UNIVERSITÄT
LEIPZIG

Institut für Informatik, Universität Leipzig

Skript zu den Vorlesungen

Automaten und Sprachen (Teile I+II)
Berechenbarkeit (Teile III+IV)

Prof. Dr. Carsten Lutz

Stand: 22. Januar 2024

Inhaltsverzeichnis

Hinweis

Dieses Skript ist als Hilfestellung für Studierende gedacht. Trotz großer Sorgfalt beim Erstellen kann keine Garantie für Fehlerfreiheit übernommen werden. Es wird ausdrücklich darauf hingewiesen, dass der prüfungsrelevante Stoff durch die Vorlesung bestimmt wird und mit dem Skriptinhalt nicht vollständig übereinstimmen muss.

Dieses Skript ist ursprünglich aus einem Vorlesungsskript von Franz Baader entstanden. Es enthält Beiträge von Thomas Schneider.

Einführung

Die theoretische Informatik beschäftigt sich mit zentralen Fragestellungen der Informatik wie etwa den prinzipiellen Grenzen der Berechenbarkeit. Zentrale Methoden sind die Abstraktion und die Modellbildung, d. h. es werden die zentralen Konzepte und Methoden der Informatik identifiziert und in abstrakter Form beschrieben und studiert. Daraus ergibt sich eine Sammlung mathematischer Theorien, die die Grundlage für zahlreiche andere Teilgebiete der Informatik bildet.

Die theoretische Informatik ist in zahlreiche Teilgebiete untergliedert, wie etwa die Komplexitätstheorie, die Algorithmentheorie, die Kryptographie und die Datenbanktheorie. Die Lehrveranstaltungen *Theoretische Informatik 1 & 2* geben eine Einführung in folgende zwei zentrale Bereiche der theoretischen Informatik:

Automatentheorie und formale Sprachen

Behandelt in *Automaten und Sprachen* – Teile I und II dieses Skriptes

Im Mittelpunkt stehen *Wörter* und *formale Sprachen* (Mengen von Wörtern). Diese sind ein nützliches Abstraktionsmittel in der Informatik. Man kann z. B. die Eingabe oder Ausgabe eines Programms als Wort betrachten und die Menge der syntaktisch korrekten Eingaben als Sprache. Wichtige Fragestellungen sind z. B.:

- Was sind geeignete Beschreibungsmittel für (meist unendliche) formale Sprachen? (z. B. Automaten und Grammatiken)
- Welche Typen von Sprachen lassen sich unterscheiden?
- Welche Eigenschaften haben die verschiedenen Sprachtypen?

Berechenbarkeit und Komplexität

Behandelt in *Berechenbarkeit* – Teile III und IV dieses Skriptes

Hier geht es darum, welche Probleme und Funktionen prinzipiell berechenbar sind und welche nicht. Außerdem wird untersucht, welcher zeitliche Aufwand zur Berechnung eines Problems/einer Funktion notwendig ist (unabhängig vom konkreten Algorithmus). Wichtige Fragestellungen sind z. B.:

- Welche Berechnungsmodelle gibt es und wie verhalten sich diese zueinander?
- Gibt es Funktionen oder Mengen, die prinzipiell nicht berechenbar sind?
- Kann man jede berechenbare Funktion mit akzeptablem Zeit- und Speicherplatzaufwand berechnen?
- Für in der Informatik häufig auftretende Probleme/Funktionen: wie viel Zeit und Speicherplatz braucht man mindestens, also bei optimalem Algorithmus?

Teil I + II: Automatentheorie und formale Sprachen

Formale Sprachen, also (endliche oder unendliche) Mengen von Wörtern, sind ein wichtiger Abstraktionsmechanismus der Informatik. Hier ein paar Anwendungsbeispiele:

- Die Menge aller wohlgeformten Programme in einer gegebenen Programmiersprache wie Java, C++ oder Haskell ist eine formale Sprache.
- Die Menge aller wohlgeformten Eingaben für ein Programm oder für ein Formular auf einer Webseite (z. B. Menge aller Kontonummern / Menge aller Geburtsdaten) ist eine formale Sprache.
- Jeder Suchausdruck (z. B. Linux Regular Expression) definiert eine formale Sprache: die Menge der Dokumente, in der der Ausdruck zu finden ist.
- Kommunikationsprotokolle: z. B. die Menge aller wohlgeformten TCP-Pakete ist eine formale Sprache.
- Das „erlaubte“ Verhalten von Soft- und Hardwaresystemen kann in sehr natürlicher Weise als formale Sprache modelliert werden.

Wir beginnen mit einem kurzen Überblick über die zentralen Betrachtungsgegenstände und Fragestellungen.

1. **Charakterisierung**:

   Nützliche und interessante formale Sprachen sind in der Regel unendlich, wie in allen obigen Beispielen (es gibt zum Beispiel unendlich viele wohlgeformte Java-Programme). Wie beschreibt man derartige Sprachen mit endlichem Aufwand?

   - *Automaten* oder *Maschinen*, die genau die Wörter der Sprache akzeptieren. Wir werden mehrere Automatenmodelle kennen lernen, wie z. B. endliche Automaten, Kellerautomaten und Turingmaschinen.
   - *Grammatiken*, die genau die Wörter der Sprache generieren; auch hier gibt es viele verschiedene Typen, z. B. rechtslineare Grammatiken und kontextfreie Grammatiken (vgl. auch VL „Praktische Informatik“: kontextfreie Grammatiken (EBNF) zur Beschreibung der Syntax von Programmiersprachen).
   - *Ausdrücke*, die beschreiben, wie man die Sprache aus Basissprachen mit Hilfe gewisser Operationen (z. B. Vereinigung) erzeugen kann.

Abhängig von dem verwendeten Automaten-/Grammatiktyp erhält man verschiedene Klassen von Sprachen. Wir werden hier die vier wichtigsten Klassen betrachten; diese sind in der **Chomsky-Hierarchie** zusammengefasst:

Klasse	Automatentyp	Grammatiktyp
Typ 0	Turingmaschine (TM)	allgemeine Chomsky-Grammatik
Typ 1	TM mit linearer Bandbeschränkung	monotone Grammatik
Typ 2	Kellerautomat	kontextfreie Grammatik
Typ 3	endlicher Automat	einseitig lineare Grammatik

Für Typ 3 existiert auch eine Beschreibung durch reguläre Ausdrücke. Am wichtigsten sind die Typen 2 und 3; beispielsweise kann Typ 2 weitgehend die Syntax von Programmiersprachen beschreiben.

2. Welche Fragen sind für eine Sprachklasse entscheidbar und mit welchem Aufwand? Die folgenden Probleme werden eine zentrale Rolle spielen:

   - *Wortproblem*: gegeben eine Beschreibung der Sprache L (z. B. durch einen Automaten, eine Grammatik, einen Ausdruck, ...) und ein Wort w. Gehört w zu L?

     Anwendungsbeispiele:

     – Programmiersprache, deren Syntax durch eine kontextfreie Grammatik beschrieben ist. Entscheide für ein gegebenes Programm P, ob dieses syntaktisch korrekt ist.

     – Suchpattern für Textdateien sind häufig reguläre Ausdrücke. Suche die Dateien (Wörter), die das Suchpattern enthalten (zu der von ihm beschriebenen Sprache gehören).

   - *Leerheitsproblem*: gegeben eine Beschreibung der Sprache L. Ist L leer?

     Anwendungsbeispiel:
     Wenn ein Suchpattern die leere Sprache beschreibt, so muss man die Dateien nicht durchsuchen, sondern kann ohne weiteren Aufwand melden, dass das Pattern nicht sinnvoll ist.

   - *Äquivalenzproblem*: Beschreiben zwei verschiedene Beschreibungen dieselbe Sprache?

     Anwendungsbeispiel:
     Jemand vereinfacht die Grammatik einer Programmiersprache, um sie übersichtlicher zu gestalten. Beschreibt die vereinfachte Grammatik wirklich dieselbe Sprache wie die ursprüngliche?

3. Welche **Abschlusseigenschaften** hat eine Sprachklasse?

   z. B. Abschluss unter Schnitt, Vereinigung und Komplement: wenn L_1, L_2 in der Sprachklasse enthalten, sind es dann auch $L_1 \cap L_2$, $L_1 \cup L_2$, $\overline{L_1}$?

   Anwendungsbeispiele:

   - Suchpattern: Suche nach Dateien, die das Pattern *nicht* enthalten (Komplement) oder die zwei Pattern enthalten (Schnitt).
   - Reduziere das Äquivalenzproblem auf das Leerheitsproblem, ohne die gewählte Klasse von Sprachen zu verlassen: Statt „$L_1 = L_2$?“ entscheidet man, ob $(L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1})$ leer ist.

Abgesehen von ihrer direkten Nützlichkeit für verschiedene Informatik-Anwendungen stellen sich alle diese Fragestellungen als mathematisch sehr interessant heraus. Zusammengenommen bilden sie eine wichtige formale Grundlage der Informatik.

I. Endliche Automaten und reguläre Sprachen

0. Grundbegriffe

Die grundlegenden Begriffe der Vorlesung „Automaten und Sprachen“ sind *Wörter* und *formale Sprachen*.

0.1. Wörter und formale Sprachen

Alphabet. Ein *Alphabet* ist eine endliche Menge von Symbolen. Beispiele sind:

- $\Sigma_1 = \{a, b, c, \ldots, z\}$
- $\Sigma_2 = \{0, 1\}$
- $\Sigma_3 = \{0, \ldots, 9\} \cup \{,\}$
- Σ_4 = {`program`, `const`, `var`, `label`, `procedure`, `function`, `type`, `begin`, `end`, `if`, `then`, `else`, `case`, `of`, `repeat`, `until`, `while`, `do`, `for`, `to`} ∪ {VAR, VALUE, FUNCTION}

Als Symbol (Platzhalter) für Alphabetssymbole benutzen wir in der Regel $a, b, c, \ldots$. Alphabete bezeichnen wir meist mit Σ.

Obwohl die Symbole von Σ_4 aus mehreren Buchstaben der üblichen Schriftsprache bestehen, betrachten wir sie doch als unteilbare Symbole. Die Elemente von Σ_4 sind genau die Schlüsselwörter der Programmiersprache Pascal. Konkrete Variablennamen, Werte und Funktionsaufrufe sind zu den Schlüsselwörtern VAR, VALUE, FUNCTION abstrahiert, um Endlichkeit des Alphabets zu gewährleisten.

Wort. Ein *Wort* ist eine endliche Folge von Symbolen. Ein Wort $w = a_1 \cdots a_n$ mit $a_i \in \Sigma$ heißt Wort *über dem Alphabet* Σ. Beispiele sind:

- $w = abc$ ist ein Wort über Σ_1.
- $w = 1000110$ ist ein Wort über Σ_2.
- $w = 10{,}0221{,}4292_{„}$ ist ein Wort über Σ_3.

- Jedes Pascal-Programm kann als Wort über Σ_4 betrachtet werden, wenn man jede konkrete Variable durch das Schlüsselwort VAR ersetzt, jeden Wert durch VALUE und jeden Funktionsaufruf durch FUNCTION.

Als Symbol für Wörter verwenden wir meist w, v, u. Die Länge eines Wortes w wird mit $|w|$ bezeichnet, es gilt also z. B. $|aba| = 3$. Manchmal ist es praktisch, auch die Anzahl der Vorkommen eines Symbols a in einem Wort w in kurzer Weise beschreiben zu können. Wir verwenden hierfür $|w|_a$, es gilt also z. B. $|aba|_a = 2$, $|aba|_b = 1$, $|aba|_c = 0$. Einen Spezialfall stellt das *leere Wort* dar, also die leere Folge von Symbolen. Dieses wird durch ε bezeichnet. Es ist das einzige Wort mit $|w| = 0$.

Formale Sprache. Eine *(formale) Sprache* ist eine Menge von Wörtern. Mit Σ^* bezeichnen wir die Sprache, die aus *allen* Wörtern über dem Alphabet Σ besteht, also z. B.:

$$\{a, b\}^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

Eine Sprache $L \subseteq \Sigma^*$ heißt *Sprache über dem Alphabet* Σ. Beispiele sind:

- $L = \emptyset$
- $L = \{abc\}$
- $L = \{a, b, c, ab, ac, bc\}$
- $L = \{w \in \{a, \dots, z\}^* \mid w \text{ ist ein Wort der deutschen Sprache}\}$
- L als Menge aller Wörter über Σ_4, die wohlgeformte Pascal-Programme beschreiben

Als Symbol für Sprachen verwenden wir meist L. Man beachte, dass Sprachen sowohl endlich als auch unendlich sein können. Interessant sind meist nur unendliche Sprachen. Als nützliche Abkürzung führen wir Σ^+ für die Menge $\Sigma^* \setminus \{\varepsilon\}$ aller nicht-leeren Wörter über Σ ein. Sowohl Σ^* als auch Σ^+ sind offensichtlich unendliche Sprachen.

0.2. Operationen auf Sprachen und Wörtern

Im Folgenden werden wir viel mit Wörtern und formalen Sprachen umgehen. Dazu verwenden wir in erster Linie die folgenden Operationen.

Präfix, Suffix, Infix: Zu den natürlichsten und einfachsten Operationen auf Wörtern gehört das Bilden von Präfixen, Suffixen und Infixen:

$$\begin{array}{lll} u \text{ ist Präfix von } v & \text{wenn} & v = uw \text{ mit } u, w \in \Sigma^*. \\ u \text{ ist Suffix von } v & \text{wenn} & v = wu \text{ mit } u, w \in \Sigma^*. \\ u \text{ ist Infix von } v & \text{wenn} & v = w_1 u w_2 \text{ mit } u, w_1, w_2 \in \Sigma^*. \end{array}$$

Präfixe und Suffixe eines Wortes w sind also beliebig lange Anfangs- bzw. Endstücke von w; Infixe sind beliebige Teilwörter, die auch in der Mitte auftreten können. Jedes Präfix und Suffix von w ist damit auch Infix. Sowohl das leere Wort als auch w selbst sind sowohl Präfix, Infix, als auch Suffix von w. Das Wort $aabbcc$ beispielsweise hat die Präfixe $\varepsilon, a, aa, aab, aabb, aabbc, aabbcc$ sowie 19 Infixe.

Konkatenation: Eine Operation, die auf Wörter sowie auf Sprachen angewendet werden kann. Auf Wörtern u und v bezeichnet die Konkatenation $u \cdot v$ das Wort uv, das man durch einfaches „Hintereinanderschreiben“ erhält. Es gilt also z. B. $abb \cdot ab = abbab$. Auf Sprachen bezeichnet die Konkatenation das Hintereinanderschreiben *beliebiger* Wörter aus den beteiligten Sprachen:

$$L_1 \cdot L_2 := \{u \cdot v \mid u \in L_1 \text{ und } v \in L_2\}$$

Es gilt also z. B.

$$\{aa, a\} \cdot \{ab, b, aba\} = \{aaab, aab, aaaba, ab, aaba\}.$$

Sowohl auf Sprachen als auch auf Wörtern wird der Konkatenationspunkt häufig weggelassen, wir schreiben also z. B. L_1L_2 statt $L_1 \cdot L_2$.

Man beachte, dass $\emptyset \cdot L = L \cdot \emptyset = \emptyset$. Die Konkatenation ist assoziativ; es gilt also $(L_1 \cdot L_2) \cdot L_3 = L_1 \cdot (L_2 \cdot L_3)$. Sie ist nicht kommutativ; im Allgemeinen gilt also nicht $L_1 \cdot L_2 = L_2 \cdot L_1$. (Finde ein Gegenbeispiel!)

Um wiederholte Konkatenation desselben Wortes zu beschreiben, verwenden wir folgende Notation: für ein Wort $w \in \Sigma^*$ und ein $n \geq 0$ bezeichnet w^n das Wort, das wir durch n-malige Konkatenation von w erhalten, also zum Beispiel $(abc)^3 = abcabcabc$ (aber $abc^3 = abccc$). Wir definieren $w^0 = \varepsilon$ für jedes Wort w.

Boolesche Operationen: Es handelt sich um die üblichen Booleschen Mengenoperationen, angewendet auf formale Sprachen:

Vereinigung	$L_1 \cup L_2$	:=	$\{w \mid w \in L_1 \text{ oder } w \in L_2\}$
Schnitt	$L_1 \cap L_2$	:=	$\{w \mid w \in L_1 \text{ und } w \in L_2\}$
Komplement	$\overline{L_1}$	:=	$\{w \mid w \in \Sigma^* \text{ und } w \notin L_1\}$

Vereinigung und Schnitt sind sowohl assoziativ als auch kommutativ.

Kleene-Stern: Der Kleene-Stern bezeichnet die beliebig (aber nur endlich) oft iterierte Konkatenation. Gegeben eine Sprache L, definiert man zunächst induktiv Sprachen $L^0, L^1, \ldots$ und darauf basierend dann die durch Anwendung des Kleene-Sterns erhaltene Sprache L^*:

$$\begin{aligned} L^0 &:= \{\varepsilon\} \\ L^{n+1} &:= L^n \cdot L \\ L^* &:= \textstyle\bigcup_{n \geq 0} L^n \end{aligned}$$

Für $L = \{a, ab\}$ gilt also z. B. $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^2 = \{aa, aab, aba, abab\}$, etc. Offensichtlich ist L^* unendlich gdw. (genau dann, wenn) $L \neq \emptyset$.

Man beachte, dass das leere Wort per Definition *immer* in L^* enthalten ist, unabhängig davon, was L für eine Sprache ist. Manchmal verwenden wir auch die Variante ohne L_0:

$$L^+ := \bigcup_{n \geq 1} L^n$$

Es ist leicht zu sehen, dass $\varepsilon \in L^+$ gdw. $\varepsilon \in L$. Einige einfache Beobachtungen sind $\emptyset^* = \{\varepsilon\}$, $(L^*)^* = L^*$ und $L^* \cdot L^* = L^*$. Es ist hier wichtig, $\emptyset$ (die leere Sprache), $\{\varepsilon\}$ (die Sprache, die das leere Wort enthält) und ε (das leere Wort) sorgsam auseinander zu halten.

Etwas informeller könnte man den Kleene-Stern also auch wie folgt definieren:

$$L^* = \{\varepsilon\} \cup \{w \mid \exists u_1, \ldots, u_n \in L : w = u_1 \cdot u_2 \cdot \cdots \cdot u_n\}.$$

$<$

1. Endliche Automaten

Endliche Automaten stellen ein einfaches und dennoch sehr nützliches Mittel zum Beschreiben von formalen Sprachen dar. Sie können als Abstraktion eines (Hardware- oder Software-)Systems aufgefasst werden. Die charakteristischen Merkmale eines endlichen Automaten sind

- eine endliche Menge von Zuständen, in denen sich der Automat befinden kann

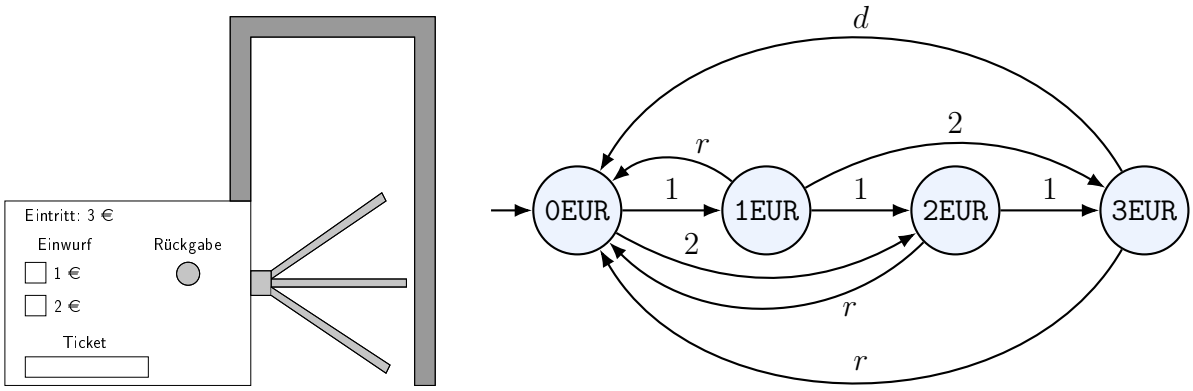
  Ein Zustand beschreibt die aktuelle Konfiguration des Systems. In unserem Kontext ist ein Zustand lediglich ein Symbol (bzw. ein Name) wie q_0, q_1, etc. Insbesondere wird nicht näher beschrieben, was genau diesen Zustand ausmacht (etwa eine bestimmte Belegung eines Registers mit einem konkreten Wert).

- feste Übergangsregeln zwischen Zuständen in Abhängigkeit von der Eingabe

  Zustandswechsel werden dabei als augenblicklich angenommen, d. h. ein eventueller Zeitverbrauch wird nicht modelliert. Ein Lauf eines Systems ist also einfach eine Folge von Zuständen.

Beispiel (Eintrittsautomat).

Eingabe: $1, 2, r, d$ *(r: Geldrückgabe, d: Drehsperre dreht sich)*
Zustände: `0EUR`, `1EUR`, `2EUR`, `3EUR`



Der dargestellte Automat regelt eine Drehsperre. Es können Münzen im Wert von 1 € oder 2 € eingeworfen werden. Nach Einwurf von 3 € wird die Arretierung der Drehsperre gelöst und der Eintritt freigegeben. Der Automat gibt kein Wechselgeld zurück, sondern nimmt einen zu hohen Betrag nicht an (Münzen fallen durch). Man kann jederzeit den Rückgabeknopf drücken, um den bereits gezahlten Betrag zurückzuerhalten.

In der schematischen Darstellung kennzeichnen die Kreise die internen Zustände und die Pfeile die Übergänge. Die Pfeilbeschriftung gibt die jeweilige Eingabe an, unter der der Übergang erfolgt. Man beachte, dass

- nur der Zustand `3EUR` einen Übergang vom Typ d erlaubt. Dadurch wird modelliert, dass nur durch Einwurf von 3 € der Eintritt ermöglicht wird.

- das Drehen der Sperre als Eingabe angesehen wird. Man könnte dies auch als Ausgabe modellieren. Wir werden in dieser Vorlesung jedoch keine endlichen Automaten mit Ausgabe (sogenannte Transduktoren) betrachten.

Die Übergänge können als festes Programm betrachtet werden, das der Automat ausführt.

Man beachte den engen Zusammenhang zu formalen Sprachen: die Menge der möglichen Eingaben $\{1, 2, r, d\}$ bildet ein Alphabet. Jede (Gesamt-)Eingabe des Automaten ist ein Wort über diesem Alphabet. Wenn man `3EUR` als Zielzustand betrachtet, so bildet die Menge der Eingaben, mittels derer dieser Zustand erreicht werden kann, eine (unendliche) formale Sprache. Diese enthält zum Beispiel das Wort $11r21$.

Wir definieren endliche Automaten nun präzise und unterscheiden dabei zwischen der *deterministischen* und der *nichtdeterministischen* Variante. Diese Unterscheidung ist ein fundamentales Konzept der theoretischen Informatik und wird uns in den Teilen II–IV bei anderen Berechnungsmodellen wieder begegnen.

1.1. Deterministische endliche Automaten

Wir beginnen mit der intuitiven deterministischen Variante, bei der es in jedem Schritt einen eindeutig bestimmten Folgezustand gibt.

Definition 1.1 (DEA)

Ein *deterministischer endlicher Automat (DEA)* ist ein Tupel $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, wobei

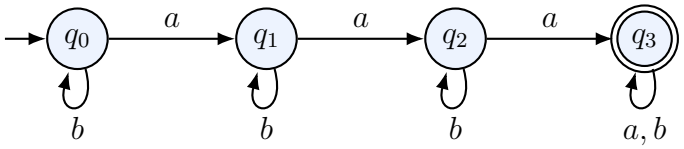
- Q eine endliche Menge von *Zuständen* ist,
- Σ ein *Eingabealphabet* ist,
- $q_0 \in Q$ der *Anfangszustand* ist,
- $\delta : Q \times \Sigma \to Q$ die *Übergangsfunktion* ist,
- $F \subseteq Q$ eine Menge von *akzeptierenden Zuständen* ist.

Beispiel 1.2

Der DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ mit den Komponenten

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b\}$
- $\delta(q_0, a) = q_1 \qquad \delta(q_1, a) = q_2 \qquad \delta(q_2, a) = \delta(q_3, a) = q_3$
  $\delta(q_i, b) = q_i \quad$ für $i \in \{0, 1, 2, 3\}$
- $F = \{q_3\}$

wird graphisch dargestellt als:



Wie im obigen Beispiel werden wir Automaten häufig als kantenbeschriftete Graphen darstellen, wobei die Zustände des Automaten die Knoten des Graphen sind und die Übergänge als Kanten gesehen werden (beschriftet mit einem Alphabetsymbol). Der Startzustand wird durch einen eingehenden Pfeil gekennzeichnet und die akzeptierenden Zustände durch einen Doppelkreis. (Manchmal werden in der Literatur die akzeptierenden Zustände auch durch einen ausgehenden Pfeil gekennzeichnet.)

Intuitiv arbeitet ein DEA, indem er im Startzustand beginnt und ein Wort Symbol für Symbol von links nach rechts liest und dabei entsprechend der Übergangsfunktion den Zustand wechselt. Zu diesem Zweck bestimmt die Übergangsfunktion für jeden Zustand und jedes mögliche gelesene Zeichen einen eindeutig bestimmten Folgezustand. Der DEA akzeptiert das Eingabewort, wenn er sich, *nachdem er das Wort vollständig gelesen hat,* in einem akzeptierenden Zustand befindet. Um dieses Verhalten präzise definieren zu können, müssen wir zunächst die Übergangsfunktion so erweitern, dass sie auch für alle möglichen gelesenen *(Teil-)Wörter* einen Folgezustand bestimmt.

Definition 1.3 (kanonische Fortsetzung von δ)

Die *kanonische Fortsetzung von* $\delta : Q \times \Sigma \to Q$ *von einzelnen Symbolen auf beliebige Wörter* ist eine Funktion $\hat{\delta} : Q \times \Sigma^* \to Q$, die induktiv (über die Wortlänge) wie folgt definiert ist:

- $\hat{\delta}(q, \varepsilon) := q$
- $\hat{\delta}(q, wa) := \delta(\hat{\delta}(q, w), a)$

Beachte: für alle Symbole $a \in \Sigma$ und Zustände $q \in Q$ ist die obige Definition von $\hat{\delta}(q, a)$ identisch mit dem ursprünglichen δ, denn $\hat{\delta}(q, a) = \delta(\hat{\delta}(q, \varepsilon), a) = \delta(q, a)$.

Als Beispiel für Definition 1.3 betrachte wieder den Automaten $\mathcal{A}$ aus Beispiel 1.2. Es gilt $\hat{\delta}(q_0, bbbabbbb) = q_1$ und $\hat{\delta}(q_0, baaab) = q_3$.

Definition 1.4 (Akzeptiertes Wort, erkannte Sprache)

Ein DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ *akzeptiert* das Wort $w \in \Sigma^*$, wenn $\hat{\delta}(q_0, w) \in F$. Die von $\mathcal{A}$ *erkannte Sprache* ist $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}$.

Man sieht leicht, dass der Automat $\mathcal{A}$ aus Beispiel 1.2 die Sprache

$$L(\mathcal{A}) = \{w \in \{a, b\}^* \mid \ |w|_a \geq 3\}$$

erkennt. Mit anderen Worten: er akzeptiert genau diejenigen Wörter über dem Alphabet $\{a, b\}$, in denen das Symbol a mindestens 3-mal vorkommt.

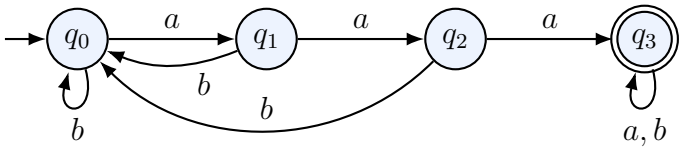
Definition 1.5 (Erkennbarkeit einer Sprache)

Eine Sprache $L \subseteq \Sigma^*$ heißt *erkennbar*, wenn es einen DEA $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

Wir haben also gerade gesehen, dass die Sprache $L = \{w \in \{a,b\}^* \mid |w|_a \geq 3\}$ erkennbar ist. Folgendes Beispiel liefert eine weitere erkennbare Sprache.

Beispiel 1.6

Der folgende DEA erkennt die Sprache $L = \{w = uaaav \mid u, v \in \Sigma^*\}$ mit $\Sigma = \{a, b\}$. Auch diese Sprache ist also erkennbar.

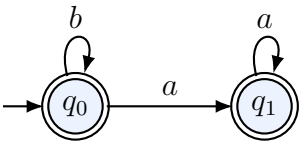


Anmerkung zur Begriffswahl. Anstelle des Begriffs „akzeptierender Zustand“ (auf Englisch: „accepting state“) wird oft „Endzustand“ (bzw. „final state“) verwendet. Dies erweckt jedoch den falschen Eindruck, dass die Berechnung zu Ende sei, sobald der DEA einen solchen Zustand erreicht. Tatsächlich ist die Berechnung aber erst zu Ende, wenn das gesamte Wort gelesen wurde (siehe Def. 1.4). Der Automat kann dabei zwischendurch einen akzeptierenden Zustand einnehmen; dies ist aber für das Akzeptieren des Wortes *am Ende der Berechnung* unerheblich.

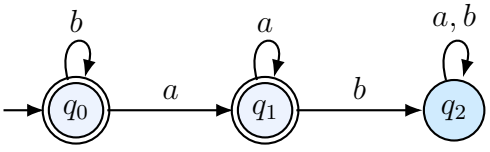
Totalheit der Übergansfunktion. Beachte: die Übergangsfunktion eines DEAs ist eine *totale Funktion*; es muss also für jede mögliche Kombination von Zustand und Symbol ein Folgezustand angegeben werden.

Beispiel 1.7

Folgendes ist also kein DEA, denn es fehlt ein Übergang für q_1 und b:



Man erhält aber leicht einen DEA durch Hinzunahme eines „Papierkorbzustandes“, der alle fehlenden Übergänge aufnimmt (und *kein* akzeptierender Zustand ist):



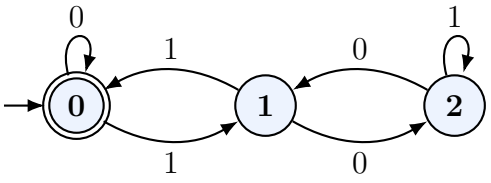
Die erkannte Sprache ist übrigens

$$L \;=\; \{b\}^* \cdot \{a\}^* \;=\; \{w \in \{a,b\}^* \mid ab \text{ ist nicht Infix von } w\}.$$

Ein komplexeres Beispiel. Wir geben nun noch ein etwas anspruchsvolleres Beispiel für eine Sprache, die mittels eines DEA erkennbar ist. Jede Zahl über dem Alphabet $\Sigma = \{0, 1\}$ repräsentiert eine Binärzahl, wobei das niederwertigste Bit ganz rechts steht und wir (links) führende Nullen ignorieren. Das Wort $1101 \in \Sigma^*$ repräsentiert also beispielsweise die Zahl 13. Das Wort ε repräsentiert dabei die Zahl 0. Wir interessieren uns für die Sprache

$$L = \{w \in \{0, 1\}^* \mid w \text{ ist Vielfaches von } 3\}.$$

Es ist nicht unmittelbar klar, wie man einen DEA $\mathcal{A}$ konstruiert so dass $L(\mathcal{A}) = L$. Wir behaupten, dass der folgende Automat $\mathcal{A}$ dies erreicht:



Auch das ist naturlich nicht unmittelbar klar und muss bewiesen werden. Dies werden wir nun tun. Die Zustandsnamen sind hier übrigens fett geschrieben, um sie von den Eingabesymbolen zu unterscheiden. Es wird gleich klar werden (Eigenschaft $(*)$ unten), warum wir diese Namen für die Zustände gewählt haben.

Wir machen zunächst zwei elementare Beobachtungen:

1. Wenn $w \in \Sigma^*$ binäre Repräsentation der Zahl n ist, dann repräsentiert $w0$ die Zahl $2n$ und $w1$ die Zahl $2n + 1$.
2. Im obigen Automaten $\mathcal{A}$ gilt:

   $(*)$ $\delta(q, a) = (2q + a) \mod 3$ für alle $q \in \{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$ und alle $a \in \{0, 1\}$.

   Dies ist leicht einzusehen, indem man einfach alle 6 relevanten Fälle prüft.

Wir zeigen nun folgende Behauptung, die unmittelbar impliziert, dass $L(\mathcal{A}) = L$.

Behauptung. $\hat{\delta}(\mathbf{0}, w) = w \mod 3$ für alle $w \in \{0, 1\}^*$.

Der Beweis ist per Induktion über die Länge $|w|$ des Wortes w.

Im Induktionsanfang ist $|w| = 0$, also $w = \varepsilon$. Nach Definition von $\hat{\delta}$ gilt $\hat{\delta}(\mathbf{0}, \varepsilon) = \mathbf{0}$. Das ist wie gewünscht denn $0 \mod 3 = 0$.

Für den Induktionsschritt betrachte das Wort wa mit $w \in \Sigma^*$ und $a \in \{0, 1\}$. Es gilt:

$$\begin{aligned}
\hat{\delta}(\mathbf{0}, wa) &= \delta(\hat{\delta}(\mathbf{0}, w), a) && (\text{Def.}\hat{\delta}) \\
&= \delta(w \bmod 3, a) && (\text{Ind. Vor.}) \\
&= (2(w \bmod 3) + a \bmod 3 && (*) \\
&= (2w + a) \bmod 3 && (\text{Rechnen mit mod}) \\
&= wa \bmod 3.
\end{aligned}$$

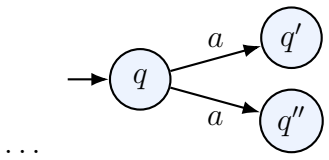
Dem Leser wird empfohlen, den „Rechnen mit modulo“ Teil kurz selbst nachzuvollziehen.

Randbemerkung. Im Prinzip sind „echte Computer“ ebenfalls endliche Automaten: sie haben nur endlich viel Speicherplatz und daher nur eine endliche Menge möglicher Konfigurationen (Prozessorzustand + Belegung der Speicherzellen). Die Konfigurationsübergänge werden bestimmt durch Verdrahtung und Eingaben (Tastatur, Peripheriegeräte).

Wegen der extrem großen Anzahl von Zuständen sind endliche Automaten aber keine geeignete Abstraktion für Rechner. Außerdem verwendet man einen Rechner (z. B. bei der Programmierung) üblicherweise nicht als endlichen Automaten; man nutzt also z. B. nicht aus, dass der Arbeitsspeicher ganz genau 16 GB groß ist. Stattdessen nimmt man den Speicher als potentiell unendlich an und verlässt sich auf Techniken wie Swapping und Paging. In einer geeigneten Abstraktion von Rechnern sollte daher auch der Speicher als unendlich angenommen werden. Das wichtigste solche Modell ist die Turingmaschine, die wir später im Detail kennen lernen werden.

1.2. Nichtdeterministische endliche Automaten

Wir generalisieren nun das Automatenmodell des DEA dadurch, dass wir *Nichtdeterminismus* zulassen. In unserem konkreten Fall bedeutet das, dass wir für einen gegebenen Zustand und ein gelesenes Symbol *mehr als einen möglichen Übergang* erlauben; Folgendes ist also möglich:



Ein Automat hat dadurch unter Umständen *mehrere* Möglichkeiten, ein Wort zu verarbeiten. Er akzeptiert seine Eingabe, wenn *eine Möglichkeit existiert*, dabei einen akzeptierenden Zustand zu erreichen.

Nichtdeterminismus ist ein fundamentales Konzept der Informatik, das nicht nur bei endlichen Automaten eine wichtige Rolle spielt. Wir werden es in dieser Vorlesung noch häufiger verwenden. Dabei werden mehrere Möglichkeiten wie oben immer durch *existentielles Quantifizieren* behandelt (also z. B. wie hier: „es existiert eine Möglichkeit, einen akzeptierenden Zustand zu erreichen“). Natürlich gibt es in der Realität keine nichtdeterministischen Maschinen. Dennoch ist Nichtdeterminismus aus folgenden Gründen von großer Bedeutung:

- Als Modellierungsmittel bei unvollständiger Information.

  Es ist häufig nicht sinnvoll, Ereignisse wie Benutzereingaben, einkommende Nachrichten von anderen Prozessen usw. im Detail zu modellieren, da man viel zu komplexe Modelle erhalten würde. Stattdessen verwendet man nichtdeterministische Übergänge ohne genauer zu spezifizieren, wann welcher Übergang verwendet wird.

- Große Bedeutung in der Komplexitätstheorie.

  In der Komplexitätstheorie (VL Berechenbarkeit) geht es unter anderem um die prinzipielle Frage, was effizient berechenbar ist und was nicht. Interessanterweise spielt dabei das zunächst praxisfern wirkende Konzept des Nichtdeterminismus eine zentrale Rolle. Paradebeispiel ist das Problem „P versus NP“ – das wichtigste ungelöste Problem der Informatik, welches uns in Teil IV begegnen wird.

NEAs ergeben sich dadurch, dass man die Übergangs*funktion* von DEAs durch eine Übergangs*relation* ersetzt. Wir definieren NEAs der Vollständigkeit halber noch einmal als Ganzes.

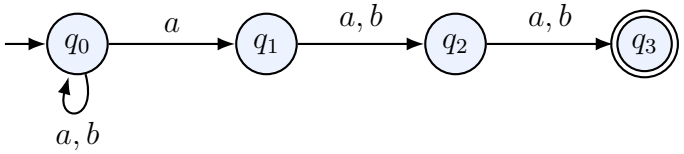
Definition 1.8 (NEA)

Ein *nichtdeterministischer endlicher Automat (NEA)* ist ein Tupel $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$, wobei

- Q eine endliche Menge von Zuständen ist,
- Σ ein Eingabealphabet ist,
- $q_0 \in Q$ der Anfangszustand ist,
- $\Delta \subseteq Q \times \Sigma \times Q$ *die Übergangsrelation ist*,
- $F \subseteq Q$ eine Menge von akzeptierenden Zuständen ist.

Beispiel 1.9

Folgenden NEA werden wir im Folgenden als durchgängiges Beispiel verwenden:



Dieser Automat ist kein DEA, da es an der Stelle q_0 für die Eingabe a zwei mögliche Übergänge gibt.

Um das Akzeptanzverhalten von NEAs zu beschreiben, verwenden wir eine etwas andere Notation als bei DEAs.

Definition 1.10 (Pfad)

Ein *Pfad* in einem NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ von einem Zustand $p_0 \in Q$ zu einem Zustand $p_n \in Q$ ist eine Folge

$$\pi \;=\; p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1 \xrightarrow{a_2}_{\mathcal{A}} \cdots \xrightarrow{a_n}_{\mathcal{A}} p_n$$

so dass $(p_i, a_{i+1}, p_{i+1}) \in \Delta$ für $i = 0, \ldots, n-1$. Der Pfad π hat die *Beschriftung* $w := a_1 \cdots a_n$. Wenn es in $\mathcal{A}$ einen Pfad von p nach q mit der Beschriftung w gibt, so schreiben wir:

$$p \overset{w}{\Longrightarrow}_{\mathcal{A}} q$$

Für $n = 0$ sprechen wir vom *leeren Pfad*, welcher die Beschriftung ε hat.

Im NEA aus Beispiel 1.9 gibt es unter anderem folgende Pfade für die Eingabe *aba*:

$$\begin{aligned} \pi_1 &= q_0 \xrightarrow{a}_{\mathcal{A}} q_1 \xrightarrow{b}_{\mathcal{A}} q_2 \xrightarrow{a}_{\mathcal{A}} q_3 \\ \pi_2 &= q_0 \xrightarrow{a}_{\mathcal{A}} q_0 \xrightarrow{b}_{\mathcal{A}} q_0 \xrightarrow{a}_{\mathcal{A}} q_1 \end{aligned}$$

Wie erwähnt basiert das Akzeptanzverhalten bei Nichtdeterminismus immer auf *existentieller Quantifizierung*:

Definition 1.11 (Akzeptiertes Wort, erkannte Sprache)
Der NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ *akzeptiert* das Wort $w \in \Sigma^*$, wenn $q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q_f$ für ein $q_f \in F$; mit anderen Worten: wenn *es einen Pfad* $p_0 \xrightarrow{a_1}_{\mathcal{A}} \cdots \xrightarrow{a_n}_{\mathcal{A}} p_n$ *gibt*, so dass $p_0 = q_0$ und $p_n \in F$. Die von $\mathcal{A}$ *erkannte Sprache* ist $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}$.

Der NEA aus Beispiel 1.9 akzeptiert also die Eingabe *aba*, weil der oben angegebene Pfad π_1 in einem akzeptierenden Zustand endet. Dabei ist es irrelevant, dass der ebenfalls mögliche Pfad π_2 in einem nicht-akzeptierenden Zustand endet. Nicht akzeptiert wird beispielsweise die Eingabe *baa*, da keiner der möglichen Pfade zu einem akzeptierenden Zustand führt. Man sieht leicht, dass der NEA aus Beispiel 1.9 die folgende Sprache akzeptiert:

$$L(\mathcal{A}) = \{w \in \{a, b\}^* \mid \text{das drittletzte Symbol in } w \text{ ist } a\}$$

Eine gute Hilfe zum Verständnis von Nichtdeterminismus ist die Metapher des *Ratens*. Intuitiv „rät“ der NEA aus Beispiel 1.9 im Zustand q_0 bei Eingabe von a, ob er sich gerade an der drittletzten Stelle des Wortes befindet oder nicht. Man beachte, dass der Automat keine Möglichkeit hat, das sicher zu wissen. Wenn er sich für „ja“ entscheidet, so wechselt er in den Zustand q_1 und *verifiziert* mittels der Kette von q_1 nach q_3, dass er richtig geraten hat:

- hat er in Wahrheit das zweitletzte oder letzte Symbol gelesen, so wird der akzeptierende Zustand nicht erreicht; der Automat akzeptiert also nicht;
- ist er weiter als drei Symbole vom Wortende entfernt, so ist in q_3 kein Übergang mehr möglich und der Automat „blockiert“ und akzeptiert ebenfalls nicht.

Die wichtigsten Eigenschaften eines solchen Rate-Ansatzes zum Erkennen einer Sprache L sind, dass (i) für Wörter $w \in L$ es die Möglichkeit gibt, richtig zu raten und (ii) für Wörter $w \notin L$ falsches Raten niemals zur Akzeptanz führt.

Da wir uns bei einem Automaten meist nur für die erkannte Sprache interessieren, bezeichnen wir zwei NEAs als *äquivalent*, wenn sie dieselbe Sprache akzeptieren.

Ohne Nichtdeterminismus, also mittels eines DEA, ist es schwieriger, die Sprache aus Beispiel 1.9 zu erkennen (Aufgabe: finde einen geeigneten NEA!). Es gilt aber interessanterweise, dass man zu jedem NEA einen äquivalenten DEA finden kann. Nichtdeterminismus trägt in diesem Fall also nicht zur Erhöhung der Ausdrucksstärke bei – das ist aber

keineswegs immer so, wie wir noch sehen werden. NEAs haben dennoch einen Vorteil gegenüber DEAs: manche Sprachen lassen sich im Vergleich zu DEAs mit erheblich (exponentiell) kleineren NEAs erkennen. Letzteres werden wir im Rahmen der Übungen kurz beleuchten. In der Vorlesung beweisen wir lediglich das folgende klassische Resultat.

Satz 1.12 (Rabin/Scott)
Zu jedem NEA kann man einen äquivalenten DEA konstruieren.

Bevor wir den Beweis dieses Satzes angeben, skizzieren wir kurz die

Beweisidee:
Der Beweis dieses Satzes verwendet die bekannte *Potenzmengenkonstruktion*: die Zustandsmenge des DEA ist die Potenzmenge 2^Q der Zustandsmenge Q des NEA. Jeder Zustand des DEA besteht also aus einer *Menge* von NEA-Zuständen; umgekehrt ist jede solche Menge ein DEA-Zustand.

Sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA. Nach der Definition von NEAs gilt $w \in L(\mathcal{A})$ gdw. die Menge $\{q \in Q \mid q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q\} \in 2^Q$ mindestens einen akzeptierenden Zustand enthält. Wir definieren also die Übergangsfunktion δ und Menge F' der akzeptierenden Zustände des DEAs so, dass für alle $w \in \Sigma^*$ gilt:

1. $\hat{\delta}(\{q_0\}, w) = \{q \mid q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q\}$ und
2. $\{q \mid q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q\}$ ist akzeptierender Zustand des DEA, wenn mindestens ein akzeptierender Zustand des ursprünglichen NEAs enthalten ist.

Intuitiv simuliert damit der eindeutige Lauf des DEAs auf einer Eingabe w *alle* möglichen Läufe des ursprünglichen NEAs auf w.

Beweis. Sei der NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ gegeben. Der DEA $\mathcal{A}' = (2^Q, \Sigma, \{q_0\}, \delta, F')$ ist definiert durch:

- $\delta(P, a) = \bigcup_{p \in P} \{p' \mid (p, a, p') \in \Delta\}$ für alle $P \in 2^Q$ und $a \in \Sigma$
- $F' = \{P \in 2^Q \mid P \cap F \neq \emptyset\}$

Wir benötigen im Folgenden die

Hilfsaussage: $q' \in \hat{\delta}(\{q_0\}, w)$ gdw. $q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q'$ $\quad(\star)$

Daraus folgt $L(\mathcal{A}) = L(\mathcal{A}')$, da:

$w \in L(\mathcal{A})$	gdw.	$\exists q \in F : q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q$	(Def. $L(\mathcal{A})$)
	gdw.	$\exists q \in F : q \in \hat{\delta}(\{q_0\}, w)$	(Hilfsaussage)
	gdw.	$\hat{\delta}(\{q_0\}, w) \cap F \neq \emptyset$	
	gdw.	$\hat{\delta}(\{q_0\}, w) \in F'$	(Def. F')
	gdw.	$w \in L(\mathcal{A}')$	

Beweis der Hilfsaussage mittels Induktion über $|w|$:

Induktionsanfang: $|w| = 0$

$q' \in \hat{\delta}(\{q_0\}, \varepsilon)$ gdw. $q_0 = q'$ gdw. $q_0 \stackrel{\varepsilon}{\Longrightarrow}_{\mathcal{A}} q'$

Induktionsvoraussetzung: Die Hilfsaussage ist bereits gezeigt für alle $w \in \Sigma^*$ mit $|w| \leq n$

Induktionsschritt: $|w| = n + 1$

Sei $w = ua$ mit $u \in \Sigma^*$, $|u| = n$ und $a \in \Sigma$. Es gilt:

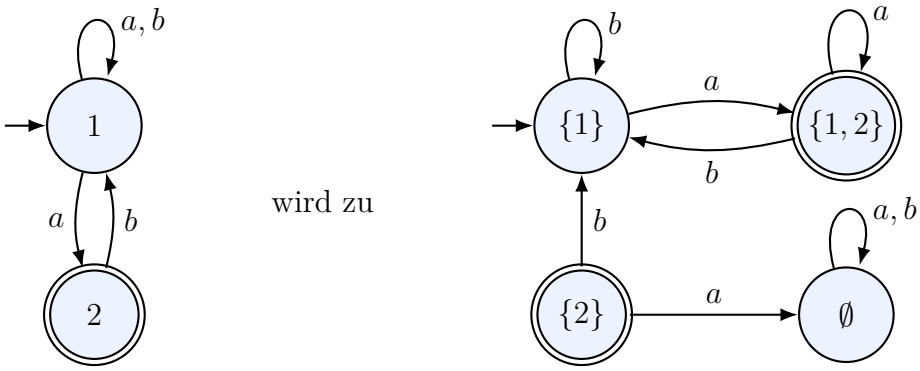
$$\begin{aligned}
\hat{\delta}(\{q_0\}, ua) &= \delta(\hat{\delta}(\{q_0\}, u), a) && \text{(Def. 1.3)} \\
&= \bigcup_{q' \in \hat{\delta}(\{q_0\}, u)} \{q'' \mid (q', a, q'') \in \Delta\} && \text{(Def. } \delta\text{)} \\
&= \bigcup_{q_0 \stackrel{u}{\Longrightarrow}_{\mathcal{A}} q'} \{q'' \mid (q', a, q'') \in \Delta\} && \text{(Induktionsvoraussetzung)} \\
&= \{q'' \mid q_0 \stackrel{ua}{\Longrightarrow}_{\mathcal{A}} q''\} && \text{(Def. Pfad)}
\end{aligned}$$

Daraus folgt sofort die Hilfsaussage für $w = ua$.

❑

Beispiel 1.13

Der NEA $\mathcal{A}$ (links) wird mit der Potenzmengenkonstruktion transformiert in den DEA $\mathcal{A}'$ (rechts):



Nachteilig an dieser Konstruktion ist, dass die Zustandsmenge stets *exponentiell* vergrößert wird. Im Allgemeinen kann man dies wie erwähnt nicht vermeiden; in manchen Fällen kommt man aber doch mit weniger Zuständen aus. Als einfache Optimierung kann man Zustände weglassen, die mit keinem Wort vom Startzustand aus erreichbar sind. In Abschnitt 6 werden wir zudem eine allgemeine Methode kennen lernen, um zu einer gegebenen erkennbaren Sprache den kleinstmöglichen DEA zu konstruieren.

Wir beenden diesen Abschnitt mit der Beschreibung einer praktischen Herangehensweise, um die Potenzmengenkonstruktion (per Hand) anzuwenden. Dabei verwenden wir als Notation eine Tabelle. Für Beispiel 1.13 sähe dies wie folgt aus:

	a	b
$\{1\}$	$\{1,2\}$	$\{1\}$
$\{1,2\}$	$\{1,2\}$	$\{1\}$

Man beginnt in der ersten Zeile der linken Spalte mit der Menge, die nur den Startzustand enthält. Dann bildet man in der ersten Zeile den Übergang für jedes Alphabetssymbol gemäß der Potenzmengenkonstruktion. Wenn eine Zustandsmenge in der Tabelle erscheint, so muß sie in der linken Spalte eingetragen und die Spalte ebenfalls vervollständigt werden. Nicht erscheinende Zustandsmengen sind nicht vom Startzustand aus erreichbar und können weggelassen werden. Es bleibt noch, die Menge der akzeptierenden Zustände zu bestimmen.

1.3. Endliche Automaten mit Wortübergängen

Wir betrachten noch zwei natürliche Varianten von NEAs, die sich in manchen technischen Konstruktionen als sehr nützlich herausstellen. Wir werden sehen, dass sie dieselben Sprachen erkennen können wie NEAs.

Definition 1.14 (NEA mit Wortübergängen, ε-NEA)
Ein *NEA mit Wortübergängen* hat die Form $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$, wobei Q, Σ, q_0, F wie beim NEA definiert sind und $\Delta \subseteq Q \times \Sigma^* \times Q$ eine endliche Menge von *Wortübergängen* ist.
Ein *ε-NEA* ist ein NEA mit Wortübergängen der Form (q, ε, q') und (q, a, q') mit $a \in \Sigma$.

Pfade, Pfadbeschriftungen und erkannte Sprache werden entsprechend wie für NEAs definiert. Zum Beispiel hat der Pfad

$$q_0 \xrightarrow{ab}_{\mathcal{A}} q_1 \xrightarrow{\varepsilon}_{\mathcal{A}} q_2 \xrightarrow{bb}_{\mathcal{A}} q_3$$

die Beschriftung $ab \cdot \varepsilon \cdot bb = abbb$.

Man beachte, dass $q \overset{a}{\Longrightarrow} p$ bedeutet, dass man von q nach p kommt, indem man zunächst beliebig viele ε-Übergänge macht, dann einen a-Übergang und danach wieder beliebig viele[1] ε-Übergänge (im Unterschied zu $q \xrightarrow{a} p$).

Satz 1.15
Zu jedem NEA mit Wortübergängen kann man einen äquivalenten NEA konstruieren.

Man zeigt Satz 1.15 mit Umweg über ε-NEAs.

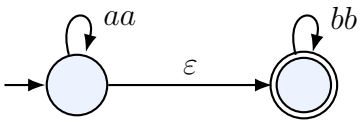
Lemma 1.16
Zu jedem NEA mit Wortübergängen kann man einen äquivalenten ε-NEA konstruieren.

[1] „Beliebig viele“ schließt dabei jeweils „keine“ mit ein.

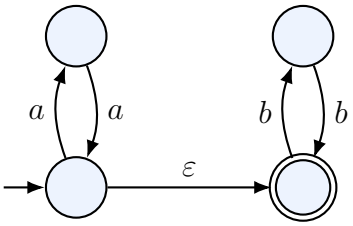
Beweis. Man ersetzt jeden Wortübergang $(q, a_1 \cdots a_n, q')$ mit $n > 1$ durch Symbolübergänge $(q, a_1, p_1), (p_1, a_2, p_2), \ldots, (p_{n-1}, a_n, q')$, wobei $p_1, \ldots, p_{n-1}$ jeweils neue Hilfszustände sind (die nicht zur Menge der akzeptierenden Zustände hinzugenommen werden). Man sieht leicht, dass dies einen äquivalenten ε-NEA liefert. ❑

Beispiel 1.17

Der NEA mit Wortübergängen, der durch die folgende Darstellung gegeben ist:



wird überführt in einen äquivalenten ε-NEA:



Lemma 1.18

Zu jedem ε-NEA kann man einen äquivalenten NEA konstruieren.

Beweis. Der ε-NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ sei gegeben. Wir konstruieren daraus einen NEA $\mathcal{A}'$ ohne ε- Übergänge wie folgt:

$\mathcal{A}' = (Q, \Sigma, q_0, \Delta', F')$, wobei

- $\Delta' := \left\{ (p, a, q) \in Q \times \Sigma \times Q \mid p \overset{a}{\Longrightarrow}_{\mathcal{A}} q \right\}$
- $F' := \begin{cases} F \cup \{q_0\} & \text{falls} \quad q_0 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} q_f \text{ für ein } q_f \in F \\ F & \text{sonst} \end{cases}$

Intuitiv gesprochen macht also der NEA $\mathcal{A}'$ die „impliziten“ a-Übergänge des ε-NEAs $\mathcal{A}$ explizit: die Übergangsrelation Δ' enthält genau dann einen a-Übergang von p nach q, wenn $\mathcal{A}$ von q nach p kommt, indem er zunächst beliebig viele ε-Übergänge macht, dann einen a-Übergang und danach wieder beliebig viele ε-Übergänge. Entsprechend muss F' nicht nur alle akzeptierenden Zustände von $\mathcal{A}$ enthalten, sondern zusätzlich den Anfangszustand, falls von diesem aus in $\mathcal{A}$ über ε-Kanten ein akzeptierender Zustand erreichbar ist.

Wir zeigen, dass $L(\mathcal{A}) = L(\mathcal{A}')$.

1. $L(\mathcal{A}') \subseteq L(\mathcal{A})$:
   Sei $w = a_1 \cdots a_n \in L(\mathcal{A}')$. Dann gibt es einen Pfad

   $$p_0 \xrightarrow{a_1}_{\mathcal{A}'} p_1 \xrightarrow{a_2}_{\mathcal{A}'} \cdots \xrightarrow{a_{n-1}}_{\mathcal{A}'} p_{n-1} \xrightarrow{a_n}_{\mathcal{A}'} p_n \quad \text{mit} \quad p_0 = q_0,\ p_n \in F'.$$

   Nach Definition von Δ' gibt es auch in $\mathcal{A}$ einen Pfad π von p_0 nach p_n mit Beschriftung w (der unter Umständen zusätzliche ε-Schritte enthält).

   1. Fall: $p_n \in F$
   Dann zeigt π, dass $w \in L(\mathcal{A})$.

   2. Fall: $p_n \in F' \setminus F$, d. h. $p_n = q_0$
   Nach Definition von F' gilt $q_0 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p$ für ein $p \in F$. Es gibt also in $\mathcal{A}$ einen Pfad von p_0 über $p_n = q_0$ nach $p \in F$ mit Beschriftung w; daher $w \in L(\mathcal{A})$.

2. $L(\mathcal{A}) \subseteq L(\mathcal{A}')$: Sei $w \in L(\mathcal{A})$.

   1. Fall: $w \neq \varepsilon$. Sei

   $$\pi = p_0 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p_0' \xrightarrow{a_1}_{\mathcal{A}} p_1 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p_1' \xrightarrow{a_2}_{\mathcal{A}} p_2 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} \cdots \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p_{n-1}' \xrightarrow{a_n}_{\mathcal{A}} p_n \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p_n'$$

   Pfad in $\mathcal{A}$ mit $p_0 = q_0$, $p_n' \in F$ und Beschriftung w. Nach Definition von Δ' ist

   $$p_0 \xrightarrow{a_1}_{\mathcal{A}'} p_1 \xrightarrow{a_2}_{\mathcal{A}'} \cdots \xrightarrow{a_{n-1}}_{\mathcal{A}'} p_{n-1} \xrightarrow{a_n}_{\mathcal{A}'} p_n'$$

   ein Pfad in $\mathcal{A}'$. Aus $p_n' \in F$ folgt $p_n' \in F'$; also $w \in L(\mathcal{A}')$.

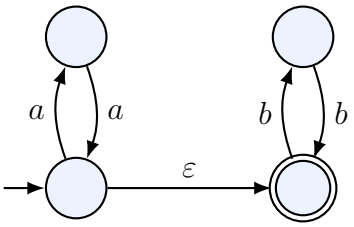
   2. Fall: $w = \varepsilon$

   Wegen $\varepsilon \in L(\mathcal{A})$ gibt es $p \in F$ mit $q_0 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p$. Also $q_0 \in F'$ nach Definition von F' und damit $\varepsilon \in L(\mathcal{A}')$.

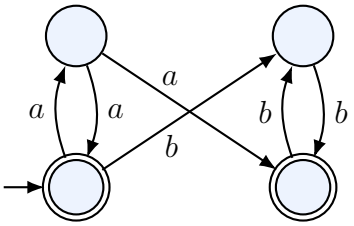
❑

Beispiel 1.19

Der ε-NEA aus Beispiel 1.17



wird in folgenden NEA überführt:



2. Nachweis der Nichterkennbarkeit

Nicht jede formale Sprache ist erkennbar. Im Gegenteil ist es so, dass nur solche Sprachen, die auf sehr reguläre Weise aufgebaut sind, erkennbar sein können. Es stellt sich also die Frage, wie man von einer Sprache nachweist, dass sie *nicht* erkennbar ist.

Um nachzuweisen, dass eine gegebene Sprache erkennbar ist, genügt es, einen endlichen Automaten (DEA oder NEA) dafür anzugeben. Der Nachweis, dass eine Sprache nicht erkennbar ist, gestaltet sich schwieriger: man kann nicht alle unendlich viele existierende Automaten durchprobieren, und es genügt auch nicht zu sagen, dass man keinen funktionierenden Automaten gefunden hat.

Darum verwendet man die folgende Strategie. Man etabliert eine allgemeine Eigenschaft, die von jeder erkennbaren Sprache erfüllt wird. Um von einer Sprache zu zeigen, dass sie nicht erkennbar ist, genügt es dann nachzuweisen, dass sie die Eigenschaft verletzt. Die wichtigste solche Eigenschaft wird durch das bekannte Pumping-Lemma beschrieben. Bevor wir es formulieren, verdeutlichen wir die zugrunde liegenden Gedanken an einer Beispielsprache.

Beispiel 2.1

$L = \{a^n b^n \mid n \geq 0\}$ ist *nicht* erkennbar.

Bevor wir den Beweis erbringen, überlegen wir uns intuitiv, was ein endlicher Automat tun müsste, um L zu erkennen. Er müsste für jedes Eingabewort w die Anzahl der gelesenen a's mit denen der danach gelesenen b's vergleichen und w genau dann akzeptieren, wenn diese Anzahlen übereinstimmen. Da ein endlicher Automat jedoch nur eine endliche Anzahl von Zuständen hat, aber die Zahl n in der Beschreibung von L unbegrenzt groß werden kann, ist dieses Zählen von keinem endlichen Automaten zu bewerkstelligen.

Man beachte, dass dieser Gedankengang nur *Intuitionen* wiedergibt. Er taugt nicht als formaler Beweis, denn er basiert auf der nur schwer exakt zu belegenden Annahme, dass die einzige Möglichkeit L zu erkennen auf dem Zählen von a's und b's basiert. Ein korrekter Beweis für die Nichterkennbarkeit von L sieht so aus:

Beweis. Angenommen, L sei erkennbar. Dann gibt es einen NEA $\mathcal{A}$ mit $L(\mathcal{A}) = L$. Dieser NEA hat eine endliche Anzahl von Zuständen, sagen wir n_0 Stück. Wir betrachten das Wort $w = a^{n_0}b^{n_0} \in L$. Da $w \in L(\mathcal{A})$, gibt es einen Pfad vom Anfangszustand q_0 zu irgendeinem Endzustand q_f in $\mathcal{A}$, der mit w beschriftet ist. Die ersten $n_0 + 1$ Zustände (einschließlich q_0) auf diesem Pfad werden durch Lesen der a's erreicht. Da $\mathcal{A}$ nur n_0 Zustände hat, muss unter diesen $n_0 + 1$ Zuständen ein Zustand q doppelt vorkommen. Der genannte Pfad lässt sich also auch schreiben als

$$q_0 \overset{x}{\Longrightarrow}_{\mathcal{A}} q \overset{y}{\Longrightarrow}_{\mathcal{A}} q \overset{z}{\Longrightarrow}_{\mathcal{A}} q_f \,,$$

wobei x, y, z Teilwörter von w sind mit $w = xyz$. Weil q in diesem Pfad doppelt auftaucht, kann man den Teilpfad zwischen diesen beiden Vorkommen verdoppeln und erhält einen Pfad

$$q_0 \overset{x}{\Longrightarrow}_{\mathcal{A}} q \overset{y}{\Longrightarrow}_{\mathcal{A}} q \overset{y}{\Longrightarrow}_{\mathcal{A}} q \overset{z}{\Longrightarrow}_{\mathcal{A}} q_f$$

auf dem Wort $xyyz$. Damit enthält $L(\mathcal{A})$ das Wort $xyyz$, welches die Form

$$a^{k_1}\, a^{2k_2}\, a^{k_3}\, b^{n_0}$$

mit $k_1 + 2k_3 + k_3 > n_0$ hat. Also hat $xyyz$ mehr a's als b's und kann damit nicht zu L gehören, obwohl es von $\mathcal{A}$ akzeptiert wird. Dies ist ein Widerspruch; also war die Annahme falsch, dass L erkennbar sei! ❏

Der dem letzten Beweis zugrunde liegende Hauptgedanke lässt sich verallgemeinern: Wenn wir annehmen, dass eine Sprache L mittels eines endlichen Automaten $\mathcal{A}$ erkennbar sei, und ein Wort $w \in L$ betrachten, das mehr Symbole enthält als $\mathcal{A}$ Zustände hat, dann muss $\mathcal{A}$ beim Akzeptieren von w einen Zustand q doppelt besuchen. Deshalb enthält L auch sämtliche Wörter, die man erhält, wenn man den Teil zwischen diesen beiden Besuchen von q beliebig vervielfacht. Dies ist im Wesentlichen die Aussage des Pumping-Lemmas.

Lemma 2.2 (Pumping-Lemma für erkennbare Sprachen)
Es sei L eine erkennbare Sprache. Dann gibt es eine natürliche Zahl $n_0 \geq 1$, so dass gilt: Jedes Wort $w \in L$ mit $|w| \geq n_0$ lässt sich zerlegen in $w = xyz$ mit

- *$y \neq \varepsilon$ und $|xy| \leq n_0$*
- *$xy^k z \in L$ für alle $k \geq 0$.*

Beweis. Sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA mit $L(\mathcal{A}) = L$. Wir wählen $n_0 = |Q|$. Sei nun $w = a_1 \cdots a_m \in L$ ein Wort mit $m \geq n_0$. Dann existiert ein Pfad

$$p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1 \xrightarrow{a_2}_{\mathcal{A}} \cdots \xrightarrow{a_{n_0}}_{\mathcal{A}} p_{n_0} \xrightarrow{a_{n_0+1}}_{\mathcal{A}} \cdots \xrightarrow{a_m}_{\mathcal{A}} p_m$$

mit $p_0 = q_0$ und $p_m \in F$. Wegen $n_0 = |Q|$ können die $n_0 + 1$ Zustände $p_0, \ldots, p_{n_0}$ nicht alle verschieden sein. Es gibt also i, j mit $0 \leq i < j \leq n_0$ und $p_i = p_j$. Wir wählen

$$x := a_1 \cdots a_i, \quad y := a_{i+1} \cdots a_j, \quad z := a_{j+1} \cdots a_m.$$

Offensichtlich gilt $y \neq \varepsilon$ (da $i < j$) und $|xy| \leq n_0$ (da $j \leq n_0$) sowie

$$q_0 = p_0 \overset{x}{\Longrightarrow}_{\mathcal{A}} p_i \overset{y}{\Longrightarrow}_{\mathcal{A}} p_i = p_j \overset{z}{\Longrightarrow}_{\mathcal{A}} p_m \in F.$$

Folglich gilt für alle $k \geq 0$ auch $p_i \overset{y^k}{\Longrightarrow}_{\mathcal{A}} p_i$, also

$$q_0 = p_0 \overset{x}{\Longrightarrow}_{\mathcal{A}} p_i \overset{y^k}{\Longrightarrow}_{\mathcal{A}} p_i = p_j \overset{z}{\Longrightarrow}_{\mathcal{A}} p_m \in F,$$

was $xy^k z \in L$ zeigt. ❏

Beispiel 2.3
Wir benutzen jetzt das Pumping-Lemma um zu zeigen:

$$L = \{a^n b^n \mid n \geq 0\} \text{ ist } \textit{nicht} \text{ erkennbar.}$$

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, L sei erkennbar. Es gibt also eine Zahl n_0 mit den in Lemma 2.2 beschriebenen Eigenschaften. Wähle das Wort

$$w = a^{n_0} b^{n_0} \in L.$$

Da $|w| \geq n_0$, gibt es eine Zerlegung $a^{n_0} b^{n_0} = xyz$ mit $|y| \geq 1$ und $|xy| \leq n_0$ sowie $xy^k z \in L$ für alle $k \geq 0$. Wegen $|xy| \leq n_0$ muss y ganz in a^{n_0} liegen. Also ist

$$x = a^{k_1}, \quad y = a^{k_2}, \quad z = a^{k_3} b^{n_0}$$

mit $k_2 > 0$ und $n_0 = k_1 + k_2 + k_3$. Damit ist aber

$$xy^0 z = xz = a^{k_1 + k_3} b^{n_0} \notin L,$$

da $k_1 + k_3 < n_0$. Widerspruch. Deshalb muss die Annahme „L ist erkennbar“ falsch sein. ❑

Beweise mit Hilfe des Pumping-Lemmas werden einfacher, wenn man die logische Struktur der Aussage genauer betrachtet: Lemma 2.2 hat die Form einer Implikation

$$\text{wenn } X, \text{ dann } Y,$$

wobei X die Aussage „L ist eine erkennbare Sprache“ und Y die Aussage „es gibt eine natürliche Zahl $n \geq 1, \ldots$“ darstellt. Anstatt wie in Beispiel 2.3 einen Widerspruchsbeweis zu führen, ist es bequemer, die *Kontraposition* der obigen Aussage zu verwenden:

$$\text{wenn nicht } Y, \text{ dann nicht } X.$$

Da eine Implikation und ihre Kontraposition logisch äquivalent sind, ergibt sich folgende direkte Konsequenz aus dem Pumping-Lemma.

Korollar 2.4 (Pumping-Lemma als Kontrapositiv)
Angenommen, für eine Sprache L gilt das Folgende:

***Für alle** natürlichen Zahlen $n_0 \geq 1$*
  ***gibt es** ein Wort $w \in L$ mit $|w| \geq n_0$, so dass*
    ***für alle** Zerlegungen $w = xyz$ mit $y \neq \varepsilon$ und $|xy| \leq n_0$ gilt:*
      ***es gibt** $k \geq 0$ mit $xy^k z \notin L$.*

*Dann ist L **nicht** erkennbar.*

Diese äquivalente Formulierung des Pumping-Lemmas legt eine *spieltheoretische Sicht* nahe: Wir spielen in Runden gegen einen Gegner. Der Gegner will zeigen, dass die fragliche Sprache L erkennbar ist; wir wollen zeigen, dass sie es nicht ist. In den Zeilen, die mit „für alle“ beginnen, ist der Gegner am Zug; in den Zeilen, die mit „es gibt“ beginnen, sind wir an der Reihe. Wenn wir das Spiel *stets gewinnen können* – und zwar unabhängig davon, was der Gegner tut – dann ist die Eigenschaft aus Korollar 2.4 erfüllt, also ist L nicht erkennbar.

Dieses Spiel verdeutlichen wir an den folgenden zwei Beispielen.

Beispiel 2.5
$L = \{a^n \mid n \text{ ist Quadratzahl}\}$ ist nicht erkennbar.

Beweis. Sei $n_0 \geq 1$ eine natürliche Zahl (vom Gegner gewählt, also können wir n_0 nicht näher bestimmen). Wir wählen das Wort $w = a^m$ mit $m = (n_0 + 1)^2$. Für dieses Wort gilt $w \in L$ und $|w| > n_0$. Sei nun xyz eine Zerlegung (vom Gegner gewählt) mit den Eigenschaften

$$y \neq \varepsilon \quad \text{und} \quad |xy| \leq n_0.$$

Wir müssen nun ein $k \geq 0$ finden, so dass $xy^kz \notin L$. Dazu beobachten wir zunächst, dass wegen $|w| > n_0$ und $|xy| \leq n_0$ gilt: $z \neq \varepsilon$. Wir wollen nun zeigen, dass wir y so „aufpumpen“ können, dass die Wortlänge die Form $s^2 + t$ bekommt, für geeignete s, t mit $0 < t < s$. So eine Zahl kann nämlich nie eine Quadratzahl sein, denn es gilt

$$s^2 \quad < \quad s^2 + t \quad < \quad s^2 + 2s + 1 \quad = \quad (s + 1)^2.$$

Wenn wir nun $k = 4 \cdot |y| \cdot |w|^2$ wählen, erreichen wir dieses Ziel, denn es gilt:

$$\begin{aligned} |xy^kz| &= |y| \cdot k + |x| + |z| \\ &= 4 \cdot |y|^2 \cdot |w|^2 + |x| + |z| \\ &= (2 \cdot |y| \cdot |w|)^2 + |x| + |z| \end{aligned}$$

Mit $s = 2 \cdot |y| \cdot |w|$ und $t = |x| + |z|$ gilt nun wie gewünscht $|xy^kz| = s^2 + t$ und $0 < t < s$. Also ist $xy^kz \notin L$, was zu zeigen war (d. h. wir gewinnen das Spiel). ❑

Der Beweis im vorangehenden Beispiel erfordert zwei kreative Schritte von uns: wir müssen (in Abhängigkeit von n_0, das der Gegner uns vorgibt) ein geeignetes Wort w wählen und dann (in Abhängigkeit von der Zerlegung $w = xyz$, die der Gegner uns vorgibt) ein geeignetes k finden, so dass wir logisch zwingend argumentieren können, dass xy^kz nicht in L sein kann. Der erste Schritt war in diesem Beispiel recht naheliegend: $w = a^{(n_0)^2}$ hätte es nicht erlaubt, $z \neq \varepsilon$ zu folgern; also haben wir $w = a^{(n_0+1)^2}$ gewählt. Im nächsten Beispiel müssen wir bei der Wahl von w etwas kreativer sein.

Beispiel 2.6
$L = \{a^n b^m \mid n \neq m\}$ ist *nicht* erkennbar.

Beweis. Sei wieder $n_0 \geq 1$ eine natürliche Zahl (vom Gegner gewählt). Wir wählen das Wort $w = a^{n_0} b^{n_0! + n_0}$, wobei $n_0! = 1 \cdot \ldots \cdot n_0$ (Fakultätsoperation). Der intuitive Grund für diese Wahl ist, dass wir (a) wie in Beispiel 2.3 erzwingen wollen, dass das y der Zerlegung vollständig in den a's liegt, und (b) wir durch „Aufpumpen“ von y erreichen müssen, dass das so gepumpte Wort genauso viele a's wie b's enthält. Da das Aufpumpen einer a-Folge der Multiplikation der Anzahl der a's entspricht, müssen wir die Anzahl der b's im Wesentlichen so wählen, dass sie durch alle Zahlen bis n_0 teilbar ist, und das ist der Fall für die Fakultätsoperation.

Sei nun xyz eine Zerlegung (vom Gegner gewählt) mit den Eigenschaften

$$y \neq \varepsilon \quad \text{und} \quad |xy| \leq n_0,$$

d. h. wie gewünscht liegt y vollständig in den a's. Also gilt:

$$x = a^{k_1}, \quad y = a^{k_2}, \quad z = a^{k_3} b^{n_0! + n_0}$$

für geeignete k_1, k_2, k_3 mit $k_1 + k_2 + k_3 = n_0$ und $k_2 > 0$ (da $y \neq \varepsilon$). Jetzt zahlt sich die Wahl der Fakultätsoperation aus: wegen $0 < k_2 \leq n_0$ gibt es nämlich eine Zahl ℓ mit:

$$\ell \cdot k_2 = n_0!$$

Wir können nun $k = \ell + 1$ wählen, denn dann ist die Anzahl a's im Wort $xy^{\ell+1}z$:

$$\begin{aligned} k_1 + (\ell + 1)k_2 + k_3 &= \underbrace{k_1 + k_2 + k_3}_{=n_0} + \underbrace{\ell \cdot k_2}_{=n_0!} \\ &= n_0 + n_0! \end{aligned}$$

Da die b's nur im Teilwort z auftreten, ist deren Anzahl unverändert gleich $n_0 + n_0!$. Also ist $xy^{\ell+1}z = a^{n_0+n_0!}b^{n_0+n_0!} \notin L$, was zu zeigen war (d. h. wir gewinnen das Spiel). ❑

Mit Hilfe des Pumping-Lemmas gelingt es leider nicht immer, die Nichterkennbarkeit einer Sprache nachzuweisen, denn es gibt Sprachen, die nicht erkennbar sind, aber trotzdem die in Lemma 2.2 beschriebene Pumping-Eigenschaft erfüllen. Anders ausgedrückt ist die Pumping-Eigenschaft aus Lemma 2.2 zwar *notwendig* für die Erkennbarkeit einer Sprache, aber nicht *hinreichend*.

Beispiel 2.7

Ist $L = \{a^m b^n c^n \mid m, n \geq 1\} \cup \{b^m c^n \mid m, n \geq 0\}$ erkennbar?

Versucht man, Nichterkennbarkeit mit Lemma 2.2 zu zeigen, so scheitert man, da L die Eigenschaft aus dem Pumping-Lemma erfüllt:

Betrachte $n_0 = 3$ (vom Gegner gewählt). Es sei nun $w \in L$ mit $|w| \geq 3$, d. h. es tritt einer der folgenden drei Fälle ein.

1. $w = a^m b^n c^n$ mit $m, n \geq 1$ (w ist aus dem „1. Teil“ von L)
2. $w = b^m c^n$ mit $m \geq 1$ und $n \geq 0$ (w ist aus dem „2. Teil“ von L und beginnt mit b)
3. $w = c^n$ mit $n \geq 1$ (w ist aus dem „3. Teil“ von L und beginnt mit c, da $|w| \geq 3$)

Wir zeigen: in jedem dieser drei Fälle lässt sich w zerlegen in $w = xyz$ mit $y \neq \varepsilon$ und $|xy| \leq n_0$ sowie $xy^k z \in L$ für alle $k \geq 0$.

1. Fall. Wenn $w = a^m b^n c^n$ mit $m, n \geq 1$, dann sei die Zerlegung

$$x = \varepsilon, \quad y = a, \quad z = a^{m-1} b^n c^n$$

betrachtet (vom Gegner gewählt). Dann ist für *jedes* $k \geq 0$ das Wort

$$xy^k z = a^{k+(m-1)} b^n c^n$$

in L, denn die Anzahlen der a's und b's sind im „1. Teil" unabhängig (und falls $(m-1)+k = 0$, ist $xy^k z$ im „2. Teil" von L).

2. Fall. Wenn $w = b^m c^n$ mit $m \geq 1$ und $n \geq 0$, dann sei die Zerlegung

$$x = \varepsilon, \quad y = b, \quad z = b^{m-1} c^n$$

betrachtet (vom Gegner gewählt). Dann ist für *jedes* $k \geq 0$ das Wort

$$xy^k z = b^{k+(m-1)} c^n$$

in L, denn die Anzahlen der b's und c's sind im „2. Teil" unabhängig.

3. Fall. Wenn $w = c^n$ mit $n \geq 1$, dann sei die Zerlegung

$$x = \varepsilon, \quad y = c, \quad z = c^{n-1}$$

betrachtet (vom Gegner gewählt). Dann ist für *jedes* $k \geq 0$ das Wort

$$xy^k z = c^{k+(n-1)}$$

in L (wiederum im „2. Teil").

Wir können also in keinem der drei Fälle erreichen, dass $xy^k z \notin L$ (d. h. wir gewinnen das Spiel nicht). Dadurch misslingt der Nachweis, dass L nicht erkennbar ist.

Im Abschnitt 6 werden wir ein Werkzeug kennen lernen, mit dem der Nachweis der Nichterkennbarkeit dieser Sprache L gelingt.

In der Literatur findet man verschärfte (und kompliziertere) Varianten des Pumping-Lemmas, die dann auch hinreichend sind (z. B. Jaffes Pumping-Lemma). Diese Varianten liefern also eine automatenunabhängige Charakterisierung der erkennbaren Sprachen.

3. Abschlusseigenschaften

Endliche Automaten definieren eine ganze *Klasse* von Sprachen: die erkennbaren Sprachen (auch *reguläre Sprachen* genannt, siehe Kapitel 5). Anstatt die Eigenschaften einzelner Sprachen zu studieren (wie z. B. Erkennbarkeit) kann man auch die Eigenschaften ganzer Sprachklassen betrachten. Wir interessieren uns hier insbesondere für Abschlusseigenschaften, zum Beispiel unter Schnitt: wenn L_1 und L_2 erkennbare Sprachen sind, dann ist auch $L_1 \cap L_2$ eine erkennbare Sprache.

Es stellt sich heraus, dass die Klasse der erkennbaren Sprachen unter den meisten natürlichen Operationen abgeschlossen ist. Diese Eigenschaft ist für viele technische Konstruktionen und Beweise sehr nützlich. Wie wir beispielsweise sehen werden, kann man manchmal die Anwendung des Pumping-Lemmas durch ein viel einfacheres Argument ersetzen, das auf Abschlusseigenschaften beruht. Später werden wir sehen, dass andere interessante Sprachklassen *nicht* unter allen natürlichen Operationen abgeschlossen sind.

Satz 3.1 (Abschlusseigenschaften erkennbarer Sprachen)
Sind L_1 und L_2 erkennbar, so sind auch die folgenden Sprachen erkennbar.

- $L_1 \cup L_2$ *(Vereinigung)*
- $\overline{L_1}$ *(Komplement)*
- $L_1 \cap L_2$ *(Schnitt)*
- $L_1 \cdot L_2$ *(Konkatenation)*
- L_1^* *(Kleene-Stern)*

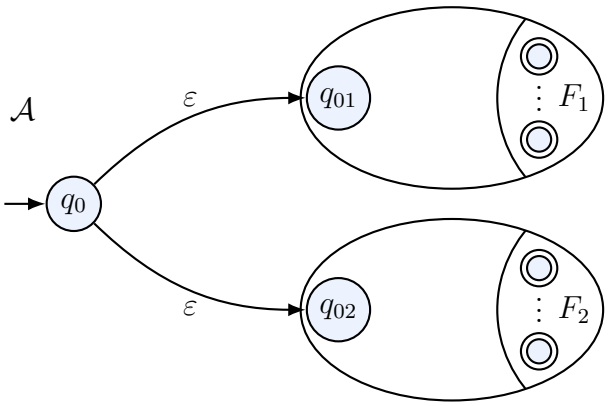
Beweis. Seien $\mathcal{A}_i = (Q_i, \Sigma, q_{0i}, \Delta_i, F_i)$ zwei NEAs für L_i $(i = 1, 2)$. O. B. d. A. gelte $Q_1 \cap Q_2 = \emptyset$.

(1) Abschluss unter Vereinigung:

Der folgende ε-NEA erkennt $L_1 \cup L_2$:
$\mathcal{A} := (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, q_0, \Delta, F_1 \cup F_2)$, wobei

- $q_0 \notin Q_1 \cup Q_2$ und
- $\Delta := \Delta_1 \cup \Delta_2 \cup \{(q_0, \varepsilon, q_{01}), (q_0, \varepsilon, q_{02})\}$.

Schematisch sieht der *Vereinigungsautomat* $\mathcal{A}$ so aus.



Mit Lemma 1.18 gibt es zu $\mathcal{A}$ einen äquivalenten NEA.

(2) Abschluss unter Komplement:

Einen DEA für $\overline{L_1}$ erhält man wie folgt.

Zunächst verwendet man die Potenzmengenkonstruktion, um zu $\mathcal{A}_1$ einen äquivalenten DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ zu konstruieren. Den DEA für $\overline{L_1}$ erhält man nun durch Vertauschen der akzeptierenden Zustände mit den nicht akzeptierenden:

$$\overline{\mathcal{A}} := (Q, \Sigma, q_0, \delta, Q \setminus F).$$

Es gilt nämlich:

$w \in \overline{L_1}$	gdw.	$w \notin L(\mathcal{A}_1)$
	gdw.	$w \notin L(\mathcal{A})$
	gdw.	$\hat{\delta}(q_0, w) \notin F$
	gdw.	$\hat{\delta}(q_0, w) \in Q \setminus F$
	gdw.	$w \in L(\overline{\mathcal{A}})$

Beachte: Diese Konstruktion funktioniert nicht für NEAs.

(3) Abschluss unter Schnitt:

Wegen $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ folgt (3) aus (1) und (2).

Da die Potenzmengenkonstruktion, die wir für $\overline{L_1}$ und $\overline{L_2}$ benötigen, recht aufwändig ist und exponentiell große Automaten liefert, kann es günstiger sein, direkt einen NEA für $L_1 \cap L_2$ zu konstruieren, den so genannten *Produktautomaten*:

$$\mathcal{A} := (Q_1 \times Q_2, \Sigma, (q_{01}, q_{02}), \Delta, F_1 \times F_2)$$

mit

$$\Delta := \{((q_1, q_2), a, (q_1', q_2')) \mid (q_1, a, q_1') \in \Delta_1 \text{ und } (q_2, a, q_2') \in \Delta_2\}$$

Ein Übergang in $\mathcal{A}$ ist also genau dann möglich, wenn der entsprechende Übergang in $\mathcal{A}_1$ *und* $\mathcal{A}_2$ möglich ist.

Behauptung. $L(\mathcal{A}) = L_1 \cap L_2$.

Sei $w = a_1 \cdots a_n$. Dann ist $w \in L(\mathcal{A})$ gdw. es gibt einen Pfad

$$(q_{1,0}, q_{2,0}) \xrightarrow{a_1}_{\mathcal{A}} (q_{1,1}, q_{2,1}) \cdots (q_{1,n-1}, q_{2,n-1}) \xrightarrow{a_n}_{\mathcal{A}} (q_{1,n}, q_{2,n})$$

mit $(q_{1,0}, q_{2,0}) = (q_{01}, q_{02})$ und $(q_{1,n}, q_{2,n}) \in F_1 \times F_2$. Nach Konstruktion von $\mathcal{A}$ ist das der Fall gdw. für jedes $i \in \{1, 2\}$

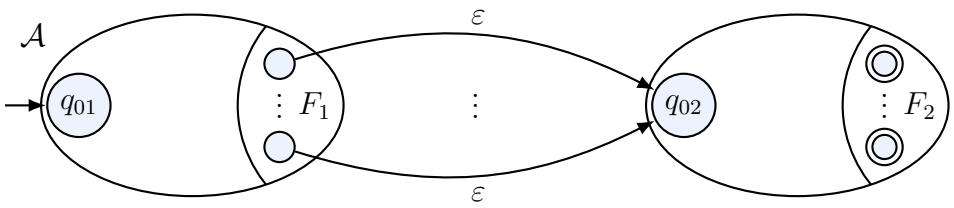
$$q_{i,0} \xrightarrow{a_1}_{\mathcal{A}_i} q_{i,1} \cdots q_{i,n-1} \xrightarrow{a_n}_{\mathcal{A}_i} q_{i,n}$$

ein Pfad ist mit q_{0i} und $q_{i,n} \in F_i$. Solche Pfade existieren gdw. $w \in L_1 \cap L_2$.

(4) Abschluss unter Konkatenation:

Der folgende ε-NEA erkennt $L_1 \cdot L_2$:

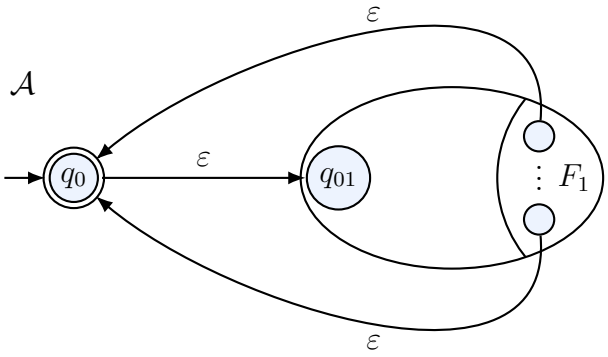
$$\mathcal{A} := (Q_1 \cup Q_2, \Sigma, q_{01}, \Delta, F_2), \quad \text{wobei}$$
$$\Delta := \Delta_1 \cup \Delta_2 \cup \{(f, \varepsilon, q_{02}) \mid f \in F_1\}$$



(5) Abschluss unter Kleene-Stern:

Der folgende ε-NEA erkennt L_1^*:

$$\mathcal{A} := (Q_1 \cup \{q_0\}, \Sigma, q_0, \Delta, \{q_0\}), \quad \text{wobei } q_0 \notin Q_1 \text{ und}$$
$$\Delta := \Delta_1 \cup \{(f, \varepsilon, q_0) \mid f \in F_1\} \cup \{(q_0, \varepsilon, q_{01})\}$$



Anmerkung: diese Konstruktion funktioniert nicht, wenn man anstelle des neuen Zustands q_0 den ursprünglichen Startzustand q_{01} verwendet (Übung!)

❑

Beachte: In den Fällen 1, 3, 4 und 5 ist der konstruierte Automat jeweils polynomiell in der Größe der Automaten für L_1 und L_2. In Fall 2 (Komplement) kann der konstruierte Automat exponentiell groß werden, wenn man mit einem NEA beginnt.

Man kann derartige Abschlusseigenschaften verwenden, um Nichterkennbarkeit einer Sprache L nachzuweisen.

Beispiel 3.2

$L := \{a^n b^m \mid n \neq m\}$ ist *nicht* erkennbar (vgl. Beispiel 2.6). Anstatt dies direkt mit dem Pumping-Lemma (Lemma 2.2) zu zeigen, kann man auch verwenden, dass bereits bekannt ist, dass die Sprache $L' := \{a^n b^n \mid n \geq 0\}$ *nicht* erkennbar ist. Wäre nämlich L erkennbar, so auch $L' = \overline{L} \cap \{a\}^* \cdot \{b\}^*$. Da wir schon wissen, dass L' nicht erkennbar ist, kann auch L nicht erkennbar sein.

Wir betrachten nun noch eine weitere Abschlusseigenschaft. Diese beruht auf dem Begriff des Homomorphismus.

Definition 3.3

Seien Σ und Γ Alphabete. Ein *Homomorphismus von* Σ^* *nach* Γ^* ist eine Abbildung $h : \Sigma^* \to \Gamma^*$, so dass $h(wv) = h(w)h(v)$ für alle $w, v \in \Sigma^*$.

