben Funktionen berechnen.

Mehrspur-Turingmaschine

– Bei *Mehrspur-Turingmaschinen* besteht das Band aus mehreren „Spuren“, die mit fest verbundenen Schreib-Lese-Köpfen bearbeitet werden. Die Köpfe bewegen sich also immer gleichzeitig über denselben Positionen wie bei konventionellen Festplattenlaufwerken und die Maschine befindet sich immer nur in einem Zustand zur Zeit. Ein- und Ausgabe würden beispielsweise über die erste Spur realisiert werden, während die anderen Spuren für „Notizen“ oder „Zwischenrechnungen“ zur Verfügung stünden.
  Man kann so eine Maschine mit n Spuren ganz simpel durch eine normale „einspurige“ Turingmaschine simulieren, indem man immer n Zeichen zu

einem zusammenfasst. Wenn die n-spurige Maschine das Bandalphabet Γ hat, dann arbeitet die simulierende Maschine mit dem Alphabet Γ^n.

Mehrband-Turingmaschine

- *Mehrband-Turingmaschinen* sind eine Erweiterung dieses Konzepts. 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“.

  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.[4]

einseitig beschränkt
Bandanfang $*$

- Das Band ist *einseitig beschränkt*. Das bedeutet, dass es ein weiteres Bandsymbol $*$ für den *Bandanfang* gibt. Der initiale Bandinhalt bei der Eingabe w ist dann nicht $\square w\square$, sondern $*w\square$, der Schreib-Lese-Kopf kann sich nicht über dieses Zeichen hinausbewegen und es darf weder überschrieben noch an anderen Positionen eingesetzt werden. So eine Maschine kann sicher durch eine Maschine „unseres“ Typs simuliert werden, indem man das neue Symbol zum Bandalphabet hinzufügt und die Übergangsfunktion entsprechend erweitert. Aber umgekehrt geht es auch: Eine einseitig beschränkte Turingmaschine kann eine unbeschränkte simulieren. Dafür muss sie lediglich immer dann, wenn die simulierte Maschine einen Schritt über den Bandanfang hinaus machen würde, den gesamten Bandinhalt um eine Position nach rechts schieben (siehe Aufgabe 11.6), um danach mit der Simulation fortzufahren.

Ein paar weitere Varianten seien nur kurz erwähnt. Wenn Sie Lust haben, können Sie sich selbst überlegen, wie die verschiedenen Maschinentypen sich gegenseitig simulieren können.

- Zusätzlich zu den akzeptierenden Endzuständen gibt es *verwerfende* Endzustände, bei denen die Maschine terminiert, ohne die Eingabe zu akzeptieren.
- Außer $\rightarrow$ und $\leftarrow$ kann die dritte Komponente der Übergangsfunktion auch $\diamondsuit$ sein. Das bedeutet, dass der Schreib-Lese-Kopf sich nicht bewegt.
- Es gibt nur ein Band, aber mehrere Schreib-Lese-Köpfe, die unabhängig voneinander gesteuert werden können.
- Die Turingmaschine arbeitet nicht auf einem eindimensionalen Band, sondern auf einer zweidimensionalen Fläche (also dem ursprünglichen „Kästchenpapier“) oder auf höherdimensionalen Verallgemeinerungen.

[4] Siehe dazu auch Lemma 17.1 später.

Aufgabe 11.6. Konstruieren Sie eine Turingmaschine nach der Definition vom Anfang des Kapitels, die ein Wort auf dem Band um eine Stelle nach rechts verschiebt, dabei aber das erste Zeichen nicht bewegt. Konkret soll aus einer Eingabe der Form $*w$ mit $w \in \Sigma_{\text{bool}}^*$ ein Bandinhalt der Form $\square^* * \square w \square^*$ werden.

Aufgabe 11.7. Begründen Sie informell, warum eine Turingmaschine mit dem Eingabealphabet Σ_{bool} und dem Bandalphabet $\Sigma_{\text{bool}} \cup \{\square\}$ jede Turingmaschine mit umfangreicheren Alphabeten simulieren kann.

Aufgabe 11.8. 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?

Aufgabe 11.9. 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 11.10**. Wie kann man mit einer einseitig beschränkten Zweispur-Turingmaschine eine unbeschränkte Turingmaschine gemäß der Definition vom Anfang simulieren, ohne Bandinhalte verschieben zu müssen?

⋆**Aufgabe 11.11**. 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?

TM2012 – eine modifizierte Turingmaschine

Für die folgenden Ausführungen verwenden wir eine leicht modifizierte Turingmaschine nebst unterstützender Software, die nach meiner Meinung angenehmer zu „programmieren“ ist als das bisher beschriebene Modell und die darauf basierenden Simulatoren wie FLACI, bei denen man sich grafische Darstellungen von Zuständen und Übergängen zusammenklickt. Natürlich ist das Geschmackssache und das Arbeiten mit Turingmaschinen ist trotzdem kein Vergnügen. Wir werden uns allerdings mithilfe dieser Variante vergleichsweise schnell davon überzeugen können, dass man über die Fähigkeiten von Turingmaschinen sprechen kann, ohne sich über Details den Kopf zu zerbrechen.

Die besagte Software kann unter der URL `https://weitz.de/files/TI.zip` heruntergeladen werden. Sie wurde bereits 2012 geschrieben,[5] sieht daher nicht mehr besonders „modern“ aus und existiert aktuell nur noch in einer Version für Microsoft Windows. Gerade aufgrund ihres Alters und der vergleichsweise schlichten Benutzeroberfläche sollten die Programme allerdings mithilfe von Software wie WINE auch auf Linux oder macOS ohne Probleme lauffähig sein.

Da es zu den Programmen jeweils ausführliche Bedienungsanleitungen als PDF gibt, werden hier im Skript ein paar Details unterschlagen und nur die wesentlichen Fakten aufgeführt. Die Turingmaschine, mit der wir nun arbeiten werden und

[5] Und zwar in COMMON LISP, falls Sie das interessiert.

TM2012

die wir TM2012 nennen, unterscheidet sich wie folgt von „richtigen“ Turingmaschinen:

(i) Es handelt sich um eine einseitig beschränkte Maschine mit dem Zeichen $*$ für den Bandanfang. Es wurde bereits begründet, warum das keine relevante Einschränkung ist.

(ii) Das Eingabealphabet Σ_{bool} ist „fest verdrahtet“ und außer der Eins und der Null gibt es für das Band nur noch das Leerzeichen (`B` für *blank*) und das Symbol $*$ für den Bandanfang. Wie man sich in Aufgabe 11.7 überlegen konnte, ist das keine wesentliche Einschränkung.

(iii) Es gibt faktisch nur einen akzeptierenden Endzustand und der wird durch den speziellen Befehl `H` (für *halt*) implementiert. Siehe dazu Aufgabe 11.8. Terminiert ein Programm in einem nicht akzeptierenden Zustand, weil ein Übergang nicht definiert ist, so wird das durch eine entsprechende Warnmeldung signalisiert.

(iv) Der Schreib-Lese-Kopf steht initial *vor* dem Bandanfangszeichen. Die Position des Kopfes am Ende (nach Erreichen des `H`-Befehls) ist irrelevant und der jeweilige Bandinhalt wird als Ausgabe betrachtet. Das ist offensichtlich nur eine triviale Vereinfachung und man könnte zur Not alle Programme so umschreiben, dass sie den Kopf zum Bandanfang bewegen, bevor sie anhalten.

(v) Es ist möglich, mehrere Wörter als Argument einzugeben und mehrere Wörter als Resultat auszugeben. Der besseren Lesbarkeit halber werden die Wörter durch das Leerzeichen getrennt, was allerdings der obigen Definition von *Berechnung* widerspricht. Das könnte man notfalls durch die Einführung eines speziellen Trennsymbols beheben. Außerdem werden wir ohnehin noch darüber sprechen, wie man mehrere Zahlen in einer codieren kann.[6]

Wir werden mit der TM2012 in erster Linie rechnen und dabei nur ganz einfache arithmetische Operationen mit natürlichen Zahlen durchführen. Die Zahlen werden binär codiert.[7] Im Prinzip kommen wir mit den folgenden beiden Aktionen aus, die sich als `decf.TURING` und `incf.TURING` in dem erwähnten ZIP-Archiv befinden.

– Von einer natürlichen Zahl wird eins subtrahiert, d. h., aus n wird $n-1$. Handelt es sich um die Eingabe $n=0$, so wird nichts geändert, weil wir nur nichtnegative Zahlen betrachten.

  Das Programm läuft von links nach rechts über die Zahl. Kommen nur Nullen vor, dann ist es bereits fertig. Kommt eine Eins vor, dann läuft es von rechts nach links zurück und macht aus allen Nullen Einsen, bis es eine Eins sieht. Diese wird zu einer Null und danach hält das Programm an.

– Zu einer natürlichen Zahl wird eins addiert, d. h., aus n wird $n+1$.

[6] Und zwar, wenn es um Gödelisierung geht.

[7] Man kann sich jedoch überlegen, dass wir im Prinzip auch im Unärsystem hätten arbeiten können: Die Zahl $n \in \mathbb{N}$ wird durch n Einsen dargestellt und die Null kann als Trennzeichen verwendet werden. Die „Programme“ wären dann sogar einfacher, dafür würden die Wörter auf dem Band deutlich länger werden. Siehe auch Übung U 32.

Das Programm läuft von links nach rechts bis zum Ende der Zahl. Dann läuft es von rechts nach links zurück und macht aus allen Einsen Nullen, bis es eine Null sieht. Diese wird zu einer Eins und danach hält das Programm an. Es kann allerdings (bei Eingaben wie `111`) passieren, dass der Schreib-Lese-Kopf nie eine Null sah. Dann wird am Anfang (denn da befindet er sich nun) eine Eins geschrieben und die Maschine durchläuft die Zahl noch einmal bis zum Ende, um eine Null anzufügen. (Aus dem Beispiel `111` würde also `1000` werden.)

Viele der folgenden Aufgaben behandeln TM2012 oder andere Software aus dem oben erwähnten ZIP-Archiv. Sie können diese Aufgaben aber auch bearbeiten, wenn Sie die zugehörige Software nicht installieren wollen oder können. Die Dokumentation besteht aus PDF-Dateien und der Quellcode ist in der Form von Textdateien abgelegt, die sich mit jedem beliebigen Texteditor öffnen lassen. Die Programme, die geschrieben werden sollen, sind alle so einfach, dass man sie wie bei anderen abstrakten Automaten auch im Kopf oder mit Zettel und Stift ablaufen lassen kann.

Aufgabe 11.12. Schreiben Sie ein Programm für TM2012, das aus der Eingabe n das Ergebnis $n-2$ berechnet. Die Eingaben $n=0$ und $n=1$ sollen nicht verändert werden.

Aufgabe 11.13. Schreiben Sie ein Programm für TM2012, das mit nur einem Zustand auskommt und das alle Einsen zu Nullen und alle Nullen zu Einsen macht.

Aufgabe 11.14. Lösen Sie Aufgabe 11.4 für TM2012.

Aufgabe 11.15. Schreiben Sie ein TM2012-Programm, das zwei Zahlen (die durch ein Leerzeichen getrennt eingegeben werden) wie folgt addiert: Mithilfe der Programme von oben wird in einer Schleife abwechselnd die erste Zahl um eins verringert und die zweite um eins erhöht. 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.

12. Die Church-Turing-These

Wir wechseln nun scheinbar völlig den Blickwinkel und schauen uns eine mehr oder weniger „ausgewachsene“ Programmiersprache an. Wir werden jedoch wieder auf Turingmaschinen zurückkommen. Vorweg wollen wir aber klären, was genau man meint (und insbesondere *nicht* meint), wenn man in der Theoretischen Informatik über *Programme* und ihre Fähigkeiten spricht.

Nicht gemeint sind die meisten Programme, mit denen wir heutzutage täglich zu tun haben, z. B. Webbrowser, Textverarbeitungssoftware oder Apps, mit denen man Musik hören, Filme anschauen oder Fotos betrachten kann. Das sind alles *interaktive* Programme, die in einer Endlosschleife auf Aktionen des Benutzers (Mausklick, Tastendruck, Berührung des Bildschirms) warten und auf diese reagieren.

Bei den Programmen, die uns auf den folgenden Seiten interessieren, geht es hingegen um das Folgende:

- Das Programm bekommt eine *Eingabe* und soll daraufhin eine *Ausgabe* produzieren. Das kann man als Frage und Antwort interpretieren.
- Es gibt klare Kriterien, anhand derer man objektiv überprüfen kann, ob die Antwort des Programms korrekt ist. Dafür müssen die Fragestellungen präzise formuliert werden. Aus diesem Grund verwenden wir formale Sprachen und die mathematische Notation. Siehe dazu die beiden Beispiele am Ende des ersten Abschnitts.
- Fragen und Antworten sollen unabhängig von Zeit und Ort sein. Es soll also beispielsweise nicht darum gehen, die Lottozahlen der nächsten Woche vorherzusagen.
- Es ist möglich, dass das Programm scheitert, indem es terminiert, ohne eine Antwort zu geben, oder indem es in eine Endlosschleife gerät.
- Wir machen uns (noch) keine Gedanken über die Effizienz bzw. die physische Realisierbarkeit, d. h., es interessiert uns momentan nicht, wie lange wir auf eine Antwort warten müssen und wie viele Ressourcen (wie beispielsweise Speicherplatz) dafür verbraucht werden.
- Wir werden uns auf Aufgaben konzentrieren, bei denen Ein- und Ausgabe Zahlen sind, meistens sogar nur natürliche Zahlen. (Das beinhaltet auch Ja-/Nein-Fragen, weil man die Antworten eins und null entsprechend interpretieren kann.)

Bevor wir zu der angekündigten Programmiersprache kommen, brauchen wir noch ein Konzept, das in der Theoretischen Informatik häufig verwendet wird und das auf einer Idee des österreichischen Mathematikers Kurt Gödel beruht, der diese zuerst 1931 im Beweis seines berühmten Unvollständigkeitssatzes verwendete – auf dem übrigens Turings Arbeit von 1936 basierte.

Definition

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

(i) g ist injektiv[8] und berechenbar.

(ii) $g[L]$ ist entscheidbar.

(iii) Die Umkehrfunktion $g^{-1}\colon g[L] \to L$ ist berechenbar.

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

In dieser Definition tauchen die Begriffe *berechenbar* und *entscheidbar* auf, die wir erst später genau definieren werden. Man kann die Anforderungen an g aber auch umgangssprachlich beschreiben:

(i) g soll verschiedenen Wörtern verschiedene Zahlen zuordnen. Und es muss ein Computerprogramm geben,[9] das für jedes Wort w von L die Gödelnummer $g(w)$ berechnen kann.

[8] Zum Begriff der Injektivität und zur Schreibweise $g[L]$ siehe z. B. den Abschnitt über Funktionen im aktuellen Matheskript.

[9] Die 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.

(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.

Aufgabe 12.1. Fällt Ihnen für die Sprache Σ^*_{lat} eine Gödelisierung ein, die in der Praxis tatsächlich verwendet wird?

Gödels ursprüngliche Gödelisierung ist prinzipiell für *alle* formalen Sprachen geeignet.[10] 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,[11] den ich als bekannt voraussetze:

kanonische Gödelisierung

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

Aufgabe 12.3. Schreiben Sie für das Beispiel aus Aufgabe 12.2 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 12.4. 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.

Die Programmiersprache TOLL, mit der wir uns nun beschäftigen werden, hat eine Syntax, die vage an die von PASCAL angelehnt ist. Wenn Sie aber überhaupt schon

TOLL

[10] Genauer: für alle Sprachen, die man überhaupt gödelisieren kann. Erinnern Sie sich an Abschnitt 3.

[11] Kapitel 8 in KMFI.

einmal „ernsthaft“ programmiert haben, sollten Sie TOLL-Programme ohne Probleme lesen können. TOLL bietet die in den meisten Sprachen üblichen Datentypen `integer` und `float` sowie Arrays solcher Werte. Im Gegensatz zu vielen gängigen Programmiersprachen können die Zahlen jedoch beliebig groß oder klein werden und die Fließkommazahlen beliebig genau.[12] Es gibt Variablen, Zuweisungen, arithmetische Ausdrücke, in denen die vier Grundrechenarten vorkommen können, sowie natürlich Konstrukte für bedingte Anweisungen und Schleifen. Zudem gibt es einen `goto`-Befehl, den viele aktuelle Sprachen nicht mehr unterstützen. Da TOLL spezifisch auf die oben beschriebene Problemstellung zugeschnitten ist, sind Beginn und Ende des Programmablaufs klar definiert: Für jedes Programm steht fest, welche und wie viele Eingabeparameter es akzeptiert, und ein planmäßiges Ende ist nur durch den Befehl `halt` möglich, der eine Zahl zurückgibt.

Dieses Programm berechnet beispielsweise die Fakultät $n!$ der Eingabe n:

```
program fakultaet(n: integer)

var i: integer,
    result: integer;

begin
  # Für negative Argumente wird -1 ausgegeben
  if n < 0 then halt(0-1) endif;
  result := 1;
  for i from 1 to n do
    result := result * i;
  halt(result)
end
```

Und das folgende Programm berechnet für die Eingabe n den n-ten Wert der Fibonacci-Folge.

```
program fibo(n: integer)

var i: integer,
    a: array[1000] of integer;

begin
  if (n < 0) then halt(0-1) endif;
  a[0] := 0;
  a[1] := 1;
  for i from 2 to n do
    a[i] := a[i-1] + a[i-2];
  halt(a[n])
end
```

[12] Während das für die ganzen Zahlen wirklich stimmt (sofern es der Speicherplatz zulässt), stellen wir uns das für die Fließkommazahlen nur vor. In der Realität arbeitet TOLL intern mit stinknormalen IEEE-754-Zahlen.

Aufgabe 12.5. Schreiben Sie das obige Programm `fibo` so um, dass es ohne Arrays funktioniert. (Sie müssen dafür nicht TOLL verwenden, sondern können eine Programmiersprache Ihrer Wahl benutzen.)

Aufgabe 12.6. Was macht das Programm `foo.TOLL`, das Sie im ZIP-Archiv finden?

TOLL soll als Muster dafür dienen, was ein typischer Computer (oder besser: jeder beliebige Computer) leisten kann. Daher müssen wir zunächst darüber reden, was diese Sprache (scheinbar) *nicht* kann. Wenn Sie sie mit „echten“ Programmiersprachen, die Sie kennen, vergleichen, werden Ihnen wahrscheinlich diverse Dinge fehlen. Darauf gehen wir gleich ein. Vorab möchte ich aber ein Argument bringen, aufgrund dessen man sich die folgende Diskussion im Prinzip größtenteils sparen kann: Sämtliche Programme, die auf Ihrem Computer laufen, bestehen aus Anweisungen in Maschinensprache, denn nur die „versteht“ der Prozessor. Die Programme wurden wahrscheinlich in einer sogenannten *Hochsprache* wie C++ oder PYTHON geschrieben, dann aber mit einem Compiler oder Interpreter in die „Muttersprache“ der CPU übersetzt. Daher müsste man TOLL eigentlich nur mit Maschinensprache vergleichen.

Trotzdem wollen wir ein paar Punkte, die von Studierenden an dieser Stelle häufig genannt werden, etwas genauer unter die Lupe nehmen.

- TOLL hat keinerlei Fähigkeiten, mit der Hardware (Tastatur, Maus, Bildschirm, Lautsprecher, etc.) zu kommunizieren. Das ist aber offensichtlich angesichts unserer Aufgabenstellung auch nicht nötig.
- TOLL hat keine Strings und auch keinen Datentyp für einzelne Zeichen. Das ist aber nicht wirklich ein Problem, da Buchstaben intern ohnehin als Bitmuster gespeichert werden, die man ebenso gut als Zahlen interpretieren kann.[13] Strings lassen sich als Arrays von Zahlen realisieren.
- TOLL hat keine komplexen Datenstrukturen wie zum Beispiel Hashtabellen, Mengen oder Datensätze. Die lassen sich aber alle mit Arrays simulieren.[14]
- TOLL bietet außer den vier Grundrechenarten, den Relationen = und < sowie den Junktoren `and`, `or` und `not` keine weiteren mathematischen und logischen Operatoren. Man kann die „fehlenden“ Funktionen aber alle mit den vorhandenen Mitteln implementieren, wenn sie benötigt werden. Bei Funktionen wie dem Sinus, bei dem ohnehin nur Näherungswerte gebraucht werden, kann man sogar ohne spezielle Tricks mit einer langsam konvergierenden Reihenentwicklung arbeiten, weil Effizienzüberlegungen momentan keine Rolle spielen. Siehe dazu auch die Aufgaben 12.7 und 12.8.
- TOLL ist nicht objektorientiert. Wer nur JAVA kennt, fällt angesichts dieser Aussage vielleicht in Ohnmacht, aber Objektorientierung ist im Endeffekt

[13] Der Unicode-Standard ordnet ebenfalls jedem Zeichen eine Zahl (den *Codepunkt*) zu und in der Programmiersprache C ist `char` einfach ein Synonym für einen ganzzahligen Wert und Strings als eigenen Datentyp gibt es gar nicht. Das ändert aber nichts daran, dass es unzählige C-Programme gibt, die Text verarbeiten können.

[14] Man könnte auch argumentieren, dass (eindimensionale) Arrays die einzige „natürliche“ Datenstruktur heutiger Computer sind, weil deren Speicher entsprechend organisiert sind.

nur eine Abstraktionsschicht, die einer Programmiersprache keine neue Funktionalität hinzufügt.[15] (Der Linux-Kernel besteht z. B. aus zig Millionen Zeilen Code in C und C ist nicht objektorientiert.)

– TOLL hat keine Funktionen oder Prozeduren. Bei größeren Programmierprojekten dürfte das wohl das größte Manko von TOLL sein – falls jemand die Sprache ernsthaft verwenden wollte. Man kann so etwas jedoch (mühsam) mit `goto` und einem Stack reproduzieren, wenn man sich ein Beispiel daran nimmt, wie Funktionen und Prozeduren auf der Ebene der Maschinensprache realisiert werden. (Den Stack kann man mit einem Array simulieren. Aber siehe auch Aufgabe 12.11.)

Langzahlarithmetik

Nicht verschweigen sollte man auch, dass TOLL Dinge kann (oder zu können vorgibt), die jemandem, der nur Sprachen wie JAVA oder C kennt, vielleicht arg übertrieben vorkommen. Die Rede ist von den beliebig genauen Zahlen. Das ist aber keine theoretische Träumerei, sondern in der Praxis durchaus üblich und wird als Langzahlarithmetik bezeichnet. PYTHON oder COMMON LISP bieten diese beispielsweise für ganze Zahlen und JULIA sowohl für ganze Zahlen als auch für Fließkommazahlen. Dafür wird intern ein gewisser Aufwand betrieben, weil die Zahlen auf mehrere Speicherzellen verteilt werden müssen, und natürlich sind solche Werte nicht *beliebig* genau, weil man irgendwann an die Grenzen des verfügbaren Speichers stößt. Mehr brauchen wir aber auch nicht. Wir hatten die Bänder von Turingmaschinen als „lang genug“ vorausgesetzt und so stellen wir uns den Speicher für unsere TOLL-Programme auch als „groß genug“ vor.

Hoffentlich hat diese lange Argumentationskette Sie davon überzeugt, dass TOLL ein ausreichendes Modell dafür ist, was ein beliebiger Computer leisten kann – im Sinne der Eingrenzung vom Anfang des Abschnitts. Denn das war der wesentliche Grund für die Einführung dieser Programmiersprache (die Sie in der Praxis wohl kaum einsetzen wollen).

Aufgabe 12.7. Schreiben Sie ein TOLL-Programm, das $a \bmod b$ für zwei ganzzahlige Werte a und b berechnet.

Aufgabe 12.8. Schreiben Sie ein TOLL-Programm, das das bitweise UND zweier natürlicher Zahlen berechnet, wobei die Zahlen als Bitfolgen gemäß ihrer Binärdarstellung aufgefasst werden. (Das entspricht einer Operation wie `a & b` in C-Syntax.)

Damit hat TOLL seine Schuldigkeit auch bereits getan. Wir werden nun nach und nach die Fähigkeiten von TOLL beschneiden und dabei jeweils begründen, warum man auf bestimmte Dinge verzichten kann, wenn man den dadurch entstehenden höheren Aufwand nicht scheut.

– Man braucht keine Fließkommazahlen. Wie Sie vielleicht wissen, werden diese intern ohnehin als rationale Zahlen gespeichert, deren Nenner Zweierpotenzen sind.[16] Man könnte also Fließkommazahlen durch Paare von

[15] Übrigens beschreibt der Klassiker *Structure and Interpretation of Computer Programs* sehr schön, wie man einer Sprache wie SCHEME, die nicht objektorientiert ist, Objektorientierung nachträglich hinzufügen kann.

[16] Recht detailliert wird das wie gesagt in den Kapiteln 12 und 13 von KMFI beschrieben.

ganzen Zahlen (Zähler und Nenner) ersetzen und hätte damit in TOLL sogar beliebig genaue rationale Zahlen. Oder man könnte die „echte“ Fließkommaarithmetik simulieren. Siehe dazu Aufgabe 12.13.

– Man braucht keine negativen ganzen Zahlen. Ganz simpel kann man eine ganze Zahl beispielsweise als zwei natürliche Zahlen speichern – eine für den Betrag und eine für das Vorzeichen, wobei letztere nur zwei Werte annehmen kann, etwa 1 für plus und 0 für minus. So werden wir es auch machen. Da TOLL aber mittels `halt` nur einen Wert zurückgeben kann, führen wir noch die Konvention ein, dass bei dieser Zahl das niedrigstwertige Bit das Vorzeichen codiert. Beispiel: Die Ausgabe 26 hat das Bitmuster `11010` und wird interpretiert als das Paar aus den Bitmustern `1101` (13) und `0`, wobei die Null am Ende für *minus* steht. „Gemeint“ ist also die Zahl −13. Siehe dazu auch Aufgabe 12.9.
– Wenn man nun nur noch natürliche Zahlen hat, kann man auf Arrays verzichten, weil man ein ganzes Array in einer Zahl unterbringen kann. Dafür verwenden wir die kanonische Gödelisierung. Siehe Aufgabe 12.10.
– Nun ersetzt man alle arithmetischen Ausdrücke, die nicht auf der rechten Seite von Zuweisungen stehen, durch neue Variablen. Beispielsweise ersetzt man einen Befehl der Form `halt(a+2*b)` durch `h:=a+2*b` gefolgt von `halt(h)`, wobei `h` eine Variable sein soll, die extra für diesen Zweck hinzugenommen wurde.
– Dann werden die verbleibenden arithmetischen Ausdrücke – evtl. unter Hinzunahme weiterer neuer Variablen – so vereinfacht, dass pro Zuweisung nur noch ein Operator vorkommt. Aus `a:=a+b*c` kann man beispielsweise `h:=b*c` gefolgt von `a:=a+h` machen.
– Ebenso werden alle Junktoren entfernt. Wenn diese in `if`- oder `while`-Konstrukten auftreten, dann kann man das immer durch mehrere Einzeltests ersetzen.
– Als Nächstes kann man auf `for`-Schleifen verzichten, weil man dafür auch `while` verwenden kann.
– Und nachdem man das erledigt hat, wird man `while` los, indem man stattdessen `if` und `goto` verwendet. Ebenso kann man mit `case` verfahren. Auch auf den `else`-Zweig in `if`-Befehlen kann man natürlich verzichten. Aufgabe 12.12 demonstriert einige der letzten Schritte an einem Beispiel.
– Wir sind noch nicht fertig! Jetzt ersetzen wir Punkt- durch Strichrechnung, d. h., dass nach diesem Schritt die Operationen `*` und `/` nicht mehr vorkommen. Wie geht das? Statt z. B. `a*b` auszurechnen, wird eine neue Variable `r` mit null initialisiert und dann wird `a`-mal `b` zu `r` addiert. Das kann man (mühsam) mit einer Zählvariablen sowie `if` und `goto` erreichen.
– Und dann werden mit einem ähnlichen Verfahren wie im letzten Schritt Ausdrücke der Form `a+b` oder `a-b` so modifiziert, dass nur noch die Addition bzw. Subtraktion der Zahl 1 zugelassen ist.

Ich habe ein paar Details weggelassen und man muss bei dem ganzen Prozess auf die Reihenfolge achten, damit man nicht in einem späteren Schritt Dinge

wieder einführt, die man in einem früheren Schritt entfernt hatte. Aber so eine „Verdummung“ von TOLL ist wirklich möglich und als Beweis finden Sie im ZIP-Archiv ein Programm, das TOLL-Code in die Sprache NSD übersetzen kann,[17] die nun beschrieben wird:

NSD

- NSD-Programme haben eine ähnliche Syntax wie TOLL, erlauben aber nur ganz wenige Befehle.
- In Variablen können nur natürliche Zahlen gespeichert werden. Daher gibt es keine Typen und Variablen müssen nicht deklariert werden. Sie müssen jedoch vor der Verwendung initialisiert werden.
- Es sind nur drei Sorten von Zuweisungen erlaubt:

  `x:=0`, `x:=x+1` und `x:=x-1`.

  Dabei steht `x` für irgendeine Variable, aber `0` und `1` dürfen *nicht* durch andere Werte ersetzt werden. Multiplikation und Division gibt es gar nicht und das Ergebnis der Rechnung 0 – 1 ist per definitionem 0.
- `if`-Befehle können nur die Form

  ```
  if x=0 then goto 20
  ```

  haben. Man darf `x` durch eine andere Variable und `20` durch eine andere Zeilennummer ersetzen, aber der Rest muss so bleiben.
- `halt`-Befehle müssen die Form `halt(x)` haben, wobei `x` irgendein Variablenname ist.
- Und das war alles! Sämtliche Befehle, die oben nicht explizit aufgeführt wurden, sind verboten!

Diese Sprache ist natürlich – ebenso wie Turingmaschinen – nicht dafür gedacht, mit ihr ernsthaft zu programmieren. Als Beispiel dafür, wie mühsam das wäre, hier ein Programm, das lediglich zwei Zahlen multipliziert:

```
program mult(x,y)
summe := 0; summand := 0; dummy := 0; helper := 0;

# weise "summand" den Wert von "x" zu
10: if x = 0 goto 5;
    summand := summand + 1;
    helper := helper + 1;
    x := x - 1;
    if dummy = 0 goto 10;

# setze "x" wieder zurück (mit Hilfe von "helper")
 5: if helper = 0 goto 6;
    helper := helper - 1;
```

[17] Da es sich nur um eine Demo handelt, gibt es ein paar Einschränkungen: Das TOLL-Programm darf keine Arrays oder Fließkommazahlen verwenden. Das war mir zu aufwendig. Außerdem wird die Übersetzung nur für vergleichsweise kleine Programme sinnvoll funktionieren, weil die resultierenden NSD-Programme sehr umfangreich und ziemlich langsam sind.

```
    x := x + 1;
    if dummy = 0 goto 5;

 6: if y = 0 goto 40;

# addiere "summand" (also "x") zu "summe"
20: if summand = 0 goto 30;
    summe := summe + 1;
    summand := summand - 1;
    if dummy = 0 goto 20;

# mache das alles "y"-mal
30: y := y - 1;
    if dummy = 0 goto 10;

40: halt(summe);
```

NSD steht übrigens augenzwinkernd für „nich so doll“, aber der ernste Hintergrund ist, dass es sich dabei im Wesentlichen um eine sogenannte *Registermaschine* handelt, ein weiteres Tierchen aus dem Zoo der abstrakten Maschinen. Und das, was ich vorhin scherzhaft Verdummung genannt habe, ist natürlich nichts anderes als das, was ein Compiler macht.

Registermaschine

Aufgabe 12.9. Warum kann man negative ganze Zahlen in TOLL nicht im weitverbreiteten Zweierkomplement darstellen?

Aufgabe 12.10. Schreiben Sie in einer Programmiersprache Ihrer Wahl (und besser *nicht* in TOLL) Funktionen die das Auslesen und Speichern von natürlichen Zahlen für ein „Array“ simulieren, das wie in Aufgabe 12.4 gödelisiert ist. (Der Inhalt des Arrays soll sich also die ganze Zeit in einer einzigen natürlichen Zahl befinden.)

Aufgabe 12.11. Implementieren Sie analog zu Aufgabe 12.10 einen Stack, der mit den Operationen `push` und `pop` modifiziert werden kann, als eine einzige Zahl.

Aufgabe 12.12. Schreiben Sie das TOLL-Programm aus der Lösung von Aufgabe 12.8 so um, dass `while`, `else` und Junktoren nicht mehr vorkommen. Verwenden Sie stattdessen `goto` und ggf. neue Variablen. Arithmetische Ausdrücke sind nur auf der rechten Seite von Zuweisungen erlaubt und dürfen jeweils höchstens einen der Operatoren `+`, `-`, `*` oder `/` enthalten.

⋆**Aufgabe 12.13**. Das TOLL-Programm `heron.TOLL` aus dem ZIP-Archiv berechnet näherungsweise die Quadratwurzel einer Zahl mit dem Heron-Verfahren. Zeigen Sie anhand dieses Beispiels folgendermaßen, dass man Gleitkommazahlen eigentlich „nicht braucht“:

Schreiben Sie das Program so um, dass für die Eingabe und sämtliche Berechnungen ausschließlich ganze Zahlen benutzt werden. Ersetzen Sie dafür alle Gleitkommazahlen jeweils durch eine ganzzahlige Mantisse und einen ganzzahligen Zehnerexponen-

ten.[18] Das Programm erhält als Eingabe also nun *zwei* Zahlen statt einer und jede `float`-Variable wird durch jeweils zwei `integer`-Variablen ersetzt.

Wenn Sie Zahlen in dieser Darstellung addieren, subtrahieren oder vergleichen wollen, müssen Sie zunächst dafür sorgen, dass die Zehnerexponenten identisch sind. Bei der Division müssen Sie darauf achten, dass Sie nur ganzzahlige Divisionen durchführen dürfen. Es wird also ggf. Rundungsfehler geben.[19] Sie können aber durch Vergrößern der Mantisse des Dividenden die Rundungsfehler klein halten.

Da man mit dem Befehl `halt` nur eine Zahl ausgeben kann, dürfen Sie ganz am Ende des Programms das Ergebnis wieder in einen `float`-Wert umrechnen. Allerdings darf dies erst dann geschehen, wenn der Näherungswert für die Wurzel bereits ausgerechnet wurde.