Ist $L \subseteq \Sigma^*$ eine Sprache, so bezeichnen wir die Sprache $h(L) := \{h(w) \mid w \in L\}$ als das *homomorphe Bild von* L *unter* h.

Informell kann man sagen: ein Homomorphismus bildet Wörter über Σ auf Wörter über Γ so ab, dass dabei die Konkatenation respektiert wird.

Aus Definition 3.3 folgt das Folgende:

- $h(\varepsilon) = \varepsilon$
- $h(a_1 \cdots a_n) = h(a_1) \cdots h(a_n)$ für alle $a_1 \cdots a_n \in \Sigma^*$.

Um Punkt 1 einzusehen ist es hilfreich, Wortlängen zu betrachten:

$$\begin{aligned} |h(\varepsilon)| &= |h(\varepsilon\varepsilon)| \\ &= |h(\varepsilon)h(\varepsilon)| \\ &= |h(\varepsilon)| + |h(\varepsilon)|. \end{aligned}$$

wobei die zweite Gleichheit aus der Definition von Homomorphismen folgt. Wenn wir nun auf beiden Seiten $|h(\varepsilon)|$ subtrahieren, ergibt such $|h(\varepsilon)| = 0$, also $h(\varepsilon) = \varepsilon$.

Wegen Punkt 2 reicht es zur Definition eines Homomorphismus h immer aus, nur $h(a)$ für alle $a \in \Sigma$ anzugeben.

Beispiel 3.4

Sei $L = \{a^n b^n \mid n \geq 0\}$. Für

- $h(a) = b$ und $h(b) = a$ gilt: $h(L) = \{b^n a^n \mid n \geq 0\}$
- $h(a) = a$ und $h(b) = a$ gilt: $h(L) = \{a^n \mid n \text{ geradzahlig }\}$
- $h(a) = a$ und $h(b) = \varepsilon$ gilt: $h(L) = \{a\}^*$

Homomorphismen liefern eine weitere wichtige Abschlusseigenschaft der regulären und kontextfreien Sprachen, welche wir hier jedoch nicht beweisen wollen (wir verweisen stattdessen auf die Literatur).

Satz 3.5

Ist $L \subseteq \Sigma^*$ eine erkennbar Sprache und $h : \Sigma^* \to \Gamma^*$ ein Homomorphismus, dann ist das homomorphe Bild $h(L)$ ebenfalls erkennbar.

Auch den Abschluss unter homomorphen Bildern kann man nutzen, um Nichterkennbarkeit zu beweisen.

Beispiel 3.6

Wir zeigen, dass $L = \{(ab)^n c^*(de)^n \mid n \geq 0\}$ nicht erkennbar ist. Dazu verwenden wie wieder, dass die Sprache $L' := \{a^n b^n \mid n \geq 0\}$ nicht erkennbar ist. Für $h(a) = a$, $h(b) = h(c) = h(d) = \varepsilon$ und $h(e) = b$ gilt nämlich $L = h(L')$. Wäre also L erkennbar, so mit Satz 3.5 auch L', was aber nicht der Fall ist.

4. Entscheidungsprobleme

Wenn man einen endlichen Automaten in einer konkreten Anwendung einsetzen will, so ist es wichtig, sich zunächst vor Augen zu führen, was *genau* man mit dem Automaten anfangen möchte. In Abhängigkeit davon kann man dann die konkreten, in dieser Anwendung zu lösenden algorithmischen Probleme bestimmen.

Wir betrachten drei typische Probleme im Zusammenhang mit erkennbaren Sprachen. Bei allen dreien handelt es sich um *Entscheidungsprobleme*, also um Probleme, für die der Algorithmus eine Antwort aus der Menge $\{\text{ja}, \text{nein}\}$ berechnen soll – präzise einführen werden wir diesen Begriff erst in Teil III. Die drei betrachteten Probleme sind die folgenden.

- **das Wortproblem:**

  Gegeben ein endlicher Automat $\mathcal{A}$ und eine Eingabe $w \in \Sigma^*$ für $\mathcal{A}$, entscheide ob $w \in L(\mathcal{A})$.

- **das Leerheitsproblem:**

  Gegeben ein endlicher Automat $\mathcal{A}$, entscheide ob $L(\mathcal{A}) = \emptyset$.

- **das Äquivalenzproblem:**

  Gegeben endliche Automaten $\mathcal{A}_1$ und $\mathcal{A}_2$, entscheide ob $L(\mathcal{A}_1) = L(\mathcal{A}_2)$.

Wir werden Algorithmen für diese Probleme betrachten und deren Laufzeit analysieren.

In den obigen Problemen können die als Eingabe gegebenen endlichen Automaten entweder DEAs oder NEAs sein. Für die Anwendbarkeit der Algorithmen macht das im Prinzip keinen Unterschied, da man zu jedem NEA ja einen äquivalenten DEA konstruieren kann (man überlegt sich leicht, dass die Potenzmengenkonstruktion aus Satz 1.12 algorithmisch implementierbar ist). Bezüglich der Laufzeit kann es aber einen erheblichen Unterschied geben, da der Übergang von NEAs zu DEAs einen exponentiell größeren Automaten liefert und damit auch die Laufzeit exponentiell größer wird.

4.1. Das Wortproblem für DEAs und NEAs

Ist der Eingabeautomat $\mathcal{A}$ für das Wortproblem ein DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, so kann man beginnend mit q_0 durch wiederholte Anwendung von δ berechnen, in welchem Zustand $\mathcal{A}$ nach dem Lesen der Eingabe w ist. Man muss dann nur noch prüfen, ob dies ein akzeptierender Zustand ist. Bei dieser Vorgehensweise muss man δ offensichtlich $|w|$-mal anwenden, und jede Anwendung benötigt $|\delta|$ Schritte (Durchsuchen von δ nach dem passenden Übergang).

Satz 4.1

Das Wortproblem für DEAs ist in Zeit $\mathcal{O}(|w| \cdot |\delta|)$ entscheidbar.

Häufig betrachtet man beim Wortproblem den Automaten $\mathcal{A}$ als *fest gegeben* und das Wort w als die alleinige Eingabe. Dadurch wird die Größe von $\mathcal{A}$ zur Konstante und trägt nicht mehr zur Laufzeit bei, die dann als $\mathcal{O}(|w|)$ abgeschätzt wird und damit zu *Linearzeit* wird. Diese Art der Komplexitätsbetrachtung nennt man auch *Datenkomplexität*. Motiviert ist die Datenkomplexität durch folgende Erwägungen:

- häufig werden viele Worte w für denselben Automaten $\mathcal{A}$ getestet; in diesem Sinne ist $\mathcal{A}$ dann in der Tat fest gegeben, das Wort w aber nicht;
- der Automat $\mathcal{A}$ ist oft verhältnismäßig klein im Vergleich zum Wort w; so könnte $\mathcal{A}$ etwa ein Suchpattern repräsentieren und w eine Datenbank, letztere wäre dann um mehrere Größenordnungen größer.

Für einen NEA ist der triviale für DEAs verwendete Algorithmus nicht möglich, da es ja mehrere mit w beschriftete Pfade geben kann. In der Tat führen die naiven Ansätze zum Entscheiden des Wortproblems für NEAs zu exponentieller Laufzeit:

- Alle Pfade für das Eingabewort durchprobieren.

  Im schlimmsten Fall gibt es $|Q|^{|w|}$ viele solche Pfade, also exponentiell viele. Wenn $w \notin L(\mathcal{A})$, werden alle diese Pfade auch tatsächlich überprüft.

- Erst Potenzmengenkonstruktion anwenden, um DEA zu konstruieren.

  Wie bereits erwähnt, führt die exponentielle Vergrößerung des Automaten zu exponentieller Laufzeit.

Es stellt sich allerdings heraus, dass auch das Wortproblem für NEAs effizient lösbar ist. Die Idee ist, ebenso wie in der Potenzmengenkonstruktion alle Pfade des NEA gleichzeitig zu verfolgen, die mit dem gegebenen Eingabewort beschriftet sind.

Dies wird realisiert durch folgenden einfachen Algorithmus, der als Eingabe einen NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ und ein Wort $w = a_1 \cdots a_n \in \Sigma^*$ erhält:

procedure akz$(\mathcal{A}, w)$
  $Q_0 := \{q_0\}$
  for $1 \leq i \leq n$ **do**
    $Q_i := \{q \in Q \mid \exists p \in Q_{i-1} : (p, a_i, q) \in \Delta\}$
  if $Q_n \cap F \neq \emptyset$ **then return** 'yes'
  else return 'no'

Der Algorithmus ist korrekt:

Lemma 4.2

akz$(\mathcal{A}, w)$ = yes gdw. w von $\mathcal{A}$ akzeptiert wird.

Beweis. Sei $\mathcal{A}'$ der DEA, der aus $\mathcal{A}$ mittels Potenzmengenkonstruktion hervorgeht und δ' seine Übergangsfunktion. Aus der Definition des Algorithmus und der Konstruktion von $\mathcal{A}'$ folgt leicht:

Behauptung: Für $0 \leq i < n$ gilt $\hat{\delta}'(q_0, w_i) = Q_i$ wobei w_i das Präfix von w der Länge i ist.

Formal beweist man das per Induktion über i. Wir lassen die Details zur Übung. Die Behauptung impliziert (ebenfalls nach der Definition des Algorithmus und der Konstruktion von $\mathcal{A}'$), dass $\text{akz}(\mathcal{A}, w) = \text{yes}$ gdw. w von $\mathcal{A}'$ akzeptiert wird. Dies impliziert Lemma 4.2 da $\mathcal{A}$ und $\mathcal{A}'$ dieselbe Sprache erkennen. ❑

Wir analysieren nun kurz die Laufzeit des obigen Algorithmus:

- die **for**-Schleife macht $|w|$ Iterationen;
- in jeder Iteration gegen wir alle Zustände $p \in Q_{i-1}$ durch und dann für jedes solche p alle Übergänge in Δ. Dies benötigt Zeit $\mathcal{O}(|Q| \times |\Delta|)$;
- eine naive Implementierung des Testes „$Q_n \cap F \neq \emptyset$“ist möglich in Zeit $\mathcal{O}(|Q|^2)$.

Satz 4.3

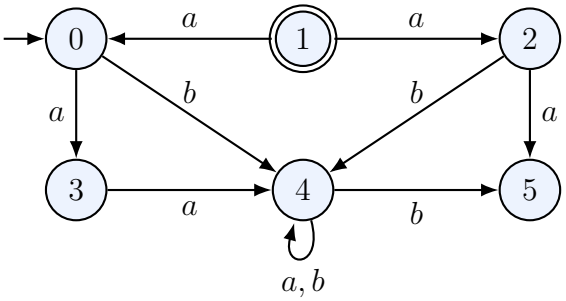
Das Wortproblem für NEAs ist entscheidbar in Zeit $\mathcal{O}((|w| \cdot |Q| \cdot |\Delta|) + |Q|^2)$.

Einen alternativen Ansatz, das Wortproblem für NEAs zu lösen, werden wir im folgenden Abschnitt kennenlernen.

4.2. Das Leerheitsproblem für DEAs und NEAs

Wir betrachten hier direkt NEAs. Da jeder DEA auch ein NEA ist, können die entwickelten Algorithmen natürlich auch für DEAs verwendet werden.

Im Gegensatz zum Wortproblem ist beim Leerheitsproblem keine konkrete Eingabe gegeben. Es scheint daher zunächst, als müsse man alle (unendlich vielen) Eingaben durchprobieren, was natürlich unmöglich ist. Ein einfaches Beispiel zeigt aber sofort, dass das Leerheitsproblem sehr einfach zu lösen ist. Betrachte den folgenden NEA $\mathcal{A}$:



Offensichtlich ist $L(\mathcal{A}) = \emptyset$, da der akzeptierende Zustand vom Startzustand aus gar nicht erreichbar ist. Man überlegt sich leicht, dass auch die umgekehrte Implikation gilt: wenn $L(\mathcal{A}) = \emptyset$, dann ist kein akzeptierender Zustand vom Startzustand aus erreichbar,

denn sonst würde die Beschriftung eines den akzeptierenden Zustand erreichenden Pfades ein Wort $w \in L(\mathcal{A})$ liefern. Das Leerheitsproblem ist also nichts weiter als ein Erreichbarkeitsproblem auf gerichteten Graphen wie dem oben dargestellten.

Formal definieren wir zu jedem NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ einen gerichteten Graph $G_\mathcal{A} = (V, E)$ durch Setzen von

$$\begin{aligned} V &= Q \\ E &= \{(q_1, q_2) \mid \exists a \in \Sigma : (q_1, a, q_2) \in \Delta\}. \end{aligned}$$

Es ist nun leicht zu sehen, dass folgendes gilt.

Lemma 4.4

$L(\mathcal{A}) = \emptyset$ gdw. in $G_\mathcal{A}$ kein $q \in F$ von q_0 aus erreichbar ist.

Das Erreichbarkeitsproblem in gerichteten Graphen ist mittels *Breitensuche* in Zeit $\mathcal{O}(|V| + |E|)$ lösbar. Dies wurde in der Vorlesung „Algorithmen und Datenstrukturen“ näher behandelt, siehe auch das Buch „Introduction to Algorithms“ [CLRS01]. Vorraussetzung ist, dass die Kantenmenge E *als Liste* repräsentiert ist (im Gegensatz zu beispielsweise: als Adjazenzmatrix).

Wenn die Übergangsrelation Δ des NEA $\mathcal{A}$ als Liste gegeben ist, dann können wir den assoziierten Graph $G_\mathcal{A} = (V, E)$ offensichtlich leicht in Zeit $\mathcal{O}(|V| + |E|)$ konstruieren, wobei E dann ebenfalls als Liste repräsentiert ist. Anwendung von Lemma 4.4 liefert folgendes Resultat.

Satz 4.5

Das Leerheitsproblem für NEAs ist in Zeit $\mathcal{O}(|Q| + |\Delta|)$ entscheidbar.

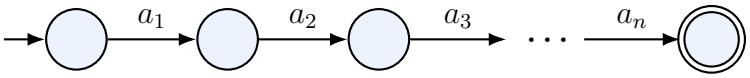
Wir betrachten nun noch einen alternativen Algorithmus für das *Wort*problem für NEAs, der in einer *Reduktion* auf das Leerheitsproblem besteht. Wir interessieren uns hauptsächlich dafür, weil Reduktionen eine wichtige Methode sind. Der Ansatz liefert jedoch gegenüber Satz auch eine leicht verbesserte Laufzeit.

Satz 4.6

Das Wortproblem für NEAs ist in Zeit $\mathcal{O}(|w| \cdot (|Q| + |\Delta|))$ entscheidbar.

Die Reduktion besteht darin, den schon gefundenen Algorithmus für das Leerheitsproblem zu verwendetn, um das Wortproblem zu lösen (mehr zu Reduktionen findet sich in den Teilen III und IV).

Beweis. Konstruiere zunächst einen Automaten $\mathcal{A}_w$, der genau das Wort $w = a_1 \cdots a_n$ akzeptiert:



Dieser Automat hat $|w|+1$ Zustände. Offenbar ist

$$w \in L(\mathcal{A}) \qquad \text{gdw.} \qquad L(\mathcal{A}) \cap L(\mathcal{A}_w) \neq \emptyset.$$

Wir können also entscheiden, ob $w \in L(\mathcal{A})$ ist, indem wir zunächst den Produktautomaten zu $\mathcal{A}$ und $\mathcal{A}_w$ konstruieren und dann unter Verwendung von Satz 4.5 prüfen, ob dieser eine nicht-leere Sprache erkennt.

Wir analysieren zunächst die Größe des Produktautomaten:

- Anzahl der Zustände: $|Q| \cdot (|w|+1)$
- Anzahl der Übergänge:

  Da $\mathcal{A}_w$ genau $|w|$ Übergänge hat, hat der Produktautomat höchstens $|w| \cdot |\Delta|$ Übergänge.

Nach Satz 4.5 ist daher der Aufwand zum Testen von $L(\mathcal{A}) \cap L(\mathcal{A}_w) \neq \emptyset$ also:

$$\mathcal{O}(|Q| \cdot (|w|+1) + |w| \cdot |\Delta|) \;=\; \mathcal{O}(|w| \cdot (|Q| + |\Delta|))$$

Auch die Konstruktion des Produktautomaten benötigt Zeit. Man überlegt sich leicht, dass auch hierfür die Zeit $\mathcal{O}(|w| \cdot (|Q| + |\Delta|)$ ausreichend ist. Als Gesamtlaufzeit ergibt sich

$$2 \cdot \mathcal{O}(|w| \cdot (|Q| + |\Delta|)) \;=\; \mathcal{O}(|w| \cdot (|Q| + |\Delta|)).$$

❑

4.3. Das Äquivalenzproblem für DEAs und NEAs

Wir verwenden sowohl für DEAs als auch für NEAs eine Reduktion auf das Leerheitsproblem:

$$L_1 = L_2 \qquad \text{gdw.} \qquad (L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$$

Im Fall des Äquivalenzproblems wollen wir auf eine ganz exakte Analyse der Laufzeit des sich ergebenden Algorithmus verzichten. Allerdings gibt es einen interessanten Unterschied zwischen DEAs und NEAs, der im folgenden Satz herausgearbeitet wird.

Satz 4.7

Das Äquivalenzproblem für DEAs ist in polynomieller Zeit entscheidbar. Für NEAs ist es in exponentieller Zeit entscheidbar.

Beweis. Die Konstruktion für Schnitt (Produktautomat) und Vereinigung ist sowohl für DEAs als auch für NEAs polynomiell. Bei der Komplementierung ist dies nur dann der Fall, wenn bereits DEAs vorliegen. Bei NEAs muss zunächst die Potenzmengenkonstruktion angewendet werden, daher kann der auf Leerheit zu testende Automat exponentiell groß sein. Damit ergibt sich exponentielle Laufzeit. ❑

Wir werden in Teil IV sehen, dass sich der exponentielle Zeitaufwand für das Äquivalenzproblem für NEAs (wahrscheinlich) nicht vermeiden lässt. Vorgreifend auf Teil IV sei erwähnt, dass das Äquivalenzproblem für NEAs PSpace-vollständig ist und damit zu einer Klasse von Problemen gehört, die wahrscheinlich nicht in polynomieller Zeit lösbar sind.

5. Reguläre Ausdrücke und Sprachen

Wir haben bereits einige *verschiedene Charakterisierungen* der Klasse der erkennbaren Sprachen gesehen:

Eine Sprache $L \subseteq \Sigma^*$ ist *erkennbar* gdw.

(1) $L = L(\mathcal{A})$ für einen DEA $\mathcal{A}$.

(2) $L = L(\mathcal{A})$ für einen NEA $\mathcal{A}$.

(3) $L = L(\mathcal{A})$ für einen ε-NEA $\mathcal{A}$.

(4) $L = L(\mathcal{A})$ für einen NEA mit Wortübergängen $\mathcal{A}$.

Im Folgenden betrachten wir eine weitere Charakterisierung mit Hilfe *regulärer Ausdrücke*. Diese Stellen eine bequeme „Sprache“ zur Verfügung, mittels derer erkennbare Sprachen beschrieben werden können. Varianten von regulären Ausdrücken werden in Tools wie `Emacs`, `Perl` und `sed` zur Beschreibung von Mustern („Patterns“) verwendet.

Definition 5.1 (Syntax regulärer Ausdrücke)
Sei Σ ein endliches Alphabet. Die Menge Reg_Σ der *regulären Ausdrücke über* Σ ist induktiv definiert:

- $\emptyset$, ε, a (für $a \in \Sigma$) sind Elemente von Reg_Σ.
- Sind $r, s \in \mathsf{Reg}_\Sigma$, so auch $(r + s), (r \cdot s), r^* \in \mathsf{Reg}_\Sigma$.

Beispiel 5.2

- $((a \cdot b^*) + \emptyset^*)^* \in \mathsf{Reg}_\Sigma$ für $\Sigma = \{a, b\}$
- $(((a \cdot b)^* + (c \cdot b)^*) + a^*) \in \mathsf{Reg}_\Sigma$ für $\Sigma = \{a, b, c\}$

Notation:
Um Klammern zu sparen, lassen wir Außenklammern weg und vereinbaren,

- dass * stärker bindet als $\cdot$
- dass $\cdot$ stärker bindet als $+$
- $\cdot$ lassen wir meist ganz wegfallen.

Der obere Ausdruck aus Beispiel 5.2 kann also geschrieben werden als $(ab^* + \emptyset^*)^*$.

Außerdem wird gleich aus Definition 5.3 folgen, dass $+$ und $\cdot$ assoziativ sind, es beschreiben also z.B. $((\alpha + \beta) + \gamma)$ und $(\alpha + (\beta + \gamma))$ dieselbe Sprache. Also lassen wir auch hier Klammern weg und schreiben $\alpha + \beta + \gamma$.

Statt $(((a \cdot b)^* + (c \cdot b)^*) + a^*)$ wie in Beispiel 5.2 schreiben wir also $(ab)^* + (cb)^* + a^*$.

Um die Bedeutung bzw. Semantik von regulären Ausdrücken zu definieren, wird jedem regulären Ausdruck r über Σ eine formale Sprache $L(r)$ zugeordnet.

Definition 5.3 (Semantik regulärer Ausdrücke)
Die durch den regulären Ausdruck r definierte Sprache $L(r)$ ist induktiv definiert:

$$\begin{aligned} L(\emptyset) &:= \emptyset & L(r+s) &:= L(r) \cup L(s) \\ L(\varepsilon) &:= \{\varepsilon\} & L(r \cdot s) &:= L(r) \cdot L(s) \\ L(a) &:= \{a\} & L(r^*) &:= L(r)^* \end{aligned}$$

Eine Sprache $L \subseteq \Sigma^*$ heißt *regulär*, falls es ein $r \in \mathsf{Reg}_\Sigma$ gibt mit $L = L(r)$.

Beispiel 5.4

- $(a+b)^*ab(a+b)^*$ definiert die Sprache aller Wörter über $\{a,b\}$, die Infix ab haben.
- $L(ab^* + b) = \{ab^i \mid i \geq 0\} \cup \{b\}$

Vereinfachung regulärer Ausdrücke

Sowohl in der Praxis als auch in theoretischen Konstruktionen ist es häufig sinnvoll, reguläre Ausdrücke soweit wie möglich zu vereinfachen. Auf dieses Thema werden wir im Folgenden einen kurzen Blick werfen.

Wir nennen zwei reguläre Ausdrücke r, s *äquivalent* falls $L(r) = L(s)$ und schreiben dann $r \equiv s$. Wenn $r \equiv s$, dann können wir also in jedem Kontext den Ausdruck r durch den (möglicherweise einfacheren) Ausdruck s ersetzen.

Einige grundlegende Äquivalenzen für r und s lauten wie folgt:

$$\begin{aligned} r + (s+t) &\equiv (r+s) + t && \text{(Assoziativität } +) \\ r \cdot (s \cdot t) &\equiv (r \cdot s) \cdot t && \text{(Assoziativität } \cdot) \\ r + s &\equiv s + r && \text{(Kommutativität } +) \\ r(s+t) &\equiv rs + rt && \text{(Distributivität)} \\ (r+s)t &\equiv rt + st && \text{(Distributivität)} \\ r + \emptyset &\equiv \emptyset + r \equiv r && (\emptyset \text{ neutrales Element für } +) \\ r \cdot \varepsilon &\equiv \varepsilon \cdot r \equiv r && (\varepsilon \text{ neutrales Element für } \cdot) \\ r \cdot \emptyset &\equiv \emptyset && (\emptyset \text{ absorbierendes Element für } \cdot) \end{aligned}$$

Für jedes Alphabet Σ sei $\mathcal{L}(\mathrm{Reg}_\Sigma)$ die Menge aller regulären Sprachen über Σ. Aus den obigen Äquivalenzen folgt, dass die Algebra $(\mathcal{L}(\mathrm{Reg}_\Sigma), +, \cdot, \emptyset, \varepsilon)$ ein Halbring mit Nullelement $\emptyset$ und Einselement ε ist. Dies ist auch der Grund, warum man für Vereinigung und Konjunktion die Symbole $+$ und $\cdot$ wählt. Es gelten aber noch einige zusätzliche Äquivalenten, die in Halbringen im allgemeinen nicht gelten müssen. Insbesondere taucht

der Kleene-Stern in den obigen Äquivalenzen und der soeben erwähnten Algebra ja gar nicht auf. Es gilt beispielsweise:

$$\begin{array}{rcll} r + r & \equiv & r & \text{(Idempotenz +)} \\ \varepsilon + rr^* & \equiv & r^* & \text{Äquivalenz 1 für Kleene Stern} \\ \varepsilon + r^*r & \equiv & r^* & \text{Äquivalenz 2 für Kleene Stern} \end{array}$$

Interessant wäre es, eine möglichst kleine Menge von Äquivalenzen zu finden, die vollständig ist in dem Sinne, dass sich alle anderen gültigen Äquivalenzen daraus herleiten lassen (was genau mir „Herleiten“gemeint ist, wollen wir hier nicht präzise machen). Es lässt sich aber beweisen, dass es eine endliche solche Menge nicht geben kann. Aus diesem Grund ist es nützlich, auch Aussagen über Ausdrücke zu treffen, die nur eine *Teil*sprache eines anderen Ausdrucks definieren (statt *dieselbe* Sprache wie bei Äquivalenz).

Wir schreiben $r \leq s$ falls $s \equiv s + r$ und lesen $r \leq s$ als *Ausdruck r definiert eine Teilsprache von Ausruck s*. Man sieht leicht, dass gilt:

- $r \leq s$ gdw. $L(r) \subseteq L(s)$ sowie
- $r \equiv s$ gdw. $r \leq s$ und $s \leq r$.

Nun gilt beispielsweise:

$$\begin{array}{ll} \text{wenn } rs \leq s, \text{ dann } r^*s \leq s & \text{Implikation 1} \\ \text{wenn } sr \leq s, \text{ dann } sr^* \leq s & \text{Implikation 2.} \end{array}$$

Aus den oben gezeigten Äquivalenzen und Implikationen lässt sich nun in der Tat jede gültige Äquivalenz herleiten. Wir wollen dies am Beispiel der Äquivalenz $r^* \equiv r^*r^*$ zeigen. Dafür machen wir zunächst eine nützliche Beobachtung.

Beobachtung: wenn $r \equiv s + t$, dann $s \leq r$

Beweis:

$$\begin{array}{rcll} r & \equiv & s + t & \text{(Annahme)} \\ & \equiv & (s + s) + t & \text{(Idempotenz +)} \\ & \equiv & s + (s + t) & \text{(Assoziativität +)} \\ & \equiv & s + r & \text{(Annahme)} \end{array}$$

Um $r^* \equiv r^*r^*$ zu zeigen, genügt es zu beweisen, dass $r^* \leq r^*r^*$ und $r^*r^* \leq r^*$ gilt.

($r^*r^* \leq r^*$). Äquivalenz 1 für Kleene-Stern liefert $r^* \equiv \varepsilon + rr^*$. Aus der Beobachtung folgt dann $rr^* \leq r$ und Implikation 1 liefert $r^*r^* \leq r^*$.

($r^* \leq r^*r^*$). Es gilt

$$\begin{aligned} r^*r^* &\equiv r^*(\varepsilon + rr^*) && \text{(Äquivalenz 1 für Kleene-Stern)} \\ &\equiv r^*\varepsilon + r^*rr^* && \text{(Distributivität)} \\ &\equiv r^* + r^*rr^* && (\varepsilon \text{ neutrales Element für } \cdot) \end{aligned}$$

Die Beobachtung liefert nun wie gewünscht $r^* \leq r^*r^*$.

Bemerkung:
Statt $L(r)$ schreiben wir im Folgenden häufig einfach r.
Dies ermöglicht es uns z. B. zu schreiben:

- $(ab)^*a = a(ba)^*$ eigentlich: $L((ab)^*a) = L(a(ba)^*)$
- $L(\mathcal{A}) = ab^* + b$ eigentlich: $L(\mathcal{A}) = L(ab^* + b)$

Wir zeigen nun, dass man mit regulären Ausdrücken genau die erkennbaren Sprachen definieren kann.

Satz 5.5 (Kleene)
Für eine Sprache $L \subseteq \Sigma^*$ sind äquivalent:

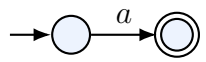
1) L ist regulär.

2) L ist erkennbar.

Beweis.

„1 $\Rightarrow$ 2“: Induktion über den Aufbau regulärer Ausdrücke

Induktionsanfang:

- $L(\emptyset) = \emptyset$ erkennbar:  ist NEA für $\emptyset$ (kein akz. Zustand).
- $L(\varepsilon) = \{\varepsilon\}$ erkennbar:  ist NEA für $\{\varepsilon\}$.
- $L(a) = \{a\}$ erkennbar:  ist NEA für $\{a\}$.

Induktionsschritt:

Weiß man bereits, dass $L(r)$ und $L(s)$ erkennbar sind, so folgt mit Satz 3.1 (Abschlusseigenschaften), dass auch $L(r + s)$, $L(r \cdot s)$ und $L(r^*)$ erkennbar sind, denn laut Definition 5.3 gilt:

- $L(r + s) = L(r) \cup L(s)$
- $L(r \cdot s) = L(r) \cdot L(s)$
- $L(r^*) = L(r)^*$

„2 ⇒ 1“: Sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA mit $L = L(\mathcal{A})$. Für alle $p, q \in Q$ und $X \subseteq Q$ sei $L^X_{p,q}$ die Sprache aller Wörter $w = a_1 \cdots a_n$, für die es einen Pfad

$$p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1 \xrightarrow{a_2}_{\mathcal{A}} \cdots \xrightarrow{a_{n-1}}_{\mathcal{A}} p_{n-1} \xrightarrow{a_n}_{\mathcal{A}} p_n$$

gibt mit $p_0 = p$, $p_n = q$ und $\{p_1, \ldots, p_{n-1}\} \subseteq X$. Offensichtlich gilt:

$$L(\mathcal{A}) = \bigcup_{q_f \in F} L^Q_{q_0, q_f}$$

Es reicht also, zu zeigen dass alle Spachen $L^X_{p,q}$ regulär sind. Dies erfolgt per Induktion über die Größe von X.

Induktionsanfang: $X = \emptyset$.

- 1. Fall: $p \neq q$

  Dann ist $L^\emptyset_{p,q} = \{a \in \Sigma \mid (p, a, q) \in \Delta\}$. Damit hat $L^\emptyset_{p,q}$ die Form $\{a_1, \ldots, a_k\}$ und der entsprechende reguläre Ausdruck ist $a_1 + \cdots + a_k$.
- 2. Fall: $p = q$

  Wie voriger Fall, außer dass $L^\emptyset_{p,q}$ (und damit auch der konstruierte reguläre Ausdruck) zusätzlich ε enthält.

Induktionsschritt: $X \neq \emptyset$.

Wähle ein beliebiges $\widehat{q} \in X$. Dann gilt:

$$L^X_{p,q} \;=\; L^{X\setminus\{\widehat{q}\}}_{p,q} \;\cup\; \left(L^{X\setminus\{\widehat{q}\}}_{p,\widehat{q}} \;\cdot\; \left(L^{X\setminus\{\widehat{q}\}}_{\widehat{q},\widehat{q}}\right)^* \;\cdot\; L^{X\setminus\{\widehat{q}\}}_{\widehat{q},q}\right) \qquad (*)$$

Für die Sprachen, die auf der rechten Seite verwendet werden, gibt es nach Induktionsvoraussetzung reguläre Ausdrücke. Außerdem sind alle verwendeten Operationen in regulären Ausdrücken verfügbar. Es bleibt also, $(*)$ zu zeigen.

„⊆“: Sei $w \in L^X_{p,q}$. Dann gibt es einen Pfad

$$p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1 \xrightarrow{a_2}_{\mathcal{A}} \cdots \xrightarrow{a_{n-1}}_{\mathcal{A}} p_{n-1} \xrightarrow{a_n}_{\mathcal{A}} p_n$$

mit $p_0 = p$, $p_n = q$ und $\{p_1, \ldots, p_{n-1}\} \subseteq X$. Wenn $\widehat{q}$ nicht in $\{p_1, \ldots, p_{n-1}\}$ vorkommt, dann $w \in L^{X\setminus\{\widehat{q}\}}_{p,q}$. Andernfalls seien $i_1, \ldots, i_k$ alle Indizes mit $p_{i_j} = \widehat{q}$ (und $i_1 < \cdots < i_k$). Offensichtlich gilt:

- $a_0 \cdots a_{i_1} \in L^{X\setminus\{\widehat{q}\}}_{p,\widehat{q}}$
- $a_{i_j+1} \cdots a_{i_{j+1}} \in L^{X\setminus\{\widehat{q}\}}_{\widehat{q},\widehat{q}}$ für $1 \leq j < k$
- $a_{i_k+1} \cdots a_n \in L^{X\setminus\{\widehat{q}\}}_{\widehat{q},q}$

„⊇“: Wenn $w \in L_{p,q}^{X\setminus\{\widehat{q}\}}$, dann offenbar $w \in L_{p,q}^{X}$. Wenn

$$w \in \left(L_{p,\widehat{q}}^{X\setminus\{\widehat{q}\}} \cdot \left(L_{\widehat{q},\widehat{q}}^{X\setminus\{\widehat{q}\}} \right)^* \cdot L_{\widehat{q},q}^{X\setminus\{\widehat{q}\}} \right),$$

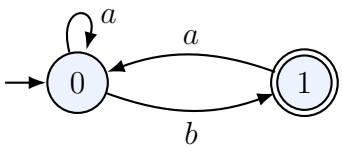
dann $w = xyz$ mit $x \in L_{p,\widehat{q}}^{X\setminus\{\widehat{q}\}}$, $y \in (L_{\widehat{q},\widehat{q}}^{X\setminus\{\widehat{q}\}})^*$, und $z \in L_{\widehat{q},q}^{X\setminus\{\widehat{q}\}}$. Setzt man die entsprechenden Pfade für x, y und z zusammen, so erhält man einen mit w beschrifteten Pfad von p nach q in $\mathcal{A}$, in dem alle Zustände außer dem ersten und letzten aus X sind. Also $w \in L_{p,q}^{X}$.

❑

Wenn man die Konstruktion aus „$2 \Rightarrow 1$“ in der Praxis anwendet, so ist es meist sinnvoll, die Zustände $\widehat{q}$ so zu wählen, dass der Automat in möglichst viele nicht-verbundene Teile zerfällt.

Beispiel 5.6

Betrachte den folgenden NEA $\mathcal{A}$:



Da 1 der einzige akzeptierende Zustand ist, gilt $L(\mathcal{A}) = L_{0,1}^{Q}$. Wir wenden wiederholt $(*)$ an:

$$\begin{aligned} L_{0,1}^{Q} &= L_{0,1}^{\{0\}} \cup L_{0,1}^{\{0\}} \cdot (L_{1,1}^{\{0\}})^* \cdot L_{1,1}^{\{0\}} \\ L_{0,1}^{\{0\}} &= L_{0,1}^{\emptyset} \cup L_{0,0}^{\emptyset} \cdot (L_{0,0}^{\emptyset})^* \cdot L_{0,1}^{\emptyset} \\ L_{1,1}^{\{0\}} &= L_{1,1}^{\emptyset} \cup L_{1,0}^{\emptyset} \cdot (L_{0,0}^{\emptyset})^* \cdot L_{0,1}^{\emptyset} \end{aligned}$$

Im ersten Schritt hätten wir natürlich auch 0 anstelle von 1 aus X eliminieren können. Der Induktionsanfang liefert:

$$\begin{aligned} L_{0,1}^{\emptyset} &= b \\ L_{0,0}^{\emptyset} &= a + \varepsilon \\ L_{1,1}^{\emptyset} &= \varepsilon \\ L_{1,0}^{\emptyset} &= a \end{aligned}$$

Einsetzen und Vereinfachen liefert nun:

$$\begin{aligned} L_{0,1}^{\{0\}} &= b + (a+\varepsilon) \cdot (a+\varepsilon)^* \cdot b = a^*b \\ L_{1,1}^{\{0\}} &= \varepsilon + a \cdot (a+\varepsilon)^* \cdot b = \varepsilon + aa^*b \\ L_{0,1}^{Q} &= a^*b + a^*b \cdot (\varepsilon + aa^*b)^* \cdot (\varepsilon + aa^*b) = a^*b(aa^*b)^* \end{aligned}$$

Der zu $\mathcal{A}$ gehörende reguläre Ausdruck ist also $a^*b(aa^*b)^*$.

Der reguläre Ausdruck, der in der Richtung „$2 \Rightarrow 1$“ aus einem NEA konstruiert wird, ist im Allgemeinen exponentiell größer als der ursprüngliche NEA. Man kann zeigen, dass dies nicht vermeidbar ist.

Beachte: Aus Satz 3.1 und Satz 5 folgt, dass es zu allen regulären Ausdrücken r und s

- einen Ausdruck t gibt mit $L(t) = L(r) \cap L(s)$;
- einen Ausdruck t' gibt mit $L(t') = \overline{L(r)}$.

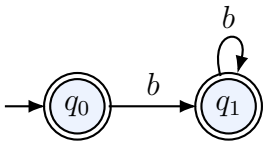
Es ist offensichtlich sehr schwierig, diese Ausdrücke direkt aus r und s (also ohne den Umweg über Automaten) zu konstruieren.

Beobachtungen über relative Größe

Satz 1.12 und Satz 5 zeigen uns, dass sich jede Sprache, die durch einen NEA erkannt werden kann, auch durch einen DEA erkennbar ist und durch einen regulären Ausdruck definierbar. Dabei kann die Größe dieser Automaten bzw. Ausdrücke jedoch sehr unterschiedlich sein. Im Folgenden werden wir diesen Aspekt etwas genauer beleuchten.

Beispiel 5.7

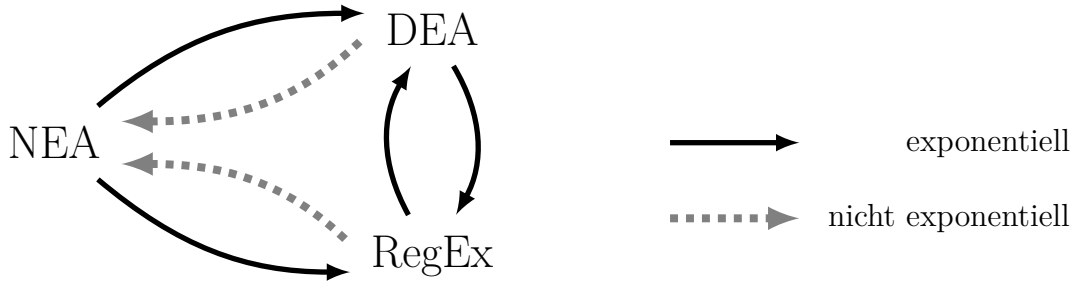
Betrachte den folgenden NEA $\mathcal{A}$:



Dieser erkennt die Sprache $\{b\}^*$ und hat 2 Zustände. Der reguläre Ausdruck $r = b + b^*$ drückt dieselbe Sprache aus, hat jedoch Länge 4.

Offensichtlich ist der Automat $\mathcal{A}$ aus Beispiel 5.7 kleiner als der reguläre Ausdruck r, in dem Sinne, dass die Anzahl der Zustände von $\mathcal{A}$ (unser Größenmaß für Automaten) kleiner ist als die Länge von r (unser Größenmaß für reguläre Ausdrücke). Besonders aufschlussreich ist dies für das Verhältnis der Größe von NEAs und regulären Ausdrücken jedoch nicht, denn einerseits können wir nicht sagen, ob der Unterschied nur an der ungünstigen Wahl des Ausdruckes liegt und andererseits läßt sich auf Basis *einen einzigen* Beispiels keine interessante Aussage über Größenverhältnisse treffen. Gilt für $n = 2$ und $m = 4$ dass $m = n + 2$, $m = 2 \cdot n$, $m = n^2$, oder sogar $m = 2^n$?

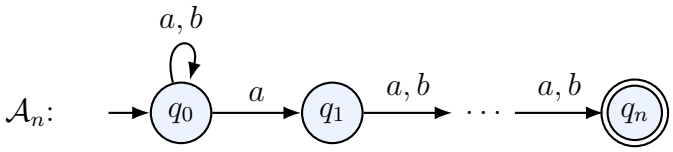
Das folgende Bild fasst die Situation bzgl. der relativen Größe von Automaten und regulären Ausdrücken zusammen, wobei ein durchgezogener Pfeil bedeutet, dass ein exponentielles Aufblähen im allgemeinen (beweisbar) unvermeidbar ist und ein gestrichelter Pfeil, dass wir bereits eine polynomielle Übersetzung kennengelernt haben:



Betrachten wir zunächst genauer den Zusammenhang zwischen NEAs und DEAs. Von DEA zu NEA braucht es keine Übersetzung, denn jeder DEA ist ein NEA. Für die andere Richtung haben wir die Potenzmengenkonstruktion verwendet, die exponentiell aufbläht. Wir wollen nun beweisen, dass das unvermeidbar ist. Dazu betrachten wir die Famile von Sprachen $(L_n)_{n\geq 1}$ mit

$$L_n = \{w \in \{a,b\}^* \mid \exists u, v \in \{a,b\}^* : w = uav \text{ und } |v| = n-1\}.$$

Die Sprache L_n enthält also alle Wörter, die mindestens Länge n haben und bei denen der n.-letzte Buchstabe ein a ist. Man sieht leicht, dass der folgende NEA $\mathcal{A}_n$ die Sprache L_n erkennt:



Es werden also nur $n+1$ Zustände benötigt. Bei DEAs ist die Situation dagegen völlig anders.

Lemma 5.8

Für alle $n \geq 1$ gilt: jeder DEA $\mathcal{A}_n$ mit $L(\mathcal{A}_n) = L_n$ hat mindestens 2^n Zustände.

Beweis. Sei $n \in \mathbb{N}$ und $\mathcal{A}_n = (Q, \{a,b\}, q_0, \delta, F)$ ein beliebiger DEA mit $L(\mathcal{A}_n) = L_n$. Angenommen $|Q| < 2^n$. Es existieren 2^n Wörter der Länge n über dem Alphabet $\{a,b\}$. Da $2^n > |Q|$ müssen zwei voneinander verschiedene Wörter $u, v \in \{a,b\}^n$ existieren, für die $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$ gilt.

Da $u = u_1 \cdots u_n$ und $v = v_1 \cdots v_n$ verschieden sind, existiert mindestens eine Stelle $j \in \{1, \ldots, n\}$ mit $u_j \neq v_j$. Wir betrachten die Worte $u' = ua^{j-1}$ und $v' = va^{j-1}$. Zunächst beobachten wir

$$\hat{\delta}(q_0, u') = \hat{\delta}(\hat{\delta}(q_0, u), a^{j-1}) = \hat{\delta}(\hat{\delta}(q_0, v), a^{j-1}) = \hat{\delta}(q_0, v') \ .$$

Also gilt $u' \in L(\mathcal{A}_n)$ genau dann wenn $v' \in L(\mathcal{A}_n)$.

Doch wie sieht es mit dem n.-letzten Buchstaben von u' und v' aus? Da $|u| = |v| = n$ und $|a^{j-1}| = j-1$ ist in beiden Wörtern der Index des n.-letzten Buchstaben $n + (j-1) - (n-1) = j$. Wir wissen jedoch $u_j \neq v_j$, also gilt entweder $u' \in L_n$, oder $v' \in L_n$, aber nicht beides. Da aber $u' \in L(\mathcal{A}_n)$ genau dann wenn $v' \in L(\mathcal{A}_n)$, folgt $L(\mathcal{A}_n) \neq L_n$. Dies ist ein Widerspruch und beendet unseren Beweis. ❑

Bei der Übersetzung von DEAs in NEAs lässt sich also ein exponentielles Aufblähen nicht vermeiden. Insgesamt können wir somit sagen: NEAs sind kompakter als DEAs.

Lemma 5.8 hilft uns auch beim Vergleich von DEA und regulären Ausdrücken: Der reguläre Ausdruck

$$r_n = (a+b)^* \cdot a \cdot \underbrace{(a+b) \cdot \ldots \cdot (a+b)}_{(n-1)-\text{mal}}$$

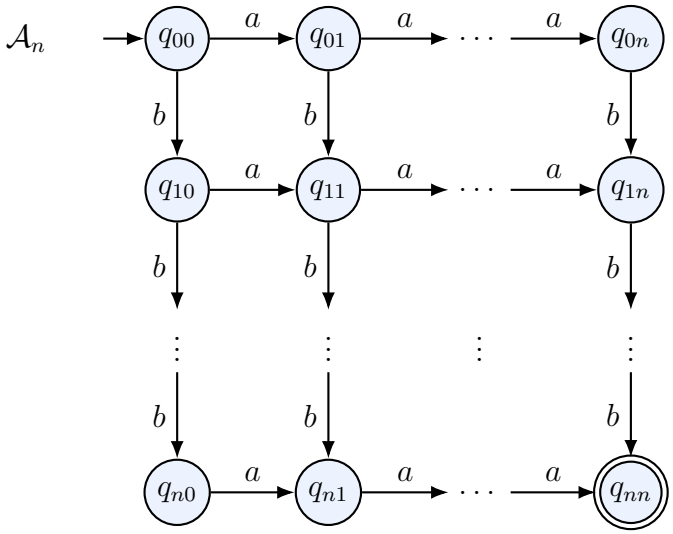
definiert ebenfalls die Sprache L_n. Die Länge von r_n ist $8 + 6 \cdot (n-1) \in \mathcal{O}(n)$. Da dies ein lineares Wachstum ist, die Größe der DEAs aber exponentiell wächst (Lemma 5.8), lässt sich ein exponentielles Aufblähen bei der Übersetzung von regulären Ausdrücken in DEA nicht vermeiden.

Bedeutet dies, dass reguläre Ausdrücke stets kompakter sind als DEAs? Das tut es nicht. Denn wie wir im Folgenden sehen, lässt sich auch bei der Übersetzung von DEA in RegEx ein exponentielles Aufblähen nicht vermeiden–nur für eine andere Familie von Sprachen.

Wir betrachten die Sprachfamilie $(L_n)_{n\geq 1}$ mit

$$L_n = \{w \in \{a,b\}^* \mid |w|_a = |w|_b = n\} \ .$$

Der folgende DEA erkennt die Sprache L_n:



Offensichtlich ist die Anzahl der Zustände $(n+1)^2 \in \mathcal{O}(n^2)$. Folgendes Lemma sagt uns, dass reguläre Ausdrücke zum Ausdrücken dieser Sprachen exponentiell mit n wachsen. Wir geben es hier ohne Beweis an, denn der ist nicht ganz einfach (Argumentationen über reguläre Ausdrücke können sehr komplex werden).

Lemma 5.9

Für alle $n \geq 1$ gilt und jeden regulären Ausdruck r_n mit $L(r_n) = L_n$ gilt: r hat Länge $2^{\Omega(n)}$.

Jede Folge von Ausrücken, die die Sprachfamilie $(L_n)_{n\geq 1}$ definiert, wächst also mindestens exponentiell.

Wir betrachten nun noch kurz die relative Größe von regulären Ausdrücken und NEAs. Da sich bei der Übersetzung von DEAs in reguläre Ausdrücke ein exponentielles Aufblähen nicht vermeiden lässt, gilt das offensichtlich ebenso für NEAs. Umgekehrt vergrößert die Übersetzung von regulären Ausdrücken in NEAs in Satz aber nur polynomiell. Genauer gesagt wird jeder Ausdruck r in einen Automat mit höchstens $2|r|$ Zuständen übersetzt. Dies zeigt man leicht unter Verwendung folgender Beobachtung: im induktiven Schritt werden Automaten für Vereinigung, Konkatenation und Kleene-Stern konstruiert, die lediglich höchstens einen einzigen Zustand zu den bestehenden Zuständen hinzugefügen. (zudem kann jeder atomare reguläre durch einen NEA mit maximal zwei Zuständen dargestellt werden).

NEAs sind also sowohl kompakter als DEAs als auch als reguläre Ausdrücke, die wiederum bezüglich ihrer relativen Größe unvergleichbar sind.

6. Minimale DEAs und die Nerode-Rechtskongruenz

6.1. Minimale DEAs

Wir werden im Folgenden ein Verfahren angeben, welches zu einem gegebenen DEA einen äquivalenten DEA mit minimaler Zustandszahl konstruiert. Das Verfahren besteht aus 2 Schritten:

1. Schritt: Eliminieren unerreichbarer Zustände

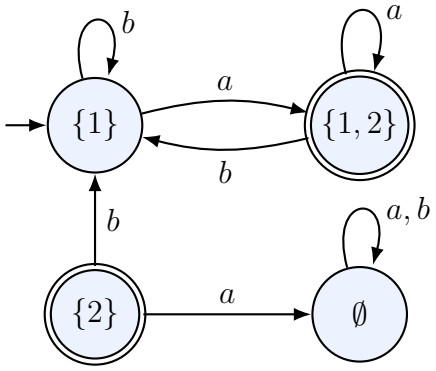
Definition 6.1 (Erreichbarkeit eines Zustandes)
Ein Zustand q des DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ heißt *erreichbar*, falls es ein Wort $w \in \Sigma^*$ gibt mit $\hat{\delta}(q_0, w) = q$. Sonst heißt q *unerreichbar*.

Da für die erkannte Sprache nur Zustände wichtig sind, welche von q_0 erreicht werden, erhält man durch Weglassen unerreichbarer Zustände einen äquivalenten Automaten:

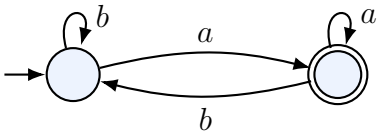
$\mathcal{A}_0 = (Q_0, \Sigma, q_0, \delta_0, F_0)$ mit

- $Q_0 = \{q \in Q \mid q \text{ ist erreichbar}\}$
- $\delta_0 = \delta|_{Q_0 \times \Sigma}$ (also: δ_0 ist wie δ, aber eingeschränkt auf die Zustände in Q_0)
- $F_0 = F \cap Q_0$

Beispiel 6.2
Betrachte als Resultat der Potenzmengenkonstruktion den Automaten $\mathcal{A}'$ aus Beispiel 1.13:



Die Zustände $\{2\}$ und $\emptyset$ sind nicht erreichbar. Durch Weglassen dieser Zustände erhält man den DEA $\mathcal{A}'_0$:

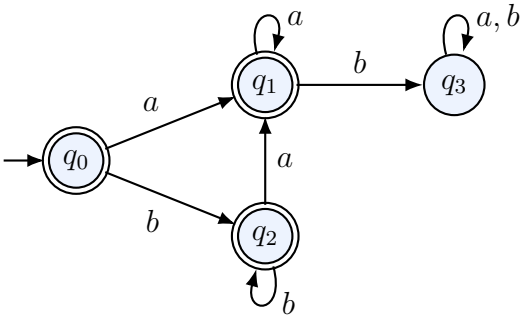


2. Schritt: Zusammenfassen äquivalenter Zustände

Ein DEA ohne unerreichbare Zustände muss noch nicht minimal sein, da er noch verschiedene Zustände enthalten kann, die sich „gleich“ verhalten in Bezug auf die erkannte Sprache.

Beispiel 6.3

Im folgenden DEA $\mathcal{A}$ mit $L(\mathcal{A}) = b^*a^*$ sind alle Zustände erreichbar. Er erkennt dieselbe Sprache wie der DEA aus Beispiel 1.7, hat aber einen Zustand mehr. Dies kommt daher, dass q_0 und q_2 äquivalent sind.



Im Allgemeinen definieren wir die Äquivalenz von Zuständen wie folgt.

Definition 6.4 (Äquivalenz von Zuständen)

Es sei $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ ein DEA. Für $q \in Q$ sei $\mathcal{A}_q = (Q, \Sigma, q, \delta, F)$. Zwei Zustände $q, q' \in Q$ heißen *$\mathcal{A}$-äquivalent* ($q \sim_{\mathcal{A}} q'$) gdw. $L(\mathcal{A}_q) = L(\mathcal{A}_{q'})$.

In Beispiel 6.3 gilt $q_0 \sim_{\mathcal{A}} q_2$, aber z. B. nicht $q_0 \sim_{\mathcal{A}} q_1$, da $b \in L(\mathcal{A}_{q_0}) \setminus L(\mathcal{A}_{q_1})$.

Um äquivalente Zustände auf mathematisch elegante Weise zusammenzufassen, nutzen wir aus, dass es sich bei der Relation $\sim_{\mathcal{A}}$ um eine Äquivalenzrelation handelt. Diese erfüllt zusätzlich einige weitere angenehme Eigenschaften.

Lemma 6.5

1) $\sim_{\mathcal{A}}$ *ist eine Äquivalenzrelation auf* Q*, d. h. reflexiv, transitiv und symmetrisch.*

2) $\sim_{\mathcal{A}}$ *ist verträglich mit der Übergangsfunktion, d. h.*

$$q \sim_{\mathcal{A}} q' \Rightarrow \forall a \in \Sigma : \delta(q, a) \sim_{\mathcal{A}} \delta(q', a)$$

3) $\sim_{\mathcal{A}}$ *kann in Polynomialzeit berechnet werden.*

Beweis.

1) ist klar, da „=“ reflexiv, transitiv und symmetrisch ist.

2) lässt sich wie folgt herleiten:

$$\begin{aligned}
q \sim_{\mathcal{A}} q' &\Rightarrow L(\mathcal{A}_q) = L(\mathcal{A}_{q'}) \\
&\Rightarrow \forall w \in \Sigma^* : \hat{\delta}(q, w) \in F \Leftrightarrow \hat{\delta}(q', w) \in F \\
&\Rightarrow \forall a \in \Sigma\ \forall v \in \Sigma^* : \hat{\delta}(q, av) \in F \Leftrightarrow \hat{\delta}(q', av) \in F \\
&\Rightarrow \forall a \in \Sigma\ \forall v \in \Sigma^* : \hat{\delta}(\delta(q, a), v) \in F \Leftrightarrow \hat{\delta}(\delta(q', a), v) \in F \\
&\Rightarrow \forall a \in \Sigma : L(\mathcal{A}_{\delta(q,a)}) = L(\mathcal{A}_{\delta(q',a)}) \\
&\Rightarrow \forall a \in \Sigma : \delta(q, a) \sim_{\mathcal{A}} \delta(q', a)
\end{aligned}$$

3) folgt unmittelbar daraus, dass das Äquivalenzproblem für DEAs in Polynomialzeit entscheidbar ist. ❑

Die $\sim_{\mathcal{A}}$-Äquivalenzklasse eines Zustands $q \in Q$ bezeichnen wir von nun an mit

$$[q]_{\mathcal{A}} := \{q' \in Q \mid q \sim_{\mathcal{A}} q'\}.$$

Auch wenn wir mit Punkt 3) des Lemma 6.5 bereits wissen, dass die Relation $\sim_{\mathcal{A}}$ berechenbar ist, geben wir hier noch eine direktere Methode an. Wir definieren eine Folge von Relationen $\sim_0, \sim_1, \sim_2, \cdots$:

- $q \sim_0 q'$ gdw. $q \in F \Leftrightarrow q' \in F$
- $q \sim_{k+1} q'$ gdw. $q \sim_k q'$ und $\forall a \in \Sigma : \delta(q, a) \sim_k \delta(q', a)$

Diese sind (Über-)Approximationen von $\sim_{\mathcal{A}}$ im folgenden Sinn.

Behauptung.

Für alle $k \geq 0$ gilt: $q \sim_k q'$ gdw. für alle $w \in \Sigma^*$ mit $|w| \leq k$: $w \in L(\mathcal{A}_q) \Leftrightarrow w \in L(\mathcal{A}_{q'})$.

Beweis.

Per Induktion über k:

Induktionsanfang: Nach Def. von $\sim_0$ gilt $q \sim_0 q'$ gdw. $\varepsilon \in L(\mathcal{A}_q) \Leftrightarrow \varepsilon \in L(\mathcal{A}_{q'})$.

Induktionsschritt:

$$\begin{aligned}
q \sim_{k+1} q' \ \text{gdw.}\ & q \sim_k q' \text{ und } \forall a \in \Sigma : \delta(q, a) \sim_k \delta(q', a) \\
\text{gdw.}\ & \forall w \in \Sigma^* \text{ mit } |w| \leq k : w \in L(\mathcal{A}_q) \Leftrightarrow w \in L(\mathcal{A}_{q'}) \text{ und} \\
& \forall a \in \Sigma : \forall w \in \Sigma^* \text{ mit } |w| \leq k : w \in L(\mathcal{A}_{\delta(q,a)}) \Leftrightarrow w \in L(\mathcal{A}_{\delta(q',a)}) \\
\text{gdw.}\ & \forall w \in \Sigma^* \text{ mit } |w| \leq k+1 : w \in L(\mathcal{A}_q) \Leftrightarrow w \in L(\mathcal{A}_{q'})
\end{aligned}$$

❑

Offensichtlich gilt $Q \times Q \supseteq \sim_0 \supseteq \sim_1 \supseteq \sim_2 \supseteq \ldots$. Da Q endlich ist, gibt es ein $k \geq 0$ mit $\sim_k = \sim_{k+1}$. Wir zeigen, dass $\sim_k$ die gewünschte Relation $\sim_{\mathcal{A}}$ ist. Nach obiger Behauptung

und Definition von $\sim_{\mathcal{A}}$ gilt offensichtlich $\sim_{\mathcal{A}} \subseteq \sim_k$. Um $\sim_k \subseteq \sim_{\mathcal{A}}$ zu zeigen, nehmen wir das Gegenteil $\sim_k \not\subseteq \sim_{\mathcal{A}}$ an. Wähle q, q' mit $q \sim_k q'$ und $q \not\sim_{\mathcal{A}} q'$. Es gibt also ein $w \in \Sigma^*$ mit $w \in L(\mathcal{A}_q)$ und $w \notin L(\mathcal{A}_{q'})$. Mit obiger Behauptung folgt $q \not\sim_n q'$ für $n = |w|$. Da $\sim_k \subseteq \sim_i$ für all $i \geq 0$ folgt $q \not\sim_k q'$, ein Widerspruch.

Beispiel 6.3 (Fortsetzung)

Für den Automaten aus Beispiel 6.3 gilt:

- $\sim_0$ hat die Klassen $F = \{q_0, q_1, q_2\}$ und $Q \setminus F = \{q_3\}$.
- $\sim_1$ hat die Klassen $\{q_1\}, \{q_0, q_2\}, \{q_3\}$.
  Zum Beispiel ist $\delta(q_0, b) = \delta(q_2, b) \in F$ und $\delta(q_1, b) \notin F$.
- $\sim_2 = \sim_1 = \sim_{\mathcal{A}}$.

Man kann diese Vorgehensweise beim Berechnen von $\sim_{\mathcal{A}}$ wie folgt als Algorithmus in Pseudocode formulieren; siehe Algorithmus 1.

Algorithmus 1 : Berechnung von $\sim_{\mathcal{A}}$

Algorithmus `equiv(`$\mathcal{A}$`):`
- **input :** DEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$
- **output :** Äquivalenzrelation $\sim_{\mathcal{A}} \subseteq Q \times Q$
- $k := 0$
- $\sim_0$ ist die Äquivalenzrelation mit den Äquivalenzklassen F und $Q \setminus F$
- **repeat**
  - Berechne $\sim_{k+1}$ aus $\sim_k$ wie oben definiert
  - $k := k + 1$
- **until** $\sim_k = \sim_{k-1}$
- **return** $\sim_{k+1}$

In der nachfolgenden Konstruktion werden äquivalente Zustände zusammengefasst, indem die Äquivalenzklasse $[q]_{\mathcal{A}}$ selbst zu den Zuständen gemacht werden. Jede Klasse verhält sich dann genau wie die in ihr enthaltenen Zustände im ursprünglichen Automaten.

Definition 6.6 (Quotientenautomat)

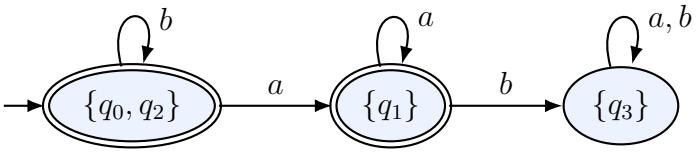
Der *Quotientenautomat* $\widetilde{\mathcal{A}} = (\widetilde{Q}, \Sigma, [q_0]_{\mathcal{A}}, \widetilde{\delta}, \widetilde{F})$ *zu* $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ ist definiert durch:

- $\widetilde{Q} := \{[q]_{\mathcal{A}} \mid q \in Q\}$
- $\widetilde{\delta}([q]_{\mathcal{A}}, a) := [\delta(q, a)]_{\mathcal{A}}$ (repräsentantenunabhängig wegen Lemma 6.5)
- $\widetilde{F} := \{[q]_{\mathcal{A}} \mid q \in F\}$ (repräsentantenunabhängig wegen Definition von $\sim_{\mathcal{A}}$ 6.4)

Nach Lemma 6.5 kann der Quotientenautomat in polynomieller Zeit konstruiert werden.

Beispiel 6.3 (Fortsetzung)

Für den Automaten aus Beispiel 6.3 ergibt sich der folgende Quotientenautomat:



Wir beweisen nun, dass der Quotientenautomat äquivalent zum ursprünglichen DEA ist. Dazu verwenden wir Punkt 1 des folgendes Lemma, der intuitiv sagt, dass sich die kanonische Fortsetzung $\widehat{\widetilde{\delta}}$ der Übergangsfunktion $\widetilde{\delta}$ von $\widetilde{\mathcal{A}}$ entsprechend der Definition von $\widetilde{\delta}$ verhält.

Lemma 6.7

1. *Für alle $q \in Q$ und $w \in \Sigma^*$ gilt: $\widehat{\widetilde{\delta}}([q_0]_{\mathcal{A}}, w) = [\hat{\delta}(q_0, w)]_{\mathcal{A}}$.*
2. *$\widetilde{\mathcal{A}}$ ist äquivalent zu $\mathcal{A}$.*

Beweis. Punkt 1 beweist man leicht per Induktion über $|w|$, die Details lassen wir als Übung. Nun gilt:

$$
\begin{array}{lll}
w \in L(\mathcal{A}) & \text{gdw.} & \hat{\delta}(q_0, w) \in F \\
& \text{gdw.} & [\hat{\delta}(q_0, w)]_{\mathcal{A}} \in \widetilde{F} \quad \text{(Def. } \widetilde{F}\text{)} \\
& \text{gdw.} & \widehat{\widetilde{\delta}}([q_0]_{\mathcal{A}}, w) \in \widetilde{F} \quad \text{(Punkt 1)} \\
& \text{gdw.} & w \in L(\widetilde{\mathcal{A}})
\end{array}
$$

❑

Die folgende Definition fasst die beiden Minimierungs-Schritte zusammen:

Definition 6.8 (reduzierter Automat zu einem DEA)

Für einen DEA $\mathcal{A}$ bezeichnet $\mathcal{A}_{\mathsf{red}}$ den *reduzierten Automaten*, den man aus $\mathcal{A}$ durch Eliminieren unerreichbarer Zustände und nachfolgendes Bilden des Quotientenautomaten erhält.

Wir wollen zeigen, dass der reduzierte Automat nicht weiter vereinfacht werden kann: $\mathcal{A}_{\mathsf{red}}$ ist der kleinste DEA (bezüglich der Zustandszahl), der $L(\mathcal{A})$ erkennt. Um den Beweis führen zu können, benötigen wir als Hilfsmittel eine Äquivalenzrelation auf Wörtern, die Nerode-Rechtskongruenz.

6.2. Die Nerode-Rechtskongruenz

Die Nerode-Rechtskongruenz ist auch unabhängig von reduzierten Automaten von Interesse und hat neben dem bereits erwähnten Beweis weitere interessante Anwendungen, von denen wir zwei kurz darstellen werden: sie liefert eine von Automaten unabhängige Charakterisierung der erkennbaren Sprachen und stellt ein weiteres Mittel zur Verfügung, um von einer Sprache nachzuweisen, dass sie *nicht* erkennbar ist.

Im Gegensatz zur Relation $\sim_{\mathcal{A}}$ auf den Zuständen eines Automaten handelt es sich hier um eine Relation *auf Wörtern*.

Definition 6.9 (Nerode-Rechtskongruenz)
Es sei $L \subseteq \Sigma^*$ eine beliebige Sprache. Für $u, v \in \Sigma^*$ definieren wir:
$u \simeq_L v$ gdw. $\forall w \in \Sigma^* : uw \in L \Leftrightarrow vw \in L$.

Man beachte, dass das Wort w in Definition 6.9 auch gleich ε sein kann. Darum folgt aus $u \simeq_L v$, dass $u \in L \Leftrightarrow v \in L$.

Beispiel 6.10
Wir betrachten die Sprache $L = b^*a^*$ (vgl. Beispiele 1.7, 6.3).

- Es gilt:
  $\varepsilon \simeq_L b$: $\forall w : \varepsilon w \in L$ gdw. $w \in L$
  gdw. $w \in b^*a^*$
  gdw. $bw \in b^*a^*$
  gdw. $bw \in L$
- $\varepsilon \not\simeq_L a$: $\varepsilon b \in L$, aber $a \cdot b \notin L$

Wir zeigen nun, dass es sich bei $\simeq_L$ wirklich um eine Äquivalenzrelation handelt. In der Tat ist $\simeq_L$ sogar eine Kongruenzrelation bezüglich Konkatenation von beliebigen Wörtern „von rechts“. Im Folgenden bezeichnet der *Index* einer Äquivalenzrelation die Anzahl ihrer Klassen.

Lemma 6.11 (Eigenschaften von $\simeq_L$)

1) $\simeq_L$ ist eine Äquivalenzrelation.

2) $\simeq_L$ ist Rechtskongruenz, d.h. zusätzlich zu 1) gilt: $u \simeq_L v \Rightarrow \forall w \in \Sigma^* : uw \simeq_L vw$.

3) L ist Vereinigung von $\simeq_L$-Klassen:

$$L = \bigcup_{u \in L} [u]_L$$

wobei $[u]_L := \{v \mid u \simeq_L v\}$.

4) Ist $L = L(\mathcal{A})$ für einen DEA $\mathcal{A}$, so ist die Anzahl der Zustände von $\mathcal{A}$ größer oder gleich dem Index von $\simeq$.

Beweis.

1) folgt aus der Definition von $\simeq_L$, da „$\Leftrightarrow$“ reflexiv, transitiv und symmetrisch ist.

2) Damit $uw \simeq_L vw$ gilt, muss für alle $w' \in \Sigma^*$ gelten:

$$uww' \in L \Leftrightarrow vww' \in L \qquad (*)$$

Wegen $ww' \in \Sigma^*$ folgt $(*)$ aus $u \simeq_L v$.

3) Dafür zeigen wir, dass für alle $v \in \Sigma^*$ gilt:

$$v \in L \qquad \text{gdw.} \qquad v \in \bigcup_{u \in L} [u]_L$$

„$\Rightarrow$“: Wenn $v \in L$, dann ist $[v]_L$ in der Vereinigung rechts; zudem gilt $v \in [v]_L$.

„$\Leftarrow$“: Sei $v \in [u]_L$ für ein $u \in L$.
Wegen $\varepsilon \in \Sigma^*$ folgt aus $u = u \cdot \varepsilon \in L$ und $v \simeq_L u$ auch $v = v \cdot \varepsilon \in L$.

4) Es sei $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ ein DEA mit $L = L(\mathcal{A})$.

Wir zeigen: $\hat{\delta}(q_0, u) = \hat{\delta}(q_0, v)$ impliziert $u \simeq_L v$:

$$\begin{aligned} \forall w : uw \in L \quad & \text{gdw.} \quad \hat{\delta}(q_0, uw) \in F \\ & \text{gdw.} \quad \hat{\delta}(\hat{\delta}(q_0, u), w) \in F \\ & \text{gdw.} \quad \hat{\delta}(\hat{\delta}(q_0, v), w) \in F \\ & \text{gdw.} \quad \hat{\delta}(q_0, vw) \in F \\ & \text{gdw.} \quad vw \in L \end{aligned}$$

Also folgt aus $u \not\simeq_L v$, dass $\hat{\delta}(q_0, u) \neq \hat{\delta}(q_0, v)$. Damit gibt es mindestens so viele Zustände wie $\simeq$-Klassen (Schubfachprinzip). ❑

Beispiel 6.10 (Fortsetzung)

$\simeq_L$ hat drei Klassen:

- $[\varepsilon]_L = b^*$
- $[a]_L = b^*aa^*$
- $[ab]_L = (a+b)^*ab(a+b)^*$

Es ist $L = [\varepsilon]_L \cup [a]_L$ (vgl. Lemma 6.11, Punkt 3) und $\Sigma^* \setminus L = [ab]_L$.

Eine interessante Eigenschaft von $\simeq_L$ ist, dass die Äquivalenzklassen zur Definition eines kanonischen Automaten $\mathcal{A}_L$ verwendet werden können, der L erkennt. Dieser Automat ergibt sich *direkt* und *auf eindeutige Weise* aus der Sprache L (im Gegensatz

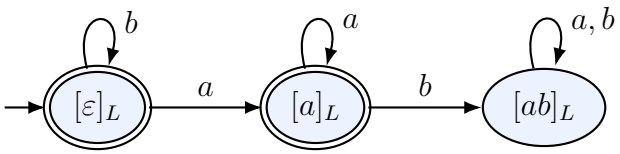
zum reduzierten Automaten, für dessen Konstruktion man bereits einen Automaten für die betrachtete Sprache haben muss). Damit wir einen endlichen Automaten erhalten, dürfen wir die Konstruktion nur auf Sprachen L anwenden, für die $\simeq_L$ nur endlich viele Äquivalenzklassen hat.

Definition 6.12 (Kanonischer DEA $\mathcal{A}_L$ zu einer Sprache L)
Sei $L \subseteq \Sigma^*$ eine Sprache, so dass $\simeq_L$ endlichen Index hat. Der *kanonische DEA* $\mathcal{A}_L = (Q', \Sigma, q_0', \delta', F')$ zu L ist definiert durch:

- $Q' := \{[u]_L \mid u \in \Sigma^*\}$
- $q_0' := [\varepsilon]_L$
- $\delta'([u]_L, a) := [ua]_L$ (repräsentantenunabhängig wegen Lemma 6.11, Punkt 2)
- $F' := \{[u]_L \mid u \in L\}$ (repräsentantenunabhängig wegen Bemerkung nach Def. 6.9)

Man beachte, dass $\mathcal{A}_L$ mit Punkt 4 von Lemma 6.11 eine minimale Anzahl von Zuständen hat: es gibt keinen DEA, der $L(\mathcal{A}_L)$ erkennt und weniger Zustände hat.

Beispiel 6.10 (Fortsetzung)
Für die Sprache $L = b^*a^*$ ergibt sich damit folgender kanonischer Automat $\mathcal{A}_L$:



Lemma 6.13
Hat $\simeq_L$ endlichen Index, so ist $\mathcal{A}_L$ ein DEA mit $L = L(\mathcal{A}_L)$.

Beweis. Es gilt:

$$\begin{aligned} L(\mathcal{A}_L) &= \{u \mid \hat{\delta}'(q_0', u) \in F'\} \\ &= \{u \mid \hat{\delta}'([\varepsilon]_L, u) \in F'\} && \text{(Def. } q_0'\text{)} \\ &= \{u \mid [u]_L \in F'\} && \text{(wegen } \hat{\delta}'([\varepsilon]_L, u) = [u]_L\text{)} \\ &= \{u \mid u \in L\} && \text{(Def. } F'\text{)} \\ &= L \end{aligned}$$

❑

Das folgende Resultat ist eine interessante Anwendung der Nerode-Rechtskongruenz und des kanonischen Automaten. Es liefert eine Charakterisierung von erkennbaren Sprachen, die vollkommen unabhängig von endlichen Automaten ist.

Satz 6.14 (Satz von Myhill und Nerode)
Eine Sprache L ist erkennbar gdw. $\simeq_L$ endlichen Index hat.

Beweis.

„⇒“: Ergibt sich unmittelbar aus Lemma 6.11, 4).

„⇐“: Ergibt sich unmittelbar aus Lemma 6.13, da $\mathcal{A}_L$ DEA ist, der L erkennt.

❑

Der Satz von Myhill und Nerode liefert uns als Nebenprodukt eine weitere Methode, von einer Sprache zu beweisen, dass sie *nicht* erkennbar ist.

Beispiel 6.15 (nicht erkennbare Sprache)
Die Sprache $L = \{a^n b^n \mid n \geq 0\}$ ist nicht erkennbar, da für $n \neq m$ gilt: $a^n \not\simeq_L a^m$. In der Tat gilt $a^n b^n \in L$, aber $a^m b^n \notin L$. Daher hat $\simeq_L$ unendlichen Index.

Wir zeigen nun, dass der reduzierte DEA minimal ist.

Satz 6.16 (Minimalität des reduzierten DEA)
Sei $\mathcal{A}$ ein DEA. Dann hat jeder DEA, der $L(\mathcal{A})$ erkennt, mindestens so viele Zustände wie der reduzierte DEA $\mathcal{A}_{\mathsf{red}}$.

Beweis. Sei $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ und $\mathcal{A}_{\mathsf{red}} = (\widetilde{Q}, \Sigma, [q_0]_{\mathcal{A}}, \widetilde{\delta}, \widetilde{F})$. Wir definieren eine injektive Abbildung π, die jedem Zustand aus $\widetilde{Q}$ eine Äquivalenzklasse von $\simeq_L$ zuordnet. Es folgt, dass $\mathcal{A}_{\mathsf{red}}$ *höchstens* so viele Zustände hat, wie $\simeq_L$ Äquivalenzklassen (Schubfachprinzip), also ist er nach Punkt 4 von Lemma 6.11 minimal.

Sei $[q]_{\mathcal{A}} \in \widetilde{Q}$. Nach Definition von $\mathcal{A}_{\mathsf{red}}$ ist q in $\mathcal{A}$ von q_0 aus erreichbar mit einem Wort w_q. Setze $\pi([q]_{\mathcal{A}}) = [w_q]_L$.

Es bleibt zu zeigen, dass π injektiv ist. Seien $[q]_{\mathcal{A}}, [p]_{\mathcal{A}} \in \widetilde{Q}$ mit $[q]_{\mathcal{A}} \neq [p]_{\mathcal{A}}$. Dann gilt $q \not\sim_{\mathcal{A}} p$ und es gibt $w \in \Sigma^*$ so dass

$$\hat{\delta}(q, w) \in F \Leftrightarrow \hat{\delta}(p, w) \in F$$

nicht gilt. Nach Wahl von w_p und w_q gilt dann aber auch

$$\hat{\delta}(q_0, w_q w) \in F \Leftrightarrow \hat{\delta}(q_0, w_p w) \in F$$

nicht und damit auch nicht

$$w_q w \in L \Leftrightarrow w_p w \in L$$

Es folgt $w_q \not\simeq_L w_p$, also $\pi([q]_{\mathcal{A}}) \neq \pi([p]_{\mathcal{A}})$ wie gewünscht. ❑

Es ist also sowohl der reduzierte Automat als auch der kanonische Automat von minimaler Größe. In der Tat ist der Zusammenhang zwischen beiden Automaten sogar noch viel enger: man kann zeigen, dass sie identisch bis auf Zustandsumbenennung sind. Dies wird durch den Begriff der Isomorphie beschrieben.

Definition 6.17 (Isomorphie von DEAs)
Zwei DEAs $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ und $\mathcal{A}' = (Q', \Sigma, q_0', \delta', F')$ sind *isomorph* (geschrieben $\mathcal{A} \simeq \mathcal{A}'$) gdw. es eine Bijektion $\pi : Q \to Q'$ gibt mit:

- $\pi(q_0) = q_0'$
- $\pi(F) = \{\pi(q) \mid q \in F\} = F'$, wobei $\pi(F) = \{\pi(q) \mid q \in F\}$
- $\pi(\delta(q, a)) = \delta'(\pi(q), a)$ für alle $q \in Q$, $a \in \Sigma$

Lemma 6.18
$\mathcal{A} \simeq \mathcal{A}' \;\Rightarrow\; L(\mathcal{A}) = L(\mathcal{A}')$

Beweis. Es sei $\pi : Q \to Q'$ der Isomorphismus. Durch Induktion über $|w|$ zeigt man leicht, dass $\pi(\hat{\delta}(q, w)) = \hat{\delta}'(\pi(q), w)$. Daher gilt:

$$\begin{aligned}
w \in L(\mathcal{A}) \quad & \text{gdw.} \quad \hat{\delta}(q_0, w) \in F \\
& \text{gdw.} \quad \pi(\hat{\delta}(q_0, w)) \in F' \quad (\text{wegen } \pi(F) = F') \\
& \text{gdw.} \quad \hat{\delta}'(\pi(q_0), w) \in F' \\
& \text{gdw.} \quad \hat{\delta}'(q_0', w) \in F' \quad (\text{wegen } q_0' = \pi(q_0)) \\
& \text{gdw.} \quad w \in L(\mathcal{A}')
\end{aligned}$$

❑

Wir können nun Minimalität und Eindeutigkeit des reduzierten Automaten zeigen.

Satz 6.19 (Isomorphie reduzierter und kanonischer DEA)
Es sei L eine erkennbare Sprache und $\mathcal{A}$ ein DEA mit $L(\mathcal{A}) = L$. Dann gilt: der reduzierte Automat $\mathcal{A}_{\mathsf{red}}$ ist isomorph zum kanonischen Automaten $\mathcal{A}_L$.

Beweis. Es sei $\mathcal{A}_{\mathsf{red}} = (Q, \Sigma, q_0, \delta, F)$ und $\mathcal{A}_L = (Q', \Sigma, q_0', \delta', F')$. Wir definieren eine Funktion $\pi : Q \to Q'$ und zeigen, dass sie ein Isomorphismus ist. Für jedes $q \in Q$ existiert (mindestens) ein $w_q \in \Sigma^*$ mit $\hat{\delta}(q_0, w_q) = q$, da in $\mathcal{A}_{\mathsf{red}}$ alle Zustände erreichbar sind. O. B. d. A. sei $w_{q_0} = \varepsilon$. Wir definieren $\pi(q) := [w_q]_L$.[2]

I) π ist injektiv:
Wir müssen zeigen, dass aus $p \neq q$ auch $[w_p]_L \neq [w_q]_L$ folgt.
Da $\mathcal{A}_{\mathsf{red}}$ reduziert ist, sind verschiedene Zustände nicht äquivalent. Es gibt also mindestens ein w, für das

$$\hat{\delta}(p, w) \in F \Leftrightarrow \hat{\delta}(q, w) \in F$$

[2] Aus Punkt 1 von Lemma 6.7 und der Definition von F folgt, dass diese Funktion π dieselbe ist wie die im Beweis von Satz 6.16 definierte Funktion π. Im Prinzip haben wir die Injektivität also bereits bewiesen.

nicht gilt. Das heißt aber, dass

$$\hat{\delta}(q_0, w_p w) \in F \Leftrightarrow \hat{\delta}(q_0, w_q w) \in F$$

nicht gilt und damit wiederum, dass $w_p w \in L \Leftrightarrow w_q w \in L$ *nicht* gilt. Also ist $w_p \not\equiv_L w_q$, d. h. $[w_p]_L \neq [w_q]_L$.

II) π ist surjektiv:
Folgt aus Injektivität und $|Q| \geq |Q'|$ (Punkt 4 Lemma 6.11).

III) $\pi(q_0) = q_0'$:
Da $w_{q_0} = \varepsilon$ und $q_0' = [\varepsilon]_L$.

IV) $\pi(F) = F'$:

$$\begin{array}{lll} q \in F & \text{gdw.} & \hat{\delta}(q_0, w_q) \in F \quad \text{(Wahl } w_q\text{)} \\ & \text{gdw.} & w_q \in L \\ & \text{gdw.} & [w_q]_L \in F' \quad \text{(Def. } F'\text{)} \\ & \text{gdw.} & \pi(q) \in F' \end{array}$$

V) $\pi(\delta(q, a)) = \delta'(\pi(q), a)$:
Als Abkürzung setze $p = \delta(q, a)$. Die Hilfsaussage

$$\hat{\delta}(q_0, w_p) = \hat{\delta}(q_0, w_q a) \qquad (*)$$

gilt wegen:

$$\begin{array}{lcll} \hat{\delta}(q_0, w_p) & = & p & \text{(Wahl } w_p\text{)} \\ & = & \delta(q, a) & \\ & = & \delta(\hat{\delta}(q_0, w_q), a) & \text{(Wahl } w_q\text{)} \\ & = & \hat{\delta}(q_0, w_q a) & \end{array}$$

Mittels $(*)$ zeigen wir nun:

$$\begin{array}{lcll} \pi(p) & = & [w_p]_L & \text{(Def. } \pi\text{)} \\ & = & [w_q a]_L & \text{(folgt aus } (*) \text{ und } L(\mathcal{A}_{\mathsf{red}}) = L\text{)} \\ & = & \delta'([w_q]_L, a) & \text{(Def. } \mathcal{A}_L\text{)} \\ & = & \delta'(\pi(q), a) & \text{(Def. } \pi\text{)} \end{array}$$

❑

Dieser Satz zeigt auch folgende interessante Eigenschaften:

- Der reduzierte Automat $\mathcal{A}_{\mathsf{red}}$ ist unabhängig vom ursprünglichen DEA $\mathcal{A}$:
  wenn $L(\mathcal{A}) = L(\mathcal{B}) = L$, dann gilt wegen $\mathcal{A}_{\mathsf{red}} \simeq \mathcal{A}_L \simeq \mathcal{B}_{\mathsf{red}}$ auch $\mathcal{A}_{\mathsf{red}} \simeq \mathcal{B}_{\mathsf{red}}$ (denn die Komposition zweier Isomorphismen ergibt wieder einen Isomorphismus).

- Für jede erkennbare Sprache L gibt es einen eindeutigen minimalen DEA:
  Wenn $L(\mathcal{A}) = L(\mathcal{B}) = L$ und $\mathcal{A}$ und $\mathcal{B}$ minimale Zustandszahl unter allen DEAs haben, die L erkennen, dann enthalten $\mathcal{A}$ und $\mathcal{B}$ weder unerreichbare noch äquivalente Zustände und der jeweilige reduzierte Automat ist identisch zum ursprünglichen Automaten. Damit gilt $\mathcal{A} = \mathcal{A}_{\mathsf{red}} \simeq \mathcal{A}_L \simeq \mathcal{B}_{\mathsf{red}} = \mathcal{B}$, also auch $\mathcal{A} \simeq \mathcal{B}$.

Im Prinzip liefert Satz 6.19 zudem eine Methode, um von zwei Automaten zu entscheiden, ob sie dieselbe Sprache akzeptieren:

Korollar 6.20

Es seien $\mathcal{A}$ und $\mathcal{A}'$ DEAs. Dann gilt: $L(\mathcal{A}) = L(\mathcal{A}')$ *gdw.* $\mathcal{A}_{\mathsf{red}} \simeq \mathcal{A}'_{\mathsf{red}}$.

Man kann die reduzierten Automaten wie beschrieben konstruieren. Für gegebene Automaten kann man feststellen, ob sie isomorph sind (teste alle Bijektionen). Da es exponentiell viele Kandidaten für eine Bijektion gibt, ist diese Methode nicht optimal.

Hat man NEAs anstelle von DEAs gegeben, so kann man diese zuerst deterministisch machen und dann das Korollar anwenden.

Schlussbemerkungen zu Teil I

Zum Abschluss von Teil I erwähnen wir einige hier aus Zeitgründen nicht behandelte Themenbereiche:

Andere Varianten von endlichen Automaten:

NEAs/DEAs mit Ausgabe (sogenannte *Transduktoren*) haben Übergänge $p \xrightarrow{a/v}_{\mathcal{A}} q$, wobei $v \in \Gamma^*$ ein Wort über einem Ausgabealphabet ist. Solche Automaten beschreiben Funktionen $\Sigma^* \to \Gamma^*$. *2-Wege-Automaten* können sich sowohl vorwärts als auch rückwärts auf der Eingabe bewegen. *Alternierende Automaten* generalisieren NEAs: zusätzlich zu den nichtdeterministischen Übergängen, die einer existentiellen Quantifizierung entsprechen, gibt es hier auch universell quantifizierte Übergänge.

Algebraische Theorie formaler Sprachen:

Jeder Sprache L wird ein Monoid M_L (syntaktisches Monoid) zugeordnet. Klassen von Sprachen entsprechen dann Klassen von Monoiden, z. B. L ist regulär gdw. M_L endlich ist. Dies ermöglicht einen sehr fruchtbaren algebraischen Zugang zur Automatentheorie.

Automaten auf unendlichen Wörtern:

Hier geht es um Automaten, die unendliche Wörter (unendliche Folgen von Symbolen) als Eingabe erhalten. Die Akzeptanz via Erreichen eines akzeptierenden Zustandes funktioniert hier natürlich nicht mehr und man studiert verschiedene Akzeptanzbedingungen wie z. B. Büchi-Akzeptanz und Rabin-Akzeptanz.

Baumautomaten:

Diese Automaten erhalten Bäume statt Wörtern als Eingabe. Eine streng lineare Abarbeitung wie bei Wörtern ist in diesem Fall natürlich nicht möglich. Man unterscheidet zwischen Top-Down- und Bottom-Up-Automaten.

II. Grammatiken, kontextfreie Sprachen und Kellerautomaten

Einführung

Der zweite Teil beschäftigt sich hauptsächlich mit der Klasse der kontextfreien Sprachen sowie mit Kellerautomaten, dem zu dieser Sprachfamilie passenden Automatenmodell. Die Klasse der kontextfreien Sprachen ist allgemeiner als die der regulären Sprachen, und dementsprechend können Kellerautomaten als eine Erweiterung von endlichen Automaten verstanden werden. Kontextfreie Sprachen spielen in der Informatik eine wichtige Rolle, da durch sie z. B. die Syntax von Programmiersprachen (zumindest in großen Teilen) beschreibbar ist.

Bevor wir uns im Detail den kontextfreien Sprachen zuwenden, führen wir Grammatiken als allgemeines Mittel zum Generieren von formalen Sprachen ein. Wir werden sehen, dass sich sowohl die regulären als auch die kontextfreien Sprachen und weitere, noch allgemeinere Sprachklassen mittels Grammatiken definieren lassen. Diese Sprachklassen sind angeordnet in der bekannten Chomsky-Hierarchie von formalen Sprachen.

7. Die Chomsky-Hierarchie

Grammatiken dienen dazu, Wörter zu erzeugen. Man hat dazu *Regeln*, die es erlauben, ein Wort durch ein anderes Wort zu ersetzen (aus ihm *abzuleiten*). Die *erzeugte Sprache* ist die Menge der Wörter, die ausgehend von einem *Startsymbol* durch wiederholtes Ersetzen erzeugt werden können.

Beispiel 7.1

Regeln:	$S \longrightarrow aSb$	(1)
	$S \longrightarrow \varepsilon$	(2)
Startsymbol:	S	

Eine mögliche Ableitung eines Wortes ist:

$S \xrightarrow{1} aSb \xrightarrow{1} aaSbb \xrightarrow{1} aaaSbbb \xrightarrow{2} aaabbb$

Das Symbol S ist hier ein Hilfssymbol (*nichtterminales Symbol*) und man ist nur an erzeugten Wörtern interessiert, die das Hilfssymbol nicht enthalten (*Terminalwörter*). Man sieht leicht, dass dies in diesem Fall genau die Wörter $a^n b^n$ mit $n \geq 0$ sind.

Definition 7.2 (Grammatik)

Eine *Grammatik* ist von der Form $G = (N, \Sigma, P, S)$, wobei

- N und Σ endliche, disjunkte Alphabete von *Nichtterminalsymbolen* bzw. *Terminalsymbolen* sind,
- $S \in N$ das *Startsymbol* ist,
- $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ eine endliche Menge von Ersetzungsregeln (*Produktionen*) ist.