Aufgabe 12.14. Im ZIP-Archiv befinden sich ein Interpreter für die Sprache NSD nebst Anleitung sowie ein paar Beispielprogramme. Schauen Sie sich die an und schreiben Sie dann ein NSD-Programm, das den größten gemeinsamen Teiler zweier Zahlen berechnet.

Wie Sie vielleicht schon geahnt haben, fügen wir noch einen weiteren Compiler hinzu:[20] NSD-Programme werden in Code für TM2012 übersetzt. Das ist wesentlich weniger Arbeit als oben, weil der NSD-Befehlssatz so überschaubar ist. Man muss nur die folgenden Schritte durchführen:

(i) Alle Variablen, die im NSD-Programm vorkommen, werden durchnummeriert. Im Folgenden nennen wir die n-te Variable `xn`. Für alle *temporären* Variablen (also die, die keine Argumente des Programms sind) wird hinter dem Ende des (beschriebenen) Tapes je eine Null angefügt.

(ii) Für Zuweisungen der Form `xn:=0` wandert TM2012 zum Start des Bandes, dann von dort zur n-ten Variable und überschreibt sie mit Nullen.

(iii) Für Zuweisungen der Form `xn:=xn-1` wandert TM2012 zum Start des Bandes, dann von dort zur n-ten Variable und subtrahiert eins. Wie das geht, haben wir auf Seite 58 gesehen.

(iv) Für Zuweisungen der Form `xn:=xn+1` wandert TM2012 zum Start des Bandes, dann von dort *vor* die n-te Variable, verschiebt den Bandinhalt ab dieser Position um eine Stelle nach rechts (um Platz für einen eventuellen Übertrag zu schaffen) und addiert schließlich eins. Auch dafür haben wir den Code bereits gesehen (den für das Verschieben in Aufgabe 11.6). Danach wird ggf. noch ein überflüssiges Leerzeichen, das durch das Verschieben entstanden sein könnte, entfernt.

(v) Für einen Befehl der Form `if xn=0 goto 10` wandert TM2012 zum Start des Bandes, von dort zur n-ten Variable und prüft, ob diese nur aus Nullen besteht. Ist das der Fall, dann wird entsprechend der Zustand gewechselt.

[18] Z. B. kann man die Zahl 4.2 als $42 \cdot 10^{-1}$ oder auch als $420 \cdot 10^{-2}$ darstellen – dabei wäre 42 oder 420 die Mantisse und -1 bzw. -2 der Zehnerexponent.

[19] Natürlich kommt es auch bei der üblichen Division mit Gleitkommazahlen in JAVA oder C zu Rundungsfehlern. Details dazu in den Kapiteln 12 und 13 von KMFI.

[20] Und den gibt es auch als ausführbares Programm im ZIP-Archiv.

(vi) Und der Befehl `halt(xn)` wird dadurch realisiert, dass alle Variablen vor und hinter `xn` entfernt werden und die Maschine dann angehalten wird.

Für kleine TOLL-Programme wie z. B. das zum Berechnen der Fakultät sollten Sie den Übersetzungsprozess wirklich einmal ausprobieren.[21] Die Checkboxen mit der Beschriftung „Parameter übersetzen“ sind dafür da, automatisch negative Zahlen bei der Ein- und Ausgabe zu konvertieren. Wenn Sie den übersetzten Code in TM2012 tatsächlich laufen lassen wollen, fangen Sie mit kleinen Werten an! Für die meisten Programme werden Sie die Verzögerung komplett abstellen und die Checkbox „Tape zeigen“ deaktivieren müssen, denn sonst warten Sie bis zum Sankt-Nimmerleins-Tag auf das Ergebnis.

Trotz dieser Einschränkungen sind Sie nun hoffentlich davon überzeugt, dass eine Turingmaschine (zumindest theoretisch) alles berechnen kann, was herkömmliche Computer berechnen können.[22] Das Ziel der kleinen *Tour de Force* durch drei sehr unterschiedliche „Computertypen“ war es aber auch, die folgende Aussage zu motivieren:

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* waren. 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.

[21] Beachten Sie, dass die Programme weder Fließkommazahlen noch Arrays verwenden dürfen.

[22] Und es sollte wohl klar sein, dass umgekehrt natürlich auch Ihr PC alles berechnen kann, was eine Turingmaschine berechnen kann. Das erkennt man schon daran, dass es Programme wie FLACI oder TM2012 gibt. Aber vielleicht denken Sie einmal darüber nach, wie Sie eine Turingmaschine in einer Programmiersprache Ihrer Wahl simulieren würden.

Aufgabe 12.15. 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?

turingvollständig

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 TOLL, NSD und 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*.

FRACTRAN

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

$$\frac{17}{91}\ \frac{78}{85}\ \frac{19}{51}\ \frac{23}{38}\ \frac{29}{33}\ \frac{77}{29}\ \frac{95}{23}\ \frac{77}{19}\ \frac{1}{7}\ \frac{11}{13}\ \frac{13}{11}\ \frac{15}{14}\ \frac{15}{2}\ \frac{55}{1} \tag{12.1}$$

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.[23] Unter dem richtigen Blickwinkel betrachtet ist (12.1) 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, finden Sie in dem am Rande verlinkten Video.

⋆**Aufgabe 12.16**. Übersetzen Sie (12.1) in ein NSD-Programm.

13. Akzeptierende Turingmaschinen und formale Sprachen

Wir arbeiten nun wieder mit Turingmaschinen. Die Überlegungen aus dem letzten Abschnitt 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 Sprache zu codieren. Wir klären als Erstes die Frage, welche Klasse von Sprachen zu den Turingmaschinen passt.

Definition

Eine Turingmaschine T mit Eingabealphabet I **akzeptiert** ein Wort $w \in I^*$, wenn die von T berechnete Funktion f_T für dieses Wort einen Funktionswert hat, wenn also $f_T(w) \neq \perp$ gilt. (Mit anderen Worten:[24] Wenn T bei der Eingabe

[23] Das gilt eigentlich für alle Programmiersprachen. Bei den üblichen haben wir uns nur an eine Art „Standardinterpretation“ gewöhnt. Was Sie auf Ihrer Tastatur tippen, erzeugt elektronische Impulse, die vom Computer übersetzt werden. Und als Antwort sorgt er dafür, dass Punkte auf Ihrem Bildschirm aufleuchten, deren Muster ihr Gehirn erkennt. Intern gibt es nur Bits.

von w in einem akzeptierenden Zustand terminiert.) Die **von T akzeptierte Sprache** ist dann definiert als

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

Für das weitere Vorgehen wird das bereits aus den vorhergehenden Kapiteln bekannte Konzept des Nichtdeterminismus hilfreich sein. Man kann aus einer gewöhnlichen Turingmaschine $T = (S, I, \Gamma, \delta, s_0, \square, F)$, die nach unserer Definition deterministisch agiert, eine *nichtdeterministische Turingmaschine* machen, indem man für δ partielle Funktionen von $S \times \Gamma$ nach $\mathscr{P}(S \times \Gamma \times \{\leftarrow, \rightarrow\})$ 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.

nichtdeterministische Turingmaschine

Aufgabe 13.1. Verständnisfrage: Bei nichtdeterministischen Turingmaschinen könnte man auf das adjektiv *partiell* für δ verzichten. Wieso?

Aufgabe 13.2. 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 13.1

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

Beweis. Sei $T = (S, I, \Gamma, \delta, s_0, \square, F)$ 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 $(S \times \Gamma) \times (S \times \Gamma \times \{\leftarrow, \rightarrow\})$ auf, bei denen $y \in \delta(x)$ gilt,[25] und nennen sie e_1 bis e_w. Jedes Mal, wenn T eine Entscheidung[26] 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 $\{1, \ldots, w\}$ beschreiben; und zwar in dem Sinne, dass das Tupel $(k_1, k_2, \ldots, k_n)$ bedeutet, dass die Turingmaschine nacheinander die „Entscheidungen“ e_{k_1}, e_{k_2} bis e_{k_n} getroffen hat.

[24] Wenn man es ganz genau nimmt, dann sind diese beiden Beschreibungen nicht äquivalent, weil wir bei der Definition von f_T verlangt hatten, dass der Bandinhalt am Ende eine bestimmte Form haben muss. Das ist aber offensichtlich ein Detail, das man vernachlässigen kann.

[25] Das sind quasi alle Einträge in einer Tabelle wie auf Seite 53, wobei im nichtdeterministischen Fall in einer Tabellenzelle jedoch mehrere Einträge stehen können.

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

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

$$(), (1), (2), (1,1), (1,2), (2,1), (2,2), (1,1,1), (1,1,2), (1,2,1), \ldots \tag{13.1}$$

Für die deterministische Turingmaschine T', die T simulieren soll, verwenden wir nun der Einfachheit halber eine Mehrband-Maschine mit drei Bändern. Auf dem ersten Band steht die ganze Zeit unverändert das Eingabewort. Auf dem zweiten Band generiert T' nacheinander die Tupel der Sequenz (13.1). 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. Kommt sie dabei in einen akzeptierenden Zustand, dann hält sie an. Wenn nicht, wird das nächste Tupel ausgewählt. (Die Folge (13.1) 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. ■

Definition

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.[27] Weil damit das leere Wort nicht abgeleitet werden kann, wird noch die ε-**Sonderregel** eingeführt: Die Produktion $S \to \varepsilon$ 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.

Aufgabe 13.3. Welche der folgenden Grammatiken sind kontextsensitiv?

(i) $S \to \mathtt{a}S \mid \mathtt{ab}T,\ \mathtt{b}T \to \mathtt{cc}$
(ii) $S \to \mathtt{a}S \mid \mathtt{ab}T,\ \mathtt{ab}T \to \mathtt{cc}$
(iii) $S \to \varepsilon \mid \mathtt{a}S \mid \mathtt{ab}T,\ \mathtt{b}T \to \mathtt{cc}$
(iv) $S \to \mathtt{a}S \mid \mathtt{ab}T,\ \mathtt{b}T \to \mathtt{cc},\ T \to \varepsilon$
(v) $S \to \varepsilon \mid \mathtt{a} \mid \mathtt{ab}T,\ \mathtt{b}T \to \mathtt{cc}$

Definition

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.

Zum Begriff „kontextsensitiv“ siehe die Erläuterung zu „kontextfrei“ am Anfang von Abschnitt 9. Woher der Begriff „rekursiv aufzählbar“ kommt, wird später noch deutlich werden.

[27] In 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.

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 13.4. Begründen Sie, dass jede kontextfreie Sprache kontextsensitiv ist.

Satz 13.2

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 *nicht*deterministische Turingmaschine konstruieren – aus der wir anschließend nach Satz 13.1 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 $\alpha \to \beta$. 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.

Nun wollen wir eine Grammatik konstruieren, die dieselbe Sprache wie eine vorgegebene deterministische Turingmaschine akzeptiert. Wir machen das exemplarisch für die TM2012.[28] Wenn Sie den Beweis verstanden haben, wird Ihnen auch klar sein, wie Sie ihn auf andere Turingmaschinen übertragen können.

[28] Akzeptanz für die TM2012 definieren wir folgendermaßen: Am Anfang wird ein Wort aus Σ^*_{bool} hinter das Bandanfangszeichen $*$ geschrieben, auf das nur noch Leerzeichen folgen. Die TM2012 startet wie üblich und das Wort wird akzeptiert, wenn der Programmablauf mit H beendet wird. Anderenfalls wird es nicht akzeptiert.

Die konkrete TM2012 die wir konvertieren wollen, hat wie üblich das Bandalphabet $\Gamma = \{0, 1, \square, *\}$ und soll n Zustände haben. Die Terminalsymbole der Grammatik sind natürlich 0 und 1. Als Nichtterminalsymbole verwenden wir 3-Tupel aus $\Gamma \times \Gamma \times \{0, \ldots, n\}$. 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 k-ten Zustand befindet. Die Produktionen werden dafür sorgen, dass nur Wörter abgeleitet werden können, in denen maximal ein Tripel mit von null verschiedener dritter Komponente vorkommt.

Ist nun $(\sigma\ \sigma'\ \mathtt{R}\ d)$ eine der Alternativen im k-ten Zustand,[29] dann fügen wir für alle $\varphi, \chi, \tau \in \Gamma$ die Produktion

$$(\varphi, \sigma, k)(\chi, \tau, 0) \to (\varphi, \sigma', 0)(\chi, \tau, k + d)$$

hinzu. Analog wird es Produktionen der Form

$$(\varphi, \tau, 0)(\chi, \sigma, k) \to (\varphi, \tau, k + d)(\chi, \sigma', 0)$$

geben, wenn wir $(\sigma\ \sigma'\ \mathtt{L}\ d)$ im k-ten Zustand finden. Und es kommt noch für alle $\varphi \in \Gamma$ die Produktion

$$(\varphi, \sigma, k) \to (\varphi, \sigma', 0)$$

dazu, wenn $(\sigma\ \sigma'\ \mathtt{H}\ d)$ zum k-ten Zustand gehört.

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 TM2012 in den Anfangszustand versetzen:

$$S \to (*, *, 1)S_2 \qquad S_2 \to (0, 0, 0)S_2 \mid (1, 1, 0)S_2 \mid S_3$$
$$S_3 \to (\square, \square, 0)S_3 \mid (\square, \square, 0)$$

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 Wort abgeleitet werden, in dem alle Tripel null als letzte Komponente haben. Ist man so weit gekommen, dann wird durch die Produktionen

$$(0, \sigma, 0) \to 0 \qquad (1, \sigma, 0) \to 1$$
$$(\square, \sigma, 0) \to \varepsilon \qquad (*, \sigma, 0) \to \varepsilon$$

der ursprüngliche Bandhinhalt, der die ganze Zeit konserviert wurde, in Form von Terminalsymbolen wiederhergestellt und die irrelevanten Bandzeichen werden entfernt. ■

[29] Zur Syntax der TM2012 siehe das entsprechende PDF im ZIP-Archiv.

Was noch fehlt, sind die kontextsensitiven Sprachen. Die spielen zwar in der Theoretischen Informatik keine so große Rolle,[30] aber wir sind ihrer Charakterisierung durch abstrakte Automaten so nahe, dass wir sie zumindest kurz skizzieren wollen.

Auf Seite 56 hatten wir schon über einseitig beschränkte Turingmaschinen (wie z. B. die TM2012) gesprochen. Man kann solch eine Maschine 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 $* w \dashv$ ist, dann kann die Maschine nur die $|w|$ Kästchen des Bandes benutzen, die w ursprünglich einnahm. Solche Turingmaschinen nennt man *linear beschränkt*.[31] Obwohl das nach einer sehr drastischen Restriktion aussieht,[32] können solche Maschinen doch noch recht viel.

Bandende $\dashv$

linear beschränkt

Satz 13.3

Eine Sprache ist genau dann kontextsensitiv, wenn sie von einer *nichtdeterministischen* linear beschränkten Turingmaschine akzeptiert wird.

Beweis. Wir orientieren uns am Beweis von Satz 13.2. Dessen ersten Teil können wir wortwörtlich übernehmen. Aufgrund der speziellen Eigenarten kontextsensitiver Grammatiken wird der Bandinhalt niemals länger werden.[33] Daher kann man die Aufgabe auch mit einer linear beschränkten Turingmaschine erledigen.

Wenden wir uns dem zweiten Teil zu.[34] 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 Anfang bzw. das direkt vor dem Bandende handelt. Alle Produktionen müssen natürlich entsprechend angepasst werden, aber das ist nur Fleißarbeit. Entscheidend ist, dass man am Anfang weder das Bandanfangszeichen 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 Nichtdeterminismus. Kann man den Wortbestandteil „nicht“ streichen? Akzeptieren deterministische linear beschränkte Turingmaschinen dieselben Sprachen wie die nichtdeterministischen? (Den Beweis von Satz 13.1 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

[30] Im Standardwerk von Hopcroft et al., das in der Literaturliste aufgeführt ist, wurde das Kapitel über kontextsensitive Sprachen beispielsweise ab der zweiten Auflage entfernt.

[31] Im Englischen *linear bounded automaton*, abgekürzt LBA.

[32] Andererseits ist das in gewissem Sinne sogar „realistischer“ als eine Standard-Turingmaschine, weil echte Computer auch nur über begrenzten Speicherplatz verfügen.

[33] Das gilt zwar nicht für die ε-Sonderregel, aber die kann die Maschine als Spezialfall zuerst abhandeln.

[34] Dafür stelle man sich TM2012 entsprechend modifiziert vor, damit daraus eine linear beschränkte Turingmaschine wird.

erstes LBA-Problem

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.

⋆**Aufgabe 13.5**. 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 13.6. 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 9.3 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 13.7. 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 Grammatik[35]

$$G = (N_1 \cup \{S\}, \Sigma, P_1 \cup \{S \to S_1 S, S \to S_1\}, S)$$

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 13.8. In den Lösungen der Aufgaben 13.6 und 13.7 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 13.9. 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 13.4 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 13.9 umwandelt, kann man den Beweis von Satz 9.3 kopieren und sich überzeugen, dass so etwas wie in Aufgabe 13.6 nicht passieren kann. ■

Lemma 13.5 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 $\varepsilon \notin L(G_1)$ (sonst muss man die Produktion $S_1 \to \varepsilon$ entfernen) und auf den linken Seiten der Produktionen seien keine Nichtterminalsymbole. (Siehe Aufgabe 13.9.) G_1' sei die Grammatik $(N_1', \Sigma, P_1', S_1')$, die aus G_1 entsteht, indem man jedes Symbol $A \in N_1$

[35] Siehe die Lösung von Aufgabe 9.7.