Der besseren Lesbarkeit halber schreiben wir Produktionen $(u, v) \in P$ gewöhnlich als $u \longrightarrow v$.

Man beachte, dass die rechte Seite von Produktionen aus dem leeren Wort bestehen darf, die linke jedoch nicht. Außerdem sei darauf hingewiesen, dass sowohl Nichtterminalsymbole als auch Terminalsymbole durch eine Ersetzungsregel ersetzt werden dürfen.

Beispiel 7.3

Folgendes Tupel ist eine Grammatik: $G = (N, \Sigma, P, S)$ mit

- $N = \{S, B\}$
- $\Sigma = \{a, b, c\}$
- $P = \{S \longrightarrow aSBc,$
  $S \longrightarrow abc,$
  $cB \longrightarrow Bc,$
  $bB \longrightarrow bb\}$

Im Folgenden schreiben wir meistens Elemente von N mit Großbuchstaben und Elemente von Σ mit Kleinbuchstaben.

Wir definieren nun, was es heißt, dass man ein Wort durch Anwenden der Regeln aus einem anderen ableiten kann.

Definition 7.4 (durch eine Grammatik erzeugte Sprache)
Sei $G = (N, \Sigma, P, S)$ eine Grammatik und x, y Wörter aus $(N \cup \Sigma)^*$.

1) *y aus x direkt ableitbar:*
   $x \vdash_G y$ gdw. $x = x_1 u x_2$ und $y = x_1 v x_2$ mit $u \longrightarrow v \in P$ und $x_1, x_2 \in (N \cup \Sigma)^*$
2) *y aus x in n Schritten ableitbar:*
   $x \vdash_G^n y$ gdw. $x \vdash_G x_1 \vdash_G \cdots \vdash_G x_{n-1} \vdash_G y$ für $x_1, \ldots, x_{n-1} \in (N \cup \Sigma)^*$
3) *y aus x ableitbar:*
   $x \vdash_G^* y$ gdw. $x \vdash_G^n y$ für ein $n \geq 0$
4) Die durch G *erzeugte Sprache* ist
   $L(G) := \{w \in \Sigma^* \mid S \vdash_G^* w\}$.

Man ist also bei der erzeugten Sprache nur an den in G aus S ableitbaren Terminalwörtern interessiert.

Beispiel 7.3 (Fortsetzung)
Zwei Beispielableitungen in der Grammatik aus Beispiel 7.3:

$$S \vdash_G abc, \qquad \text{d. h. } abc \in L(G)$$

$$\begin{aligned} S &\vdash_G aSBc \\ &\vdash_G aaSBcBc \\ &\vdash_G aaabcBcBc \\ &\vdash_G aaabBccBc \\ &\vdash_G^2 aaabBBccc \\ &\vdash_G^2 aaabbbccc, \qquad \text{d. h. } aaabbbccc \in L(G) \end{aligned}$$

Die erzeugte Sprache ist $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

Beweis.

„$\supseteq$“: Für $n = 1$ ist $abc \in L(G)$ klar. Für $n > 1$ sieht man leicht:

$$S \vdash_G^{n-1} a^{n-1} S (Bc)^{n-1} \vdash_G a^n bc (Bc)^{n-1} \vdash_G^* a^n b B^{n-1} c^n \vdash_G^{n-1} a^n b^n c^n$$

„$\subseteq$“: Sei $S \vdash_G^* w$ mit $w \in \Sigma^*$. Offenbar wird die Regel $S \longrightarrow abc$ in der Ableitung von w genau einmal angewendet. Vor dieser Anwendung können nur die Regeln

$S \longrightarrow aSBc$ und $cB \longrightarrow Bc$ angewendet werden, da noch keine b's generiert wurden. Die Anwendung von $S \longrightarrow abc$ hat damit die Form

$$a^n Su \vdash a^{n+1}bcu \text{ mit } u \in \{c, B\}^* \text{ und } |u|_B = |u|_c = n$$

(formaler Beweis per Induktion über die Anzahl der Regelanwendungen). Nun sind nur noch $cB \longrightarrow Bc$ und $bB \longrightarrow bb$ anwendbar. Alle dabei entstehenden Wörter haben die folgende Form (formaler Beweis per Induktion über die Anzahl der Regelanwendungen):

$$a^{n+1}b^{k+1}v \text{ mit } v \in \{c, B\}^*, \quad |v|_B = n - k \text{ und } |v|_c = n + 1.$$

Jedes Terminalwort dieser Form erfüllt $|v|_B = 0$, also $n = k$, und damit hat das Wort die Form $a^n b^n c^n$, $n \geq 1$.

❑

Beispiel 7.5

Betrachte die Grammatik $G = (N, \Sigma, P, S)$ mit

- $N = \{S, B\}$
- $\Sigma = \{a, b\}$
- $P = \{S \longrightarrow aS,$
  $S \longrightarrow bS,$
  $S \longrightarrow abB,$
  $B \longrightarrow aB,$
  $B \longrightarrow bB,$
  $B \longrightarrow \varepsilon\}$

Die erzeugte Sprache ist $L(G) = \Sigma^* \cdot \{a\} \cdot \{b\} \cdot \Sigma^*$.

Die Grammatiken aus Beispiel 7.5, 7.3 und 7.1 gehören zu unterschiedlichen Sprachklassen, die alle durch Grammatiken erzeugt werden können. Sie sind angeordnet in der Chomsky-Hierarchie.

Definition 7.6 (Chomsky-Hierarchie, Typen von Grammatiken)

Es sei $G = (N, \Sigma, P, S)$ eine Grammatik.

- Jede Grammatik G ist Grammatik vom **Typ 0.**
- G ist Grammatik vom **Typ 1 (monoton),** falls alle Regeln *nicht verkürzend* sind, also die Form $w \longrightarrow u$ haben wobei $w, u \in (\Sigma \cup N)^+$ und $|u| \geq |w|$.

  Ausnahme: Die Regel $S \longrightarrow \varepsilon$ ist erlaubt, wenn S in keiner Produktion auf der rechten Seite vorkommt.
- G ist Grammatik vom **Typ 2 (kontextfrei),** falls alle Regeln die Form $A \longrightarrow w$ haben mit $A \in N, w \in (\Sigma \cup N)^*$.
- G ist Grammatik vom **Typ 3 (rechtslinear),** falls alle Regeln die Form $A \longrightarrow uB$ oder $A \longrightarrow u$ haben mit $A, B \in N$, $u \in \Sigma^*$.

Die *kontextfreien* Sprachen heißen deshalb so, weil die linke Seite jeder Produktion nur aus einem Nichtterminalsymbol A besteht, das unabhängig vom *Kontext im Wort* (also dem Teilwort links von A und dem Teilwort rechts von A) ersetzt wird. Bei monotonen Grammatiken sind hingegen Regeln

$$u_1 A u_2 \longrightarrow u_1 w u_2$$

erlaubt, wenn $|w| \geq 1$. Hier ist die Ersetzung von A durch w abhängig davon, dass der richtige Kontext (u_1 links und u_2 rechts, beides aus $(N \cup \Sigma)^*$) im Wort vorhanden ist. Man kann sogar zeigen, dass es keine Beschränkung der Allgemeinheit ist, monotone Grammatiken ausschließlich durch Regeln der obigen Form zu definieren. Die durch monotone Grammatiken erzeugten Sprachen nennt man daher auch *kontextsensitive Sprachen*.

Eine sehr wichtige Eigenschaft von monotonen Grammatiken ist, dass die Anwendung einer Produktion das Wort nicht verkürzen kann. Vor diesem Hintergrund ist auch die Ausnahme $S \longrightarrow \varepsilon$ zu verstehen: sie dient dazu, das leere Wort generieren zu können, was ohne Verkürzen natürlich nicht möglich ist. Wenn diese Regel verwendet wird, dann sind Regeln wie $aAb \to aSb$ aber implizit verkürzend, da Sie das Ableiten von ab aus aAb erlauben. Um das zu verhindern, darf in der Gegenwart von $S \longrightarrow \varepsilon$ das Symbol S nicht auf der rechten Seite von Produktionen verwendet werden.

Beispiel 7.7

Die Grammatik aus Beispiel 7.1 ist vom Typ 2. Sie ist nicht vom Typ 1, da $S \longrightarrow \varepsilon$ vorhanden ist, aber S auf der rechten Seite von Produktionen verwendet wird. Es gibt aber eine Grammatik vom Typ 1, die dieselbe Sprache erzeugt:

$$\begin{array}{rcl} S & \longrightarrow & \varepsilon \\ S & \longrightarrow & S' \\ S' & \longrightarrow & ab \\ S' & \longrightarrow & aS'b \end{array}$$

Die Grammatik aus Beispiel 7.3 ist vom Typ 1. Wir werden später sehen, dass es keine Grammatik vom Typ 2 gibt, die die Sprache aus diesem Beispiel generiert. Die Grammatik aus Beispiel 7.5 ist vom Typ 3.

Die unterschiedlichen Typen von Grammatiken führen zu unterschiedlichen *Typen von Sprachen*.

Definition 7.8 (Klasse der Typ-i-Sprachen)

Für $i = 0, 1, 2, 3$ ist die *Klasse der Typ-i-Sprachen definiert* als

$$\mathcal{L}_i := \{L(G) \mid G \text{ ist Grammatik vom Typ } i\}.$$

Nach Definition kann eine Grammatik von Typ i auch Sprachen höheren Typs $j \geq i$ generieren (aber nicht umgekehrt). So erzeugt beispielsweise die folgende Typ-1-Grammatik die Sprache $\{a^n b^n \mid n \geq 0\}$, welche nach Beispiel 7.1 von Typ 2 ist:

$$\begin{aligned} S &\longrightarrow \varepsilon \\ S &\longrightarrow ab \\ S &\longrightarrow aXb \\ aXb &\longrightarrow aaXbb \\ X &\longrightarrow ab \end{aligned}$$

Offensichtlich ist jede Grammatik von Typ 3 auch eine von Typ 2 und jede Grammatik von Typ 1 auch eine von Typ 0. Da Grammatiken von Typ 2 und 3 das Verkürzen des abgeleiteten Wortes erlauben, sind solche Grammatiken nicht notwendigerweise von Typ 1. Wir werden jedoch später sehen, dass jede Typ-2-Grammatik in eine Typ-1-Grammatik gewandelt werden kann, die dieselbe Sprache erzeugt. Daher bilden die assoziierten Sprachtypen eine Hierarchie. Diese ist sogar strikt.

Lemma 7.9

$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$

Beweis. Nach Definition der Grammatiktypen gilt offenbar $\mathcal{L}_3 \subseteq \mathcal{L}_2$ und $\mathcal{L}_1 \subseteq \mathcal{L}_0$. Die Inklusion $\mathcal{L}_2 \subseteq \mathcal{L}_1$ werden wir später zeigen (Korollar 9.12). Auch die Striktheit der Inklusionen werden wir erst später beweisen. ❑

8. Rechtslineare Grammatiken und reguläre Sprachen

Wir zeigen, dass rechtslineare Grammatiken genau die erkennbaren Sprachen generieren. Damit haben wir eine weitere, unabhängige Charakterisierung dieser Klasse von Sprachen gefunden, und alle Resultate, die wir bereits für die erkennbaren Sprachen bewiesen haben, gelten auch für Typ-3-Sprachen.

Satz 8.1

Die Typ-3-Sprachen sind genau die regulären/erkennbaren Sprachen, d. h.:

$$\mathcal{L}_3 = \{L \mid L \text{ ist regulär}\}$$

Beweis. Der Beweis wird in zwei Richtungen durchgeführt:

1. **Jede Typ-3-Sprache ist erkennbar:**

   Sei $L \in \mathcal{L}_3$, d. h. $L = L(G)$ für eine Typ-3-Grammatik $G = (N, \Sigma, P, S)$. Es gilt $w_1 \cdots w_n \in L(G)$ gdw. es eine Ableitung

   $$S = B_0 \vdash_G w_1 B_1 \vdash_G w_1 w_2 B_2 \vdash_G \ldots \vdash_G w_1 \ldots w_{n-1} B_{n-1} \vdash_G w_1 \ldots w_{n-1} w_n \quad (*)$$

   gibt mit Produktionen $B_{i-1} \longrightarrow w_i B_i \in P$ $(i = 1, \ldots, n)$ und $B_{n-1} \longrightarrow w_n \in P$.

   Diese Ableitung ähnelt dem Lauf eines NEA auf dem Wort $w_1 \cdots w_n$, wobei die Nichtterminale die Zustände sind und die Produktionen die Übergänge beschreiben.

   Wir konstruieren nun einen *NEA mit Wortübergängen*, der die Nichtterminalsymbole von G als Zustände hat:

   $\mathcal{A} = (N \cup \{\Omega\}, \Sigma, S, \Delta, \{\Omega\})$, wobei

   - $\Omega \notin N$ akzeptierender Zustand ist und
   - $\Delta = \{(A, w, B) \mid A \longrightarrow wB \in P\} \cup \{(A, w, \Omega) \mid A \longrightarrow w \in P\}$.

   Ableitungen der Form $(*)$ entsprechen nun genau Pfaden in $\mathcal{A}$:

   $$(S, w_1, B_1)(B_1, w_2, B_2) \ldots (B_{n-2}, w_{n-1}, B_{n-1})(B_{n-1}, w_n, \Omega)$$

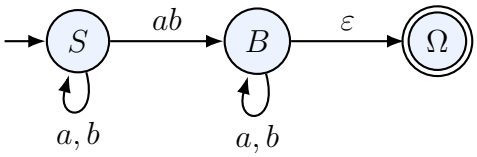
   Dies zeigt $L(\mathcal{A}) = L(G)$.

   Beispiel 7.5 (Fortsetzung)

   Die Grammatik

   $$\begin{aligned} P = \{ & S \longrightarrow aS, \\ & S \longrightarrow bS, \\ & S \longrightarrow abB, \\ & B \longrightarrow aB, \\ & B \longrightarrow bB, \\ & B \longrightarrow \varepsilon\} \end{aligned}$$

   liefert den folgenden NEA mit Wortübergängen:



2. Jede erkennbare Sprache ist eine Typ-3-Sprache:

Sei $L = L(\mathcal{A})$ für einen NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$. Wir definieren daraus eine Typ-3-Grammatik $G = (N, \Sigma, P, S)$ wie folgt:

$$\begin{aligned} N &:= Q \\ S &:= q_0 \\ P &:= \{p \longrightarrow aq \mid (p, a, q) \in \Delta\} \cup \{p \longrightarrow \varepsilon \mid p \in F\} \end{aligned}$$

Ein Pfad in $\mathcal{A}$ der Form

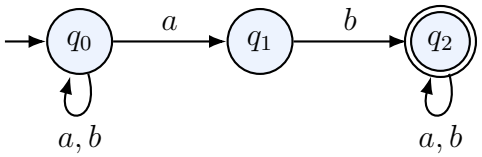
$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_n} q_n$$

mit $q_n \in F$ entspricht nun genau einer Ableitung

$$q_0 \vdash_G a_1 q_1 \vdash_G a_1 a_2 q_2 \vdash_G \ldots \vdash_G a_1 \ldots a_n q_n \vdash_G a_1 \ldots a_n.$$

Beispiel 8.2

Der NEA



liefert die Grammatik mit den rechtslinearen Produktionen

$$\begin{aligned} P = \{ & q_0 \longrightarrow aq_0, \\ & q_0 \longrightarrow bq_0, \\ & q_0 \longrightarrow aq_1, \\ & q_1 \longrightarrow bq_2, \\ & q_2 \longrightarrow aq_2, \\ & q_2 \longrightarrow bq_2, \\ & q_2 \longrightarrow \varepsilon\} \end{aligned}$$

❏

Korollar 8.3

$\mathcal{L}_3 \subset \mathcal{L}_2$.

Beweis. Wir wissen bereits, dass $\mathcal{L}_3 \subseteq \mathcal{L}_2$ gilt. Außerdem haben wir mit Beispiel 7.1 $L := \{a^n b^n \mid n \geq 0\} \in \mathcal{L}_2$. Wir haben bereits gezeigt, dass L nicht erkennbar/regulär ist, d. h. mit Satz 8.1 folgt $L \notin \mathcal{L}_3$. ❏

Beispiel 8.4

Als ein weiteres Beispiel für eine kontextfreie Sprache, die nicht regulär ist, betrachten wir $L = \{a^n b^m \mid n \neq m\}$ (vgl. Beispiele 2.7, 3.2). Man kann diese Sprache mit der folgenden kontextfreien Grammatik erzeugen:

$G = (N, \Sigma, P, S)$ mit

- $N = \{S, S_{\geq}, S_{\leq}\}$
- $\Sigma = \{a, b\}$
- $P = \{S \longrightarrow aS_{\geq}, \quad S \longrightarrow S_{\leq}b,$
  $S_{\geq} \longrightarrow aS_{\geq}b, \quad S_{\leq} \longrightarrow aS_{\leq}b,$
  $S_{\geq} \longrightarrow aS_{\geq}, \quad S_{\leq} \longrightarrow S_{\leq}b,$
  $S_{\geq} \longrightarrow \varepsilon, \quad S_{\leq} \longrightarrow \varepsilon\}$

Es gilt nun:

- $S_{\geq} \vdash_G^* w \in \{a, b\}^* \quad \Rightarrow \quad w = a^n b^m$ mit $n \geq m$,
- $S_{\leq} \vdash_G^* w \in \{a, b\}^* \quad \Rightarrow \quad w = a^n b^m$ mit $n \leq m$,

woraus sich ergibt:

- $S \vdash_G^* w \in \{a, b\}^* \quad \Rightarrow \quad w = aa^n b^m$ mit $n \geq m$ oder $w = a^n b^m b$ mit $n \leq m$,

d. h. $L(G) = \{a^n b^m \mid n \neq m\}$.

9. Normalformen und Entscheidungsprobleme

Es existieren verschiedene *Normalformen* für kontextfreie Grammatiken, bei denen die syntaktische Form der Regeln weiter eingeschränkt wird, ohne dass die Klasse der erzeugten Sprachen sich ändert. Wir werden insbesondere die *Chomsky-Normalform* kennen lernen und sie verwenden, um einen effizienten Algorithmus für das Wortproblem für kontextfreie Sprachen zu entwickeln. Zudem ist jede kontextfreie Grammatik in Chomsky-Normalform auch eine Typ-1-Grammatik, was die noch ausstehende Inklusion $\mathcal{L}_2 \subseteq \mathcal{L}_1$ zeigt. Wir werden auch das Leerheitsproblem und das Äquivalenzproblem diskutieren.

Zwei Grammatiken heißen *äquivalent*, falls sie dieselbe Sprache erzeugen.

9.1. Vereinfachung kontextfreier Grammatiken und das Leerheitsproblem

Zunächst zeigen wir, wie man „überflüssige“ Symbole aus kontextfreien Grammatiken eliminieren kann. Das ist zum späteren Herstellen der Chomsky-Normalform nicht unbedingt notwendig; es ist aber trotzdem ein natürlicher erster Schritt zum Vereinfachen einer Grammatik.

Definition 9.1 (terminierende, erreichbare Symbole; reduzierte Grammatik)
Es sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik.

1) $A \in N$ heißt *terminierend*, falls es ein $w \in \Sigma^*$ gibt mit $A \vdash_G^* w$.
2) $A \in N$ heißt *erreichbar*, falls es $u, v \in (\Sigma \cup N)^*$ gibt mit $S \vdash_G^* uAv$.
3) G heißt *reduziert*, falls alle Elemente von N *erreichbar* und *terminierend* sind.

Die folgenden zwei Lemmata bilden die Grundlage zum Wandeln einer kontextfreien Grammatik in eine reduzierte kontextfreie Grammatik.

Lemma 9.2
Für jede kontextfreie Grammatik $G = (N, \Sigma, P, S)$ kann man die Menge der terminierenden Symbole berechnen.

Beweis. Wir definieren dazu

$$T_1 := \{A \in N \mid \exists w \in \Sigma^* : A \longrightarrow w \in P\}$$
$$T_{i+1} := T_i \cup \{A \in N \mid \exists w \in (\Sigma \cup T_i)^* : A \longrightarrow w \in P\}$$

Es gilt

$$T_1 \subseteq T_2 \subseteq \cdots \subseteq N.$$

Da N endlich ist, gibt es ein k mit $T_k = T_{k+1} = \bigcup_{i \geq 1} T_i$.

Behauptung: $T_k = \{A \in N \mid A \text{ ist terminierend}\}$, denn:

„$\subseteq$“: Zeige durch Induktion über i: alle Elemente von T_i, $i \geq 1$, sind terminierend:

- $i = 1$: $\quad A \in T_1 \Rightarrow A \vdash_G w \in \Sigma^*$, also ist A terminierend.
- $i \to i+1$:

  Wenn $A \in T_{i+1}$, dann gibt es nach Definition T_{i+1} zwei Fälle:
  - $A \in T_i$: Dann ist A nach Induktionsvoraussetzung (IV) terminierend.
  - $A \to w \in P$ für ein $w \in (\Sigma \cup T_i)^*$. Sei $w = u_1 B_1 u_2 B_2 \cdots u_n B_n u_{n+1}$ mit $u_j \in \Sigma^*$ und $B_j \in T_i$. Nach IV sind alle B_j terminierend, also $B_j \vdash_G^* w_j \in \Sigma^*$. Zusammen mit $A \to w \in P$ folgt $A \vdash_G^* u_1 w_1 u_2 w_2 \cdots u_n w_n u_{n+1} \in \Sigma^*$, also ist A terminierend.

„$\supseteq$“: Zeige durch Induktion über i:

$A \vdash_G^{\leq i} w \in \Sigma^* \Rightarrow A \in T_i$.

- $i = 1$: $\quad A \vdash_G^{\leq 1} w \in \Sigma^* \Rightarrow A \longrightarrow w \in P \Rightarrow A \in T_1$
- $i \to i+1$: $A \vdash_G^{\leq i+1} w \in \Sigma^*$

  $\Rightarrow A \vdash_G u_1 B_1 \cdots u_n B_n u_{n+1} \vdash_G^{\leq i} u_1 w_1 \cdots u_n w_n u_{n+1} = w$,
  mit $u_j \in \Sigma^*$ und $B_j \vdash_G^{\leq i} w_j \in \Sigma^*$

  $\Rightarrow B_j \in T_i$ für alle j (Induktion)

  $\Rightarrow A \in T_{i+1}$

❏

Beispiel 9.3

$$P = \{ \quad S \longrightarrow A, \quad S \longrightarrow aCbBa, \quad A \longrightarrow aBAc, \quad B \longrightarrow Cab, \quad C \longrightarrow AB, \quad C \longrightarrow aa \quad \}$$

$$T_1 = \{C\} \quad \subset \quad T_2 = \{C, B\} \quad \subset \quad T_3 = \{C, B, S\} \quad = \quad T_4$$

Es ist also A das einzige nichtterminierende Symbol.

Das Leerheitsproblem für kontextfreie Grammatiken besteht darin, für eine gegebene kontextfreie Grammatik G zu entscheiden, ob $L(G) = \emptyset$. Lemma 9.2 hat folgende interessante Konsequenz.

Satz 9.4

Das Leerheitsproblem ist für kontextfreie Grammatiken in polynomieller Zeit entscheidbar.

Beweis. Offenbar gilt $L(G) \neq \emptyset$ gdw. $\exists w \in \Sigma^* : S \vdash_G^* w$ gdw. S ist terminierend. Außerdem ist leicht zu sehen, dass der Algorithmus zum Berechnen aller terminierenden Symbole in polynomieller Zeit läuft. ❏

Wir werden in der VL *Berechenbarkeit* sehen, dass das Leerheitsproblem für Grammatiken der Typen 0 und 1 nicht entscheidbar ist.

Lemma 9.5

Für jede kontextfreie Grammatik $G = (N, \Sigma, P, S)$ kann man die Menge der erreichbaren Nichtterminalsymbole berechnen.

Beweis. Wir definieren dazu

$$\begin{aligned} E_0 &:= \{S\} \\ E_{i+1} &:= E_i \cup \{A \mid \exists B \in E_i \text{ mit Regel } B \longrightarrow u_1 A u_2 \in P\} \end{aligned}$$

Es gilt

$$E_0 \subseteq E_1 \subseteq E_2 \subseteq \cdots \subseteq N.$$

Da N endlich ist, gibt es ein k mit $E_k = E_{k+1}$ und damit $E_k = \bigcup_{i \geq 0} E_i$.

Behauptung: $E_k = \{A \in N \mid A \text{ ist erreichbar}\}$, denn:

„$\subseteq$“: Zeige durch Induktion über i: E_i enthält nur erreichbare Symbole.

„$\supseteq$“: Zeige durch Induktion über i: $S \vdash_G^i uAv \Rightarrow A \in E_i$.

❑

Beispiel 9.6

$$\begin{aligned} P \;=\; \{ \quad & S \longrightarrow aS, \quad A \longrightarrow ASB, \\ & S \longrightarrow SB, \quad A \longrightarrow C, \\ & S \longrightarrow SS, \quad B \longrightarrow Cb, \\ & S \longrightarrow \varepsilon \qquad\qquad\quad \} \end{aligned}$$

$$E_0 = \{S\} \quad \subset \quad E_1 = \{S, B\} \quad \subset \quad E_2 = \{S, B, C\} \quad = \quad E_3$$

Es ist also A das einzige unerreichbare Symbol.

Lemma 9.2 und 9.5 zusammen zeigen, wie man unerreichbare und nichtterminierende Symbole eliminieren kann.

Satz 9.7

Zu jeder kontextfreien Grammatik G mit $L(G) \neq \emptyset$ kann man eine äquivalente reduzierte kontextfreie Grammatik konstruieren.

Beweis. Sei $G = (N, \Sigma, P, S)$.

Erster Schritt: Eliminieren nichtterminierender Symbole.

G' is die Einschränkung von G auf terminierende Nichtterminale, also
$G' := (N', \Sigma, P', S)$ mit

- $N' := \{A \in N \mid A \text{ ist terminierend in } G\}$
- $P' := \{A \longrightarrow w \in P \mid A \in N', w \in (N' \cup \Sigma)^*\}$

Beachte: Wegen $L(G) \neq \emptyset$ ist S terminierend, also $S \in N'$!

Zweiter Schritt: Eliminieren unerreichbarer Symbole.

$G'' := (N'', \Sigma, P'', S)$ ist die Einschränkung von G' auf erreichbare Nichtterminale, wobei

- $N'' := \{A \in N' \mid A \text{ ist erreichbar in } G'\}$
- $P'' := \{A \longrightarrow w \in P' \mid A \in N''\}$

Man sieht leicht, dass $L(G) = L(G') = L(G'')$ und G'' reduziert ist. ❑

Vorsicht: Die Reihenfolge der beiden Schritte ist wichtig, dann das Eliminieren nichtterminierender Symbole kann zusätzliche Symbole unerreichbar machen (aber nicht umgekehrt). Betrachte zum Beispiel die Grammatik $G = (N, \Sigma, P, S)$ mit

$$P = \{S \longrightarrow \varepsilon, \quad S \longrightarrow AB, \quad A \longrightarrow a\}.$$

In G sind alle Symbole erreichbar. Eliminiert man also zuerst die unerreichbaren Symbole, so ändert sich die Grammatik nicht. Das einzige nichtterminierende Symbol ist B und dessen Elimination liefert

$$P' = \{S \to \varepsilon, \quad A \longrightarrow a\}$$

Diese Grammatik ist nicht reduziert, da nun A unerreichbar ist.

Beachte auch, dass eine Grammatik G mit $L(G) = \emptyset$ niemals reduziert sein kann, da jede Grammatik ein Startsymbol S enthalten muss und S in G nicht terminierend sein kann.

9.2. Die Chomsky-Normalform

Wie zeigen nun, dass jede Grammatik in eine äquivalente Grammatik in *Chomsky-Normalform* gewandelt werden kann. Dies geschieht in drei Schritten:

- Eliminieren von Regeln der Form $A \longrightarrow \varepsilon$ (ε-*Regeln*)
- Eliminieren von Regeln der Form $A \longrightarrow B$ (*Kettenregeln*)
- Aufbrechen langer Wörter auf den rechten Seiten von Produktionen und Aufheben der Mischung von Terminalen und Nichtterminalen

Am Ende werden wir die *Chomsky-Normalform* hergestellt haben, bei der alle Regeln die Form $A \longrightarrow BC$ und $A \longrightarrow a$ haben. Wie bei Typ-1-Grammatiken ist die Ausnahme $S \longrightarrow \varepsilon$ erlaubt, wenn S nicht auf der rechten Seite von Produktionen vorkommt.

Wir beginnen mit dem Eliminieren von ε-Regeln.

Definition 9.8 (ε-freie kontextfreie Grammatik)

Eine kontextfreie Grammatik heißt ε-*frei*, falls gilt:

1) Sie enthält keine Regeln $A \longrightarrow \varepsilon$ für $A \neq S$.

2) Ist $S \longrightarrow \varepsilon$ enthalten, so kommt S nicht auf der rechten Seite einer Regel vor.

Um eine Grammatik ε-frei zu machen, eliminieren wir zunächst *alle* ε-Regeln. Wir erhalten eine Grammatik G' mit $L(G') = L(G) \setminus \{\varepsilon\}$. Das Fehlen von ε kann man später leicht wieder ausgleichen.

Lemma 9.9

Es sei G eine kontextfreie Grammatik. Dann lässt sich eine Grammatik G' ohne ε-Regeln konstruieren mit $L(G') = L(G) \setminus \{\varepsilon\}$.

Beweis.

1) Finde alle $A \in N$ mit $A \vdash_G^* \varepsilon$:

$$N_1 := \{A \in N \mid A \longrightarrow \varepsilon \in P\}$$
$$N_{i+1} := N_i \cup \{A \in N \mid A \longrightarrow B_1 \cdots B_n \in P \text{ mit } B_1, \ldots, B_n \in N_i\}$$

Es gibt ein k mit $N_k = N_{k+1} = \bigcup_{i \geq 1} N_i$. Für dieses k gilt:

Behauptung: $N_k = \{A \mid A \vdash_G^* \varepsilon\}$, denn:

„$\subseteq$“: Zeige durch Induktion über i: Wenn $A \in N_i$, dann $A \vdash_G^* \varepsilon$.

„$\supseteq$“: Zeige durch Induktion über i: Wenn $A \vdash_G^i \varepsilon$, dann $A \in N_i$.

2) Eliminiere in G alle Regeln $A \longrightarrow \varepsilon$. Um dies auszugleichen, nimmt man für alle Regeln

$$A \longrightarrow u_1 B_1 \cdots u_n B_n u_{n+1} \text{ mit } B_1, \ldots, B_n \in N_k \text{ und } u_1, \ldots, u_{n+1} \in (\Sigma \cup N \setminus N_k)^*$$

die Regeln

$$A \longrightarrow u_1 \beta_1 u_2 \cdots u_n \beta_n u_{n+1}$$

hinzu für alle $\beta_1 \in \{B_1, \varepsilon\}, \ldots, \beta_n \in \{B_n, \varepsilon\}$ mit $u_1 \beta_1 u_2 \cdots u_n \beta_n u_{n+1} \neq \varepsilon$.

❑

Beispiel 9.10

$$\begin{aligned}
P \;=\; & \{S \longrightarrow aS,\ S \longrightarrow SS,\ S \longrightarrow bA,\\
& A \longrightarrow BB,\\
& B \longrightarrow CC,\ B \longrightarrow aAbC,\\
& C \longrightarrow \varepsilon\}\\
N_1 \;=\; & \{C\},\\
N_2 \;=\; & \{C, B\},\\
N_3 \;=\; & \{C, B, A\} \;=\; N_4\\
P' \;=\; & \{S \longrightarrow aS,\ S \longrightarrow SS,\ S \longrightarrow bA,\ S \longrightarrow b,\\
& A \longrightarrow BB,\ A \longrightarrow B,\\
& B \longrightarrow CC,\ B \longrightarrow C,\\
& B \longrightarrow aAbC,\ B \longrightarrow abC,\ B \longrightarrow aAb, B \longrightarrow ab\}
\end{aligned}$$

Die Ableitung $S \vdash bA \vdash bBB \vdash bCCB \vdash bCCCC \vdash^* b$ kann nun in G' direkt durch $S \vdash b$ erreicht werden.

Satz 9.11

Zu jeder kontextfreien Grammatik G kann eine äquivalente ε-freie Grammatik konstruiert werden.

Beweis. Konstruiere G' wie im Beweis von Lemma 9.9 beschrieben. Ist $\varepsilon \notin L(G)$ (d. h. $S \not\vdash_G^* \varepsilon$, also $S \notin N_k$), so ist G' die gesuchte ε-freie Grammatik. Sonst erweitere G' um ein neues Startsymbol S_0 und die Produktionen $S_0 \longrightarrow S$ und $S_0 \longrightarrow \varepsilon$. ❑

Korollar 9.12

$\mathcal{L}_2 \subseteq \mathcal{L}_1$.

Beweis. Offenbar ist jede ε-freie kontextfreie Grammatik eine Typ-1-Grammatik, da keine der verbleibenden Regeln verkürzend ist – mit Ausnahme von $S \longrightarrow \varepsilon$, wobei dann S aber wie auch bei Typ 1 gefordert auf keiner rechten Regelseite auftritt. ❑

Der folgende Satz zeigt, dass man auf Kettenregeln verzichten kann.

Satz 9.13

Zu jeder kontextfreien Grammatik kann man eine äquivalente kontextfreie Grammatik konstruieren, die keine Kettenregeln enthält.

Beweis.

1) Bestimme die Relation $K := \{(A, B) \mid A \vdash_G^* B\}$:

$$\begin{aligned} K_0 &:= \{(A, A) \mid A \in N\} \\ K_{i+1} &:= K_i \cup \{(A, B) \in N \times N \mid \exists (A, B') \in K_i : B' \longrightarrow B \in P\} \end{aligned}$$

Es gibt ein k mit $K_k(A) = K_{k+1}(A)$. Für dieses k gilt

Behauptung: $K_k = K$, denn:

„$\subseteq$“: Zeige durch Induktion über i: Wenn $(A, B) \in K_i$, dann $A \vdash_G^* B$.

„$\supseteq$“: Zeige durch Induktion über i: Wenn $A \vdash_G^i B$, dann $(A, B) \in K_i$.

2) Entferne alle Kettenregeln. Um das auszugleichen, füge für jede verbleibende Regel $B \longrightarrow w$ und jedes $(A, B) \in K$ auch $A \longrightarrow w$ hinzu:

$$P' = \{A \longrightarrow w \mid (A, B) \in K,\ B \longrightarrow w \in P \text{ und } w \notin N\}$$ ❑

Beispiel 9.14
Sei

$$P \;=\; \{S \longrightarrow A,\; A \longrightarrow B,\; B \longrightarrow aA,\; B \longrightarrow b\}.$$

Dann ist

$$\begin{aligned}
K_0 &= \{(S,S),\,(A,A),\,(B,B)\} \\
K_1 &= \{(S,S),\,(A,A),\,(B,B),\,(S,A),\,(A,B)\} \\
K_2 &= \{(S,S),\,(A,A),\,(B,B),\,(S,A),\,(A,B),\,(S,B)\} \;=\; K_3 \;=\; K, \qquad \text{also} \\
P' &= \{B \longrightarrow aA,\; A \longrightarrow aA,\; S \longrightarrow aA,\; B \longrightarrow b, \quad A \longrightarrow b, \quad S \longrightarrow b\}.
\end{aligned}$$

Wir etablieren nun die Chomsky-Normalform.

Satz 9.15 (Chomsky-Normalform)
Jede kontextfreie Grammatik lässt sich umformen in eine äquivalente Grammatik, die nur Regeln der folgenden Form enthält:

- $A \longrightarrow a,\; A \longrightarrow BC$ *mit* $A, B, C \in N,\; a \in \Sigma$
- *und eventuell* $S \longrightarrow \varepsilon$*, wobei* S *nicht rechts vorkommt*

Eine derartige Grammatik heißt dann Grammatik in Chomsky-Normalform.

Beweis.

1) Konstruiere zu der gegebenen Grammatik eine äquivalente ε-freie ohne Kettenregeln. (Dabei ist die Reihenfolge wichtig!)
2) Führe für jedes $a \in \Sigma$ ein neues Nichtterminalsymbol X_a und die Produktion $X_a \longrightarrow a$ ein.
3) Ersetze in jeder Produktion $A \longrightarrow w$ mit $w \notin \Sigma$ alle Terminalsymbole a durch die zugehörigen X_a.
4) Produktionen $A \longrightarrow B_1 \cdots B_n$ für $n > 2$ werden ersetzt durch

$$A \longrightarrow B_1C_1,\; C_1 \longrightarrow B_2C_2,\; \ldots,\; C_{n-2} \longrightarrow B_{n-1}B_n$$

wobei die C_i jeweils neue Symbole sind. ❑

Beachte: Das Herstellen der Chomsky-Normalform wie im Beweis angegeben kann dazu führen, dass Nichtterminale überflüssig im Sinne von Definition 9.1 sind – dies verstößt nicht gegen die Chomsky-Normalform. Man kann die entstandene Grammatik nochmals reduzieren (siehe Abschnitt 9.1), ohne die Chomsky-Normalform zu verletzen. Es empfiehlt sich aber trotzdem, die Grammatik bereits *vor* dem Herstellen der Chomsky-Normalform zu reduzieren, da dies die Schritte aus diesem Abschnitt deutlich vereinfachen kann.

Komplexitätsanalyse. Das präsentierte Verfahren für die Umwandlung eine kontextfreien Grammatik in Chomsky-Normalform kann die Grammatik exponentiell vergrößern. Ursache ist das Entfernen von ε-Regeln. Nehmen wir beispielsweise an, dass

$$A \longrightarrow B_1 \cdots B_n \text{ eine Regel ist und } B_1, \ldots, B_n \in N_k,$$

wobei N_k die beim Entfernen von ε-Regeln produzierte Menge von Nichtterminalsymbolen ist. Dann kann jedes B_i entweder beibehalten oder entfernt werden, es dürfen nur nicht alle B_i gleichzeitig entfernt werden. Es ergeben sich also $2^n - 1$ viele Regeln.

Eine exponentielles Aufblasen lässt sich aber sehr leicht verhindern, indem man einfach Schritt 3 der Umwandlung in Chomsky-Normalform *vor* den Schritten 1 und 2 ausführt. Es sind dann nur noch Regeln der Form $A \longrightarrow BC$ vom ε-Freimachen betroffen und daraus werden dann je höchstens 3 Regeln.

Jede kontextfreie Grammatik lässt sich also in eine nur polynomiell größere Grammatik in Chomsky-Normalform wandeln. Die Umwandlung ist zudem in polynomieller Zeit möglich, wie man leicht prüfen kann.

9.3. Das Wortproblem für kontextfreie Sprachen

Wir betrachten nun das *Wortproblem* für kontextfreie Sprachen. Im Gegensatz zu den regulären Sprachen fixieren wir dabei eine kontextfreie Sprache L, die durch eine Grammatik G gegeben ist, anstatt G als Eingabe zu betrachten. Die zu entscheidende Frage lautet dann: gegeben ein Wort $w \in \Sigma^*$, ist $w \in L$? Das Fixieren der Sprache ist für kontextfreie Sprachen in vielen Anwendungen durchaus sinnvoll: wenn man z. B. einen Parser für eine Programmiersprache erstellt, so tut man dies in der Regel für eine einzelne, fixierte Sprache und betrachtet nur das in der Programmiersprache formulierte Programm (= Wort) als Eingabe, nicht aber die Grammatik für die Programmiersprache selbst.

Wir nehmen an, dass die Grammatik in Chomsky-Normalform vorliegt. Wie wir gleich sehen werden, ist dies eine Voraussetzung, um den bekannten CYK-Algorithmus anwenden zu können. Zuvor soll aber kurz eine angenehme Eigenschaft der Chomsky-Normalform erwähnt werden, die auf sehr einfache Weise einen (wenngleich ineffizienten) Algorithmus für das Wortproblem liefert: wenn G eine Grammatik in Chomsky-Normalform ist, dann hat jede Ableitung eines Wortes $w \in L(G)$ *höchstens die Länge* $2|w| + 1$:

- Produktionen der Form $A \longrightarrow BC$ verlängern um 1, d. h. sie können maximal $(|w| - 1)$-mal angewendet werden.
- Produktionen der Form $A \longrightarrow a$ erzeugen genau ein Terminalsymbol von w, d. h. sie werden genau $|w|$-mal angewendet.

Die „$+1$“ in „$2|w| + 1$“ wird nur wegen des leeren Wortes benötigt, das Länge 0 hat, aber einen Ableitungsschritt benötigt. Im Gegensatz dazu kann man zu kontextfreien Grammatiken, die *nicht* in Chomsky-Normalform sind, *überhaupt keine* Schranke für die maximale Länge von Ableitungen angeben. Im Wesentlichen liegt dies daran, dass Kettenregeln und ε-Regeln gestattet sind.

Für Grammatiken in Chomsky-Normalform liefert die obige Beobachtung folgenden Algorithmus für das Wortproblem: gegeben ein Wort w kann man rekursiv *alle möglichen Ableitungen* der Länge $\leq 2|w| + 1$ durchprobieren, denn davon gibt es nur endlich viele. Wie schon erwähnt ist dieses Verfahren aber exponentiell. Einen besseren Ansatz liefert die folgende Überlegung:

Definition 9.16

Es sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik in Chomsky-Normalform und $w = a_1 \cdots a_n \in \Sigma^*$. Wir definieren:

- $w_{ij} := a_i \cdots a_j \quad$ (für $i \leq j$)
- $N_{ij} := \{A \in N \mid A \vdash_G^* w_{ij}\}$

Mit dieser Notation gilt nun:

1)	$S \in N_{1n}$	gdw.	$w \in L(G)$
2)	$A \in N_{ii}$	gdw.	$A \vdash_G^* a_i$
		gdw.	$A \longrightarrow a_i \in P$
3)	$A \in N_{ij}$ für $i < j$	gdw.	$A \vdash_G^* a_i \cdots a_j$
		gdw.	$\exists A \longrightarrow BC \in P$ und ein k mit $i \leq k < j$ mit $B \vdash_G^* a_i \cdots a_k$ und $C \vdash_G^* a_{k+1} \cdots a_j$
		gdw.	$\exists A \longrightarrow BC \in P$ und k mit $i \leq k < j$ mit $B \in N_{ik}$ und $C \in N_{(k+1)j}$

Diese Überlegungen liefern einen Algorithmus zur Berechnung von N_{1n} nach der Methode *Divide and Conquer* („Teile und Herrsche“); siehe Algorithmus 2. Die generelle Idee dabei ist, das eigentliche Problem in Teilprobleme zu zerlegen und diese dann beginnend mit den einfachsten und fortschreitend zu immer komplexeren Teilproblemen zu lösen. Im vorliegenden Fall sind die Teilprobleme das Berechnen der N_{ij} mit $i \leq j$. Die „einfachsten“ Teilprobleme sind dann diejenigen mit $i = j$, und die Teilprobleme werden immer schwieriger, je größer die Teilwortlänge $j - i$ wird.

Beachte: In der innersten Schleife sind N_{ik} und $N_{(k+1)j}$ bereits berechnet, da die Teilwortlängen $k - i$ und $j - k + 1$ kleiner als das aktuelle ℓ.

Satz 9.17

Für jede Grammatik G in Chomsky-Normalform entscheidet der CYK-Algorithmus die Frage „gegeben w, ist $w \in L(G)$?“ in Zeit $\mathcal{O}(|w|^3)$.

Beweis. Die erste for-Schleife macht n Schritte; der Rest des Algorithmus besteht aus drei geschachtelte Schleifen, die jeweils $\leq |w| = n$ Schritte machen; also werden in der innersten Schleife $\leq |w|^3$ Schritte gemacht.