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

$$S \to T_1 \mid \varepsilon \qquad T_1 \to S_1 T_2 \mid S_1 \qquad T_2 \to S_1' T_1 \mid S_1'$$

enthält. Durch das „Abwechseln“ von G_1 und G_1' wird dafür gesorgt, dass so etwas wie in Aufgabe 13.7 nicht passieren kann. ■

Lemma 13.6

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 13.3 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, [36] 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 13.7

Satz von Immerman und Szelepcsényi
Die Menge der kontextsensitiven Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen die Bildung von Komplementen.

Für diesen Satz erhielten seine beiden Entdecker 1995 den Gödel-Preis.[37] 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.

[36] Oder entsprechend größerem Bandalphabet, siehe Aufgabe 13.5.

[37] Der Slowake Róbert Szelepcsényi war ein 20 Jahre alter Student, als er – unabhängig von dem Amerikaner Neil Immerman, der zeitgleich dasselbe schaffte – den Beweis fand.

V

Berechenbarkeitstheorie

In der Berechenbarkeitstheorie, die man auch *Rekursionstheorie* nennt, geht es um die grundsätzliche Frage, was man (mit Computern) überhaupt berechnen kann. Wir werden ab jetzt 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.

14. Berechenbarkeit und Entscheidbarkeit

Wir beginnen mit ein paar technischen Definitionen, um die Begriffe festzulegen, die wir danach meistens eher informell verwenden werden. Nebenbei soll es auch darum gehen, dass Sie diese Termini schon einmal gehört haben, wenn Sie Lehrbücher oder Fachartikel lesen.

Definition

Sei Σ ein Alphabet. Eine partielle Funktion $f\colon \Sigma^* \rightharpoonup \Sigma^*$ heißt **berechenbar**, wenn es eine Turingmaschine T zum Alphabet Σ gibt, für die $f_T = f$ gilt.

Bei der Definition der Gödelisierung hatten wir Funktionen g von einer Sprache L nach $\mathbb{N}$ als berechenbar bezeichnet. Das passt technisch nicht zu der obigen Definition, weil $\mathbb{N}$ keine formale Sprache ist. Es lässt sich aber leicht reparieren, indem man zwei Zeichen des Alphabets von L als null und eins interpretiert und entsprechende Ausgaben, die nur diese beiden Zeichen verwenden, als Binärzahlen. (Für den theoretisch möglichen Fall, dass das Alphabet aus nur einem Zeichen besteht, verwenden wir das Unärsystem.) Analog geht man mit der in der besagten Definition erwähnten Funktion von $g[L]$ nach L um.

Die folgende Definition ist ebenfalls möglichst allgemein gehalten, damit sie für jedes Alphabet anwendbar ist. Die Idee ist, dass die in der Definition vorkommenden Wörter w_1 und w_2 als *ja* und *nein* interpretiert werden.

Definition

Sei Σ ein Alphabet. 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 zum Alphabet Σ gibt, für die für alle $w \in \Sigma^*$ das Folgende gilt:

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

Wichtig ist, dass f_T auf ganz Σ^* definiert ist, dass T also bei *jeder* Eingabe terminiert. Auch die Verwendung des Begriffs der Entscheidbarkeit in der Definition der Gödelisierung lässt sich offenbar leicht anpassen, indem wir die dort auftretende Zahlenmenge $g[L]$ binär über dem Alphabet Σ_{bool} darstellen.

Wir werden uns im Folgenden auf (partielle) Funktionen beschränken, die mit natürlichen Zahlen 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.

Definition

Eine Turingmaschine T mit dem Eingabealphabet Σ_{bool} **berechnet** die partielle Funktion $g\colon \mathbb{N} \rightharpoonup \mathbb{N}$, wenn[1]

$$f_T(w_1) = \begin{cases} w_2 & g(n_{w_1}) = n_{w_2} \\ \bot & g(n_{w_1}) = \bot \end{cases}$$

für alle $w_1, w_2 \in \Sigma^+_{\text{bool}}$ und $f_T(\varepsilon) = \bot$ gilt. Wir verwenden dann auch T als Synonym für g, d. h., wir schreiben $T(n_{w_1}) = n_{w_2}$ bzw. $T(n_{w_1}) = \bot$.
Wenn es eine solche Maschine gibt, dann nennen wir g **berechenbar**.

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 12.4 gödelisieren. Für eine Maschine T, die aus der Eingabe 2^a3^b die Ausgabe $a+b$ macht, würden wir also $T(a,b) = a+b$ schreiben.

[1] Zur Schreibweise n_w siehe Seite 7. 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.

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 14.1. Geben Sie noch weitere Beispiele an.

Aufgabe 14.2. 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 mit *ja* antworten (oder mit etwas, das man als *ja* interpretieren kann), aber ihr Verhalten im negativen Fall ist unbekannt.

Aufgabe 14.3. Warum ist jede entscheidbare Menge semi-entscheidbar?

Aufgabe 14.4. 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 $f : \mathbb{N} \to \mathbb{N}$ gibt, deren Wertebereich A ist.[3]

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]

Satz 14.1

Eine Menge $A \subseteq \mathbb{N}$ ist genau dann semi-entscheidbar, wenn sie rekursiv aufzählbar ist.

[2] Das kann man wegen Satz 13.2 bereits ahnen.

[3] Anders 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.

[4] Ausführliche Beispiele dazu in PYTHON findet man in Kapitel 18 von KMFI.

Beweis. Wir begründen zunächst, warum jede rekursiv aufzählbare Menge A semi-entscheidbar ist. Dafür sei T eine Turingmaschine, die eine Funktion $f : \mathbb{N} \to \mathbb{N}$ 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. ■

Satz 14.2 Sind $A \subseteq \mathbb{N}$ und $\mathbb{N} \setminus A$ rekursiv aufzählbar, dann ist A entscheidbar.

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*. ■

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 der TM2012 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:

`0`	`1`	`B`	`*`	`L`	`R`	`H`	`-`	`(`	`)`
0	1	2	3	4	5	6	7	8	9

Ein Programm für die TM2012 ist nun nichts weiter als eine Folge dieser Zahlen, die wir als Dezimalzahl interpretieren können, weil wir mit zehn Zeichen auskommen. Die „Sprungadressen“ im Programm werden binär codiert.

[5] Auch dazu mehr in Kapitel 18 von KMFI.

```
(((0 0 R 1)              8 8 8 0 0 5    1 9
  (1 0 L 0)                 8 1 0 4    0 9
  (* * R 2))                8 3 3 5  1 0 9 9
 ((B B L -1)              8 8 2 2 4 7   1 9
  (0 0 H 0)))               8 0 0 6    0 9 9 9
```

Diesem Beispielprogramm entspricht also die Zahl 8880051…060999.

Aufgabe 14.5. 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 12.4 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.

Maschinenbeschreibung

Betrachtet man die Menge aller Turingmaschinen zum selben Eingabe- und Bandalphabet, so kann man deren Maschinenbeschreibungen aufsteigend sortieren.[6] Das würde man die *Standardaufzählung* dieser Menge nennen.

Standardaufzählung

15. Das Halteproblem und der Satz von Rice

Wir betrachten in diesem Abschnitt nur noch Turingmaschinen mit Eingabealphabet Σ_{bool} und Bandalphabet $\Sigma_{\text{bool}} \cup \{\square\}$, was nach Aufgabe 11.7 keine wesentliche Einschränkung ist. 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 12.4 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 14.3 „erst recht“ nicht

[6] Für 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.)

entscheidbar). Aber kann man solche Mengen konkret angeben? Und gibt es „interessante“ Mengen, die nicht (semi-)entscheidbar sind?

Definition

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**.

Satz 15.1

Es gibt eine universelle Turingmaschine.

Beweis. Als Turing 1936 diese Idee hatte, war deren Beschreibung, die die Leser überzeugen sollte, sehr aufwendig. Mit unserem heutigen Wissen über Computer und speziell nach der Lektüre von Abschnitt 12 entlockt uns das aber wohl nur noch ein Achselzucken. Das ZIP-Archiv aus selbigem Abschnitt enthält ja einen (in COMMON LISP geschriebenen) Simulator für die TM2012 und wir haben uns überzeugt, dass wir aus so einem Programm im Prinzip ein Programm für die TM2012 machen können. Das wäre 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). 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.[7]

Die entscheidende Aussage aus Turings Arbeit von 1936 ist die folgende:

Satz 15.2

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.

Problem

[7] Allgemein 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 93.) Für Tupel von Zahlen wie in der Definition von H greifen wir auf die kanonische Gödelisierung derselben zurück.

Wir betrachten zunächst

$$H^* = \{ n \in \mathbb{N} : M_n \text{ hält bei der Eingabe } n \text{ an} \} \tag{15.1}$$

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 14.3 und 14.4.) 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. ■

Definition

Die Menge $\{ n \in \mathbb{N} : M_n \text{ hält bei jeder Eingabe an} \}$ wird als **gleichmäßiges Halteproblem** bezeichnet.

Satz 15.3

Das gleichmäßige Halteproblem ist nicht entscheidbar.

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 (15.1) entscheiden. ■

Die obigen Aussagen über die Nichtentscheidbarkeit einzelner Probleme lassen sich dramatisch verallgemeinern.

Satz 15.4

Satz von Rice
Sei $\mathscr{P}$ die Menge aller partiellen berechenbaren Funktionen von $\mathbb{N}$ nach $\mathbb{N}$ und $\mathscr{A}$ eine nichtleere echte Teilmenge von $\mathscr{P}$. Dann ist

$$A = \{ i \in \mathbb{N} : M_i \text{ berechnet eine Funktion aus } \mathscr{A} \}$$

nicht entscheidbar.

Beweis. Wir können o.B.d.A. annehmen, dass die Funktion $\emptyset$ nicht zu $\mathscr{A}$ gehört.[8] (Sonst betrachten wir $\mathbb{N} \setminus A$, siehe Aufgabe 14.2.) Da $\mathscr{A}$ nicht leer ist, gibt es eine Zahl $a \in A$.

Analog zum Beweis von Satz 15.3 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 (15.1) 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 $T(i) \in A$. 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 Rice[9] ist, dass sich Fragen wie die folgenden niemals „vollautomatisiert“ klären lassen:

- Berechnet das Programm M eine bestimmte Funktion $f : \mathbb{N} \rightharpoonup \mathbb{N}$, 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 15.1. Warum gilt der Satz von Rice nicht für die Fälle $\mathscr{A} = \emptyset$ und $\mathscr{A} = \mathscr{P}$?

In den obigen Beweisen haben wir mehrfach auf eine bestimmte Art ausgenutzt, dass die Menge H^* aus (15.1) nicht entscheidbar ist. Daraus kann man ein allgemeines Prinzip machen.

[8] $\emptyset$ ist die Funktion, die immer den Funktionswert $\perp$ hat.
[9] Der amerikanische Mathematiker Henry Gordon Rice bewies ihn 1953.

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 15.5

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 $g : \mathbb{N} \to \mathbb{N}$ 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*. ■

Aufgabe 15.2. Welches Problem wurde im Beweis des Satzes von Rice auf welches reduziert? Und wie war die entsprechende Rollenverteilung beim gleichmäßigen Halteproblem?

16. 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.

Satz 16.1

Alle kontextsensitiven Sprachen sind entscheidbar.

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. $S \to \varepsilon$ – 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 102.) ■

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 16.2 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 7 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 16.3 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 16.1 gibt es zu jedem $i \in \mathbb{N}$ eine Turingmaschine N_i, die immer anhält und für die $L(N_i) = L_i$ gilt.[10] Man überlege sich, dass die Folge der zugehörigen Maschinenbeschreibungen berechenbar ist. Nach Lemma 16.2 gibt es eine entscheidbare Sprache L, die von keiner der Maschinen aus dieser Aufzählung akzeptiert wird und die daher nicht kontextsensitiv sein kann. ■

Wie steht es mit den Abschlusseigenschaften?

Satz 16.4 Die Menge der rekursiv aufzählbaren Sprachen über einem festen Alphabet Σ ist abgeschlossen gegen Vereinigung, Durchschnitt, Konkatenation und kleenesche Hülle.

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

[10] Um 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.

auf und hängen die beiden erhaltenen Wörter hintereinander. Damit erzeugen wir nach und nach sicher alle Wörter aus $L_1 \circ L_2$.[11]

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 $n = 0,1,2,\ldots$ 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:

$$
\begin{aligned}
&(),\\
&(0),\\
&(1),(0,0),(0,1),(1,0),(1,1),\\
&(2),(0,2),(1,2),(2,0),(2,1),(2,2),(0,0,0),(0,0,1),\ldots
\end{aligned}
$$

Der Rest sollte dann klar sein.

Den Teil des Beweises für Vereinigung und Durchschnitt können Sie sich selbst überlegen – siehe Übung U 39. (Tipp: In diesen Fällen ist es ggf. leichter mit akzeptierenden statt mit aufzählenden Turingmaschinen zu arbeiten.) ■

Aufgabe 16.1. Wenn L eine rekursiv aufzählbare Sprache über Σ ist, ist $\Sigma^* \setminus L$ dann auch immer rekursiv aufzählbar?

[11] Das 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.

VI

Komplexitätstheorie

Im letzten Kapitel 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 anschneiden können. Ausführlicher wird es in den Aufzeichnungen der Vorlesung aus dem Sommersemester 2014 behandelt.