Algorithmus 2 : CYK-Algorithmus von Cocke, Younger, Kasami

for $i := 1$ **to** n **do**
  $N_{ii} := \{A \mid A \longrightarrow a_i \in P\}$

for $\ell := 1$ **to** $n-1$ **do** `// wachsende Teilwortlänge` $\ell = j - i$
  for $i := 1$ **to** $n - \ell$ **do** `// Startposition` i `von Teilwort`
    $j := i + \ell$ `// Endposition` j `von Teilwort`
    $N_{ij} := \emptyset$
    for $k := i$ **to** $j - 1$ **do** `// Mögliche Trennpositionen` k
      $N_{ij} := N_{ij} \cup \{A \mid \exists A \longrightarrow BC \in P \text{ mit } B \in N_{ik} \text{ und } C \in N_{(k+1)j}\}$

Es ist zu beachten, dass die Suche nach den Produktionen $A \longrightarrow a_i$ $A \longrightarrow BC$ und im Inneren der Schleifen auch nur konstante Zeit benötigt, denn wir haben die Größe von G hier als konstant angenommen (fest vorgegebenes G). ❑

Beispiel 9.18

$P = \{S \longrightarrow SA,\ S \longrightarrow a,$
$A \longrightarrow BS,$
$B \longrightarrow BB,\ B \longrightarrow BS,\ B \longrightarrow b,\ B \longrightarrow c\}$

und $w = abacba$:

i \ j	1	2	3	4	5	6
1	S	$\emptyset$	S	$\emptyset$	$\emptyset$	S
2		B	A, B	B	B	A, B
3			S	$\emptyset$	$\emptyset$	S
4				B	B	A, B
5					B	A, B
6						S
$w =$	a	b	a	c	b	a

$S \in N_{1,6} = \{S\}$
$\Rightarrow w \in L(G)$

Wir werden in der VL *Berechenbarkeit* beweisen, dass das Äquivalenzproblem für kontextfreie Sprachen unentscheidbar ist, siehe Satz 17.23 dieses Skriptes. Es gibt also keinen Algorithmus, der für zwei gegebene kontextfreie Grammatiken G_1 und G_2 entscheidet, ob $L(G_1) = L(G_2)$.

Man beachte, dass es sich bei der Aussage in Satz 9.17 um Datenkomplexität handelt, vergleiche Abschnitt 4.1.

9.4. Die Greibach-Normalform

Eine weitere interessante Normalform für kontextfreie Grammatiken ist die *Greibach-Normalform*, bei der es für jedes Wort in der Sprache eine Ableitung gibt, die die Terminale *streng von links nach rechts* erzeugt, und zwar *genau ein* Terminal pro Ableitungsschritt. Wir geben den folgenden Satz ohne Beweis an.

Satz 9.19 (Greibach-Normalform)
Jede kontextfreie Grammatik lässt sich umformen in eine äquivalente Grammatik, die nur Regeln der Form

- $A \longrightarrow aw \quad (A \in N, a \in \Sigma, w \in N^*)$
- *und eventuell* $S \longrightarrow \varepsilon$*, wobei* S *nicht rechts vorkommt*

enthält.
Eine derartige Grammatik heißt Grammatik in Greibach-Normalform.

10. Abschlusseigenschaften und Pumping-Lemma

Die kontextfreien Sprachen verhalten sich bezüglich der Abschlusseigenschaften nicht ganz so gut wie die regulären Sprachen. Wir beginnen mit positiven Resultaten.

Satz 10.1

Die Klasse $\mathcal{L}_2$ der kontextfreien Sprachen ist unter Vereinigung, Konkatenation und Kleene-Stern abgeschlossen.

Beweis. Es seien $L_1 = L(G_1)$ und $L_2 = L(G_2)$ die Sprachen für kontextfreie Grammatiken $G_i = (N_i, \Sigma, P_i, S_i), i = 1, 2$. Wir nehmen o. B. d. A. an, dass $N_1 \cap N_2 = \emptyset$.

1) $G := (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \longrightarrow S_1, S \longrightarrow S_2\}, S)$ mit $S \notin N_1 \cup N_2$ ist eine kontextfreie Grammatik mit $L(G) = L_1 \cup L_2$.
2) $G' := (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \longrightarrow S_1S_2\}, S)$ mit $S \notin N_1 \cup N_2$ ist eine kontextfreie Grammatik mit $L(G') = L_1 \cdot L_2$.
3) $G'' := (N_1 \cup \{S\}, \Sigma, P_1 \cup \{S \longrightarrow \varepsilon, S \longrightarrow SS_1\}, S)$ mit $S \notin N_1$ ist eine kontextfreie Grammatik für L_1^*.

❑

Wir werden zeigen, dass Abschluss unter *Schnitt* und *Komplement nicht* gilt. Dazu benötigen wir zunächst eine geeignete Methode, von einer Sprache nachzuweisen, dass sie *nicht* kontextfrei ist. Dies gelingt wieder mit Hilfe eines Pumping-Lemmas. In dessen Beweis stellt man Ableitungen als Bäume dar, so genannte *Ableitungsbäume*.

Beispiel 10.2

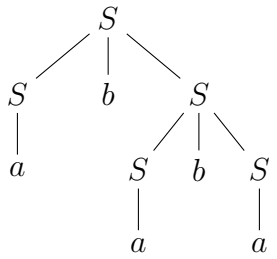
$P = \{S \longrightarrow SbS, S \longrightarrow a\}$

Drei *Ableitungen* des Wortes *ababa*:

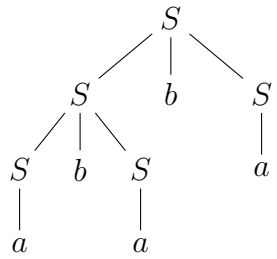
1) $S \vdash SbS \vdash abS \vdash abSbS \vdash ababS \vdash ababa$
2) $S \vdash SbS \vdash abS \vdash abSbS \vdash abSba \vdash ababa$
3) $S \vdash SbS \vdash Sba \vdash SbSba \vdash Sbaba \vdash ababa$

Die zugehörigen *Ableitungsbäume*:

für 1) und 2):



für 3):



Ein Ableitungsbaum kann also für mehr als eine Ableitung stehen, und dasselbe Wort kann verschiedene Ableitungsbäume haben.

Wir verzichten auf eine exakte Definition von Ableitungsbäumen, da diese eher kompliziert als hilfreich ist. Stattdessen geben wir nur einige zentrale Eigenschaften an.

Allgemein:

Die Knoten des Ableitungsbaumes sind mit Elementen aus $\Sigma \cup N$ beschriftet. Dabei dürfen Terminalsymbole nur an den Blättern vorkommen (also an den Knoten ohne Nachfolger) und Nichtterminale überall (um auch partielle Ableitungen darstellen zu können). Ein mit A beschrifteter Knoten kann mit $\alpha_1, \ldots, \alpha_n$ beschriftete Nachfolgerknoten haben, wenn $A \longrightarrow \alpha_1 \cdots \alpha_n \in P$ ist.

Ein Ableitungsbaum, dessen Wurzel mit A beschriftet ist und dessen Blätter (von links nach rechts) mit $\alpha_1, \ldots, \alpha_n \in \Sigma \cup N$ beschriftet sind, beschreibt eine Ableitung $A \vdash_G^* \alpha_1 \cdots \alpha_n$.

Lemma 10.3 (Pumping-Lemma für kontextfreie Sprachen)

Für jede kontextfreie Sprache L gibt es ein $n_0 \geq 1$, so dass gilt:
für jedes $z \in L$ mit $|z| \geq n_0$ existiert eine Zerlegung $z = uvwxy$ mit:

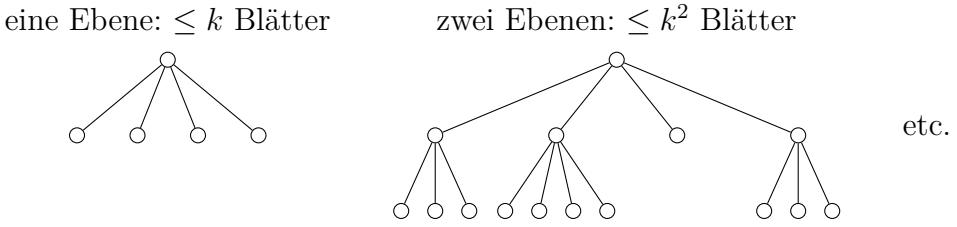
- $vx \neq \varepsilon$ *und* $|vwx| \leq n_0$
- $uv^i wx^i y \in L$ *für alle* $i \geq 0$.

Beweis. Sei G eine kontextfreie Grammatik mit $L(G) = L$. Nach Satz 9.11 und 9.13 können wir o. B. d. A. annehmen, dass G ε-frei ist und keine Kettenregeln enthält. Sei

- m die Anzahl der Nichtterminale in G;
- k die maximale Länge von rechten Regelseiten in G
  (da G ε-frei ist, muss $k \geq 1$ sein[1]);
- $n_0 = k^{m+1}$
  (wegen $k \geq 1$ ist $n_0 \geq 1$).

Wir verfahren nun wie folgt.

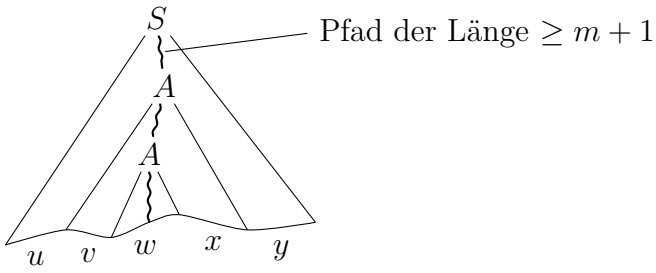
1) Ein Baum der Tiefe $\leq t$ und der Verzweigungszahl $\leq k$ hat maximal k^t viele Blätter:



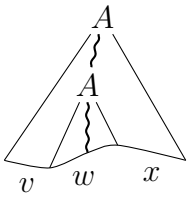
[1] Genau genommen muss man hier zwei Fälle unterscheiden: Wenn G nur die Produktion $S \to \varepsilon$ oder gar keine Produktionen besitzt, dann gilt die Aussage des Lemmas trivialerweise mit $n_0 = 1$ (denn es gibt kein $z \in L$ mit $|z| \geq 1$). Anderenfalls besitzt G mindestens eine Produktion $u \to v$, die nicht die Form $S \to \varepsilon$ hat; wegen der ε-Freiheit muss dann $|v| \geq 1$ sein.

Der Ableitungsbaum für z hat $|z| \geq k^{m+1}$ Blätter; also gibt es einen Pfad (Weg von Wurzel zu Blatt) der Länge $\geq m+1$ (gemessen in der Anzahl der Kanten).

2) Auf diesem Pfad kommen $> m+1$ Symbole vor, davon $> m$ Nichtterminale. Da es nur m verschiedene Nichtterminale gibt, kommt ein Nichtterminal A zweimal vor. Dies führt zu folgender Wahl von u, v, w, x, y:



Wir wählen hier o. B. d. A. die erste Wiederholung eines Nichtterminals A von den Blättern aus gesehen; mit den Argumenten in 1) hat dann der Teilbaum



die Tiefe $\leq m+1$, was $|vwx| \leq k^{m+1} = n_0$ zeigt.

3) Es gilt: $S \vdash_G^* uAy, \quad A \vdash_G^* vAx, \quad A \vdash_G^* w$, woraus folgt:

$$S \vdash_G^* uAy \vdash_G^* uv^iAx^iy \vdash_G^* uv^iwx^iy.$$

4) $vx \neq \varepsilon$: Da G ε-frei ist, wäre sonst $A \vdash_G^* vAx$ nur bei Anwesenheit von Regeln der Form $A \longrightarrow B$ möglich. ❑

Wir verwenden nun das Pumping-Lemma, um beispielhaft von einer Sprache nachzuweisen, dass sie nicht kontextfrei ist.

Lemma 10.4
$L = \{a^nb^nc^n \mid n \geq 1\} \notin \mathcal{L}_2$.

Beweis. Angenommen, $L \in \mathcal{L}_2$. Dann gibt es eine ε-freie kontextfreie Grammatik G ohne Regeln der Form $A \longrightarrow B$ für L. Es sei n_0 die zugehörige Zahl aus Lemma 10.3. Wir betrachten $z = a^{n_0}b^{n_0}c^{n_0} \in L = L(G)$. Mit Satz 10.3 gibt es eine Zerlegung

$$z = uvwxy, \ vx \neq \varepsilon \text{ und } uv^iwx^iy \in L \text{ für alle } i \geq 0.$$

1. Fall: v enthält verschiedene Symbole. Man sieht leicht, dass dann

$$uv^2wx^2y \notin a^*b^*c^* \supseteq L.$$

2. Fall: x enthält verschiedene Symbole. Dies führt zu entsprechendem Widerspruch.

3. Fall: v enthält lauter gleiche Symbole und x enthält lauter gleiche Symbole. Dann gibt es ein Symbol aus $\{a, b, c\}$, das in xv nicht vorkommt. Daher kommt dieses in $uv^0wx^0y = uwy$ weiterhin n_0-mal vor. Aber es gilt $|uwy| < 3n_0$, was $uwy \notin L$ zeigt. ❏

Dieses Beispiel zeigt auch, dass die kontextfreien Sprachen eine *echte* Teilmenge der Typ-1-Sprachen sind.

Satz 10.5

$\mathcal{L}_2 \subset \mathcal{L}_1$.

Beweis. Wir haben bereits gezeigt, dass $\mathcal{L}_2 \subseteq \mathcal{L}_1$ gilt (Korollar 9.12). Es bleibt zu zeigen, dass die Inklusion echt ist. Dafür betrachten wir die Sprache $L = \{a^n b^n c^n \mid n \geq 1\}$. Nach Lemma 10.4 ist $L \notin \mathcal{L}_2$. Nach Beispiel 7.3 gilt aber $L \in \mathcal{L}_1$. ❏

Außerdem können wir nun zeigen, dass die kontextfreien Sprachen unter zwei wichtigen Operationen nicht abgeschlossen sind.

Korollar 10.6

Die Klasse $\mathcal{L}_2$ der kontextfreien Sprachen ist nicht *unter Schnitt und Komplement abgeschlossen.*

Beweis.

1) Die Sprachen $\{a^n b^n c^m \mid n \geq 1, m \geq 1\}$ und $\{a^m b^n c^n \mid n \geq 1, m \geq 1\}$ sind in $\mathcal{L}_2$:
   - $\{a^n b^n c^m \mid n \geq 1, m \geq 1\} = \underbrace{\underbrace{\{a^n b^n \mid n \geq 1\}}_{\in \mathcal{L}_2} \cdot \underbrace{\{c^m \mid m \geq 1\}}_{= c^+ \in \mathcal{L}_3 \subseteq \mathcal{L}_2}}_{\in \mathcal{L}_2 \text{ (Konkatenation)}}$
   - $\{a^m b^n c^n \mid n \geq 1, m \geq 1\}$ — analog
2) $\{a^n b^n c^n \mid n \geq 1\} = \{a^n b^n c^m \mid n, m \geq 1\} \cap \{a^m b^n c^n \mid n, m \geq 1\}$.
   Wäre $\mathcal{L}_2$ unter $\cap$ abgeschlossen, so würde $\{a^n b^n c^n \mid n \geq 1\} \in \mathcal{L}_2$ folgen.
   Widerspruch zu Teil 1) des Beweises von Satz 10.5.
3) $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.
   Wäre $\mathcal{L}_2$ unter Komplement abgeschlossen, so auch unter $\cap$, da $\mathcal{L}_2$ unter $\cup$ abgeschlossen ist. Widerspruch zu 2). ❏

Beachte:

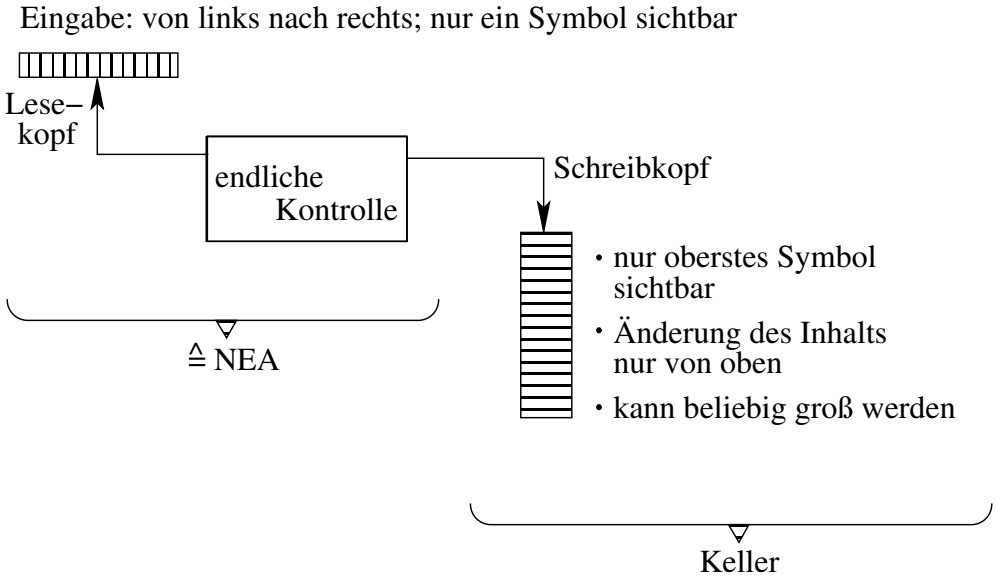
Man kann daher das Äquivalenzproblem für kontextfreie Sprachen nicht einfach auf das Leerheitsproblem reduzieren (dazu braucht man sowohl Schnitt als auch Komplement). Wie bereits erwähnt werden wir später sogar sehen, dass das Äquivalenzproblem für kontextfreie Sprachen *unentscheidbar* ist.

11. Kellerautomaten

Bisher hatten wir kontextfreie Sprachen nur mit Hilfe von Grammatiken charakterisiert. Wir haben gesehen, dass endliche Automaten *nicht* in der Lage sind, alle kontextfreien Sprachen zu akzeptieren.

Um die *Beschreibung von kontextfreien Sprachen* mit Hilfe von endlichen Automaten zu ermöglichen, muss man diese um eine unbeschränkte *Speicherkomponente*, einen so genannten *Keller (englisch: Stack)* erweitern. Dieser Keller speichert zwar zu jedem Zeitpunkt nur endlich viel Information, kann aber unbeschränkt wachsen.

Die folgende Abbildung zeigt eine schematische Darstellung eines *Kellerautomaten*:



Diese Idee wird in folgender Weise formalisiert.

Definition 11.1 (Kellerautomat)

Ein *Kellerautomat (pushdown automaton, kurz PDA)* hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F)$, wobei

- Q eine endliche Menge von *Zuständen* ist,
- Σ das *Eingabealphabet* ist,
- Γ das *Kelleralphabet* ist,
- $q_0 \in Q$ der *Anfangszustand* ist,
- $Z_0 \in \Gamma$ das *Kellerstartsymbol* ist,
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Gamma^* \times Q$ eine endliche *Übergangsrelation* ist und
- $F \subseteq Q$ eine Menge von *akzeptierenden Zuständen* ist.

Anschaulich bedeutet die Übergangsrelation:

(q, a, Z, γ, q')**:** Im Zustand q mit aktuellem Eingabesymbol a und oberstem Kellersymbol Z darf der Automat Z durch γ ersetzen und in den Zustand q' und zum nächsten Eingabesymbol übergehen.

$(q, \varepsilon, Z, \gamma, q')$**:** wie oben, nur dass das aktuelle Eingabesymbol *nicht relevant* ist und man nicht zum nächsten Eingabesymbol übergeht (der Lesekopf ändert seine Position nicht).

Übergänge der zweiten Form können nicht so einfach eliminiert werden, wie wir das für ε-Übergänge von ε-NEAs kennen, denn durch ε-Übergänge kann ein PDA beliebig viele Kellersymbole löschen, ohne ein weiteres Symbol im Eingabewort zu lesen.[2]

Im Gegensatz zu den ε-Übergängen von ε-NEAs können bei PDAs Übergänge der zweiten Form im Allgemeinen nicht eliminiert werden (denn durch ε-Übergänge kann ein PDA beliebig viele Kellersymbole löschen, ohne ein weiteres Symbol im Eingabewort zu lesen).

Um die Sprache zu definieren, die von einem Kellerautomaten erkannt wird, brauchen wir den Begriff der *Konfiguration*, die den aktuellen Stand einer Kellerautomatenberechnung erfasst. Diese ist bestimmt durch.

- den *noch zu lesenden Rest* $w \in \Sigma^*$ der Eingabe (Lesekopf steht auf dem ersten Symbol von w)
- den *Zustand* $q \in Q$
- den *Kellerinhalt* $\gamma \in \Gamma^*$ (Schreibkopf steht auf dem ersten Symbol von γ)

Definition 11.2

Eine *Konfiguration* von $\mathcal{A}$ hat die Form

$$\mathcal{K} = (q, w, \gamma) \in Q \times \Sigma^* \times \Gamma^*.$$

Die Übergangsrelation ermöglicht die folgenden *Konfigurationsübergänge*:

- $(q, aw, Z\gamma) \vdash_{\mathcal{A}} (q', w, \beta\gamma)$ falls $(q, a, Z, \beta, q') \in \Delta$
- $(q, w, Z\gamma) \vdash_{\mathcal{A}} (q', w, \beta\gamma)$ falls $(q, \varepsilon, Z, \beta, q') \in \Delta$
- $\mathcal{K} \vdash_{\mathcal{A}}^* \mathcal{K}'$ gdw. $\exists n \geq 0\ \exists \mathcal{K}_0, \ldots, \mathcal{K}_n$ mit

$$\mathcal{K}_0 = \mathcal{K},\ \mathcal{K}_n = \mathcal{K}' \text{ und } \mathcal{K}_i \vdash_{\mathcal{A}} \mathcal{K}_{i+1} \text{ für } 0 \leq i < n.$$

Eine solchen Konfigurationsfolge $\mathcal{K}_0, \ldots, \mathcal{K}_n$ nennen wir manchmal auch eine *Berechnung*. Der Automat $\mathcal{A}$ *akzeptiert* das Wort $w \in \Sigma^*$ gdw.

$$(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \gamma) \quad \text{für ein } q \in F \text{ und } \gamma \in \Gamma^*.$$

[2] Man kann jedoch die Greibach-Normalform für kontextfreie Grammatiken (Satz 9.19) sowie die im Folgenden gezeigte Äquivalenz von PDAs und kontextfreien Grammatiken (Satz 11.9) benutzen um zu zeigen, dass es zu jedem PDA einen äquivalenten PDA ohne ε-Übergänge gibt. Dies gilt wiederum nicht für die deterministische Variante (Def. 11.12): dPDAs ohne ε-Übergänge sind echt schwächer als dPDAs mit.

Dabei bedeutet die Konfiguration (q, ε, γ), dass $\mathcal{A}$ sich in einem akzeptierenden Zustand befindet, das Eingabewort ganz gelesen hat, und dass im Keller noch beliebige Symbole stehen dürfen. Wir nennen (q_0, w, Z_0) auch die *initial Konfiguration* von $\mathcal{A}$ auf w. Die von $\mathcal{A}$ *erkannte Sprache* ist

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}.$$

Man beachte, dass es sich bei PDAs wie oben definiert um ein *nichtdeterministisches* Automatenmodell handelt: für eine gegebene Konfiguration sind unter umständen verschiedene Folgekonfigurationen möglich. Akzeptanz ist dann wie bei NEAs *existentiell* definiert: es genügt, dass es mindestens eine Konfigurationsfolge von der initialen Konfiguration zu einer Konfiguration der Form (q, ε, γ) gibt.

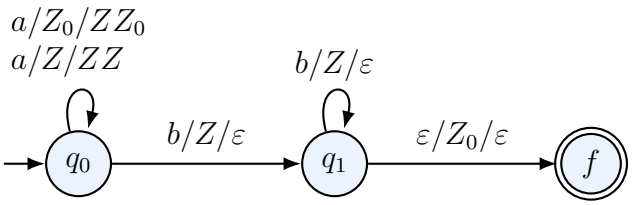
Im Folgenden zwei einfache Beispiele für Kellerautomaten.

Beispiel 11.3

Ein PDA für $\{a^n b^n \mid n \geq 1\}$.

- $Q = \{q_0, q_1, f\}$
- $\Gamma = \{Z, Z_0\}$
- $\Sigma = \{a, b\}$
- $\Delta = \{(q_0, a, Z_0, ZZ_0, q_0)$, (erstes a, speichere Z)
  (q_0, a, Z, ZZ, q_0), (weitere a's, speichere Z)
  $(q_0, b, Z, \varepsilon, q_1)$, (erstes b, entnimm Z)
  $(q_1, b, Z, \varepsilon, q_1)$, (weitere b's, entnimm Z)
  $(q_1, \varepsilon, Z_0, \varepsilon, f)\}$ (lösche das Kellerstartsymbol)
- $F = \{f\}$

Wir stellen einen Kellerautomaten graphisch in der folgenden Weise dar, wobei die Kantenbeschriftung $a/Z/\gamma$ bedeutet, dass der Automat bei Z als oberstem Kellersymbol das Eingabesymbol a lesen und Z durch γ ersetzen kann.



Betrachten wir beispielhaft einige Konfigurationsübergänge:

1) $(q_0, aabb, Z_0) \vdash_{\mathcal{A}} (q_0, abb, ZZ_0) \vdash_{\mathcal{A}} (q_0, bb, ZZZ_0) \vdash_{\mathcal{A}} (q_1, b, ZZ_0) \vdash_{\mathcal{A}} (q_1, \varepsilon, Z_0) \vdash_{\mathcal{A}} (f, \varepsilon, \varepsilon)$
   – akzeptiert

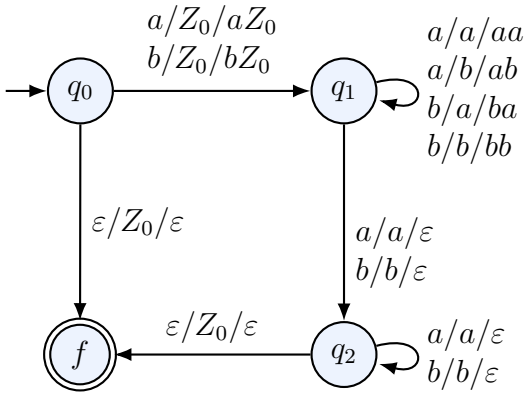
2) $(q_0, aab, Z_0) \vdash_{\mathcal{A}}^* (q_0, b, ZZZ_0) \vdash_{\mathcal{A}} (q_1, \varepsilon, ZZ_0)$
   – kein Übergang mehr möglich, nicht akzeptiert

3) $(q_0, abb, Z_0) \vdash_{\mathcal{A}} (q_0, bb, ZZ_0) \vdash_{\mathcal{A}} (q_1, b, Z_0) \vdash_{\mathcal{A}} (f, b, \varepsilon)$
   – nicht akzeptiert (weil das Wort nicht zu Ende gelesen wurde – „b“ in (f, b, ε))

Beispiel 11.4

Ein PDA für $L = \{ww^R \mid w \in \{a, b\}^*\}$ (wobei für $w = a_1 \cdots a_n$ gilt $w^R = a_n \cdots a_1$).

- $Q = \{q_0, q_1, q_2, f\}$
- $\Gamma = \{a, b, Z_0\}$
- $\Sigma = \{a, b\}$
- $F = \{f\}$
- $\Delta =$



In q_1 wird die erste Worthälfte im Keller gespeichert. Der nichtdeterministische Übergang von q_1 nach q_2 „rät“ die Wortmitte. In q_2 wird die zweite Hälfte des Wortes gelesen und mit dem Keller verglichen.

Die in den Definitionen 11.1 und 11.2 eingeführte Version von Kellerautomaten akzeptiert wie auch ein NEA *per akzeptierendem Zustand*. Man kann stattdessen auch Akzeptanz *per leerem Keller* definieren.

Definition 11.5

Ein *PDA mit Akzeptanz per leerem Keller* ist ein Tupel

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta),$$

wobei alle Komponenten wie in Definition 11.1 sind. Ein solcher PDA *akzeptiert* ein Eingabewort $w \in \Sigma^*$ gdw. $(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$ für ein $q \in Q$.

Dabei bedeutet die Konfiguration $(q, \varepsilon, \varepsilon)$ also, dass sich $\mathcal{A}$ in einem beliebigen Zustand befinden kann, das Eingabewort zu Ende gelesen haben muss und den Keller vollständig geleert haben muss (einschließlich des Kellerstartsymbols Z_0).

Es ist nicht schwer zu zeigen, dass PDAs mit Akzeptanz per leerem Keller und solche mit akzeptierenden Zuständen dieselbe Sprachklasse definieren.

Satz 11.6

Für jede formale Sprache L sind äquivalent:

1. *L with von einem PDA (mit akzeptierenden Zuständen) erkannt.*
2. *L wird von einem PDA mit Akzeptanz per leerem Keller erkannt.*

Beweis. „$1 \Rightarrow 2$“. Sei

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F)$$

ein PDA, der per akzeptierenden Zuständen akzeptiert. Wir konstruieren einen äquivalenten PDA $\mathcal{A}'$, der per leerem Keller akzeptiert, indem wir $\mathcal{A}$ wie folgt modifizieren: füge ε-Transitionen hinzu, die ausgehend von jedem akzeptierenden Zustand $f \in F$ den Keller leeren.

Es gibt dabei jedoch eine Schwierigkeit: während einer Berechnung von $\mathcal{A}$ kann der Keller auch in einem nicht-akzeptierenden Zustand leer werden. Der ursprüngliche PDA $\mathcal{A}$ verwirft dann, der neue PDA $\mathcal{A}'$ akzeptiert aber.

Die Lösung besteht darin, für $\mathcal{A}'$ ein neues Kellerstartsymbol einzuführen, das während der gesamten Berechnung von $\mathcal{A}$ auf dem Keller bleibt, so dass dieser niemals unerwünschterweise leer werden kann.

Wir definieren daher

$$\mathcal{A}' = (Q \cup \{q_0', \ell\}, \Sigma, \Gamma \cup \{Z_0'\}, q_0', Z_0', \Delta')$$

wobei

$$\begin{aligned}
\Delta' \;=\; & \Delta \cup \{(q_0', \varepsilon, Z_0', Z_0 Z_0', q_0)\} && \text{(Altes Startsymbol auf Keller)} \\
& \cup \{(f, \varepsilon, Z, \varepsilon, \ell) \mid f \in F \text{ und } Z \in \Gamma \cup \{Z_0'\}\} && \text{(Zustand } \ell \text{ leert Keller)} \\
& \cup \{(\ell, \varepsilon, Z, \varepsilon, \ell) \mid Z \in \Gamma \cup \{Z_0'\}\}. && \text{(Keller leeren)}
\end{aligned}$$

Man kann zeigen, das $L(\mathcal{A}) = L(\mathcal{A}')$.

„$2 \Rightarrow 1$“. Sei

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$$

ein PDA, der per leerem Keller akzeptiert. Wir konstruieren einen äquivalenten PDA $\mathcal{A}'$, der per Zustand akzeptiert, indem wir $\mathcal{A}$ wie folgt modifizieren: füge akzeptierenden Zustand f hinzu, wechsle zu f sobald der Keller leer ist.

Auch hier gibt es aber wieder eine Schwierigkeit: wenn der Keller erstmal leer ist, dann können wir nicht mehr zu f wechseln, denn jeder Übergang eines PDA *muss* per Definition ein oberstes Kellersymbol lesen.

Die Lösung besteht auch hier darin, ein zusätzliches Kellerstartsymbol einzuführen. Wenn dieses Kellerstartsymbol oben liegt, dann repräsentiert das den leeren Keller und es ist ein weiterer Übergang möglich.

Wir definieren daher

$$\mathcal{A}' = (Q \cup \{q_0', f\}, \Sigma, \Gamma \cup \{Z_0'\}, q_0', Z_0', \Delta', \{f\})$$

wobei

$$\begin{aligned} \Delta' &= \Delta \cup \{(q_0', \varepsilon, Z_0', Z_0 Z_0', q_0)\} \quad \text{(Altes Startsymbol auf Keller)} \\ &\quad \cup \{(q, \varepsilon, Z_0', \varepsilon, f) \mid q \in Q\} \quad \text{(bei leerem Keller zu } f\text{)} \end{aligned}$$

Man kann zeigen, das $L(\mathcal{A}) = L(\mathcal{A}')$.

❑

Wir werden nun zeigen, dass man mit Kellerautomaten genau die kontextfreien Sprachen erkennen kann. Dazu führen wir zunächst den Begriff der Linksableitung ein.

Definition 11.7

Sei G eine kontextfreie Grammatik. Eine Ableitung $S = w_0 \vdash_G w_1 \vdash_G w_2 \vdash_G \cdots \vdash_G w_n$ heißt *Linksableitung*, wenn sich w_{i+1} aus w_i durch Anwendung einer Regel auf das am weitesten links stehende Nichtterminal in w_i ergibt, für alle $i < n$.

Das folgende Lemma zeigt, dass sich die von einer kontextfreien Grammatik erzeugte Sprache nicht ändert, wenn man statt beliebigen Ableitungen nur noch Linksableitungen zulässt.

Lemma 11.8

Für jede kontextfreie Grammatik G gilt:

$$L(G) = \{w \in \Sigma^* \mid w \textit{ kann in } G \textit{ mit Linksableitung erzeugt werden}\}$$

Zum Beweis von Lemma 11.8 genügt es zu zeigen, dass jedes $w \in L(G)$ mittels einer Linksableitung erzeugt werden kann. Wir argumentieren nur informell: wenn $w \in L(G)$, dann gibt es eine Ableitung von w in G. Diese kann als Ableitungsbaum dargestellt werden. Aus dem Ableitungsbaum lässt sich nun wiederum eine Linksableitung von w ablesen (zum Beispiel durch Traversieren des Baumes in Pre-Order).

Satz 11.9

Für jede formale Sprache L sind äquivalent:

1) *L wird von einer kontextfreien Grammatik erzeugt.*
2) *L wird von einem PDA erkannt.*

Beweis.

„1 ⇒ 2“.

Es sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. Wegen Satz 11.6 genügt es, einen PDA $\mathcal{A}$ *mit Akzeptanz per leerem Keller* zu konstruieren, so dass $L(\mathcal{A}) = L(G)$. Die Idee ist, dass $\mathcal{A}$ die Linksableitungen von G auf dem Keller simulieren soll.

Dazu definieren wir $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ wobei $Q = \{q\}$, $\Gamma = \Sigma \cup N$, $q_0 = q$, $Z_0 = S$ und Δ aus folgenden Übergängen besteht:

- Übergänge zum Anwenden von Produktionen auf das oberste Kellersymbol:

  für jede Regel $A \to \gamma$ den Übergang $(q, \varepsilon, A, \gamma, q)$

- Übergänge zum Entfernen bereits erzeugter Terminalsymbole von der Kellerspitze, wenn sie in der Eingabe vorhanden sind:

  für jedes Terminalsymbol $a \in \Sigma$ den Übergang $(q, a, a, \varepsilon, q)$

Beispiel:

$P = \{S \longrightarrow \varepsilon,\ S \longrightarrow aSa,\ S \longrightarrow bSb\}$ liefert die Übergänge:

$(q,$	$\varepsilon,$	$S,$	$\varepsilon,$	$q),$	$(S \longrightarrow \varepsilon)$
$(q,$	$\varepsilon,$	$S,$	$aSa,$	$q),$	$(S \longrightarrow aSa)$
$(q,$	$\varepsilon,$	$S,$	$bSb,$	$q),$	$(S \longrightarrow bSb)$
$(q,$	$a,$	$a,$	$\varepsilon,$	$q),$	(a entfernen)
$(q,$	$b,$	$b,$	$\varepsilon,$	$q)$	(b entfernen)

Die Ableitung $S \vdash_G aSa \vdash_G abSba \vdash_G abba$ entspricht der *Konfigurationsfolge*

$$(q, abba, S) \vdash_{\mathcal{A}} (q, abba, aSa) \vdash_{\mathcal{A}} (q, bba, Sa) \vdash_{\mathcal{A}} (q, bba, bSba) \vdash_{\mathcal{A}} (q, ba, Sba) \vdash_{\mathcal{A}} (q, ba, ba) \vdash_{\mathcal{A}} (q, a, a) \vdash_{\mathcal{A}} (q, \varepsilon, \varepsilon)$$

Zu zeigen:

Für alle $w \in \Sigma^*$ gilt: $S \vdash_G^* w$ mittels Linksableitung gdw. $(q, w, S) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$.

Beweis der Behauptung.

„⇒“: Sei

$$S = w_0 \vdash_G w_1 \vdash_G \cdots \vdash_G w_n = w$$

eine Linksableitung. Für alle $i \leq n$ sei $w_i = u_i \alpha_i$ so dass u_i nur aus Terminalsymbolen besteht und α_0 mit Nichtterminal beginnt oder leer ist. Jedes u_i ist Präfix von w. Sei v_i das zugehörige Suffix, also $w = u_i v_i$.

Wir zeigen, dass

$$(q, w, S) = (q, v_0, \alpha_0) \vdash_{\mathcal{A}}^* (q, v_1, \alpha_1) \vdash_{\mathcal{A}}^* \cdots \vdash_{\mathcal{A}}^* (q, v_n, \alpha_n) = (q, \varepsilon, \varepsilon).$$

Sei $i < n$ und $A \to \gamma$ die Produktion für $w_i \vdash_G w_{i+1}$. Also beginnt α_i mit A. Dann ergibt sich $(q, v_i, \alpha_i) \vdash_{\mathcal{A}}^* (q, v_{i+1}, \alpha_{i+1})$ durch folgende Übergänge:

- zunächst verwende $(q, \varepsilon, A, \gamma, q)$;

- verwende dann Übergänge $(q, a, a, \varepsilon, q)$ solange das oberste Stacksymbol ein Terminalsymbol a ist.

„⇐“: Gelte umgekehrt

$$(q, w, S) = (q, v_0, \alpha_0) \vdash_{\mathcal{A}}^* (q, v_1, \alpha_1) \vdash_{\mathcal{A}}^* \cdots \vdash_{\mathcal{A}}^* (q, v_n, \alpha_n) = (q, \varepsilon, \varepsilon).$$

Wir können o. B. d. A. annehmen, dass

- jede der Teilberechnungen $(q, v_i, \alpha_i) \vdash_{\mathcal{A}}^* (q, v_{i+1}, \alpha_{i+1})$ genau einen Übergang der Form $(q, \varepsilon, A, \gamma, q)$ und beliebig viele der Form $(q, a, a, \varepsilon, q)$ verwendet und
- das erste Symbol von α_i ein Nichtterminal ist.

Jedes v_i ist Suffix von w. Sei u_i das zugehörige Präfix, also $w = u_i v_i$.

Wir zeigen, dass

$$S = u_0\alpha_0 \vdash_G u_1\alpha_1 \vdash_G \cdots \vdash_G u_n\alpha_n.$$

Sei $i < n$ und $(q, \varepsilon, A, \gamma, q)$ der erste Übergang in $(q, v_i, \alpha_i) \vdash_G^* (q, v_{i+1}, \alpha_{i+1})$ gefolgt von $(q, a_1, a_1, \varepsilon, q), \ldots, (q, a_m, a_m, \varepsilon, q)$. Dann hat α_i die Form $A\alpha_i'$ und γ die Form $a_1 \cdots a_m B\gamma'$ mit $B \in N$. Anwendung der Regel $A \to \gamma$ auf $u_i\alpha_i$ ergibt $u_i\gamma\alpha_i' = u_i a_1 \cdots a_m B\gamma'\alpha_i' = u_{i+1}\alpha_{i+1}$.

„2 ⇒ 1“.

Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ ein PDA mit Akzeptanz per leerem Keller. Wir wollen eine kontextfreie Grammatik G konstruieren mit $L(G) = L(\mathcal{A})$. Die Nichtterminalsymbole von G sind alle Tripel $[p, Z, q] \in Q \times \Gamma \times Q$.

Idee:

Es soll gelten: $[p, Z, q] \vdash_G^* u \in \Sigma^*$ gdw.

1. $\mathcal{A}$ erreicht *vom Zustand p aus den Zustand q* (in beliebig vielen Schritten)
2. durch *Lesen von u* auf dem Eingabeband und
3. *Löschen von Z* aus dem Keller (ohne dabei die Symbole unter Z anzutasten).

Die Produktionen beschreiben die intendierte Bedeutung jedes Nichtterminals $[p, Z, q]$ auf folgende Weise: um q von p aus unter Lesen der Eingabe $u = av$ und Löschen des Stacksymbols Z zu erreichen, braucht man einen Übergang $(p, a, Z, X_1 \cdots X_n, p_0)$, die Z durch Symbole $X_1 \cdots X_n$ ersetzt und das erste Symbol a von u liest (hier ist auch $a = \varepsilon$ möglich). Nun muss man noch den neu erzeugten Stackinhalt $X_1 \cdots X_n$ loswerden. Das tut man Schritt für Schritt mittels Zerlegung des Restwortes $v = v_1 \cdots v_n$ und über Zwischenzustände $p_1, \ldots, p_{n-1}$, so dass

- $[p_0, X_1, p_1]$ unter Lesen von v_1;
- $[p_1, X_2, p_2]$ unter Lesen von v_2;
- $\cdots$

- $[p_{n-1}, X_n, q]$ unter Lesen von v_n.

(Hier ist $[p, X, q]$ jeweils gemäss den obigen Punkten 1-3 zu lesen).

Da die benötigten Zwischenzustände $p_1, \ldots, p_{n-1}$ nicht bekannt sind, fügt man einfach eine Produktion

$$[p, Z, q] \longrightarrow a[p_0, X_1, p_1] \cdots [p_{n-1}, X_n, q]$$

für *alle möglichen* Zustandsfolgen $p_1, \ldots, p_{n-1}$ hinzu. In einer Ableitung der resultierenden Grammatik kann man dann die Regel mit den „richtigen" Zwischenzuständen auswählen (und eine falsche Auswahl führt einfach zu keiner Ableitung).

Präzise Definition:

$$\begin{aligned}
G &:= (N, \Sigma, P, S) \text{ mit} \\
N &:= \{S\} \cup \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \\
P &:= \{S \longrightarrow [q_0, Z_0, q] \mid q \in Q\} \cup \\
&\quad \{[p, Z, q] \longrightarrow a \mid (p, a, Z, \varepsilon, q) \in \Delta \text{ mit } a \in \Sigma \cup \{\varepsilon\}\ \} \cup \\
&\quad \{[p, Z, q] \longrightarrow a[p_0, X_1, p_1][p_1, X_2, p_2] \ldots [p_{n-1}, X_n, q] \mid \\
&\qquad (p, a, Z, X_1 \ldots X_n, p_0) \in \Delta,\ q \in Q \text{ sowie} \\
&\qquad a \in \Sigma \cup \{\varepsilon\}, \\
&\qquad p_1, \ldots, p_{n-1} \in Q, \\
&\qquad n \geq 1\}
\end{aligned}$$

Beachte:

Für $n = 0$ hat man den Übergang $(p, a, Z, \varepsilon, q) \in \Delta$, welcher der Produktion $[p, Z, q] \longrightarrow a$ entspricht.

Behauptung:

Für alle $p, q \in Q, u \in \Sigma^*, Z \in \Gamma, \gamma \in \Gamma^*$ gilt:

$$(\star) \qquad [p, Z, q] \vdash_G^* u \ \text{ gdw. } \ (p, u, Z) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$$

Für $p = p_0$ und $Z = Z_0$ folgt daraus:

$$\vdash_G [q_0, Z_0, q] \vdash_G^* u \ \text{ gdw. } \ S(q_0, u, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$$

d. h. $u \in L(G)$ gdw. $u \in L(\mathcal{A})$.

Der Beweis dieser Behauptung kann durch Induktion über die Länge der Konfigurationsfolge („$\Rightarrow$") bzw. über die Länge der Ableitung („$\Leftarrow$") geführt werden.

❑

Beispiel:

Gegeben sei der PDA aus Beispiel 11.3, diesmal aber mit Akzeptanz per leerem Keller;

also ist f zwar weiterhin ein Zustand, aber kein akzeptierender mehr. Dieser PDA erkennt ebenfalls $\{a^n b^n \mid n \geq 1\}$.[3] Die Berechnung dieses PDA

$$(q_0, ab, Z_0) \overset{(q_0,a,Z_0,ZZ_0,q_0)}{\vdash_{\mathcal{A}}} (q_0, b, ZZ_0) \overset{(q_0,b,Z,\varepsilon,q_1)}{\vdash_{\mathcal{A}}} (q_1, \varepsilon, Z_0) \overset{(q_1,\varepsilon,Z_0,\varepsilon,f)}{\vdash_{\mathcal{A}}} (f, \varepsilon, \varepsilon)$$

entspricht der Ableitung

$$S \vdash_G [q_0, Z_0, f] \vdash_G a[q_0, Z, q_1][q_1, Z_0, f] \vdash_G ab[q_1, Z_0, f] \vdash_G ab.$$

Aus Satz 11.9 ergibt sich leicht folgendes Korollar. Wir nennen zwei PDAs $\mathcal{A}$ und $\mathcal{A}'$ *äquivalent*, wenn $L(\mathcal{A}) = L(\mathcal{A}')$.

Korollar 11.10

Zu jedem PDA $\mathcal{A}$ gibt es einen PDA $\mathcal{A}'$ mit Akzeptanz per leerem Keller, so dass $L(\mathcal{A}) = L(\mathcal{A}')$ und $\mathcal{A}'$ nur einen Zustand hat.

Beweis. Gegeben einen PDA $\mathcal{A}$ kann man erst die Konstruktion aus dem Teil „$2 \Rightarrow 1$“ des Beweises von Satz 11.9 anwenden und dann die aus dem Teil „$1 \Rightarrow 2$“. Man erhält einen äquivalenten PDA, der nach Konstruktion nur einen einzigen Zustand enthält. ❑

Wegen der gerade gezeigten Äquivalenz zwischen kontextfreien Sprachen und PDA-akzeptierbaren Sprachen kann man Eigenschaften von kontextfreien Sprachen mit Hilfe von Eigenschaften von Kellerautomaten zeigen. Als Beispiel betrachten wir den Durchschnitt von kontextfreien Sprachen mit regulären Sprachen. Wir wissen: Der Durchschnitt zweier kontextfreier Sprachen muss nicht kontextfrei sein. Dahingegen gilt:

Satz 11.11

Es sei $L \subseteq \Sigma^*$ kontextfrei und $R \subseteq \Sigma^*$ regulär. Dann ist $L \cap R$ kontextfrei.

Beweis. Es sei $L = L(\mathcal{A})$ für einen PDA $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F)$ (also mit akzeptierenden Zuständen) und $R = L(\mathcal{A}')$ für einen DEA $\mathcal{A}' = (Q', \Sigma, q_0', \delta', F')$. Wir wenden eine Produktkonstruktion an, um einen PDA zu konstruieren, der $L \cap R$ erkennt:

$$\begin{aligned}
\mathcal{B} &:= (Q \times Q', \Sigma, \Gamma, (q_0, q_0'), Z_0, \Delta', F \times F') \text{ mit} \\
\Delta' &:= \{((p,p'), a, Z, \gamma, (q,q')) \mid (p, a, Z, \gamma, q) \in \Delta \text{ und } \delta(p', a) = q'\} \cup \\
&\quad\ \{((p,p'), \varepsilon, Z, \gamma, (q,p')) \mid (p, \varepsilon, Z, \gamma, q) \in \Delta\}
\end{aligned}$$

Man zeigt nun leicht (durch Induktion über k):
$((p,p'), uv, \gamma) \vdash_{\mathcal{B}}^k ((q,q'), v, \beta)$ gdw. $(p, uv, \gamma) \vdash_{\mathcal{A}}^k (q, v, \beta)$ und $p' \xrightarrow{u}_{\mathcal{A}'} q'$ ❑

Beachte: mit zwei PDAs als Eingabe funktioniert eine solche Produktkonstruktion nicht, da die beiden PDAs den Keller im Allgemeinen nicht „synchron“ nutzen (der eine kann das obere Kellersymbol löschen, während der andere Symbole zum Keller hinzufügt).

[3] Das klappt bei diesem PDA gerade, weil der Übergang von q_2 zu f der einzige ist, bei dem das Kellerstartsymbol gelöscht wird.

Deterministische Kellerautomaten

Analog zu endlichen Automaten kann man auch bei Kellerautomaten eine deterministische Variante betrachten. Intuitiv ist der PDA aus Beispiel 11.3 deterministisch, da es zu jeder Konfiguration höchstens eine Folgekonfiguration gibt. Der PDA aus Beispiel 11.4 ist hingegen nichtdeterministisch, da er die Wortmitte „rät“. Interessanterweise stellt es sich heraus, dass bei im Gegensatz zu DEAs/NEAs bei PDAs die deterministische Variante echt schwächer ist als die nichtdeterministische. Daher definieren die deterministischen PDAs eine eigene Sprachklasse, die *deterministisch kontextfreien Sprachen*.

Deterministische PDAs akzeptieren immer per akzeptierendem Zustand (aus gutem Grund, wie wir noch sehen werden).

Definition 11.12 (deterministischer Kellerautomat)

Ein *deterministischer Kellerautomat (dPDA)* ist ein PDA

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F),$$

der folgende Eigenschaften erfüllt:

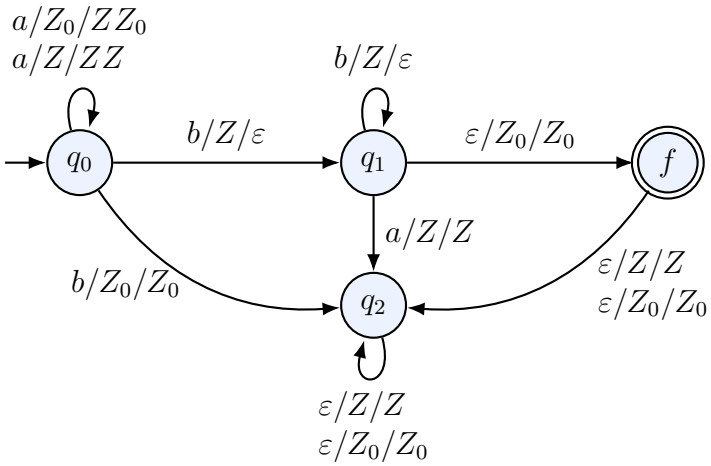
1. Für alle $q \in Q$, $a \in \Sigma$ und $Z \in \Gamma$ gibt es genau einen Übergang der Form (q, a, Z, γ, q') oder $(q, \varepsilon, Z, \gamma, q')$.
2. Wenn ein Übergang das Kellerstartsymbol Z_0 entfernt, so muss er es direkt wieder zurückschreiben; alle Übergänge, in denen Z_0 vorkommt, müssen also die Form $(q, a, Z_0, \gamma Z_0, q')$ haben, mit $\gamma \in \Gamma^*$ beliebig.

Man kann leicht sehen, dass es zu jeder Konfiguration eines dPDA, bei der der Keller nicht leer ist, genau eine Folgekonfiguration gibt. Die Bedingung 2 ist notwendig, damit der Keller tatsächlich nie leer wird (denn eine Konfiguration mit leerem Keller kann keine Folgekonfiguration haben). Wie ein PDA (mit akzeptierenden Zuständen) akzeptiert ein dPDA $\mathcal{A}$ ein Wort w gdw. $(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (q_f, \varepsilon, \gamma)$ für ein $q_f \in F$ und $\gamma \in \Gamma^*$.

Beispiel 11.13

Als Beispiel für einen dPDA betrachte die folgende Variante des PDAs aus Beispiel 11.3, die ebenfalls $L = \{a^n b^n \mid n \geq 1\}$ erkennt:

- $Q = \{q_0, q_1, q_2, f\}$
- $\Gamma = \{Z, Z_0\}$
- $\Sigma = \{a, b\}$
- $F = \{f\}$
- $\Delta =$



Im Unterschied zum PDA aus Beispiel 11.3 entfernt der Übergang von q_1 nach f nicht mehr Z_0 vom Keller (weil das nicht erlaubt wäre), und der „Papierkorbzustand“ q_2 ist hinzugekommen.

Da die Arbeitsweise von dPDAs durchaus etwas subtil ist, hier zwei Hinweise zum vorhergehenden Beispiel:

- Auf manchen Eingaben gibt es mehr als eine Berechnung. Als Beispiel betrachte man die Eingabe *aabb*. Nachdem diese gelesen wurde, befindet sich der PDA im Zustand q_1, der kein akzeptierender Zustand ist. Man kann jedoch den akzeptierenden Zustand f in einem weiteren Schritt erreichen, also wird die Eingabe *aabb* akzeptiert. Danach kann man im Prizip noch den nicht-akzeptierenden Zustand q_2 erreichen und in diesem beliebig oft loopen (was aber nicht zur Akzeptanz beiträgt – und damit auch nicht zur erkannten Sprache).
- Trotz des Determinismus können manche Eingaben wie z. B. *ba* nicht vollständig gelesen werden.

Eine interessante Eigenschaft von deterministischen PDAs ist, dass für sie das Wortproblem in *Linearzeit* entschieden werden kann. Aus diesem Grund spielen dPDAs im Gebiet des Compilerbaus eine wichtige Rolle.

Definition 11.14

Eine formale Sprache L heißt *deterministisch kontextfrei*, wenn es einen dPDA $\mathcal{A}$ gibt mi $L(\mathcal{A}) = L$. Die Menge aller deterministisch kontextfreien Sprachen ist $\mathcal{L}_2^d$.

Folgende Einordnung der deterministisch kontextfreien Sprachen ist leicht vorzunehmen.

Satz 11.15

$\mathcal{L}_3 \subset \mathcal{L}_2^d \subseteq \mathcal{L}_2$.

Beweis. Es gilt $\mathcal{L}_3 \subset \mathcal{L}_2^d$, da jeder DEA $\mathcal{A}$ als dPDA ohne ε-Übergänge und mit nur einem Kellersymbol Z_0 betrachtet werden kann, der zudem seinen Keller nie modifiziert:

aus jedem Übergang $\delta(q, a) = q'$ des DEA wird der Übergang (q, a, Z_0, Z_0, q') des dPDA. Die Inklusion ist echt, da mit Beispiel 11.13 $L = \{a^n b^n \mid n \geq 1\} \in \mathcal{L}_2^d$, wohingegen $L \notin \mathcal{L}_3$. Die Inklusion $\mathcal{L}_2^d \subseteq \mathcal{L}_2$ gilt wegen Satz 11.6. ❑

Wie bereits erwähnt sind dPDAs echt schwächer als PDAs, d. h. die deterministisch kontextfreien Sprachen sind eine *echte Teilmenge* der kontextfreien Sprachen. Der Beweis beruht auf dem folgenden Resultat. Wir verzichten hier auf den etwas aufwändigen Beweis und verweisen z. B. auf [Koz07].

Satz 11.16

$\mathcal{L}_2^d$ ist unter Komplement abgeschlossen.

Zur Erinnerung: die kontextfreien Sprachen selbst sind mit Korollar 10.6 nicht unter Komplement abgeschlossen. Man kann zeigen, dass die deterministisch kontextfreien Sprachen nicht unter Schnitt, Vereinigung, Konkatenation und Kleene-Stern abgeschlossen sind.

Satz 11.17

$\mathcal{L}_2^d \subset \mathcal{L}_2$.

Beweis. Mit Satz 11.16 ist der folgende sehr einfache Beweis möglich: wäre $\mathcal{L}_2^d = \mathcal{L}_2$, so wäre mit Satz 11.16 $\mathcal{L}_2$ unter Komplement abgeschlossen, was jedoch ein Widerspruch zu Korollar 10.6 ist.

Dieser Beweis liefert jedoch keine konkrete Sprache, die kontextfrei aber nicht deterministisch kontextfrei ist. Eine solche findet man beispielsweise wie folgt: in der Übung zeigen wir, dass die Sprache

$$L = \{w \in \{a, b\}^* \mid \forall v \in \{a, b\}^* : w \neq vv\}$$

kontextfrei ist (durch Angeben einer Grammatik), ihr Komplement

$$\overline{L} = \{w \in \{a, b\}^* \mid \exists v \in \{a, b\}^* : w = vv\}$$

aber nicht (Pumping-Lemma für kontextfreie Sprachen). Wäre $L \in \mathcal{L}_2^d$, so wäre mit Satz 11.16 auch $\overline{L} \in \mathcal{L}_2^d \subseteq \mathcal{L}_2$, womit ein Widerspruch hergestellt ist. ❑

Auch die Sprache $\{ww^R \mid w \in \{a, b\}^*\}$ aus Beispiel 11.4 ist kontextfrei, aber nicht deterministisch kontextfrei; der Beweis ist allerdings aufwändig. Intuitiv ist der Grund aber, dass das nicht-deterministische „Raten“ der Wortmitte essentiell ist. Dies wird auch dadurch illustriert, dass die Sprache $\{wcw^R \mid w \in \{a, b\}^*\}$, bei der die Wortmitte explizit durch das Symbol c angezeigt wird, sehr einfach mit einem dPDA erkannt werden kann.

Zum Abschluss bemerken wir noch, dass Akzeptanz per leerem Keller bei dPDAs zu Problemen führt.

Lemma 11.18

Es gibt keinen dPDA mit Akzeptanz per leerem Keller, der die endliche (also reguläre) Sprache $L = \{a, aa\}$ erkennt.

Beweis. Angenommen, der dPDA $\mathcal{A}$ mit Akzeptanz per leerem Keller erkennt L. Da $a \in L$, gibt es eine Konfigurationsfolge

$$\Omega = (q_0, a, Z_0) \vdash_{\mathcal{A}} (q_1, w_1, \gamma_1) \vdash_{\mathcal{A}} \cdots \vdash_{\mathcal{A}} (q_n, w_n, \gamma_n)$$

mit $w_n = \gamma_n = \varepsilon$. Also gibt es auch eine Konfigurationsfolge

$$\Omega' = (q_0, aa, Z_0) \vdash_{\mathcal{A}} (q_1, u_1, \gamma_1) \vdash_{\mathcal{A}} \cdots \vdash_{\mathcal{A}} (q_n, u_n, \gamma_n)$$

mit $u_n = a$ und $\gamma_n = \varepsilon$; da $\mathcal{A}$ deterministisch ist, ist das die *einzige* Konfigurationsfolge, die das Präfix a der Eingabe aa verarbeitet. Wegen $\gamma_n = \varepsilon$ hat (q_n, u_n, γ_n) keine Folgekonfiguration, also wird aa nicht akzeptiert (dies kann per Definition nur nach Lesen der gesamten Eingabe geschehen). Widerspruch. ❑

Man kann diese Probleme beheben, indem man ein explizites Symbol für das Wortende einführt, das auch in Übergängen von dPDAs verwendet werden kann. Mit einem solchen Symbol sind Akzeptanz per akzeptierendem Zustand und Akzeptanz per leerem Keller auch für dPDAs äquivalent.

12. Die Struktur kontextfreier Sprachen

In diesem Abschnitt gehen wir der Frage nach, was die kontextfreien von den regulären Sprachen unterscheidet. Wir können kontextfreie Sprachen mit Typ-2-Grammatiken erzeugen und mit Kellerautomaten erkennen; das Pumping-Lemma zeigt die Grenzen der kontextfreien Sprachen auf. Eine wichtige Erkenntnis war dabei, dass PDAs (bzw. kontextfreie Grammatiken) unbeschränkt zählen können, NEAs jedoch nicht. Wir haben auch viele einzelne Beispiele kennen gelernt, wie z. B.

- $\{a^n b^n \mid n \geq 0\}$
- Klammersprachen, auch gennant *Dyck-Sprachen,*[4] die durch kontextfreie Grammatiken der Form

$$S \longrightarrow \underset{1}{(}S\underset{1}{)},\ \ldots,\ S \longrightarrow \underset{n}{(}S\underset{n}{)},\ S \longrightarrow SS,\ S \longrightarrow \varepsilon$$

  erzeugt werden.

Aber haben wir damit wirklich schon ein umfassendes Bild der kontextfreien Sprachen? Gibt es ganz andere Arten kontextfreier Sprachen, die wir noch nicht gesehen haben?

In diesem Abschnitt wollen wir zeigen, dass jede kontextfreie Sprache dargestellt werden kann als der Schnitt einer Dyck-Sprache und einer regulären Sprache, plus „ein wenig Umbenennung“.

Beispiel 12.1

Es gilt $\{a^n b^n \mid n \geq 0\} = h(D_1 \cap R)$, wobei

- D_1 die *Dyck-Sprache mit einem Klammerpaar* ist, erzeugt durch die kontextfreie Grammatik $G = (N, \Sigma, P, S)$ mit $N = \{S\}$, $\Sigma = \{(,)\}$ und $P = \{S \longrightarrow (S),\ S \longrightarrow SS,\ S \longrightarrow \varepsilon\}$;
- R die reguläre Sprache $(^*)^*$ ist;
- h ein Homomorphismus ist, die das Symbol `(` in a und das Symbol `)` in b umbenennt.

Das zentrale Resultat dieses Abschnitts besagt nun, dass jede kontextfreie Sprache das homomorphe Bild des Schnittes einer Dyck-Sprache und einer regulären Sprache ist. Genauer:

Satz 12.2 (Chomsky-Schützenberger)

Für jede kontextfreie Sprache L gibt es eine Dyck-Sprache D, eine reguläre Sprache R und einen Homomorphismus h, so dass:

$$L = h(D \cap R)$$

[4] Dyck-Sprachen sind nach Walther von Dyck (1856–1934), einem deutschen Mathematiker, benannt und bilden die Grundlage von Programmiersprachen, HTML und XML.

Beweis. Sei L eine kontextfreie Sprache und $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik mit $L(G) = L$. Wegen Satz 9.15 können wir o. B. d. A. annehmen, dass G *in Chomsky-Normalform* ist; jede Produktion $\pi \in P$ hat also die Form

(1) $A \longrightarrow BC$ mit $A, B, C \in N$ oder

(2) $A \longrightarrow a$ mit $A \in N$ und $a \in \Sigma$.

Wir ignorieren dabei die mögliche Produktion $S \longrightarrow \varepsilon$. Für jede Produktion $\pi \in P$ führen wir nun zwei neue Klammerpaare $\overset{1}{\underset{\pi}{(}}, \overset{1}{\underset{\pi}{)}}, \overset{2}{\underset{\pi}{(}}, \overset{2}{\underset{\pi}{)}}$ ein und definieren:

$$\pi' = \begin{cases} A \longrightarrow \overset{1}{\underset{\pi}{(}} B \overset{1}{\underset{\pi}{)}} \overset{2}{\underset{\pi}{(}} C \overset{2}{\underset{\pi}{)}} & \text{wenn } \pi = A \longrightarrow BC \\ A \longrightarrow \overset{1}{\underset{\pi}{(}} \overset{1}{\underset{\pi}{)}} \overset{2}{\underset{\pi}{(}} \overset{2}{\underset{\pi}{)}} & \text{wenn } \pi = A \longrightarrow a \end{cases}$$

Aus diesen neuen Produktionen bilden wir nun die Grammatik $G' = (N, \Gamma, P', S)$ mit

$$\Gamma' = \{\overset{1}{\underset{\pi}{(}}, \overset{1}{\underset{\pi}{)}}, \overset{2}{\underset{\pi}{(}}, \overset{2}{\underset{\pi}{)}} \mid \pi \in P\} \quad \text{und}$$
$$P' = \{\pi' \mid \pi \in P\}.$$

Des Weiteren betrachten wir D_Γ, die Dyck-Sprache mit den Klammern aus Γ.

Anhand der Definition der π' sieht man nun leicht, dass $L(G') \subseteq D_\Gamma$ ist; die Umkehrung gilt jedoch nicht: zum Beispiel sind die Klammerwörter

$$\overset{1}{\underset{\pi}{(}}\overset{1}{\underset{\pi}{(}}\overset{1}{\underset{\pi}{)}}\overset{1}{\underset{\pi}{)}} \quad \text{und} \quad \overset{2}{\underset{\pi}{(}}\overset{2}{\underset{\pi}{)}}\overset{1}{\underset{\pi}{(}}\overset{1}{\underset{\pi}{)}}$$

in D_Γ, aber nicht von G' erzeugbar.

Alle Wörter $w \in L(G')$ erfüllen jedoch folgende zusätzliche Eigenschaften.

1. Auf jedes $\overset{1}{\underset{\pi}{)}}$ folgt $\overset{2}{\underset{\pi}{(}}$.
2. Auf $\overset{2}{\underset{\pi}{)}}$ folgt nie eine öffnende Klammer.
3. Wenn $\pi = A \longrightarrow BC$, dann
   - folgt auf $\overset{1}{\underset{\pi}{(}}$ immer $\overset{1}{\underset{\rho}{(}}$ mit $\rho = B \longrightarrow \cdots$;
   - folgt auf $\overset{2}{\underset{\pi}{(}}$ immer $\overset{1}{\underset{\sigma}{(}}$ mit $\sigma = C \longrightarrow \cdots$.
4. Wenn $\pi = A \longrightarrow a$, dann folgt auf $\overset{1}{\underset{\pi}{(}}$ immer $\overset{1}{\underset{\pi}{)}}$ und auf $\overset{2}{\underset{\pi}{(}}$ immer $\overset{2}{\underset{\pi}{)}}$.

Diese Eigenschaften sind direkt aus der Definition der π' einzusehen; man kann sie recht bequem per Induktion über die Anzahl der Regelanwendungen beim Ableiten von w beweisen.

Weiterhin gilt für jedes Nichtterminal $A \in N$ und alle Wörter w mit $A \vdash_{G'}^* :$

5_A. w beginnt mit $\overset{1}{\underset{\pi}{(}}$, wobei $\pi = A \longrightarrow \cdots$

Eigenschaften 1–4 und 5_A lassen sich jedoch durch reguläre Ausdrücke beschreiben! Für jedes $A \in N$ ist also die Sprache

$$R_A := \{w \in \Gamma^* \mid w \text{ erfüllt Eigenschaften 1–4 und } 5_A\}$$

regulär. Wir wollen als nächstes zeigen, dass sich $L(G')$ von D_Γ *nur* durch Eigenschaften 1–4 und 5_A unterscheidet, Dann ist nämlich $L(G') = D_\Gamma \cap R_S$, und wir können durch Umbenennen der Klammern in Terminale auch L wie gewünscht darstellen. Zu diesem Zweck zeigen wir:

Behauptung. Für alle $A \in N$ gilt: $\quad A \vdash^*_{G'} w \quad$ gdw. $\quad w \in D_\Gamma \cap R_A$

„⇒“ zeigt man leicht per Induktion über die Anzahl der Regelanwendungen beim Ableiten von w.

„⇐“: per Induktion über die Länge von w.

Induktionsanfang. $w = \varepsilon$ erfüllt Eigenschaft 5_A nicht, ist also nicht in R_A. Damit ist die Behauptung trivialerweise erfüllt.

Induktionsschritt. Sei $w \in D_\Gamma \cup R_A$ mit $w \neq \varepsilon$. Weil w wohlgeklammert ist und Eigenschaften 1–4 und 5_A erfüllt, zeigt man leicht:

$$w = \overset{1}{\underset{\pi}{(}} u \overset{1}{\underset{\pi}{)}} \overset{2}{\underset{\pi}{(}} v \overset{2}{\underset{\pi}{)}} \quad \text{für } u, v \in \Gamma^* \text{ und } \pi = A \longrightarrow \cdots$$

Nun kann π eine von zwei Formen haben.

- Wenn $\pi = A \longrightarrow BC$, dann sind auch u und v wohlgeklammert und erfüllen die Eigenschaften 1–4 und 5_B bzw. 5_C (z. B. folgt 5_B für u aus 3 für w). Die Induktionsvoraussetzung liefert nun $B \vdash^*_{G'} u$ und $C \vdash^*_{G'} v$, und damit erhalten wir

$$A \;\vdash_{G'}\; \overset{1}{\underset{\pi}{(}} B \overset{1}{\underset{\pi}{)}} \overset{2}{\underset{\pi}{(}} C \overset{2}{\underset{\pi}{)}} \;\vdash^*_{G'}\; \overset{1}{\underset{\pi}{(}} u \overset{1}{\underset{\pi}{)}} \overset{2}{\underset{\pi}{(}} v \overset{2}{\underset{\pi}{)}} \;=\; w$$

  wie gewünscht.

- Wenn $\pi = A \longrightarrow a$, dann liefert Eigenschaft 4 für w, dass $u = v = \varepsilon$ ist. Nach Definition von G' erhalten wir

$$A \;\vdash_{G'}\; \overset{1}{\underset{\pi}{(}} \overset{1}{\underset{\pi}{)}} \overset{2}{\underset{\pi}{(}} \overset{2}{\underset{\pi}{)}} \;=\; w$$

  wie gewünscht.

Aus der Behauptung folgt nun insbesondere:

$$L(G') = D_\Gamma \cap R_S$$

Wir definieren nun den Homomorphismus h wie folgt.

- $h\Big(\begin{smallmatrix}1\\(\\\pi\end{smallmatrix}\Big) = h\Big(\begin{smallmatrix}1\\)\\\pi\end{smallmatrix}\Big) = h\Big(\begin{smallmatrix}2\\(\\\pi\end{smallmatrix}\Big) = h\Big(\begin{smallmatrix}2\\)\\\pi\end{smallmatrix}\Big) = \varepsilon, \qquad$ falls $\pi = A \longrightarrow BC$
- $h\Big(\begin{smallmatrix}1\\(\\\pi\end{smallmatrix}\Big) = a$ und $h\Big(\begin{smallmatrix}1\\)\\\pi\end{smallmatrix}\Big) = h\Big(\begin{smallmatrix}2\\(\\\pi\end{smallmatrix}\Big) = h\Big(\begin{smallmatrix}2\\)\\\pi\end{smallmatrix}\Big) = \varepsilon, \qquad$ falls $\pi = A \longrightarrow a$.

Für jede Produktion $\pi \in P$ gilt dann: h angewendet auf π' ergibt π. Damit erhalten wir wie gewünscht:

$$L(G) = h(L(G')) = h(D_\Gamma \cap R_S)$$

❏

Anschaulich gesprochen bedeutet der Satz von Chomsky-Schützenberger, dass jede kontextfreie Sprache „quasi“ (genauer: modulo Umbenennen durch Homomorphismen) eine durch eine reguläre Sprache beschreibbare Teilmenge einer Dyck-Sprache ist.

Überblick Automaten und Sprachen (Teile I & II)

Zum Abschluss von Teil II geben wir hier einen Überblick über die in *Automaten und Sprachen* (Teile I & II) behandelten Themen.

Themenübersicht

- Sprachklassen (Chomsky-Hierarchie, Typen 0–3)
  - $\mathcal{L}_3$: regulär = erkennbar = rechtslinear
  - $\mathcal{L}_2^d$: deterministisch kontextfrei
  - $\mathcal{L}_2$: kontextfrei
  - $\mathcal{L}_1$: kontextsensitiv
  - $\mathcal{L}_0$
- Automatenmodelle zur Beschreibung von Sprachen
  - nichtdeterministische endliche Automaten (NEAs)
  - deterministische endliche Automaten (DEAs)
  - NEAs mit ε- und Wortübergängen
  - Kellerautomaten (PDAs)
  - deterministische Kellerautomaten (dPDAs)
- Andere Mechanismen, um Sprachen endlich zu beschreiben
  - reguläre Ausdrücke
  - Grammatiken (Typen 0–3: allgemeine, monotone, kontextfreie, reguläre)
- Eigenschaften von Sprachklassen
  - Abschlusseigenschaften
  - Entscheidbarkeit und Komplexität von Problemen
- Konstruktionen und Beweistechniken
  - Potenzmengenkonstruktion
  - Produktautomat
  - Quotientenautomat
  - Nerode-Rechtskongruenz
  - Pumping-Lemma für reguläre Sprachen
  - Pumping-Lemma für kontextfreie Sprachen
  - Normalformen von Grammatiken
  - etc.

Überblick Abschlusseigenschaften und Entscheidungsprobleme

Siehe Anhang **??**.

III. Berechenbarkeit

Einführung

Aus der Sicht der Theorie der *formalen Sprachen* (Teile I + II dieses Skriptes) geht es in diesem Teil darum, die Typ-0- und die Typ-1-Sprachen zu studieren und folgende Fragen zu beantworten:

- Was sind geeignete Automatenmodelle?
- Welche Abschlusseigenschaften gelten?
- Wie lassen sich die zentralen Entscheidungsprobleme (Wortproblem, Leerheitsproblem, Äquivalenzproblem) lösen?

Eine viel wichtigere Sicht auf den Inhalt von Teil III ist aber eine andere, nämlich als Einführung in die *Theorie der Berechenbarkeit*. Hierbei handelt es sich um eine zentrale Teildisziplin der theoretischen Informatik, in der Fragen wie die folgenden studiert werden:

- Gibt es Probleme, die prinzipiell nicht berechenbar sind?
- Was für Berechnungsmodelle gibt es?
- Sind alle Berechnungsmodelle (verschiedene Rechnerarchitekturen, Programmiersprachen, mathematische Modelle) gleichmächtig?
- Welche Ausdrucksmittel von Programmiersprachen sind verzichtbar, weil sie zwar der Benutzbarkeit dienen, aber die Berechnungsstärke nicht erhöhen?

In diesem Zusammenhang interessieren wir uns für

- die *Berechnung (partieller oder totaler) Funktionen*

  $$f : \mathbb{N}^k \to \mathbb{N}$$

  wobei k die *Stelligkeit* der Funktion bezeichnet; Beispiele sind etwa:
  - die konstante Nullfunktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(x) = 0$ für alle $x \in \mathbb{N}$
  - die binäre Additionsfunktion $f : \mathbb{N}^2 \to \mathbb{N}$ mit $f(x, y) = x + y$

  Manchmal betrachten wir auch Funktionen $f : (\Sigma^*)^k \to \Sigma^*$, wobei Σ ein Alphabet ist. Ein Beispiel ist die Konkatenationsfunktion.

- *Entscheidungsprobleme*, also Fragen, die nur mit „ja“ oder „nein“ beantwortet werden können, wie beispielsweise das Leerheitsproblem für kontextfreie Grammatiken: gegeben eine Grammatik G, ist $L(G) = \emptyset$?

  Wir werden Entscheidungsprobleme als Mengen $P \subseteq \Sigma^*$ formalisieren, für ein geeignetes Alphabet Σ. Ein Entscheidungsproblem ist also nichts anderes als eine formale Sprache! Das erwähnte Leerheitsproblem für kontextfreie Grammatiken würde dann dargestellt als Menge

  $$\{\mathsf{code}(G) \mid G \text{ ist kontextfreie Grammatik mit } L(G) = \emptyset\},$$

  wobei $\mathsf{code}(G)$ eine geeignete Kodierung der Grammatik G als Wort ist.

Intuitiv heißt eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ *berechenbar*, wenn es einen Algorithmus gibt, der bei Eingabe $(x_1, \ldots, x_k) \in \mathbb{N}^k$ nach endlicher Zeit den Funktionswert $f(x_1, \ldots, x_k)$ ausgibt. Ein Entscheidungsproblem $P \subseteq \Sigma^*$ heißt *entscheidbar*, wenn es einen Algorithmus gibt, der bei Eingabe w nach endlicher Zeit „ja“ zurückgibt, wenn $w \in P$, und „nein“ sonst. Man beachte: diese Beschreibungen von Berechenbarkeit und Entscheidbarkeit sind so noch *keine* präzisen Definitionen, da wir nicht eindeutig festgelegt haben, was unter dem Begriff „Algorithmus“ zu verstehen ist.

Um sich klarzumachen, dass eine gegebene Funktion berechenbar oder ein Problem entscheidbar ist, genügt es, einen entsprechenden Algorithmus für die Funktion bzw. das Problem anzugeben. Dies kann in Form eines Pascal-, Java- oder C-*Programms* oder in Form einer *abstrakten Beschreibung* der Vorgehensweise bei der Berechnung geschehen. Zum Beispiel haben wir in den Teilen I und II die Entscheidbarkeit von verschiedenen Problemen (Wortproblem, Leerheitsproblem, Äquivalenzproblem, ...) dadurch begründet, dass wir auf abstrakte Weise beschrieben haben, wie man die Probleme mit Hilfe eines Rechenverfahrens entscheiden kann. Um etwa das Wortproblem für kontextfreie Grammatiken zu lösen, kann man zunächst die Grammatik in Chomsky-Normalform wandeln (wir haben im Detail beschrieben, wie diese Wandlung realisiert werden kann) und dann den CYK-Algorithmus anwenden (den wir in Form von Pseudocode beschrieben haben). Aus dieser und ähnlichen Beschreibungen kann man jeweils leicht ein Pascal-, Java-, etc. Programm zur Entscheidung des Problems gewinnen. Für derartige Argumente wird keine formale Definition des Begriffes „Algorithmus“ benötigt, denn jede vernünftige Definition dieser Art würde die erwähnte Programme einschließen.

Eine fundamentale Einsicht der Theorie der Berechenbarkeit ist, dass es wohldefinierte (und für die Informatik hochgradig relevante!) Funktionen gibt, die nicht berechenbar sind, und analog dazu Entscheidungsprobleme, die nicht entscheidbar sind. Beim Nachweis der *Nicht*berechenbarkeit bzw. *Nicht*entscheidbarkeit ist es nicht mehr ausreichend, einen intuitiven und nicht näher spezifizierten Begriff von Algorithmus zu verwenden: die Aussage, dass es *keinen Algorithmus für ein Entscheidungsproblem gibt*, bezieht sich implizit auf *alle* Algorithmen (für jeden Algorithmus gilt: er entscheidet nicht das betrachtete Problem). Aus diesem Grund benötigt man für das Studium der *Grenzen der Berechenbarkeit* eine formale Definition dessen, was man unter einem Algorithmus versteht. Zunächst gibt es scheinbar sehr viele Kandidaten für das zugrunde gelegte

Berechnungsmodell: Eine Programmiersprache? Welche der vielen Sprachen ist geeignet? Imperativ, funktional, objektorientiert? Verwendet man ein hardwarenahes Modell, wie etwa das Modell eines Mikroprozessors? Oder ein abstraktes mathematisches Modell?

Ein geeignetes Berechnungsmodell sollte folgende Eigenschaften erfüllen:

1) es sollte **einfach** sein, damit formale Beweise erleichtert werden (z. B. nicht die Programmiersprache Java),
2) es sollte **berechnungsuniversell** sein, d. h. alle intuitiv berechenbaren Funktionen können damit berechnet werden (bzw. alle intuitiv entscheidbaren Mengen können entschieden werden) – also keine endlichen Automaten, denn deren Berechnungsstärke ist viel zu schwach.

Wir werden drei Berechnungsmodelle betrachten:

- **Turingmaschinen** als sehr einfaches, aber dennoch berechnungsuniverselles Modell
- **WHILE-Programme** als Abstraktion imperativer Programmiersprachen (im Prinzip handelt es sich um eine möglichst einfache, aber immer noch berechnungsuniverselle solche Sprache)
- **μ-rekursive Funktionen** als funktionsbasiertes mathematisches Berechnungsmodell

Es gibt noch eine Vielzahl anderer Modelle:

- Automaten-/Maschinenmodelle: Registermaschinen, Kellerautomaten mit mindestens 2 Kellern, k-Zählermaschinen mit $k \geq 2$ usw.
- Programmiersprachen: GOTO-Programme, Java-Programme usw.
- Umformungssysteme: Typ-0-Grammatiken, Markov-Algorithmen usw.
- funktionsbasierte mathematische Modelle: λ-Kalkül usw.

Es hat sich aber interessanterweise herausgestellt, dass all diese vollkommen unterschiedlichen Modelle *äquivalent* sind, d. h. dieselbe Klasse von Funktionen berechnen (bzw. Problemen entscheiden). Zudem ist es bisher niemandem gelungen, ein formales Berechnungsmodell zu finden, das

- realistisch erscheint (also im Prinzip in der wirklichen Welt realisierbar ist),
- Funktionen berechnen kann, die in den genannten Modellen *nicht* berechenbar sind.

Aus diesen beiden Gründen geht man davon aus, dass die genannten Modelle *genau den intuitiven Berechenbarkeitsbegriff* formalisieren. Diese Überzeugung nennt man die

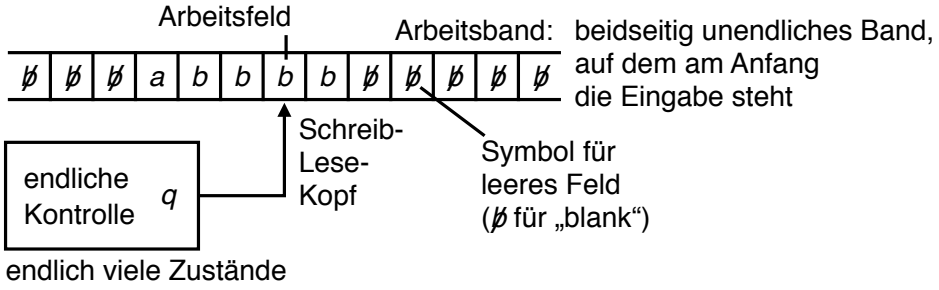
Church-Turing-These:
Die (intuitiv) berechenbaren Funktionen sind genau die mit Turingmaschinen (und damit mit WHILE-, Java-Programmen, Registermaschinen, ...) berechenbaren Funktionen.

Man könnte die These äquivalent auch für Entscheidungsprobleme formulieren. Sie ist eine *These* und kein *Satz*, da es nicht möglich ist, ihre Aussage formal zu beweisen. Dies liegt daran, dass der *intuitive* Berechenbarkeitsbegriff ja *nicht formal* definierbar ist. Es gibt aber gute Indizien, die für die Richtigkeit der These sprechen, insbesondere die Vielzahl existierender Berechnungsmodelle, die sich als äquivalent herausgestellt haben.

13. Turingmaschinen

Turingmaschinen wurden um 1936 von dem englischen Mathematiker und Informatiker Alan Turing als besonders einfaches Berechnungsmodell vorgeschlagen. In den Worten des bekannten Komplexitätstheoretikers Christos Papadimitriou: „It is amazing how little you need to have everything“. Wir verwenden Turingmaschinen einerseits als universelles Berechnungsmodell und andererseits als Werkzeug zum Definieren von formalen Sprachen. Insbesondere werden wir sehen, dass Turingmaschinen genau die Typ-0-Sprachen erkennen und eine entsprechend eingeschränkte Variante von Turingmaschinen als Automatenmodell für Typ-1-Sprachen verwendet werden kann.

Die schematische Darstellung einer Turingmaschine ist wie folgt:



Das Arbeitsband ist beidseitig unendlich. Zu jedem Zeitpunkt sind jedoch nur *endlich viele* Symbole auf dem Band verschieden von $\not\emptyset$. Das Verhalten der Turingmaschine hängt ab vom aktuellen Zustand und vom Alphabetsymbol, das sich unter dem Schreib-Lese-Kopf befindet. Ein Schritt der Maschine besteht darin, das Zeichen unter dem Schreib-Lese-Kopf zu ersetzen und dann den Kopf nach rechts oder links (oder gar nicht) zu bewegen.

Definition 13.1 (Turingmaschine)

Eine *Turingmaschine über dem Eingabealphabet* Σ hat die Form $M = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, wobei

- Q endliche *Zustandsmenge* ist,
- Σ das *Eingabealphabet* ist,
- Γ das *Arbeitsalphabet* ist mit $\Sigma \subseteq \Gamma,\ \not\emptyset \in \Gamma \setminus \Sigma$,
- $q_0 \in Q$ der *Anfangszustand* ist,
- $F \subseteq Q$ die *Menge der akzeptierenden Zustände* ist und
- $\Delta \subseteq Q \times \Gamma \times \Gamma \times \{r, l, n\} \times Q$ die *Übergangsrelation* ist.

Dabei bedeutet der Übergang $(q, a, a', \begin{matrix} r \\ l \\ n \end{matrix}, q')$:

- Im Zustand q
- mit a auf dem gerade gelesenen Feld (Arbeitsfeld)

kann die Turingmaschine M

- das Symbol a durch a' ersetzen,
- in den Zustand q' gehen und
- den Schreib-Lese-Kopf entweder um ein Feld nach rechts (r), links (l) oder gar nicht (n) bewegen.

Anstelle des Begriffes „Übergang“ benutzen wir auch oft den Begriff „*Transition*“.

Die Maschine M heißt *deterministisch*, falls es für jedes Paar $(q, a) \in Q \times \Gamma$ *höchstens ein* Tupel der Form $(q, a, \cdot, \cdot, \cdot) \in \Delta$ gibt.

NTM steht im Folgenden für (möglicherweise) nichtdeterministische Turingmaschinen und *DTM* für deterministische.

Bei einer DTM gibt es also zu jedem Berechnungszustand höchstens einen Folgezustand, während es bei einer NTM mehrere geben kann.

Einen Berechnungszustand (*Konfiguration*) einer Turingmaschine kann man beschreiben durch ein Wort $\alpha q \beta$ mit $\alpha, \beta \in \Gamma^*,\ q \in Q$:

- q ist der momentane Zustand.
- α ist die Beschriftung des Bandes links vom Arbeitsfeld.
- β ist die Beschriftung des Bandes beginnend beim Arbeitsfeld nach rechts.

Dabei werden (um endliche Wörter α, β zu erhalten) unendlich viele Blanks weggelassen, d. h. α und β umfassen mindestens den Bandabschnitt, auf dem Symbole $\neq \not b$ stehen.

Beispiel:
Der Zustand der Maschine im obigen Bild wird durch die Konfiguration *aqbbbb*, aber auch durch $\not b\not b aqbbbb\not b$ beschrieben.

Formal werden Zustandsübergänge durch die Relation $\vdash_M$ auf der Menge aller Konfigurationen beschrieben. Genauer gesagt ermöglicht die Übergangsrelation Δ die folgenden *Konfigurationsübergänge*:

Es seien $\alpha, \beta \in \Gamma^*,\ a, b, a' \in \Gamma,\ q, q' \in Q$. Es gilt

$$
\begin{array}{rcll}
\alpha q a \beta & \vdash_M & \alpha a' q' \beta & \text{falls} \quad (q, a, a', r, q') \in \Delta \\
\alpha q & \vdash_M & \alpha a' q' & \text{falls} \quad (q, \not b, a', r, q') \in \Delta \\
\alpha b q a \beta & \vdash_M & \alpha q' b a' \beta & \text{falls} \quad (q, a, a', l, q') \in \Delta \\
q a \beta & \vdash_M & q' \not b a' \beta & \text{falls} \quad (q, a, a', l, q') \in \Delta \\
\alpha q a \beta & \vdash_M & \alpha q' a' \beta & \text{falls} \quad (q, a, a', n, q') \in \Delta \\
\alpha q & \vdash_M & \alpha q' a' & \text{falls} \quad (q, \not b, a', n, q') \in \Delta
\end{array}
$$

Weitere Bezeichnungen:

- Gilt $k \vdash_M k'$, so heißt k' *Folgekonfiguration* von k.
- Die Konfiguration $\alpha q \beta$ heißt *akzeptierend*, falls $q \in F$.
- Die Konfiguration $\alpha q \beta$ heißt *Stoppkonfiguration*, falls sie keine Folgekonfiguration hat.
- Eine *Berechnung* von M ist eine endliche oder unendliche Konfigurationsfolge

$$k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \cdots$$

Offensichtlich gibt es für DTMs nur eine einzige maximale Berechnung, die in einer fixen Konfiguration k_0 beginnt; für NTMs kann es dagegen mehrere solche Berechnungen geben.

Die folgende Definition präzisiert beide Anwendungen von Turingmaschinen: das Berechnen von Funktionen und das Erkennen von Sprachen.