17. 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
```

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 kennen, aber der Vollständigkeit halber folgt hier eine Definition.

Landau-Symbole

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 17.1. Lesen Sie sich noch einmal durch, worum es bei der Landau-Notation geht, falls Sie das vergessen haben. (Siehe z. B. Kapitel 39 von KMFI.)

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 gröberes 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 17.2.** 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 Exponentialfunktion[4] wächst.

Aufgabe 17.3. Fallen Ihnen noch andere Funktionen ein, die wie die aus Aufgabe 17.2 *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.

Definition

Ein **Entscheidungsproblem** ist eine formale Sprache L. Und eine Turingmaschine, die L entscheidet, wird als **Lösung** des Problems bezeichnet.

$E = \{ w \in \Sigma_{\text{bool}}^+ : n_w \equiv 0 \pmod 2 \}$ ist z. B. ein (ganz einfaches) Entscheidungsproblem. Es geht lediglich darum, festzustellen, ob man die Binärdarstellung einer geraden Zahl vor sich hat.

[1] Man 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.

[2] Zumindest theoretisch könnte man sämtliche Lösungen tabellieren. So etwas ist in der Informatik durchaus üblich.

[3] Rein 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.

[4] Natürlich 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 13) oder wie in der Definition von *entscheidbar* auf Seite 80.

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 **P** ist.

P steht für *polynomiales* Laufzeitverhalten und **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

[5] Und 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.

Lemma 17.1

Wenn eine Mehrband-Turingmaschine ein Entscheidungsproblem in polynomialer Zeit[7] 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.

Beweis. Auf Seite 56 wurde skizziert, wie man eine Mehrband-Turingmaschine T mit einer Mehrspur-Turingmaschine T' simulieren 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 auf Seite 55 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 11.7) 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] ■

[6] Beachten Sie, dass es schon schwierig ist, überhaupt zu definieren, was mit der „durchschnittlichen“ Laufzeit überhaupt gemeint ist.

[7] Bei 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 übetragen wir diese Bezeichungen auch auf andere Rechnermodelle, wobei ein sinnvolles Maß für die Anzahl der Rechenschritte vorausgesetzt wird.

[8] Nebenbei bemerkt wird das Eingabewort natürlich auch entsprechend länger.

Satz 17.2 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. 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.[9] 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` 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.

[9] Natürlich 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.

- 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 17.1 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 17.4. Vergleichen Sie den im Beweis von Satz 17.2 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 17.5. Müssen im Beweis von Satz 17.2 die Zellen auf dem Band, das den Speicher simuliert, nach Adressen sortiert sein?

Aufgabe 17.6. Turingmaschinen haben (auf ihren Bändern) theoretisch unbegrenzten Speicherplatz zur Verfügung. Für den simulierten Computer aus Satz 17.2 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 17.2 und Lemma 17.1 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.

18. 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 Zahlschriftt 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 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.[10]

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 18.1 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.

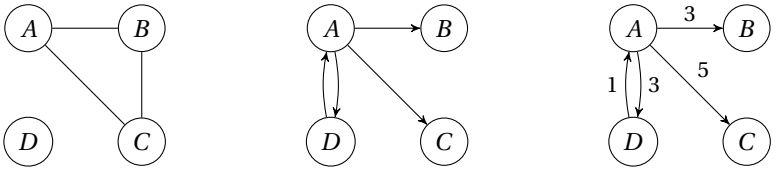
Definition

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 $f : E \to \mathbb{R}$ hinzufügt, die jeder Kante eine Zahl (ihr *Gewicht*) zuordnet.

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

[10] Wäre n die entscheidende Größe, dann würde das nach unserer Definition bedeuten, dass wir die Eingabe unär auf das Band der Turingmaschine schreiben.

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 $f = \{((A, B), 3), ((A, C), 5), ((A, D), 3), ((D, A), 1)\}$.



Definition

Ein **Weg** in einem Graphen ist eine Folge von Knoten des Graphen, in der jeweils zwei aufeinanderfolgende Knoten durch eine Kante verbunden sind.[11] In einem gewichteten Graphen bezeichnet man die Summe der Kantengewichte als **Länge** des Weges. Gibt es keine Gewichte, so werden einfach die Kanten gezählt.[12]

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 18.1**. Inwiefern passt dieses Problem nicht zu denen, über die wir in diesem Kapitel bisher gesprochen haben?

Aufgabe 18.1 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.

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

[11] In 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).

[12] Das entspricht einer Funktion, die jeder Kante das Gewicht 1 zuordnet.

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.

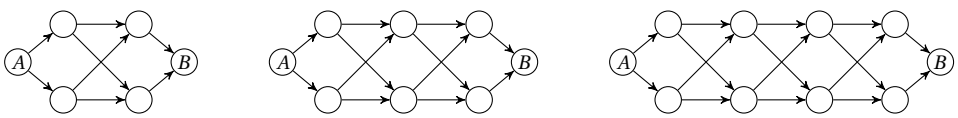
Aufgabe 18.2. Fallen Ihnen mathematische Probleme ein, für die ein Entscheidungsproblem signifkant einfacher ist als ein eng verwandtes Problem, bei dem es nicht nur um *ja* oder *nein* geht?

Doch zurück zur Graphentheorie. Der folgende, absichtlich „naiv“ gestaltete Algorithmus findet für einen gerichteten Graphen (V, E) mit einer Gewichtsfunktion $f : E \to \mathbb{R}_{\geq 0}$ einen kürzesten Weg vom Knoten A zum Knoten B.

Algorithmus 18.1

$Q \leftarrow \{A\}$, $P \leftarrow \{(A, 0)\}$
while $Q \neq \emptyset$ **do**
  $s \leftarrow$ some element of Q
  $Q \leftarrow Q \setminus \{s\}$
  for each $n \in V$ with $(s, n) \in E$ **do**
    for each $(p, w) \in P$ **do**
      if p ends with s and doesn't contain n **then**
        $P \leftarrow P \cup \{(pn, w + f((s, n)))\}$
        $Q \leftarrow Q \cup \{n\}$
      end if
    end for
  end for
end while
if $P' = \{(p, w) \in P : p \text{ ends with } B\} \neq \emptyset$ **then**
  return $(p, w) \in P'$ with $w \leq w'$ for all $(p', w') \in P'$
else
  return `"no path from A to B"`
end if

Das Problem dieses Algorithmus ist, dass er *alle* von A ausgehenden Wege durchprobiert. Die folgende Grafik (bzw. Aufgabe 18.3) zeigt jedoch, dass das zu exponentiellem Wachstum der Laufzeit führen würde.



Aufgabe 18.3. 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 18.2

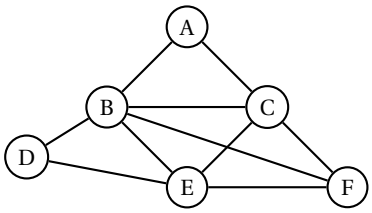
$Q \leftarrow V$, $d[A] \leftarrow 0$, $d[v] \leftarrow \infty$ for all $v \in V \setminus \{A\}$, $p[v] = \perp$ for all $v \in V$
while $Q \neq \emptyset$ **do**
  $u \leftarrow$ some element of $\{v \in Q : d[v] \leq d[w]$ for every $w \in Q\}$
  if $u = B$ **then**
    $r \leftarrow \varepsilon$
    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. Cliquen mit mehr als vier Knoten gibt es nicht.



Die Frage, wie man graphentheoretische Probleme sinnvoll codiert, wird in der Übung U 40 behandelt. Auf einem herkömmlichen Rechner lässt sich **CLIQUE** z. B. folgendermaßen lösen.

Algorithmus 18.3

```
for each V' ⊆ V with |V'| = k do
    s ← true
    for each (v, w) ∈ V' × V' with v ≠ w do
        if {v, w} ∉ E then
            s ← false
        end if
    end for
    if s then
        return "yes"
    end if
end for
return "no"
```

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 Polynom[13] 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 18.4. Begründen Sie, dass der obige Algorithmus 18.3 für **CLIQUE** für *festes k* eine Laufzeit hat, die polynomial von der Anzahl der Ecken des Graphen abhängt.

⋆**Aufgabe 18.5**. 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 Übung U 41.)

⋆**Aufgabe 18.6**. 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.

[13] Zumindest dann, wenn n groß genug ist.

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 18.4

$V' \leftarrow \emptyset$
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 18.8.) Damit ist klar, dass dieser Algorithmus polynomiale Laufzeit hat.

Aufgabe 18.7. Warum so kompliziert? Könnte man in den Pseudocode von Algorithmus 18.4 statt des **select**-Blocks nicht einfach einfügen, dass das Programm sich eine k-elementige Teilmenge von V aussuchen soll?

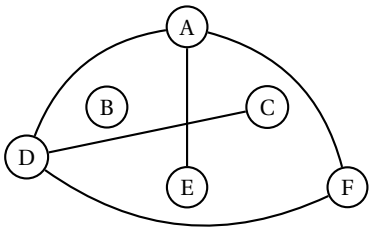
Aufgabe 18.8. Begründen Sie etwas ausführlicher, dass der Test der Cliqueneigenschaft in Algorithmus 18.4 von der Ordnung $\mathcal{O}(n^2)$ ist.

Aufgabe 18.9. Satz 13.1 demonstriert, dass man nichtdeterministische Algorithmen durch deterministische simulieren kann. Wie würde das Laufzeitverhalten eines deterministischen Algorithmus aussehen, der den nichtdeterministischen Algorithmus 18.4 für **CLIQUE** simuliert?

Definition

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?

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 18.10. Welcher Zusammenhang besteht zwischen Cliquen und stabilen Mengen? Wenn Sie es nicht direkt der Definition entnehmen können, dann vergleichen Sie die beiden Beispielskizzen.

Aufgabe 18.11. Was müsste man gemäß Aufgabe 18.10 im Algorithmus 18.3 ändern, damit er **INDSET** statt **CLIQUE** löst?

Aufgabe 18.12. Unter der URL `https://weitz.de/indset/` finden Sie eine interaktive Anwendung, mit der man sich mit **INDSET** vertraut machen kann.

Aufgabe 18.11 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:

Algorithmus 18.5

$E' = \emptyset$
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
return (V, E')

Offensichtlich erhalten wir einen deterministischen Algorithmus für **INDSET**, indem wir Algorithmus 18.5 ausführen und dessen Ausgabe als Eingabe für Algorithmus 18.3 verwenden. Ebenso erhalten wir einen nichtdeterministischen Algorithmus für **INDSET**, indem wir Algorithmus 18.5 ausführen und dessen Ausgabe an Algorithmus 18.4 weitergeben. Diesen „Trick“ wollen wir nun formalisieren.[14]

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 $L_1 \leq_p L_2$.

[14] Man erinnere sich an die Definition des Begriffs *reduzierbar* von Seite 87.

Lemma 18.1

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 18.1 wird noch einmal bestätigt, was durch Aufgabe 18.11 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.

Zertifikat

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.

Definition

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.[15]

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,[16] so kann eine Turingmaschine leicht überprüfen, dass die Formel mit dieser Belegung erfüllt wird. Für die zweite Formel gibt es kein solches Zertifikat.

Aufgabe 18.13. Wie würde ein Zertifikat für **CLIQUE** aussehen?

Satz 18.2

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 13.1 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.

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.

[15] Dabei 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.

[16] 100 steht natürlich für $x_1 = 1$, $x_2 = 0$, $x_3 = 0$.

19. 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,[17] 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 19.1

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 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 $Q(x_1,\ldots,x_m)$ 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:

$$\begin{aligned}&(x_1 \vee x_2 \vee x_3 \vee x_4)\\ &\quad \wedge (x_1 \implies \neg(x_2 \vee x_3 \vee x_4)) \wedge (x_2 \implies \neg(x_1 \vee x_3 \vee x_4))\\ &\quad \wedge (x_3 \implies \neg(x_1 \vee x_2 \vee x_4)) \wedge (x_4 \implies \neg(x_1 \vee x_2 \vee x_3))\end{aligned} \tag{19.1}$$

Und offensichtlich ist die Länge von $Q(x_1,\ldots,x_m)$ von der Ordnung $\mathcal{O}(m^2)$.

[17] Im Englischen **NP**-*hard*, was manchmal falsch als „**NP**-hart“ übersetzt wird.

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	$\cdots$	Zelle $p(n)$
1	1	1	a_1	a_2	$\cdots$	$\square$
2	?	?	?	?	$\cdots$	?
3	?	?	?	?	$\cdots$	?
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
$p(n)$	?	h	1	$\square$	$\cdots$	$\square$

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 $Q(s_{k,1},\ldots,s_{k,p(n)})$, 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.[18] 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.

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:

$$s_{42,17} \wedge b_{42,17,23} \wedge z_{42,5} \implies s_{43,18} \wedge b_{43,17,12} \wedge z_{43,3} \tag{19.2}$$

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.

[18] Die endlich vielen Zustände sowie die Symbole des Bandalphabets werden entsprechend durchnummeriert. Deren Anzahl hängt nur von T und nicht von der Eingabe ab.

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 $1\square\ldots\square$ 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 19.1. Überzeugen Sie sich, dass die Länge der im Beweis von Satz 19.1 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 19.2

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 19.2. ■

⋆**Aufgabe 19.2**. Geben Sie zu (19.1) bzw. zu (19.2) 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**.

Satz 19.3

CLIQUE ist **NP**-vollständig.

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

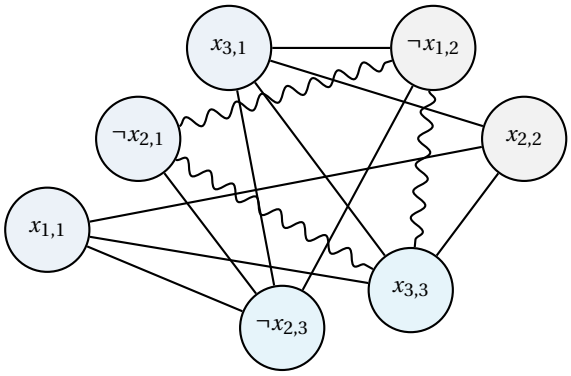
$$V = \{x_{k,m} : x_k \text{ kommt in der } m\text{-ten Klausel vor}\}$$
$$\cup \{\neg x_{k,m} : \neg x_k \text{ kommt in der } m\text{-ten Klausel vor}\}.$$

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 19.4

INDSET ist **NP**-vollständig.

Beweis. Algorithmus 18.5 eignet sich offenbar nicht nur, um **INDSET** $\leq_p$ **CLIQUE** zu zeigen, sondern man kann ihn auch für **CLIQUE** $\leq_p$ **INDSET** verwenden. ■

⋆**Aufgabe 19.3**. Dass **CNF-SAT** **NP**-vollständig ist, kann man auch anders als im Beweis von Korollar 19.2 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 (19.1) und (19.2) Zeitin-Transformationen durch.

⋆**Aufgabe 19.4**. **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 **NP** $\setminus$ **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,[19] 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.[20] 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.

Sollte man bei **NP**-vollständigen Problemen also alle Hoffnung auf eine effiziente Lösung aufgeben, wenn man sich der Meinung der Experten anschließt und **P** ≠ **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.[21] Diverse Beispiele für solche Ansätze findet man in der am Anfang des Kapitels erwähnten Playlist.

[19] Das kann man sich überlegen, indem man Aufgabe 18.9 verallgemeinert.
[20] Ersetzt man 2 durch 3, dann stimmt das nicht mehr. Siehe Aufgabe 19.4.
[21] Das 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. $\mathtt{a}^4 = \mathtt{a}\circ\mathtt{a}^3 = \mathtt{a}\circ\mathtt{a}\circ\mathtt{a}^2 = \mathtt{a}\circ\mathtt{a}\circ\mathtt{a}\circ\mathtt{a} = \mathtt{aaaa}$. $v^3 = \mathtt{ababab}$. $\mathtt{a}^2 v^2 = \mathtt{aaabab}$. $\mathtt{ab}^3 = \mathtt{abbb}$. $(\mathtt{ab})^3 = \mathtt{ababab}$. (Beachten Sie, dass diese beiden Ausdrücke unterschiedlich sind.) $|v^{10}| = 10 \cdot |v| = 20$.

Lösung 1.4. $\circ$ ist nicht kommutativ. Beispielsweise sind $\mathtt{00}\circ\mathtt{1} = \mathtt{001}$ und $\mathtt{1}\circ\mathtt{00} = \mathtt{100}$ 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. $X \circ Y = \{\mathtt{abbc}, \mathtt{abd}, \mathtt{abe}, \mathtt{acbc}, \mathtt{acd}, \mathtt{ace}\}$.

Lösung 1.7. Der Reihe nach:

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

$\Sigma^1 = \{\mathtt{a}, \mathtt{b}\}$

$\Sigma^2 = \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$

$\Sigma^3 = \{\mathtt{aaa}, \mathtt{aba}, \mathtt{baa}, \mathtt{bba}, \mathtt{aab}, \mathtt{abb}, \mathtt{bab}, \mathtt{bbb}\}$

Lösung 1.8. Es ist die Menge aller Wörter der Länge k, die man mit Symbolen aus Σ bilden kann.

Lösung 1.9. Das ist eine Kombinatorik-Aufgabe, erstes Semester Mathematik. Die Antwort ist $|\Sigma|^k$.

Lösung 1.10. $X^1 = X^0 \circ X = \{\varepsilon\} \circ X = \{\varepsilon w : w \in X\} = \{w : w \in X\} = X$.

Lösung 1.11. 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 $X = Y = \{\mathtt{a}, \mathtt{aa}\}$ gilt $X \circ Y = \{\mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}\}$, also $3 = |X \circ Y| < |X| \cdot |Y| = 4$.

Lösung 1.12. Es ist die Menge aller Wörter, die man mit Symbolen aus Σ bilden kann. Siehe Aufgabe 1.8.

Lösung 1.13. So geht es los:

$$X^0 = \{\varepsilon\} \qquad Y^0 = \{\varepsilon\}$$
$$X^1 = \{\mathtt{a}\} \qquad Y^1 = \{\varepsilon, \mathtt{a}\}$$
$$X^2 = \{\mathtt{aa}\} \qquad Y^2 = \{\varepsilon, \mathtt{a}, \mathtt{aa}\}$$
$$X^3 = \{\mathtt{aaa}\} \qquad Y^3 = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}\}$$

Allgemein gilt offensichtlich $Y^k = X^0 \cup \cdots \cup X^k$. X^* und Y^* sind dann wieder identisch $X^* = Y^* = \{\mathtt{a}^k : k \in \mathbb{N}\}$. Aber: $Y^+ = Y^* \neq \{\mathtt{a}^k : k \in \mathbb{N}^+\} = X^+$.

Lösung 1.14. Es gilt grundsätzlich nicht, wenn X das leere Wort enthält. Das hätte Ihnen beim Lösen von Aufgabe 1.13 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 1.15. 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 und dass Ausdrücke wie w^* oder w^+ weder definiert sind noch Sinn ergeben, wenn w ein Wort ist.

Lösung 1.16. 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 1.9) und ist damit endlich.

Lösung 1.17. 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 1.18. `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 $L(G) = \{\varepsilon, \mathtt{a}\}$.

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 $S \to S$ und $T \to$ a besteht.

Lösung 2.5. Das könnte so aussehen:

$$S \Rightarrow_G SS \Rightarrow_G S\mathtt{[}S\mathtt{]} \Rightarrow_G S\mathtt{[[}S\mathtt{]]} \Rightarrow_G S\mathtt{[[]]} \Rightarrow_G \mathtt{[}S\mathtt{][[]]} \Rightarrow_G \mathtt{[][[]]}$$

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 hinbekommt. 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 $S \to \varepsilon \mid \mathtt{a}S$ besteht.

Lösung 2.9. Nein. Man kann unter anderem das Wort aa nicht herleiten. Die von dieser Grammatik erzeugte Sprache ist $\{(\mathtt{ab})^n : n \in \mathbb{N}\}$.

Lösung 2.12. 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 4.1. Es handelt sich um $\{\mathtt{abb}\}^* = \{(\mathtt{abb})^n : n \in \mathbb{N}\}$.

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 $T \to \mathtt{c}$ durch die beiden Produktionen $T \to \mathtt{c}U$ und $U \to \varepsilon$ ersetzen.

Lösung 4.5. Man kann ein neues, bisher nicht genutztes Nichtterminalsymbol U einführen und die Produktion $T \to \mathtt{ab}S$ durch die beiden Produktionen $T \to \mathtt{a}U$ und $U \to \mathtt{b}S$ ersetzen.

Lösung 4.6. Wenn die Sprache aus den Wörtern w_1 bis w_n besteht, dann verwenden wir die Produktionen $T_i \to w_i U$ für $i = 1, \ldots, n$ sowie zusätzlich $U \to \varepsilon$ und $S \to T_1 \mid \cdots \mid T_n$. Offensichtlich erzeugt diese Grammatik die Sprache. Sie ist nicht regulär, aber das lässt sich – siehe Aufgabe 4.5 – leicht beheben.

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 $\{\mathtt{b}^2\} \circ \{\mathtt{a},\mathtt{b}\}^*$ 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 ist[22] $\{\mathtt{a}\} \circ \{\mathtt{bca}\}^*$.

Lösung 5.4. Mit $\{s_0, s_1\}$ ergibt sich

$$\{\varepsilon\} \cup \{\mathtt{a}(\mathtt{bca})^m(\mathtt{bc})^n : m \in \mathbb{N} \wedge n \in \{0,1\}\},$$

mit $\{s_0\}$ erhält man $\{\mathtt{abc}\}^*$.

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.

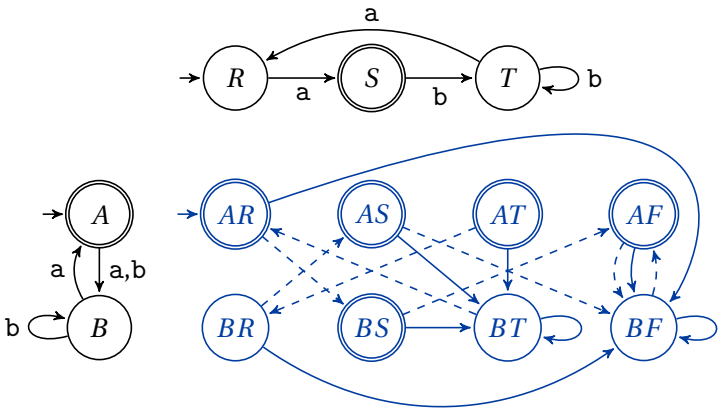
Lösung 5.8. Wenn man aus den entsprechenden Produktionen Übergänge macht, dann kann es passieren, dass von einem Zustand mehrere Pfeile mit demselben

[22] Natürlich gibt es immer mehr als eine Möglichkeit, so eine Menge aufzuschreiben. Ihre Antwort kann ruhig anders aussehen, sie muss nur dieselbe Menge beschreiben.

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 5.9. Man muss lediglich die Menge der akzeptierenden Zustände ändern, indem man F durch das Komplement $S \setminus F$ ersetzt.

Lösung 5.10. Hier wird nur eine Skizze eines Produktautomaten gezeigt. Man spricht natürlich deshalb von *Produkt*automaten, weil es um das aus der Mathevorlesung bekannte Mengen*produkt* 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 5.11. 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.)

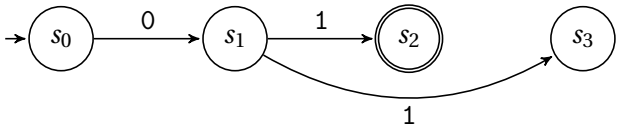
Auch für die Differenz kann man einfach mengentheoretisch argumentieren, nämlich mit $L_1 \setminus L_2 = L_1 \cap (I^* \setminus L_2)$.

Lösung 6.1. Das sieht so aus:

$$\delta^*(\{s\}, a) = \delta^*(\{s\}, a\varepsilon) = \bigcup_{s' \in \{s\}} \delta^*(\delta(s', a), \varepsilon) = \delta^*(\delta(s, a), \varepsilon) = \delta(s, a)$$

Lösung 6.2. Es handelt sich um $\{\mathtt{a}\}^+ \cup \left(\{\mathtt{ab}\} \circ \{\mathtt{a}\}^*\right)^+$.

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.

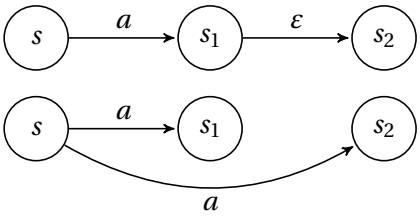
Lösung 6.5. Die wesentliche Idee ist: Ein Wort der Länge m muss vom Startzustand zu einem akzeptierenden Zustand m Pfeile und damit $m+1$ Zustände passieren. Ist m mindestens so groß wie $|S|$, dann wird mindestens ein Zustand s auf diesem Weg mehrfach besucht. Die „Schleife" von s nach s kann man sich 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 $s_1, s_2, \ldots, s_n$ mit $n \geq 2$ und $s_{i+1} \in \delta(s_i, \varepsilon)$ 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 $s_2 \in \delta(s_1, \varepsilon)$ und $s_1 \neq s_2$, so entferne s_2 aus $\delta(s_1, \varepsilon)$ 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.



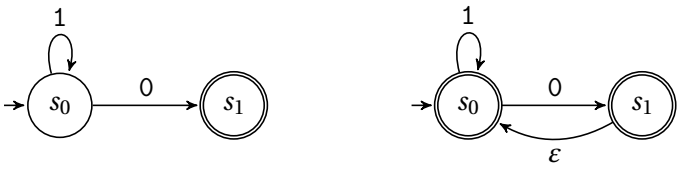
In jedem Schritt wird ein ε-Übergang entfernt und kein neuer hinzugefügt. Der Algorithmus muss also abbrechen und hinterlässt dann einen Automaten ohne ε-Übergänge. Es sollte klar sein, dass der neue Automat dieselbe Sprache wie der alte akzeptiert.

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 U 7.) 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. Fährt 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 6.10. 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 6.11. 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. Es handelt sich um $\{11\}$, $\{11,0\}$ und $\{11,0\}^*$.

Lösung 7.2. Die erste Aussage ist wahr. Die zweite Aussage ist falsch, denn die Sprache $\mathscr{L}(0(0)^*) = \{0\} \circ \{0\}^*$ 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 Konkatenation keine Rolle spielt, da alle Wörter nur aus Nullen bestehen.

Lösung 7.3. 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 anderen beiden Aussagen sind wahr.

Lösung 7.4. Der in PERL erlaubte reguläre Ausdruck `([01]*)\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.

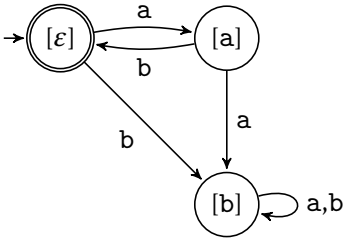
Lösung 8.1. Für die drei Wörter ε, a und ab gibt es jeweils genau eine Möglichkeit, aus ihnen durch Konkatenation ein L-Wort zu machen, und diese drei Möglichkeiten sind paarweise verschieden. Alle anderen Wörter sind nicht der Anfang eines L-Wortes. Es gibt somit vier verschiedene Klassen: $[\varepsilon]_L$, $[\mathtt{a}]_L$ und $[\mathtt{ab}]_L$ und $[\mathtt{b}]_L$. (Die letzte hätte man natürlich auch als $[\mathtt{aa}]_L$ oder $[\mathtt{b}^7]_L$ oder auf beliebig viele andere Arten schreiben können.)

Lösung 8.2. 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 $L \circ \{\mathtt{a}\}$ äquivalent zueinander, denn an sie kann man genau die Wörter aus $\{\mathtt{b}\} \circ L$ 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: $[\varepsilon]_L$, $[\mathtt{a}]_L$ und $[\mathtt{b}]_L$.

Lösung 8.3. Die Wörter $\mathtt{a}^m\mathtt{b}$ und $\mathtt{a}^n\mathtt{b}$ sind nicht äquivalent, wenn $m \neq n$ gilt, denn man kann das erste Wort mit $\mathtt{ba}^m$ zu einem Wort der Sprache ergänzen, das zweite jedoch nicht. Allein damit hat man schon unendlich viele Äquivalenzklassen.

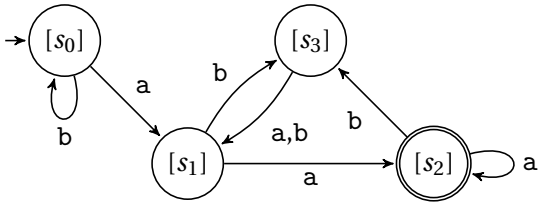
Lösung 8.4. Die Zustände sind $[\varepsilon]_L$, $[\mathtt{a}]_L$ und $[\mathtt{b}]_L$, wobei $[\varepsilon]_L$ sowohl der Startzustand als auch der einzige Endzustand ist. Die Übergangsfunktion sieht so aus:

	$[\varepsilon]$	[a]	[b]
a	$[\varepsilon\mathtt{a}] = [\mathtt{a}]$	$[\mathtt{aa}] = [\mathtt{b}]$	$[\mathtt{ba}] = [\mathtt{b}]$
b	$[\varepsilon\mathtt{b}] = [\mathtt{b}]$	$[\mathtt{ab}] = [\varepsilon]$	$[\mathtt{bb}] = [\mathtt{b}]$



Lösung 8.5. Die Wörter $\mathtt{a}^m$ und $\mathtt{a}^n$ sind nicht äquivalent, wenn $m \neq n$ gilt. Durch Konkatenation von $\mathtt{b}^m$ 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 8.7. 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 $\mathtt{b}^*\mathtt{a}(\mathtt{b}(\mathtt{a}+\mathtt{b}))^*\mathtt{a}(\mathtt{a}+\mathtt{b}(\mathtt{a}+\mathtt{b})(\mathtt{b}(\mathtt{a}+\mathtt{b}))^*\mathtt{a})^*$.

Lösung 8.8. Die Äquivalenzklassen der Nerode-Relation sind $[\varepsilon]_L$, $[\mathtt{b}]_L$, $[\mathtt{bb}]_L$, $[\mathtt{a}]_L$ und $[\mathtt{ab}]_L$. Daher muss der Minimalautomat fünf Zustände haben.

Lösung 8.9. 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 $\mathtt{0}^3$. 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 $\mathtt{0}^1\mathtt{0}^2$ und $\mathtt{0}^1\mathtt{0}^4$ Wörter aus L sind, $\mathtt{0}^1\mathtt{0}^0$ und $\mathtt{0}^1\mathtt{0}^3$ aber nicht. Da alle 15 Tabellenzeilen verschieden sind,[23] hat die Nerode-Relation für L mindestens 15 verschiedene Äquivalenzklassen. Da andererseits zwei Wörter aus $\{\mathtt{0}\}^*$ 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 8.10. Ist $n \in \mathbb{N}^+$, so wird ein Wort der Form $\mathtt{ab}^n$ durch Konkatenieren von $\mathtt{c}^n$ ein Wort von L, durch Konkatenieren von $\mathtt{c}^{n+1}$ 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 9.1. Ist $n \in \mathbb{N}^+$, so wird ein Wort der Form $\mathtt{[}^n$ durch Konkatenieren von $\mathtt{]}^n$ ein Wort der Dyck-Sprache, durch Konkatenieren von $\mathtt{]}^{n+1}$ jedoch nicht. Damit hat man schon unendlich viele Äquivalenzklassen der Nerode-Relation.

Lösung 9.2. Die Grammatik könnte aus den Produktionen $S \to XC$, $X \to \mathtt{a}X\mathtt{b} \mid \varepsilon$ und $C \to \mathtt{c}C \mid \varepsilon$ bestehen.

Lösung 9.3. 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 9.4. So ungefähr sollte es aussehen:

$$S \to \varepsilon \mid XC \mid X_1V_\mathtt{b} \mid V_\mathtt{a}V_\mathtt{b} \mid V_\mathtt{c}C \mid \mathtt{c} \quad C \to V_\mathtt{c}C \mid \mathtt{c} \quad X_1 \to V_\mathtt{a}X$$
$$X \to X_1V_\mathtt{b} \mid V_\mathtt{a}V_\mathtt{b} \quad V_\mathtt{a} \to \mathtt{a} \quad V_\mathtt{b} \to \mathtt{b} \quad V_\mathtt{c} \to \mathtt{c}$$

[23] Um das zu sehen hätten sogar drei statt sieben Wörter gereicht.

Lösung 9.6. 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.

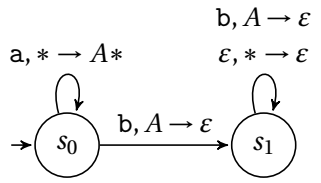
Lösung 9.7. Eine kontextfreie Grammatik für L_1^* erhält man durch

$$(N_1 \cup \{S\}, \Sigma, P_1 \cup \{(S, S_1 S), (S, \varepsilon)\}, S).$$

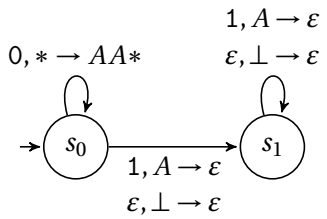
Lösung 9.8. Als Schnittmenge ergibt sich die Sprache, von der direkt nach dem Pumping-Lemma 9.2 gezeigt wurde, dass sie nicht kontextfrei ist. Damit ist gezeigt, dass der Durchschnitt zweier kontextfreier Mengen im Allgemeinen nicht kontextfrei ist.[24]

Lösung 9.9. Wäre $\Sigma^* \setminus L$ kontextfrei für alle kontextfreien Sprachen L, dann wäre nach Satz 9.3 $\Sigma^* \setminus (L_1 \cap L_2) = (\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2)$ und damit auch $L_1 \cap L_2$ kontextfrei, wenn L_1 und L_2 kontextfrei wären. Das widerspricht jedoch Aufgabe 9.8.

Lösung 10.1. 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 10.2. 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.



Lösung 10.3. Zunächst konstruieren wir zu $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ den Automaten $K' = (S', I, \Gamma', \delta', s_0', \bot', \emptyset)$ mit $L(K') = L^*(K)$. Wir fügen zwei neue Zustände s_0' und

[24] Die 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.

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 $\bot'$: Indem wir mit $\delta'(s'_0, \varepsilon, \bot') = \{(s_0, \bot\bot')\}$ anfangen, haben wir immer einen „Sicherheitspuffer“ ganz unten auf dem Stack. (Für alle Argumente, die bisher nicht erwähnten wurden, soll δ' sich wie δ verhalten. Und mit Γ' ist natürlich $\Gamma \cup \{\bot'\}$ mit $\bot' \notin \Gamma$ gemeint.)

Und jetzt konstruieren wir zu vorgegebenem $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ einen Kellerautomaten $K' = (S', I, \Gamma', \delta', s'_0, \bot', F')$ 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 $\Gamma' = \Gamma \cup \{\bot'\}$ mit $\bot' \notin \Gamma$ gelten. Für alle Argumente aus dem Definitionsbereich von δ soll sich δ' genau wie δ verhalten. (K' „simuliert“ gleichsam K.) Hinzu kommt wie oben $\delta'(s'_0, \varepsilon, \bot') = \{(s_0, \bot\bot')\}$, weil wir nun einen leeren Stack von K daran erkennen können, dass $\bot'$ das oberste Stacksymbol von K' ist. Die Menge F' der Endzustände soll $\{s'_e\}$ sein und dafür setzen wir

$$\delta'(s, \varepsilon, \bot') = \{(s'_e, \varepsilon)\}$$

für alle $s \in S$.

Lösung 10.5. Die Grundidee, die in der theoretischen Informatik öfter angewendet wird, ist, dass man Wörter einer begrenzten Länge durch Symbole ersetzt.[25] Das wird im Folgenden skizziert, aber nicht bis ins Detail ausgearbeitet.

Sei $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ ein beliebiger vorgegebener Kellerautomat. Da sowohl der Definitionsbereich der Übergangsfunktion δ als auch deren Funktionswerte endlich sind, gibt es eine maximale Anzahl n von Symbolen, um die der Stack in einem einzigen Übergang anwachsen kann. Wir definieren durch

$$\Gamma' = \Gamma^1 \cup \Gamma^2 \cup \cdots \cup \Gamma^n$$

ein neues Alphabet.[26] 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 $(\tau_1, \ldots, \tau_k)$ verwenden wir die Schreibweise $[\tau_1 \ldots \tau_k]$. Die Idee ist, dass so ein *Symbol* Stellvertreter für das *Wort* $\tau_1 \ldots \tau_k$ sein soll.

[25] Das 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.

[26] Erinnern Sie sich, dass wir ganz am Anfang Alphabete sehr allgemein definiert hatten.

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 10.6. Das sollte so aussehen:

$$\begin{aligned}(s_0,\mathtt{aab},\bot) &\Rightarrow_K (s_0,\mathtt{aab},S) \Rightarrow_K (s_0,\mathtt{aab},\mathtt{a}S\mathtt{b}) \Rightarrow_K (s_0,\mathtt{ab},S\mathtt{b})\\ &\Rightarrow_K (s_0,\mathtt{ab},\mathtt{a}S\mathtt{b}) \Rightarrow_K (s_0,\mathtt{b},S\mathtt{b}) \Rightarrow_K (s_0,\mathtt{b},\mathtt{b}) \Rightarrow_K (s_0,\varepsilon,\varepsilon)\end{aligned}$$

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 10.8. Die Menge der Nichtterminalsymbole würde aus dem neuen Startsymbol S_0 sowie zwölf[27] weiteren Symbolen wie z. B. $[s_0As_1]$ oder $[s_1\bot s_1]$ 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 $S_0\to[s_0\bot s_1]$ 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.

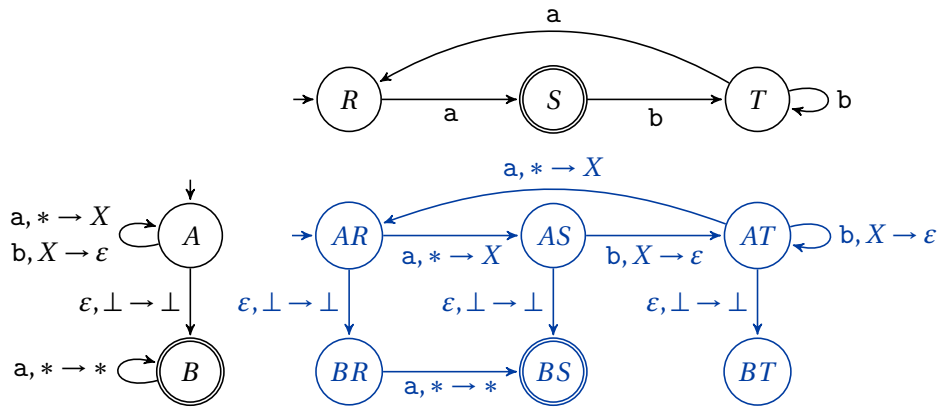
$[s_0\bot s_1]\to\mathtt{a}[s_0As_1][s_1\bot s_1]$	$\{(s_0,A\bot)\}\in\delta(s_0,\mathtt{a},\bot)$
$[s_0As_1]\to\mathtt{b}$	$\{(s_1,\varepsilon)\}\in\delta(s_0,\mathtt{b},A)$
$[s_1\bot s_1]\to\varepsilon$	$\{(s_1,\varepsilon)\}\in\delta(s_1,\varepsilon,\bot)$

Mit diesen insgesamt vier Produktionen kann man nun aus S_0 exemplarisch das Wort ab ableiten.

Lösung 10.9. Für L_2 gibt es einen Kellerautomaten K mit $L_2=L^*(K)$ (siehe Aufgabe 10.3) 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 5.11) 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 5.10) verdeutlicht hoffentlich das Prinzip.

[27] Die vier Paare (s_0,s_0), (s_0,s_1), (s_1,s_0) und (s_1,s_1) werden mit den drei Stacksymbole $\bot$, A und B auf alle möglichen Arten kombiniert.



Lösung 10.11. 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 $\delta(s,\varepsilon,\sigma)$ leer ist.

(ii) $\delta(s,a,\sigma)$ ist leer und $\delta(s,\varepsilon,\sigma)$ hat genau ein Element, wodurch auch wiederum festgelegt ist, wie es weitergeht.

(iii) $\delta(s,a,\sigma)$ und $\delta(s,\varepsilon,\sigma)$ 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 $K = (S, I, \Gamma, \delta, s_0, \bot, F)$ nach dem Konsumieren des letzten Zeichens in den Zustand s mit dem Wort $\sigma\alpha$ oben auf dem Stack über und ist $\delta(s,\varepsilon,\sigma) = (s',\varepsilon)$, dann gilt sowohl $(s_0, w, \bot) \Rightarrow_K^* (s, \varepsilon, \sigma\alpha)$ als auch $(s_0, w, \bot) \Rightarrow_K^* (s', \varepsilon, \alpha)$. 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 10.12. 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 anderes Wortes der Sprache ist.

präfixfrei

Lösung 10.13. Das lässt sich wohl am einfachsten mit einer Grammatik erledigen:

$$S \to 0S_1 1 \mid 0S_2 11$$
$$S_1 \to 0S_1 1 \mid \varepsilon \qquad\qquad S_2 \to 0S_2 11 \mid \varepsilon$$

Lösung 11.1. Der Ablauf sieht so aus:

$$\square[s_0]11\square \Rightarrow_{T_1} \square 1[s_0]1\square \Rightarrow_{T_1} \square 11[s_0]\square \Rightarrow_{T_1} \square 1[s_1]10$$

Lösung 11.2. Weil s_2 ein Endzustand ist, muss dort nach Punkt (iv) der Definition $\perp$ stehen. Man hätte die Spalte auch weglassen können.

Lösung 11.3. 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 11.4. 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	$(s_1, 1, \rightarrow)$	$(s_1, 1, \rightarrow)$	$(s_3, \square, \leftarrow)$	$(s_3, 1, \leftarrow)$	$(s_3, 0, \leftarrow)$
$\square$	$(s_5, \square, \rightarrow)$	$(s_2, \square, \leftarrow)$	$\perp$	$(s_0, \square, \rightarrow)$	$\perp$

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.

Wenn man es besonders „schön“ machen wollte, müsste man nun noch dafür sorgen, dass die Maschine am Ende an der „richtigen“ Stelle steht. Das bekommen Sie aber sicher selbst hin.

Lösung 11.6. 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	$(s_1, 0, \rightarrow)$	$(s_1, 1, \rightarrow)$	$(s_1, \square, \rightarrow)$
$\square$	$(s_e, 0, \leftarrow)$	$(s_e, 1, \leftarrow)$	$(s_e, \square, \leftarrow)$

Die Zustände s_0 bzw. s_1 stehen dafür, dass das letzte gelesene Zeichen eine Null bzw. Eins war. Beim Übergang in den Endzustand wird in diesem Fall nicht darauf geachtet, den Schreib-Lese-Kopf vorher zum Anfang des Bandes zurückzuführen, weil es laut Aufgabenstellung nicht um eine *Ausgabe* im definierten Sinne ging. (Und das wäre wegen des Leerzeichens auch gar nicht möglich.)

Lösung 11.7. Die Idee, dass so etwas prinzipiell möglich sein muss, sollte Menschen des 21. Jahrhunderts vertraut sein, weil die allgegenwärtigen Digitalcomputer jede Form von Information (Zahlen, Texte, Töne, Bilder, Filme und so weiter) speichern und verarbeiten können, obwohl sie intern nur mit Nullen und Einsen arbeiten.

Um konkret eine Turingmaschine mit Bandalphabet Σ zu simulieren, werden die Symbole dieses Alphabets durch Sequenzen von jeweils k Nullen und Einsen codiert, wobei k so gewählt wird, dass $2^k \geq |\Gamma|$ gilt. Vorgänge wie das Lesen oder Schreiben eines Symbols müssen dann kleinteilig in entsprechend viele Schritte zerlegt werden und die Menge der Zustände, die das alles koordinieren muss, wird ggf. stark anwachsen. Es wird aber keine prinzipiellen Schwierigkeiten bei der Implementation einer solchen Simulation geben.

Lösung 11.8. 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.

Die Definition wurde jedoch so allgemein gehalten, weil man in der Literatur auch Varianten ohne die Bedingung $\delta(s, \sigma) = \bot$ 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 11.10. Eine Spur simuliert den Teil des Bandes rechts von der Anfangsposition des Schreib-Lese-Kopfes und die andere Spur simuliert den Teil links davon.

Lösung 11.11. 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 $v[s]w$ der simulierten Turingmaschine entsprechen nun den Konfigurationen des Kellerautomaten, bei denen w auf

dem zweiten und v auf dem ersten Stack steht. Zustände und Übergangsfunktion können mehr oder weniger direkt denen der Turingmaschine nachempfunden werden.

Lösung 11.12. Ein Lösungsvorschlag befindet sich als Datei `decf2.TURING` im ZIP-Archiv.

Lösung 11.13. Ein Lösungsvorschlag befindet sich als Datei `invert.TURING` im ZIP-Archiv.

Lösung 11.14. Ein Lösungsvorschlag befindet sich als Datei `remove.TURING` im ZIP-Archiv. (Allerdings werden dort Nullen entfernt.)

Lösung 11.15. Ein Lösungsvorschlag befindet sich als Datei `add.TURING` im ZIP-Archiv.

Lösung 12.1. 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 1 716 471. Heutzutage verwendet man eher Unicode, aber das Prinzip ist dasselbe.

Lösung 12.2. $g(\texttt{abba}) = p_1^1 p_2^2 p_3^2 p_4^1 = 2^1 \cdot 3^2 \cdot 5^2 \cdot 7^1 = 3150$

Lösung 12.3. 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 12.4. Man bildet einfach das Tupel $t = (n_1, n_2, \ldots, n_k)$ auf

$$g(t) = p_1^{n_1+1} p_2^{n_2+1} \cdots p_k^{n_k+1}$$

ab. 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 12.5. In TOLL könnte es so aussehen:

```
program fibo(n: integer)

var i: integer,
    last: integer,
    nextToLast: integer,
    temp: integer;

begin
  if (n < 0) then halt(0-1) endif;
  if (n = 0) then halt(0) endif;
  nextToLast := 0;
  last := 1;
  for i from 2 to n do
    begin
      temp := last;
      last := last + nextToLast;
      nextToLast := temp
    end;
  halt(last)
end
```

In Sprachen, die parallele Mehrfachzuweisungen erlauben, kann man das noch etwas eleganter schreiben.

Lösung 12.6. Es prüft, ob die eingegebene ganze Zahl eine Quadratzahl ist, und gibt 1 oder 0 für *ja* bzw. *nein* zurück.

Lösung 12.7. Kurz und bündig zum Beispiel so:

```
program mod(a: integer, b: integer)

begin
  halt(a - b * (a / b))
end
```

Hier hat das Ergebnis dasselbe Vorzeichen wie der Dividend. Bekanntlich sind sich die Programmiersprachen in dieser Frage nicht einig.

Lösung 12.8. Meine Lösung sieht so aus:

```
program bitand(a: integer, b: integer)

var r: integer, f: integer;

begin
  r := 0; f := 1;
  while (0 < a) and (0 < b) do
  begin
    r := r + f*(a-2*(a/2))*(b-2*(b/2));
    a := a/2; b := b/2; f := 2*f
  end;
  halt(r);
end
```

Lösung 12.9. Das Zweierkomplement setzt eine feste Anzahl von Stellen voraus, weil das Vorzeichenbit immer das höchstwertige Bit ist.[28] Mit Langzahlarithmetik, bei der die Bitlänge beliebig ist, verträgt sich das nicht.

Lösung 12.10. 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 12.11. 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)
```

[28] Das hängt mit der modularen Arithmetik zusammen. Warum und wie das funktioniert, wird in den ersten vier Kapiteln von KMFI erklärt.

```
    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 12.12. Ich hatte an so etwas gedacht:

```
program bitand(a: integer, b: integer)

var r: integer, f: integer, t: integer, h: integer, k: integer;

begin
      r := 0; f := 1;
  10: t := 0;
      if (0 < a) then t := 1 endif;
      if (b = 0) then t := 0 endif;
      if t = 0 then goto 20 endif;
      h := a/2; h := 2*h; h := a-h; h := f*h;
      k := b/2; k := 2*k; k := b-k; k := h*k;
      r := r + k; a := a/2; b := b/2; f := 2*f;
      goto 10;
  20: halt(r);
end
```

Lösung 12.13. Die Datei `heron_nofloat.TURING` im ZIP-Archiv enthält einen Lösungsvorschlag.

Lösung 12.15. Nein, jedenfalls nicht im Sinne der am Anfang des Abschnitts umrissenen 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 13.1. Weil nun der Funktionswert $\emptyset$ möglich ist, der die Rolle des „Funktionswertes“ $\perp$ übernehmen kann.

Lösung 13.2. Die Bedingung ist nun nicht mehr, dass ein bestimmtes Tripel der entsprechende Funktionswert von δ *ist*, sondern dass es ein *Element* des Funktionswertes ist. Ansonsten sieht es so aus wie vorher:

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

Lösung 13.3. Die zweite Grammatik ist nicht kontextsensitiv, weil in der Produktion $\mathtt{ab}T \to \mathtt{cc}$ die linke Seite länger als die rechte ist. Die dritte ist nicht kontextsensitiv, weil es die Produktion $S \to \varepsilon$ gibt und S auf der rechten Seite von $S \to \mathtt{a}S$ auftritt. Die vierte ist wegen $T \to \varepsilon$ 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 13.4. Sie haben hoffentlich nicht einfach gedacht, dass jede kontextfreie Grammatik „offensichtlich“ kontextsensitiv ist. Das stimmt nämlich nicht. Selbst ganz einfache reguläre Grammatiken wie $S \to \mathtt{a}S \mid \varepsilon$ sind nicht kontextsensitiv. Wir können jedoch zu jeder kontextfreien Sprache L nach Lemma 9.1 eine kontextfreie Grammatik G in Chomsky-Normalform angeben, die L erzeugt. Und G ist dann tatsächlich kontextsensitiv.

Lösung 13.5. 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 13.6. Als Alphabet nehmen wir Σ_{bool}. Die beiden Grammatiken seien

$$G_1 \colon S_1 \to \mathtt{1} \quad \text{und} \quad G_2 \colon S_2 \to \mathtt{0} \quad \mathtt{1}S_2 \to \mathtt{01}.$$

Offenbar gilt $L(G_1) \circ L(G_2) = \{\mathtt{1}\} \circ \{\mathtt{0}\} = \{\mathtt{10}\}$. Die Produktion $\mathtt{1}S_2 \to \mathtt{01}$ kann in G_2 nicht verwendet werden. Weil die Mengen der Nichtterminalsymbole bereits disjunkt sind, würden wir nach der erwähnten Methode einfach die Produktion $S \to S_1 S_2$ hinzufügen und S zum neuen Startsymbol machen. Die so entstandene Grammatik G erlaubt jedoch die Ableitung $S \Rightarrow_G S_1 S_2 \Rightarrow_G \mathtt{1}S_2 \Rightarrow_G \mathtt{01}$.

Der Hinweis diente dazu, Beispiele wie das folgende auszuschließen: Man wähle für $i = 1, 2$ die Grammatik G_i, die nur die eine Produktion $S_i \to \varepsilon$ hat. Die besagte Methode liefert dann eine Grammatik, die *nicht* kontextsensitiv ist (warum?), aber darum ging es in der Frage nicht.

Lösung 13.7. Als Alphabet nehmen wir wieder Σ_{bool}. G_1 habe die beiden Produktionen $S_1 \to \mathtt{0}$ und $\mathtt{0}S_1 \to \mathtt{11}$. Offenbar gilt $L(G_1)^+ = \{\mathtt{0}\}^+$. Die Produktion $\mathtt{0}S_1 \to \mathtt{11}$ kann nicht verwendet werden. Die Grammatik G erlaubt jedoch die Ableitung $S \Rightarrow_G S_1 S \Rightarrow_G S_1 S_1 \Rightarrow_G \mathtt{0}S_1 \Rightarrow_G \mathtt{11}$.

Lösung 13.8. Das erste Beispiel könnte erweitert so aussehen:

$$G_1 \colon S_1 \to \mathtt{1} \quad \text{und} \quad G_2 \colon S_2 \to \mathtt{0} \mid \mathtt{11}S_2 \quad \mathtt{1}S_2 \to \mathtt{01}.$$

Nach wie vor gilt $\mathtt{01} \notin L(G_1) \circ L(G_2) = \{\mathtt{1}\} \circ \{\mathtt{0}, \mathtt{101}\} = \{\mathtt{10}, \mathtt{1101}\}$.

Für das zweite Beispiel könnte man für G_1

$$S_1 \to \mathtt{0} \mid \mathtt{00}S_1 \quad \text{und} \quad \mathtt{0}S_1 \to \mathtt{11}$$

nehmen. $L(G_1)^+$ enthielte dann weiterhin nicht das Wort `11`.

Lösung 13.9. Man führt für alle Terminalsymbole a ein neues Nichtterminalsymbol V_a sowie die Produktion $V_a \to a$ 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 13.6 würde z. B. $S_2 \to \mathtt{0}, V_1S_2 \to \mathtt{01}, V_1 \to \mathtt{1}$ werden.

Lösung 14.2. 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 14.3. 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 14.4. Das ist ähnlich einfach wie Aufgabe 14.3. 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 14.5. Da beim Decodieren der Zahl klar ist, wann man sich bei der dritten Komponente eines Tupels befindet, kann man statt `L`, `R` und `H` auch die Zeichen `0`, `1` und $*$ wiederverwenden. Damit hat man schon drei Zeichen gespart.

Lösung 15.1. Jede Turingmaschine berechnet eine Funktion aus $\mathscr{P}$, weil diese Menge so definiert ist. Im Fall $\mathscr{A} = \emptyset$ wäre die Menge A also die leere Menge und im anderen Fall ganz $\mathbb{N}$. Dass diese beiden Mengen entscheidbar sind, ist offensichtlich.

Lösung 15.2. In beiden Fällen war die Menge H^* aus (15.1) das Problem, das auf das jeweilige Problem aus der Formulierung des Satzes reduziert wurde. Wir haben die Implikation von Lemma 15.5 in den besagten Beweisen immer umgekehrt verwendet: Ist A nicht entscheidbar, dann ist B „erst recht“ nicht entscheidbar.

Lösung 16.1. Nein. Wenn das für alle Sprachen stimmen würde, dann wären nach Satz 14.2 alle rekursiv aufzählbaren Mengen natürlicher Zahlen entscheidbar. Aber wir kennen durch das Halteproblem und den Satz von Rice genügend Mengen, die nicht entscheidbar sind.

Lösung 17.2. 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

$$\lim_{x\to\infty} \frac{(\ln x)^2}{x} = \lim_{x\to\infty} \frac{2\ln x}{x} = \lim_{x\to\infty} \frac{2}{x} = 0,$$

d. h., für alle $\varepsilon > 0$ gibt es ein $n_0 \in \mathbb{N}$ mit $(\ln n)^2 < n\varepsilon$ und damit

$$n \ln 2 - (\ln n)^2 > n \ln 2 - n\varepsilon = n(\ln 2 - \varepsilon) \tag{L-1}$$

für alle $n \geq n_0$. Insbesondere ist damit der Term auf der linken Seite von (L-1) unbeschränkt und daher ist

$$\frac{2^n}{n^{\ln n}} = \frac{\mathrm{e}^{n \ln 2}}{\mathrm{e}^{(\ln n)^2}} = \mathrm{e}^{n \ln 2 - (\ln n)^2}$$

ebenfalls unbeschränkt und es folgt $2^n \notin \mathcal{O}(n^{\ln n})$. Offenbar gilt dieselbe Aussage auch für andere Basen als 2.

Lösung 17.3. Man könnte statt ln einen anderen Logarithmus nehmen oder z. B. $n^{\sqrt{n}}$ betrachten.

Lösung 17.5. 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.[29] Man kann sich jedoch überlegen, dass auch dieser Aufwand nichts am polynomialen Zeitverhalten ändert.

Lösung 17.6. 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 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.[30] Mit etwas Aufwand lässt sich dann zeigen, dass Satz 17.2 nach wie vor gilt.

[29] Da 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.

[30] Das ist ja auch realistisch, weil der Prozessor die Befehle aus dem Speicher holen muss.

Lösung 18.1. Bisher ging es nur um Entscheidungsprobleme im Sinne der Definition von Seite 93, also um Fragen, die man mit *ja* oder *nein* beantworten kann.

Lösung 18.2. 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 18.3. 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 18.4. 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 18.7. Nein. Im Prinzip macht das Programm das zwar, aber man muss sich an dieser Stelle daran erinnern, wie nichtdeterministische Turingmaschinen in Abschnitt 13 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 18.8. Das Programm muss für alle zweielementigen Teilmengen von V' testen, ob sie zu E gehören. Wenn wir davon ausgehen, dass ein entsprechender 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 18.9. 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

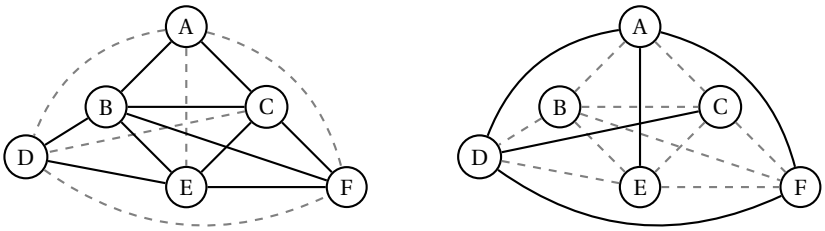
durchspielen muss. Wenn jede von denen ein Laufzeitverhalten von $\mathcal{O}(n^2)$ hat, ergibt sich insgesamt $\mathcal{O}(n^2 2^n)$.

Lösung 18.10. Ist $G = (V, E)$ ein Graph, dann bezeichnet man den Graphen

$$(V, \{\{v, w\} : v, w \in V \text{ und } v \neq w \text{ und } \{v, w\} \notin E\})$$

Komplementgraph

als den *Komplementgraphen* von G. Das ist also der Graph mit denselben Knoten, der genau dort Kanten hat, wo G keine hat.



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 18.11. Im ersten **if**-Block muss $\{v, w\} \notin E$ durch $\{v, w\} \in E$ ersetzt werden. Alles andere kann so bleiben.

Lösung 18.13. Das ist ganz einfach. Ein Zertifikat wäre einfach eine Clique der Größe k.

Lösung 19.2. Das sieht so aus:

$$\begin{aligned}
&(x_1 \vee x_2 \vee x_3 \vee x_4) \\
&\quad \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \\
&\quad \wedge (\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4)
\end{aligned}$$

$$\begin{aligned}
&(\neg s_{42,17} \vee \neg b_{42,17,23} \vee \neg z_{42,5} \vee s_{43,18}) \\
&\quad \wedge (\neg s_{42,17} \vee \neg b_{42,17,23} \vee \neg z_{42,5} \vee b_{43,17,12}) \\
&\quad \wedge (\neg s_{42,17} \vee \neg b_{42,17,23} \vee \neg z_{42,5} \vee z_{43,3})
\end{aligned}$$

Übungsaufgaben

Die Übungsaufgaben in diesem Abschnitt sollen selbstständig nach den Vorlesungen bearbeitet werden. In der darauf folgenden Veranstaltung werden dann Ihre Lösungen gemeinsam besprochen. Es wird jeweils angesagt, welche Aufgaben bis zum Folgetermin bearbeitet werden sollen.

Ich erinnere noch einmal daran, was man schon in der Gebrauchsanweisung lesen konnte: Aufgaben sind nicht als „Training“ für die Modulprüfung gedacht, sondern sie sollen Ihnen helfen, den Stoff zu verstehen.

Aufgabe U 1. Geben Sie zwei *verschiedene* Sprachen L_1 und L_2 über demselben Alphabet an, für die $L_1 \circ L_2 = L_2 \circ L_1$ gilt.

Aufgabe U 2. Geben Sie drei Sprachen L_1, L_2 und L_3 über dem Alphabet Σ_{bool} an, für die *nicht* $L_1 \circ (L_2 \cap L_3) = (L_1 \circ L_2) \cap (L_1 \circ L_3)$ gilt. [Hinweis: Denken Sie nicht an komplizierte unendliche Sprachen. Man kommt mit insgesamt vier Wörtern aus.]

Aufgabe U 3. Je zwei der folgenden acht Sprachen sind identisch. Finden Sie die entsprechenden Paare.

$L_1 = \{10\} \circ \{0\}^*$ $\quad$ $L_2 = \{ w \in \Sigma^* : |w|_0 = 3 \}$

$L_3 = \{01\}^*$ $\quad$ $L_4 = \{0,1\}^0 \cup \{0,1\}^1 \cup \{0,1\}^2$

$L_5 = \{ (01)^n : n \in \mathbb{N} \}$ $\quad$ $L_6 = \{ w \in \Sigma^* : |w| < 3 \}$

$L_7 = \{1\} \circ \{0\}^+$ $\quad$ $L_8 = \{ w \in \Sigma^* : |w|_1 = |w| - 1 \} \circ \{ w \in \Sigma^* : |w|_0 = 2 \}$

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

⋆**Aufgabe U 4**. Schreiben Sie für jede der folgenden Sprachen L ein Programm in einer Programmiersprache Ihrer Wahl, das entscheiden kann, ob ein Wort über dem Alphabet Σ zu L gehört.

$\Sigma = \Sigma_{\text{bool}}$ $\quad$ $L = \{ 0^n 1^{2n} 0^n : n \in \mathbb{N} \}$

$\Sigma = \Sigma_{\text{lat}}$ $\quad$ $L = \{ w \in \Sigma^+ : w$ ist ein Palindrom$\}$

$\Sigma = \{0,1,2\}$ $\quad$ $L = \{ w \in \Sigma^* : |w|_0 \cdot |w|_1$ ist ungerade$\}$

$\Sigma = \{0,1,2,\ldots,9,\mathtt{X}\}$ $\quad$ $L = \{ w \in \Sigma^* : w$ ist eine zehnstellige ISBN$\}$

Zum Thema ISBN siehe Kapitel 6 in KMFI.

Aufgabe U 5. In der folgenden Grammatik sind ⟨Subjekt⟩, ⟨Prädikat⟩ und so weiter Nichtterminalsymbole, ⟨Satz⟩ ist das Startsymbol und `Bär`, `Kekse`, `dicke` und so weiter sind Terminalsymbole.[31]

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

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?

Aufgabe U 6. Geben Sie die Sprache an, die von dieser Grammatik erzeugt wird:

$$
\begin{aligned}
S &\to \texttt{0}S \mid \texttt{1}S \mid \texttt{0}A \\
A &\to \texttt{1}B \\
B &\to \texttt{0}C \\
C &\to \varepsilon
\end{aligned}
$$

Aufgabe U 7. Geben Sie Grammatiken für die folgenden Sprachen an:

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

(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. Denken Sie daran, dass Sie das Verhalten solcher Grammatiken mit FLACI überprüfen können.

⋆**Aufgabe U 8**. Geben Sie eine Grammatik an, die die Sprache $\{\texttt{0}^n\texttt{1}^n\texttt{0}^n : n \in \mathbb{N}\}$ über Σ_{bool} erzeugt.

Aufgabe U 9. Begründen Sie mithilfe des Pumping-Lemmas 4.1, dass die Sprache

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

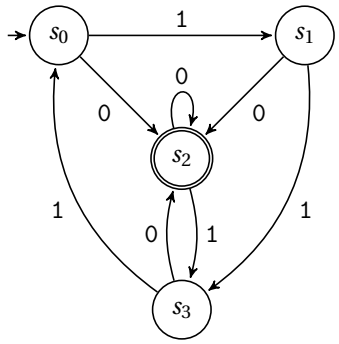
nicht regulär ist. Schreiben Sie eine vollständige Begründung in ganzen Sätzen auf, so dass jemand, der sich mit der Materie auskennt, sie nachvollziehen kann.

⋆**Aufgabe U 10**. Begründen Sie mithilfe des Pumping-Lemmas 4.1, dass die Sprache[32] $\{\texttt{1}^p : p \in \mathbb{P}\}$ über dem Alphabet $\{\texttt{1}\}$ nicht regulär ist.

[31] Anders als im Rest des Skripts bestehen Symbole hier also aus mehreren Buchstaben.

[32] Mit $\mathbb{P}$ ist die Menge der Primzahlen gemeint. Siehe Kapitel 8 in KMFI.

Aufgabe U 11. Geben Sie für den folgenden endlichen Automaten die von ihm akzeptierte Sprache in korrekter Schreibweise an.



Aufgabe U 12. 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 U 7

(iii) Sprache (ii) aus Aufgabe U 7

Aufgabe U 13. 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 $\texttt{f}^4$.

Aufgabe U 14. Sei A ein endlicher Automat mit dem Eingabealphabet I. Welche Möglichkeiten für $L(A)$ gibt es, wenn A nur einen einzigen Zustand hat?

Aufgabe U 15. 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 U 16. Der nichtdeterministische endliche Automat

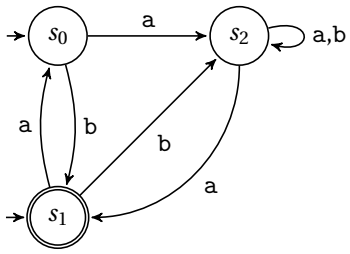
$$A = (\{s_0, s_1, s_2, s_3\}, \Sigma_{\text{bool}}, \delta, \{s_2\}, \{s_1\})$$

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 U 17. Konstruieren Sie wie im Beweis von Satz 6.1 zu dem folgenden nichtdeterministischen endlichen Automaten einen deterministischen, der dieselbe Sprache akzeptiert.



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

Aufgabe U 18. Einige der folgenden Aussagen über reguläre Ausdrücke sind falsch und einige richtig. Identifizieren Sie die falschen und geben Sie jeweils ein Wort an, das in der einen und nicht in der anderen Sprache enthalten ist.

$$(01)^* \equiv 0^*1^*$$
$$(10)^* \equiv (1^*0^*)^*$$
$$(0+1)^* \equiv 0^* + 1^*$$
$$1^* \equiv (1^*)^*$$

Aufgabe U 19. Beschreiben Sie die beiden folgenden Sprachen jeweils durch reguläre Ausdrücke:

$$L_1 = \{ w \in \Sigma_{\text{bool}}^* : |w| = 3 \}$$
$$L_2 = \{ w \in \Sigma_{\text{bool}}^* : |w|_1 \text{ ist gerade} \}$$

Aufgabe U 20. Geben Sie einen deterministischen endlichen Automaten an, der die durch den regulären Ausdruck $(\mathtt{a}^* + \mathtt{b})\mathtt{b}$ beschriebene Sprache akzeptiert. (Sie müssen dabei nicht das umständliche Verfahren aus dem Beweis von Satz 7.3 anwenden.)

Aufgabe U 21. 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.[33]

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

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

⋆**Aufgabe U 22**. Implementieren Sie den regulären Ausdruck aus Aufgabe U 21 in der in der Industrie gebräuchlichen PERL-Syntax und probieren Sie ihn aus. (Siehe dazu die Links in Aufgabe 7.4.)

Aufgabe U 23. Wie viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben sich für $L = \{\mathtt{0}(\mathtt{011})^{2n} : n \in \mathbb{N}\}$ und wie sehen sie aus?

Aufgabe U 24. Begründen Sie, dass sich für $L = \{\mathtt{0}^n\mathtt{1}^m\mathtt{0}^{n-1} : m, n \in \mathbb{N}^+\}$ unendlich viele Äquivalenzklassen bzgl. der Nerode-Relation ergeben.

Aufgabe U 25. Geben Sie für die Sprache aus Aufgabe U 23 einen regulären Ausdruck und die Anzahl der Zustände des Minimalautomaten an.

Aufgabe U 26. Konstruieren Sie nach dem in Abschnitt 8 vorgestellten Algorithmus den Minimalautomaten für den Automaten $(\{s_0, \ldots, s_5\}, \Sigma_{\text{bool}}, \delta, s_0, \{s_0, s_5\})$, 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. Idealiter arbeiten Sie nur mit der Tabelle, ohne Automaten zu zeichnen.

Aufgabe U 27. Erzeugen Sie für die Grammatik mit den Produktionen $S \to ASA \mid \mathtt{a}B$, $A \to B \mid S$, $B \to \mathtt{b} \mid \varepsilon$ eine Chomsky-Normalform mit dem Verfahren aus dem Beweis von Lemma 9.1.

⋆**Aufgabe U 28**. Begründen Sie mithilfe des Pumping-Lemmas 9.2, dass die Sprache aus Aufgabe U 10 auch nicht kontextfrei ist.

Aufgabe U 29. Konstruieren Sie einen Kellerautomaten für $\{w \in \Sigma_{\text{bool}}^* : |w|_0 = |w|_1\}$.

Aufgabe U 30. 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,[34] bei Eingaben wie `1100` hingegen `0` (für „nein“).

Aufgabe U 31. Was macht die Turingmaschine

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

deren Übergangsfunktion durch die folgende Tabelle definiert ist?

	q_0	q_1	q_2	q_4
0	$(q_0, 0, \to)$	$(q_1, 0, \to)$	$(q_2, 0, \leftarrow)$	$(q_4, 1, \leftarrow)$
1	$(q_1, 1, \to)$	$(q_1, 1, \to)$	$(q_2, 1, \leftarrow)$	$(q_2, 0, \leftarrow)$
$\square$	$(q_2, \square, \leftarrow)$	$(q_4, \square, \leftarrow)$	$(q_3, \square, \to)$	$\bot$

[34] Es sollen also Palindrome ungerader und auch gerader Länge erkannt werden.

Aufgabe U 32. 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.[35] 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 den dahingehend verbessern kann, dass möglichst wenig Speicher „verschwendet“ wird.

Aufgabe U 33. Schreiben Sie ein TM2012-Programm, das Übung U 30 löst. (Zur Kontrolle finden Sie einen Lösungsvorschlag als Datei `palindrome.TURING` im ZIP-Archiv.)

⋆**Aufgabe U 34**. Die Lösung von Aufgabe 11.15 braucht m Schleifendurchläufe zur Berechnung von $m+n$. Schreiben Sie ein TM2012-Programm, das dieselbe Aufgabe effizienter löst, indem wie bei der schriftlichen Addition gerechnet wird. (Zur Kontrolle finden Sie einen Lösungsvorschlag als Datei `add2.TURING` im ZIP-Archiv.)

Aufgabe U 35. Begründen Sie, warum die Zahl 42 nicht als Funktionswert der Funktion g aus Aufgabe 12.4 vorkommen kann.

⋆**Aufgabe U 36**. Im ZIP-Archiv finden Sie ein NSD-Programm zur Berechnung des n-ten Gliedes der Fibonacci-Folge. Machen Sie daraus ein FRACTRAN-Programm. Noch besser: Schreiben Sie ein FRACTRAN-Programm, das analog zu (12.1) *alle* Werte der Fibonacci-Folge nacheinander ausgeben kann. (Sie können Ihre Programme mit dem PYTHON-Code aus der Vorlesung testen.)

Aufgabe U 37. Begründen Sie, warum es zu jeder entscheidbaren Menge von natürlichen Zahlen unendlich viele Programme für die TM2012 gibt, die sie entscheiden.

Aufgabe U 38. 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.

Aufgabe U 39. Begründen Sie, warum sowohl der Durchschnitt als auch die Vereinigung von zwei rekursiv aufzählbaren Mengen (bzw. Sprachen) wieder rekursiv aufzählbar sind.

[35] Der Speicher ist also ebenfalls beliebig groß.

Aufgabe U 40. Geben Sie das Cliquen-Problem als formale Sprache an, so dass es ein Entscheidungsproblem im Sinne der Definition von Seite 93 ist. (Mit dem Alphabet muss man also Paare der Form (G, k) darstellen können, wobei G ein Graph und k eine natürliche Zahl ist.)

Aufgabe U 41. Begründen Sie anhand der formalen Sprache aus Übung U 40 die folgende Aussage: Gibt es eine Turingmaschine T, die das Cliquen-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 Cliquen-Problems zu **P** bzw. **NP** hängt per definitionem von der Codierung ab (siehe Seite 98), aber bei einer sinnvollen Codierung ist die Anzahl der Ecken die relevante Größe.

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 Priese, 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

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.