- Beim Berechnen einer (partiellen) Funktion f steht die Eingabe $(w_1, \ldots, w_n)$ in Form des Wortes $w_1 \not\flat w_2 \not\flat \cdots \not\flat w_n$ auf dem Band, wobei sich der Kopf zu Anfang auf dem ersten (am weitesten links stehenden) Symbol von w_1 befindet. Nachdem die Maschine eine Stoppkonfiguration erreicht hat, findet sich die Ausgabe (und damit der Funktionswert $f(w_1, \ldots, w_n)$) ab der Kopfposition bis zum ersten Symbol aus $\Gamma \setminus \Sigma$. Terminiert die Berechnung jedoch nicht, dann ist der Funktionswert $f(w_1, \ldots, w_n)$ undefiniert.
- Beim Erkennen einer Sprache L steht das Eingabewort w auf dem Band und der Kopf befindet sich anfangs über dem ersten Symbol von w. Ein positives Berechnungsergebnis ($w \in L$) wird dann über das Stoppen in einem *akzeptierenden* Zustand signalisiert. Ist hingegen $w \notin L$, dann kann die Maschine entweder in einem nicht-akzeptierenden Zustand stoppen oder nicht terminieren.

Definition 13.2 (Turing-berechenbar, Turing-erkennbar)

1) Die partielle Funktion $f : (\Sigma^*)^n \to \Sigma^*$ heißt *Turing-berechenbar*, falls es eine DTM M gibt mit
   a) der Definitionsbereich $\mathrm{dom}(f)$ von f besteht aus den Tupeln $(w_1, \ldots, w_n) \in (\Sigma^*)^n$, so dass M ab der Konfiguration

$$k_0 = q_0 w_1 \not\flat w_2 \not\flat \ldots \not\flat w_n$$

      eine Stoppkonfiguration erreicht.
   b) wenn $(x_1, \ldots, x_n) \in \mathrm{dom}(f)$, dann hat die von k_0 erreichte Stoppkonfiguration k die Form $uqxv$ mit
      - $x = f(w_1, \ldots, w_n)$
      - $v \in (\Gamma \setminus \Sigma) \cdot \Gamma^* \cup \{\varepsilon\}$

2) Die von der NTM M *erkannte Sprache* ist

$$L(M) = \{w \in \Sigma^* \mid q_0 w \vdash_M^* k, \text{ wobei } k \textit{ akzeptierende Stoppkonfiguration} \text{ ist}\}.$$

Eine Sprache $L \subseteq \Sigma^*$ heißt *Turing-erkennbar*, falls es eine NTM M gibt mit $L = L(M)$.

Nach Punkt b) dürfen vor und nach der Ausgabe des Funktionswertes noch Überbleibsel der Berechnung stehen. Beispielsweise entspricht die Stoppkonfiguration *aaaqbaabc*$\not b$*acbca* der Ausgabe *baabc*, wenn $\Sigma = \{a, b, c\}$. Per Definition werden akzeptierende Zustände nur für das Erkennen von Sprachen verwendet, aber *nicht* für das Berechnen von Funktionen.

Beachte:

1) Wir verwenden *partielle* Funktionen, da Turingmaschinen nicht anhalten müssen; für manche Eingaben ist der Funktionswert daher *nicht definiert*.
2) Bei berechenbaren Funktionen betrachten wir nur *deterministische* Maschinen, da sonst der Funktionswert nicht eindeutig sein müsste.
3) Bei $|\Sigma| = 1$ kann man Funktionen von $(\Sigma^*)^n \to \Sigma^*$ als Funktionen von $\mathbb{N}^k \to \mathbb{N}$ auffassen (a^k entspricht k). Wir unterscheiden im Folgenden nicht immer explizit zwischen beiden Arten von Funktionen.
4) Es gibt *zwei Arten*, auf die eine Turingmaschine ein Eingabewort *verwerfen* kann: entweder sie erreicht eine Stoppkonfiguration mit einem nicht-akzeptierenden Zustand, oder sie stoppt nicht (wir sagen auch: sie terminiert nicht).

Beispiel:
Die Funktion

$$f : \mathbb{N} \to \mathbb{N}, \quad n \mapsto 2n$$

ist Turing-berechenbar. Wie kann eine Turingmaschine die Anzahl der a's auf dem Band verdoppeln?

Idee:

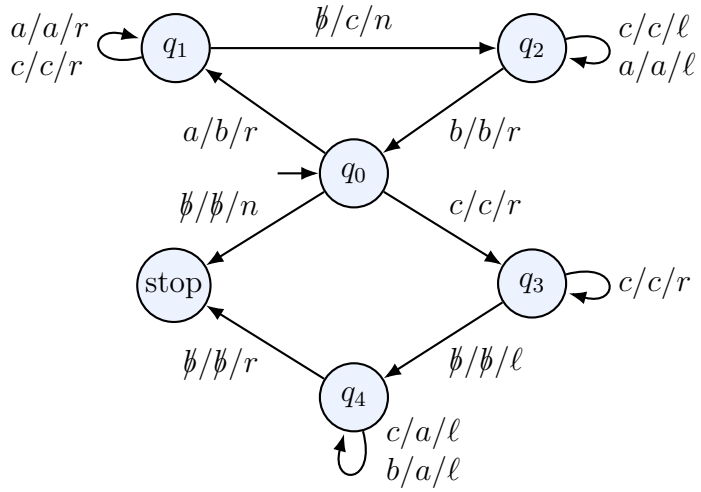
- Ersetze das erste a durch b.
- Laufe nach rechts bis zum ersten Blank, ersetze dieses durch c.
- Laufe zurück bis zum zweiten a (unmittelbar rechts vom b); ersetze dieses durch b.
- Laufe nach rechts bis zum ersten Blank etc.
- Sind alle a's aufgebraucht, so ersetze noch die b's und c's wieder durch a's.

Dies wird durch die folgende Übergangsrelation realisiert:

$(q_0,$	$\not\flat,$	$\not\flat,$	$n,$	stop$)$	$2 \cdot 0 = 0$
$(q_0,$	$a,$	$b,$	$r,$	$q_1)$	ersetze a durch b $\quad(\star)$
$(q_1,$	$a,$	$a,$	$r,$	$q_1)$	laufe nach rechts über a's
$(q_1,$	$c,$	$c,$	$r,$	$q_1)$	und bereits geschriebene c's
$(q_1,$	$\not\flat,$	$c,$	$n,$	$q_2)$	schreibe weiteres c
$(q_2,$	$c,$	$c,$	$l,$	$q_2)$	laufe zurück über c's und
$(q_2,$	$a,$	$a,$	$l,$	$q_2)$	a's
$(q_2,$	$b,$	$b,$	$r,$	$q_0)$	bei erstem b eins nach rechts und weiter wie $(\star)$ oder
$(q_0,$	$c,$	$c,$	$r,$	$q_3)$	alle a's bereits ersetzt
$(q_3,$	$c,$	$c,$	$r,$	$q_3)$	laufe nach rechts bis Ende der c's
$(q_3,$	$\not\flat,$	$\not\flat,$	$l,$	$q_4)$	letztes c erreicht
$(q_4,$	$c,$	$a,$	$l,$	$q_4)$	ersetze c's und b's
$(q_4,$	$b,$	$a,$	$l,$	$q_4)$	durch a's
$(q_4,$	$\not\flat,$	$\not\flat,$	$r,$	stop$)$	bleibe am Anfang der erzeugten $2n$ a's stehen

Beachte, dass „stop“ hier einen ganz normalen Zustand bezeichnet. Da er in keinem Tupel der Übergangsrelation ganz links erscheint, ist jede Konfiguration der Form $\alpha\,\mathrm{stop}\,\beta$ eine Stoppkonfiguration.

Die graphische Darstellung von Turingmaschinen ist ähnlich der von NEAs und PDAs (siehe Abschnitte 1 und 11). Die obige Turingmaschine stellen wir also wie folgt dar; dabei bedeutet beispielsweise die Kantenbeschriftung $a/b/r$, dass das a auf dem Arbeitsfeld durch b ersetzt wird und sich der Kopf einen Schritt nach rechts bewegt.



Wie bei den endlichen Automaten kennzeichnen wir den Startzustand durch einen eingehenden Pfeil und akzeptierende Zustände durch einen Doppelkreis. Da obige DTM eine

Funktion berechnet (im Gegensatz zu: eine Sprache erkennt), spielen die akzeptierenden Zustände hier jedoch keine Rolle.

Man sieht, dass das Programmieren von Turingmaschinen recht umständlich ist. Wie bereits erwähnt, betrachtet man solche einfachen (und unpraktischen) Modelle, um das Führen von Beweisen zu erleichtern. Wir werden im Folgenden häufig nur die Arbeitsweise einer Turingmaschine beschreiben, ohne die Übergangsrelation in vollem Detail anzugeben.

Beispiel:
Die Sprache $L = \{a^n b^n c^n \mid n \geq 0\}$ ist Turing-erkennbar.

Die Turingmaschine, die L erkennt, geht wie folgt vor:

- Sie ersetzt das erste a durch a', das erste b durch b' und das erste c durch c'.
- Sie läuft zurück und wiederholt diesen Vorgang.
- Falls dabei ein a rechts von einem b oder c steht, verwirft die TM direkt (indem sie in eine nicht-akzeptierende Stoppkonfiguration wechselt); ebenso, wenn ein b rechts von einem c steht.
- Dies wird solange gemacht, bis nach erzeugtem c' ein $\not\emptyset$ steht.
- Zum Schluss wird zurückgelaufen und überprüft, dass keine unersetzten a oder b übrig geblieben sind.

Eine solche Turingmaschine erkennt tatsächlich L: sie akzeptiert gdw.

1. die Eingabe dieselbe Anzahl a's wie b's wie c's hat (denn es wurde jeweils dieselbe Anzahl ersetzt und danach waren keine a's, b's und c's mehr übrig);
2. in der Eingabe alle a's vor b's vor c's stehen.

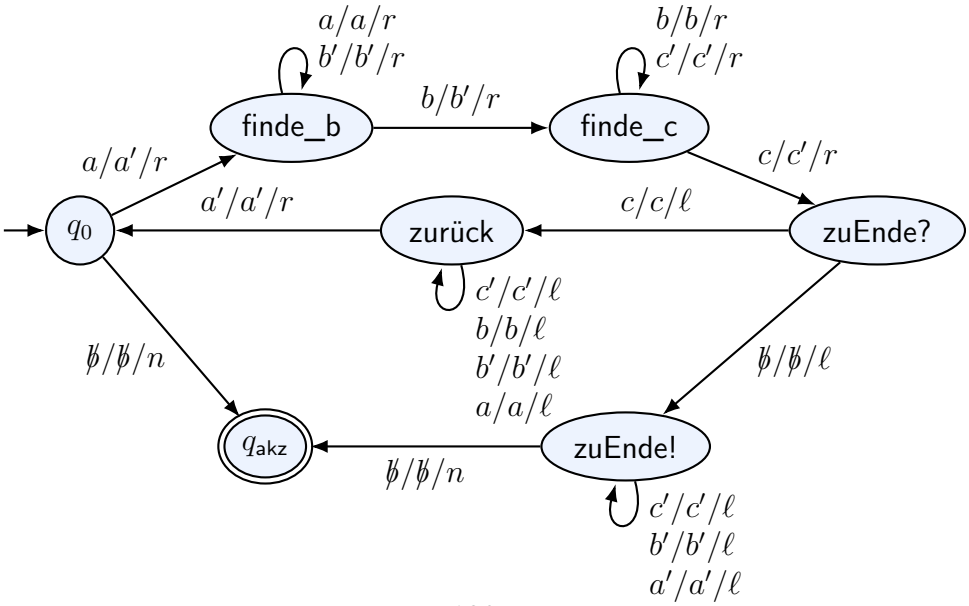
Im Detail definiert man die Turingmaschine M wie folgt:

$$
\begin{aligned}
M &= (Q, \Sigma, \Gamma, q_0, \Delta, \{q_{\mathsf{akz}}\}) \\
&\text{mit} \\
Q &= \{q_0,\ q_{\mathsf{akz}},\ \mathsf{finde\_b},\ \mathsf{finde\_c},\ \mathsf{zuEnde?},\ \mathsf{zuEnde!},\ \mathsf{zurück}\} \\
\Sigma &= \{a, b, c\} \\
\Gamma &= \{a, a', b, b', c, c', \not\emptyset\} \\
&\text{und}
\end{aligned}
$$

$$
\begin{aligned}
\Delta \;=\; \{\; & (q_0, && \not b, && \not b, && n, && q_{\mathsf{akz}}),\\
& (q_0, && a, && a', && r, && \mathsf{finde\_b}),\\
& (\mathsf{finde\_b}, && a, && a, && r, && \mathsf{finde\_b}),\\
& (\mathsf{finde\_b}, && b', && b', && r, && \mathsf{finde\_b}),\\
& (\mathsf{finde\_b}, && b, && b', && r, && \mathsf{finde\_c}),\\
& (\mathsf{finde\_c}, && b, && b, && r, && \mathsf{finde\_c}),\\
& (\mathsf{finde\_c}, && c', && c', && r, && \mathsf{finde\_c}),\\
& (\mathsf{finde\_c}, && c, && c', && r, && \mathsf{zuEnde?}),\\
& (\mathsf{zuEnde?}, && c, && c, && l, && \mathsf{zurück}),\\
& (\mathsf{zurück}, && c', && c', && l, && \mathsf{zurück}),\\
& (\mathsf{zurück}, && b, && b, && l, && \mathsf{zurück}),\\
& (\mathsf{zurück}, && b', && b', && l, && \mathsf{zurück}),\\
& (\mathsf{zurück}, && a, && a, && l, && \mathsf{zurück}),\\
& (\mathsf{zurück}, && a', && a', && r, && q_0),\\
& (\mathsf{zuEnde?}, && \not b, && \not b, && l, && \mathsf{zuEnde!}),\\
& (\mathsf{zuEnde!}, && c', && c', && l, && \mathsf{zuEnde!}),\\
& (\mathsf{zuEnde!}, && b', && b', && l, && \mathsf{zuEnde!}),\\
& (\mathsf{zuEnde!}, && a', && a', && l, && \mathsf{zuEnde!}),\\
& (\mathsf{zuEnde!}, && \not b, && \not b, && n, && q_{\mathsf{akz}}) && \}
\end{aligned}
$$

Beachte: wenn die Maschine z. B. im Zustand finde_c ist und ein a liest, so ist sie in einer Stoppkonfiguration. Da finde_c kein akzeptierender Zustand ist, handelt es sich um eine verwerfende Stoppkonfiguration. Also verwirft die Maschine, wenn in der Eingabe nach einer Folge von a's noch ein b erscheint.

In graphischer Darstellung sieht diese Maschine wie folgt aus:



Varianten von Turingmaschinen

In der Literatur werden verschiedene Versionen von Turingmaschinen definiert, die aber alle äquivalent zueinander sind, d. h. dieselben Sprachen erkennen und dieselben Funktionen berechnen. Hier zwei Beispiele:

- Turingmaschinen mit nach links begrenztem und nur nach rechts unendlichem Arbeitsband
- Turingmaschinen mit mehreren Bändern und Schreib-Lese-Köpfen

Die Äquivalenz dieser Modelle ist ein Indiz für die Gültigkeit der Church-Turing-These.

Wir betrachten das zweite Beispiel genauer und zeigen Äquivalenz zur in Definition 13.1 eingeführten 1-Band-TM.

Definition 13.3 (k-Band-TM)

Eine *k-Band-NTM* hat die Form $M = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ mit

- $Q, \Sigma, \Gamma, q_0, F$ wie in Definition 13.1 und
- $\Delta \subseteq Q \times \Gamma^k \times \Gamma^k \times \{r, l, n\}^k \times Q$.

Dabei bedeutet $(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$:

- Vom Zustand q aus
- mit $a_1, \ldots, a_k$ auf den Arbeitsfeldern der k Bänder

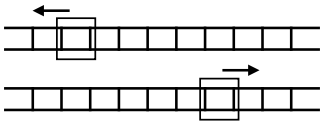
kann M

- das Symbol a_i auf dem i-ten Band durch b_i ersetzen,
- in den Zustand q' gehen und
- die Schreib-Lese-Köpfe der Bänder entsprechend d_i bewegen.

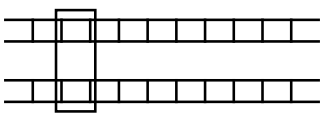
Das erste Band wird (o. B. d. A.) als Ein- und Ausgabeband verwendet.

Beachte:

Die Köpfe der verschiedenen Bänder können sich unabhängig voneinander bewegen:



Wären die Köpfe gekoppelt, so hätte man im Prinzip nicht mehrere Bänder, sondern ein Band mit mehreren Spuren:



k Spuren erhält man einfach, indem man eine normale NTM (nach Definition 13.1) verwendet, die als Bandalphabet Γ^k statt Γ hat.

Offenbar kann man jede 1-Band-NTM (nach Definition 13.1) durch eine k-Band-NTM ($k > 1$) simulieren, indem man nur das erste Band wirklich verwendet. Der folgende Satz zeigt, dass auch die Umkehrung gilt:

Satz 13.4

Wird die Sprache L durch eine k-Band-NTM erkannt, so auch durch eine 1-Band-NTM.

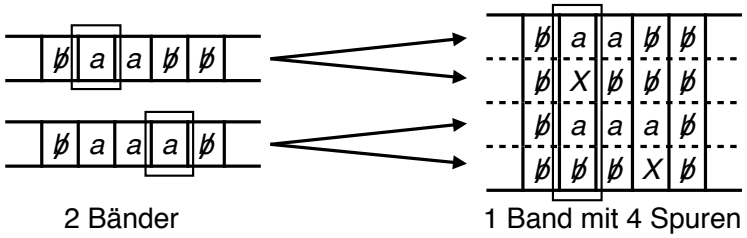
Beweis. Es sei M eine k-Band-NTM. Gesucht ist eine 1-Band-NTM M' mit $L(M) = L(M')$. Wähle ein beliebiges $X \in \Gamma$.

Arbeitsalphabet von M': $\Gamma^{2k} \cup \Sigma \cup \{\not b\}$:

- $\Sigma \cup \{\not b\}$ wird für die Eingabe benötigt.
- Γ^{2k} sorgt dafür, dass $2k$ Spuren auf dem Band von M' zur Verfügung stehen.

Idee: Jeweils 2 Spuren kodieren ein Band von M.

- Die erste Spur enthält die Bandbeschriftung.
- Die zweite Spur enthält eine Markierung X (und sonst Blanks), die zeigt, wo das Arbeitsfeld des Bandes ist, z. B.

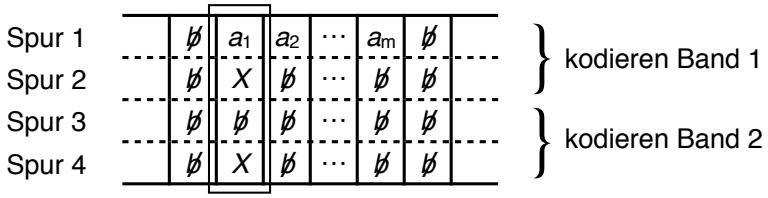


Die 1-Band-NTM M' macht jeweils mehrere Schritte, um einen einzelnen Schritt von M zu simulieren. Im Detail arbeitet M' wie folgt.

Initialisierung: Zunächst wird die Anfangskonfiguration

$$q_0 a_1 \dots a_m$$

von M' in die Repräsentation der entsprechenden Anfangskonfiguration von M umgewandelt (durch geeignete Übergänge):



Simulation eines Überganges von M':

- Von links nach rechts suche die mit X markierten Felder. Dabei merke man sich (im Zustand von M') die Symbole, die jeweils über dem X stehen. Außerdem zählt man (im Zustand von M'), wie viele X man schon gelesen hat, um festzustellen, wann das k-te (und letzte) X erreicht ist. Man ermittelt so das aktuelle Tupel $(a_1, \ldots, a_k)$.
- entscheide nichtdeterministisch, welcher Übergang

  $$(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$$

  von M stattfindet.
- Von rechts nach links gehend ersetze die a_i bei den X-Marken jeweils durch das entsprechende b_i und verschiebe die X-Marken gemäß d_i.
- Bleibe bei der am weitesten links stehenden X-Markierung und gehe in Zustand q'.

In dieser Konstruktion hat M' im Prinzip dieselben Zustände wie M. Also hält M auf Eingabe w in akzeptierender Stoppkonfiguration gdw. M' in akzeptierender Stoppkonfiguration hält. ❑

Bemerkung 13.5

1) War M deterministisch, so liefert obige Konstruktion auch eine deterministische 1-Band-Turingmaschine.
2) Diese Konstruktion kann auch verwendet werden, wenn man sich für die berechnete Funktion interessiert. Dazu muss man am Schluss (wenn M in Stoppkonfiguration ist) in der Maschine M' noch die Ausgabe geeignet aufbereiten.

Wegen Satz 13.4 und Bemerkung 13.5 können wir von nun an ohne Beschränkung der Allgemeinheit bei der Konstruktion von Turingmaschinen eine beliebige (aber feste) Zahl von Bändern verwenden.

Bei der Definition von „Turing-berechenbar“ haben wir uns von vornherein auf *deterministische* Turingmaschinen beschränkt. Der folgende Satz zeigt, dass man dies auch für „Turing-erkennbar“ tun kann, ohne an Ausdrucksstärke zu verlieren.

Satz 13.6

Zu jeder NTM gibt es eine DTM, die dieselbe Sprache erkennt.

Beweis. Es sei $M = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ eine NTM. Wegen Satz 13.4 und Bemerkung 13.5 genügt es, eine deterministische 3-Band-Turingmaschine M' zu konstruieren, so dass $L(M) = L(M')$.

Die Maschine M' soll für wachsendes n auf dem dritten Band jeweils alle Berechnungen

$$k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \ldots \vdash_M k_n$$

beginnend mit der Startkonfiguration $k_0 = q_0 w$ erzeugen, also erst alle solchen Berechnungen der Länge 1, dann alle der Länge 2 usw.

Die Kontrolle, dass tatsächlich alle solchen Folgen erzeugt werden, wird auf dem zweiten Band vorgenommen. Das erste Band speichert das Eingabewort w, damit man stets weiß, was k_0 sein muss.

Genauer: Es sei

$$r = \text{ maximale Anzahl von Transitionen in } \Delta \text{ pro festem Paar } (q, a) \in Q \times \Gamma.$$

Dies entspricht dem maximalen Verzweigungsgrad der nichtdeterministischen Berechnung und kann direkt aus Δ abgelesen werden.

Eine Indexfolge $i_1, \ldots, i_n$ mit $i_j \in \{1, \ldots, r\}$ beschreibt dann von k_0 aus für n Schritte die Auswahl der jeweiligen Transition, und somit von k_0 aus eine feste Berechnung

$$k_0 \vdash_M k_1 \vdash_M \ldots \vdash_M k_n$$

(Wenn es in einer Konfiguration k_j weniger als i_j mögliche Nachfolgerkonfigurationen gibt, dann beschreibt $i_1, \ldots, i_n$ keine Berechnung und wird einfach übersprungen.)

Zählt man alle endlichen Wörter über $\{1, \ldots, r\}$ (die nichts anderes sind als die o.g. Indexfolgen) in aufsteigender Länge auf und erzeugt zu jedem Wort $i_1 \ldots i_n$ die zugehörige Berechnung, so erhält man eine Aufzählung aller endlichen Berechnungen.

M' realisiert dies auf den drei Bändern wie folgt:

- Auf Band 1 bleibt die Eingabe gespeichert.
- Auf dem zweiten Band werden sukzessive alle Wörter $i_1 \ldots i_n \in \{1, \ldots, r\}^*$ erzeugt (z. B. durch Zählen zur Basis r).
- Für jedes dieser Wörter wird auf dem dritten Band die zugehörige Berechnung erzeugt (wenn sie existiert). Erreicht man dabei eine akzeptierende Stoppkonfiguration von M, so geht auch M' in eine akzeptierende Stoppkonfiguration.

  Falls dieses Aufzählen aller endlichen Berechnungen endet (weil alle Berechnungen von M endlich sind) und dabei keine akzeptierende Stoppkonfiguration erreicht wurde, so geht M' in eine nicht-akzeptierende Stoppkonfiguration.

Es ist sofort einzusehen, dass M' korrekt arbeitet, also $L(M') = L(M)$:

Wenn $w \in L(M)$ für ein beliebiges Eingabewort w, dann gibt es eine endliche Berechnung von M auf w. Sei n die Länge dieser Berechung. Diese Berechnung wird von M' nach endlich vielen Schritten erzeugt und dann die akzeptierende Stoppkonfiguration erreicht, also akzeptiert M' die Eingabe w ebenfalls. Ist hingegen $w \notin L(M)$, dann wird in keiner der erzeugten Berechnungen eine akzeptierende Stoppkonfiguration erreicht, also geht M' in eine nicht-akzeptierende Stoppkonfiguration, wenn alle Berechnungen von M endlich sind, oder in eine Endlosschleife, akzeptiert also w nicht. ❑

14. Zusammenhang zwischen Turingmaschinen und Grammatiken

Wir zeigen zunächst, dass die *Turing-erkennbaren Sprachen* genau die *Typ-0-Sprachen* sind, schränken dann das Modell der Turingmaschine so ein, dass genau die *Typ-1-Sprachen* erkannt werden, und untersuchen schließlich die Abschlusseigenschaften sowie erste Entscheidungsprobleme der Typ-0- und Typ-1-Sprachen.

14.1. Typ-0-Sprachen

Der Zusammenhang zwischen *Typ-0-Sprachen* und *Turing-erkennbaren Sprachen* beruht darauf, dass es eine Entsprechung von Ableitungen einer Typ-0-Grammatik einerseits und Berechnungen von Turingmaschinen andererseits gibt. Beim Übergang von der Turingmaschine zur Grammatik dreht sich allerdings die Richtung um:

- Eine akzeptierende Berechnung beginnt mit dem zu akzeptierenden Wort.
- Eine Ableitung endet mit dem erzeugten Wort.

Wir werden im Folgenden sehen, wie man diese Schwierigkeit löst.

Satz 14.1

Eine Sprache L gehört zu $\mathcal{L}_0$ gdw. sie Turing-erkennbar ist.

Beweis.

„⇒“. Es sei $L = L(G)$ für eine Typ-0-Grammatik $G = (N, \Sigma, P, S)$. Wir beschreiben eine 2-Band-NTM M, die $L(G)$ erkennt (und nach Satz 13.4 äquivalent zu einer 1-Band-NTM ist).

1. Band: speichert Eingabe w

2. Band: Es wird nichtdeterministisch und Schritt für Schritt eine Ableitung von G erzeugt.

Es wird verglichen, ob auf Band 2 irgendwann w (d. h. der Inhalt von Band 1) entsteht. Wenn ja, so geht M in eine akzeptierende Stoppkonfiguration; anderenfalls werden weitere Ableitungsschritte vorgenommen.

Die Maschine M geht dabei wie folgt vor:

1) Schreibe S auf Band 2, gehe nach links auf das $\not\emptyset$ vor S.

2) Gehe auf Band 2 nach rechts und wähle (nichtdeterministisch) eine Stelle aus, an der die linke Seite der anzuwendenden Produktion beginnen soll.

3) Wähle (nichtdeterministisch) eine Produktion $u \longrightarrow v$ aus P aus, die angewendet werden soll.

4) Überprüfe, ob u tatsächlich die Beschriftung des Bandstücks der Länge $|u|$ ab der gewählten Bandstelle ist.

5) Falls der Test erfolgreich war, so ersetze u durch v. Vorher müssen
   - falls $|u| < |v|$, die Symbole rechts von u um $|v| - |u|$ Positionen nach rechts bzw.
   - falls $|u| > |v|$, um $|u| - |v|$ Positionen nach links geschoben werden.

6) Gehe nach links bis zum ersten $\not\emptyset$ und vergleiche, ob die Beschriftung auf dem Band 1 mit der auf Band 2 übereinstimmt.

7) Wenn ja, so gehe in akzeptierende Stoppkonfiguration. Sonst fahre fort bei 2).

„⇐“. Es sei $L = L(M)$ für eine NTM $M = (Q, \Sigma, \Gamma, q_0, \Delta, F)$.

Wir konstruieren eine Grammatik G, die jedes Wort $w \in L(M)$ wie folgt erzeugt:

1. Phase: Erst wird w mit „genügend vielen“ $\not\emptyset$-Symbolen links und rechts davon erzeugt (dies passiert für jedes w, auch für $w \notin L(M)$).

„Genügend viele“ bedeutet dabei: so viele, wie M beim Akzeptieren von w vom Arbeitsband benötigt.

2. Phase: Auf dem so erzeugten Arbeitsband simuliert G die Berechnung von M bei Eingabe w.

3. Phase: War die Berechnung akzeptierend, so erzeuge nochmals das ursprüngliche w.

Damit man in der zweiten Phase das in der dritten Phase benötigte w nicht vergisst, verwendet man als Nichtterminalsymbole Paare aus

$$(\Sigma \cup \{\not\emptyset\}) \times \Gamma$$

wobei das Paar $[a, b]$ zwei Zwecken dient:

- In der ersten Komponente merkt man sich die ursprüngliche Eingabe w.
- In der zweiten Komponente simuliert man die Berechnung (bei der die Eingabe ja überschrieben werden kann).

Genaue Definition:

$$N = \{S, A, B, E\} \cup Q \cup ((\Sigma \cup \{\not\emptyset\}) \times \Gamma),$$

wobei

- S, A, B zum Aufbau des Rechenbandes am Anfang,
- E zum Löschen am Schluss,
- Q zur Darstellung des aktuellen Zustandes,
- $\Sigma \cup \{\not\emptyset\}$ zum Speichern von w und
- Γ zur M-Berechnung

dienen.

Regeln:

1. Phase: Erzeuge w und ein genügend großes Arbeitsband.

$$\begin{aligned} S &\longrightarrow Bq_0A \\ A &\longrightarrow [a,a]A \text{ für alle } a \in \Sigma \\ A &\longrightarrow B \\ B &\longrightarrow [\not b, \not b]B \\ B &\longrightarrow \varepsilon \end{aligned}$$

Man erhält also somit für alle $a_1 \dots a_n \in \Sigma^*, k, \ell, \geq 0$:

$$S \vdash_G^* [\not b, \not b]^k q_0 [a_1, a_1] \dots [a_n, a_n] [\not b, \not b]^\ell$$

2. Phase: simuliert TM-Berechnung in der „zweiten Spur“:

- $p[a,b] \longrightarrow [a,b']q$
  falls $(p,b,b',r,q) \in \Delta,\ a \in \Sigma \cup \{\not b\}$
- $[a,c]p[a',b] \longrightarrow q[a,c][a',b']$
  falls $(p,b,b',l,q) \in \Delta,\ a,a' \in \Sigma \cup \{\not b\},\ c \in \Gamma$
- $p[a,b] \longrightarrow q[a,b']$
  falls $(p,b,b',n,q) \in \Delta,\ a \in \Sigma \cup \{\not b\}$

Beachte:

Da wir in der ersten Phase genügend viele Blanks links und rechts von $a_1 \dots a_n$ erzeugen können, muss in der zweiten Phase das „Nachschieben“ von Blanks am Rand nicht mehr behandelt werden.

3. Phase: Aufräumen und Erzeugen von $a_1 \dots a_n$, wenn M akzeptiert hat

- $\begin{aligned} q[a,b] &\longrightarrow EaE \quad \text{für} \quad a \in \Sigma,\ b \in \Gamma \\ q[\not b, b] &\longrightarrow E \quad\ \ \text{für} \quad b \in \Gamma \end{aligned}$
  falls $q \in F$ und es keine Transition der Form $(q,b,\cdot,\cdot,\cdot) \in \Delta$ gibt (d. h. akzeptierende Stoppkonfiguration erreicht)
- $E[a,b] \longrightarrow aE$ für $a \in \Sigma, b \in \Gamma$
  (Aufräumen nach rechts)
- $[a,b]E \longrightarrow Ea$ für $a \in \Sigma, b \in \Gamma$
  (Aufräumen nach links)
- $E[\not b, b] \longrightarrow E$ für $b \in \Gamma$
  (Entfernen des zusätzlich benötigten Arbeitsbandes nach rechts)
- $[\not b, b]E \longrightarrow E$ für $b \in \Gamma$
  (Entfernen des zusätzlich benötigten Arbeitsbandes nach links)
- $E \longrightarrow \varepsilon$

Man sieht nun leicht, dass für alle $w \in \Sigma^*$ gilt:

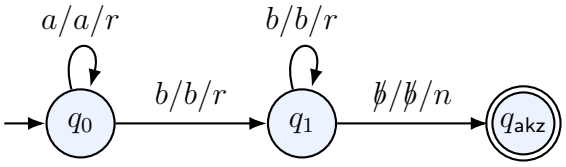
$$w \in L(G) \text{ gdw. } M \text{ akzeptiert } w.$$

❑

Beispiel: Betrachte die Turingmaschine mit den Übergängen

q_0	a	a	r	q_0
q_0	b	b	r	q_1
q_1	b	b	r	q_1
q_1	$\not b$	$\not b$	n	q_{akz}

und der graphischen Darstellung:



Offenbar wird das Wort $w = aab$ von M akzeptiert. Eine zugehörige Ableitung der konstruierten Grammatik G sieht wie folgt aus.

$$\begin{array}{lll}
S & \vdash_G^* & Bq_0[a,a][b,b][b,b]A \qquad \text{Phase 1} \\
& \vdash_G^* & q_0[a,a][b,b][b,b][\not b,\not b] \\
& \vdash_G & [a,a]q_0[b,b][b,b][\not b,\not b] \qquad \text{Phase 2} \\
& \vdash_G^* & [a,a][b,b][b,b]q_{\mathsf{akz}}[\not b,\not b] \\
& \vdash_G & [a,a][b,b][b,b]E \qquad \text{Phase 3} \\
& \vdash_G & [a,a][b,b]Eb \\
& \vdash_G^* & Eabb \\
& \vdash_G & abb
\end{array}$$

14.2. Linear beschränkte Automaten und Typ-1-Sprachen

Von den Sprachklassen aus der Chomsky-Hierarchie sind nun alle bis auf $\mathcal{L}_1$ durch geeignete Automaten-/Maschinenmodelle charakterisiert. Im Folgenden identifizieren wir auch ein geeignetes Maschinenmodell für $\mathcal{L}_1$. Nach Definition enthalten monotone (Typ-1-) Grammatiken nur Regeln, die *nicht verkürzend* sind, also Regeln $u \longrightarrow v$ mit $|v| \geq |u|$. Wenn man ein Terminalwort w mit einer solchen Grammatik ableitet, so wird die Ableitung also niemals ein Wort enthalten, dessen Länge größer als $|w|$ ist.

Diese Beobachtung legt folgende Modifikation von Turingmaschinen nahe: die Maschinen dürfen nicht mehr als $|w|$ Zellen des Arbeitsbandes verwenden, also nur auf dem Bandabschnitt arbeiten, auf dem anfangs die Eingabe steht. Um ein Überschreiten der dadurch gegebenen Bandgrenzen zu verhindern, verwendet man Randmarker $\not\in$, \$.

Definition 14.2 (linear beschränkter Automat)

Ein *linear beschränkter Automat (LBA)* ist eine NTM $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, so dass

- $\$, \not\in \Gamma \setminus \Sigma$
- Übergänge $(q, \not\in, \cdot, \cdot, \cdot)$ sind nur in der Form $(q, \not\in, \not\in, r, q')$ erlaubt (linker Rand darf nicht überschritten werden).
- Übergänge $(q, \$, \cdot, \cdot, \cdot)$ sind nur in der Form $(q, \$, \$, l, q')$ erlaubt.
- $\not\in$ und $\$$ dürfen nicht geschrieben werden.

Ein gegebener LBA $\mathcal{A}$ erkennt die Sprache

$$L(\mathcal{A}) := \{w \in \Sigma^* \mid \not\in q_0 w \$ \vdash^* k, \text{ wobei } k \text{ akzeptierende Stoppkonfiguration ist}\}.$$

Offensichtlich muss auch bei einem LBA nicht jede Berechnung terminieren. Wie bei Turingmaschinen gilt nach der obigen Definition: stoppt ein LBA $\mathcal{A}$ auf einer gegebenen Eingabe w nicht, so ist $w \notin L(\mathcal{A})$.

Korollar 14.3

Eine Sprache L gehört zu $\mathcal{L}_1$ gdw. sie von einem LBA erkannt wird.

Beweis.

„⇒“: Verwende die Konstruktion aus dem Beweis von Satz 14.1.

Da die Produktionen von monotonen Grammatiken nicht verkürzend sind (mit Ausnahme $S \longrightarrow \varepsilon$), muss man auf dem zweiten Band nur ableitbare Wörter bis zur Länge $|w|$ erzeugen (aus längeren kann nie mehr w abgeleitet werden). Daher kommt man mit $|w|$ vielen Feldern aus.

Beachte:

Zwei Bänder liefern nicht ein doppelt so langes Band, sondern ein größeres Arbeitsalphabet, siehe den Beweis von Satz 13.4.

„⇐“: Durch Modifikation der Beweisidee von Satz 14.1 gelingt es, zu einem LBA eine Grammatik zu konstruieren, die nur nicht verkürzende Regeln hat.

Idee:

Da man mit $|w|$ Arbeitsfeldern auskommt, muss man keine $[\not\in, \not\in]$ links und rechts von w erzeugen. Dadurch fallen dann auch die folgenden verkürzenden Regeln weg:

$$E[\not\in, \not\in] \longrightarrow E$$

$$[\not\in, \not\in]E \longrightarrow E$$

Es gibt allerdings noch zwei technische Anforderungen aufgrund der Definition von LBAs:

- Man muss die Randmarker $\not\in$ und \$ einführen und am Schluss löschen.
- Man muss das Hilfssymbol E und den Zustand q löschen.

Zu diesem Zweck führt man die Symbole $\not\in$, \$ sowie E und den Zustand q *nicht* als zusätzliche Symbole ein, sondern kodiert sie in die anderen Symbole hinein:

$$\text{z. B. statt } [a,b]q[a',b'][a'',b''] \text{ verwende } [a,b][q,a',b'][a'',b''].$$

Basierend auf dieser Idee kann man die Konstruktion aus dem Beweis von Satz 14.1 so modifizieren, dass eine monotone Grammatik erzeugt wird.

❑

14.3. Abschlusseigenschaften und Entscheidungsprobleme

Wir analysieren nun die Abschlusseigenschaften von Typ-0- und Typ-1-Sprachen und betrachten die Entscheidbarkeit und Komplexität von Wortproblem, Leerheitsproblem und Äquivalenzproblem. Wir beginnen mit den Typ-0-Sprachen.

Satz 14.4

1) $\mathcal{L}_0$ *ist abgeschlossen unter* $\cup, \cdot, ^*$ *und* $\cap$.

2) $\mathcal{L}_0$ *ist nicht abgeschlossen unter Komplement.*

Beweis.

1) Für die regulären Operationen $\cup, \cdot, ^*$ zeigt man dies im Prinzip wie für $\mathcal{L}_2$ durch Konstruktion einer entsprechenden Grammatik.
Damit sich die Produktionen der verschiedenen Grammatiken nicht gegenseitig beeinflussen, genügt es allerdings nicht mehr, nur die Nichtterminalsymbole der Grammatiken disjunkt zu machen.

Zusätzlich muss man die Grammatiken in die folgende Form bringen:
Die Produktionen sind von der Form

$$\begin{aligned} u &\longrightarrow v && \text{mit } u \in N_i^+ \text{ und } v \in N_i^* \\ X_a &\longrightarrow a && \text{mit } X_a \in N_i \text{ und } a \in \Sigma \end{aligned}$$

Für Abgeschlossenheit unter Schnitt verwendet man Turingmaschinen:

Die NTM für $L_1 \cap L_2$ verwendet zwei Bänder und simuliert zunächst auf dem ersten die Berechnung der NTM für L_1 und dann auf dem anderen die der NTM für L_2.

Wenn beide zu akzeptierenden Stoppkonfigurationen der jeweiligen Turingmaschinen führen, so geht die NTM für $L_1 \cap L_2$ in eine akzeptierende Stoppkonfiguration.

Beachte:

Es kann sein, dass die NTM für L_1 auf einer Eingabe w nicht terminiert, die NTM für $L_1 \cap L_2$ also gar nicht dazu kommt, die Berechnung der NTM für L_2 zu simulieren. Aber dann ist ja w auch nicht in $L_1 \cap L_2$.

2) Wir werden später sehen, dass die Klasse der Turing-erkennbaren Sprachen nicht unter Komplement abgeschlossen ist (Satz 18.6).

❑

Wir werden später außerdem zeigen, dass für die Turing-erkennbaren Sprachen (und damit für $\mathcal{L}_0$) das Wortproblem, Leerheitsproblem und Äquivalenzproblem unentscheidbar sind (Sätze 17.6, 17.15, 17.16). Die Begriffe „entscheidbar“ und „unentscheidbar“ werden wir in Kürze genau definieren. Intuitiv bedeutet Unentscheidbarkeit, dass es keinen Algorithmus gibt, der das Problem löst.

Satz 14.5 (siehe Abschnitt 17)

Für $\mathcal{L}_0$ sind das Leerheitsproblem, das Wortproblem und das Äquivalenzproblem unentscheidbar.

Wir wenden uns nun den Typ-1-Sprachen zu und beginnen mit den Abschlusseigenschaften. Genauso wie die Typ-3-Sprachen ist die Klasse der Typ-1-Sprachen unter allen betrachteten Operationen abgeschlossen.

Satz 14.6

$\mathcal{L}_1$ ist abgeschlossen unter $\cup, \cdot, ^*, \cap$ und Komplement.

Beweis. Für $\cup, \cdot, ^*$ und $\cap$ verwende Grammatiken bzw. LBAs, analog zu $\mathcal{L}_0$.

Komplement: der Beweis ist schwierig und wird an dieser Stelle nicht geführt. Abschluss unter Komplement von $\mathcal{L}_1$ war lange ein offenes Problem und wurde dann in den 1980ern unabhängig von zwei Forschern bewiesen (Immerman und Szelepcsenyi). ❑

Satz 14.7

Für monotone Grammatiken sind

1) das Wortproblem entscheidbar,

2) das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.

Beweis.

1) Da die Produktionen von monotonen Grammatiken (bis auf Spezialfall $S \longrightarrow \varepsilon$) nicht verkürzend sind, muss man zur Entscheidung „$w \in L(G)\,?$“ nur systematisch die Menge aller aus S ableitbaren Wörter über $(N \cup \Sigma)^*$ der Länge $\leq |w|$ erzeugen und dann nachsehen, ob w in dieser Menge ist. Dieses Verfahren terminiert, da es nur *endlich viele* solche Wörter gibt.

2) Werden wir später beweisen (Satz 17.21). ❑

Es ist ein sehr schwieriges offenes Forschungsproblem, ob deterministische LBAs genauso stark wie nichtdeterministische LBAs (im Sinne von Satz 13.6) sind.

15. LOOP- und WHILE-Programme

In diesem Abschnitt betrachten wir Berechnungsmodelle, die an imperative Programmiersprachen angelehnt, aber auf wesentliche Elemente reduziert sind. Dies dient zweierlei Zweck: Erstens werden wir eine abstrakte Programmiersprache identifizieren, die dieselbe Berechnungsstärke wie Turingmaschinen aufweist – wir sagen auch, dass diese Programmiersprache ein *berechnungsvollständiges* Berechnungsmodell ist. Dies ist ein gutes Indiz für die Gültigkeit der Church-Turing-These und zeigt, dass es sinnvoll ist, Berechenbarkeit anhand von Turingmaschinen (anstelle von „echten Programmiersprachen“) zu studieren. Zweitens erlauben uns die erzielten Resultate, in präziser Weise diejenigen Elemente von imperativen Programmiersprachen zu identifizieren, die für die Berechnungsvollständigkeit verantwortlich sind. Wir trennen sie damit vom bloßen „Beiwerk“, das zwar angenehm für die Programmierung, aber in Bezug auf die Berechnungsstärke verzichtbar ist.

Wir definieren zunächst eine sehr einfache Programmiersprache LOOP und erweitern sie in einem zweiten Schritt zur Programmiersprache WHILE. Wir betrachten in diesem Abschnitt nur Funktionen f der Form

$$f : \mathbb{N}^n \to \mathbb{N}.$$

Dies entspricht dem Spezialfall $|\Sigma| = 1$ bei Wortfunktionen, ist aber keine echte Einschränkung, da es berechenbare Kodierungsfunktionen π gibt, die Wörter über beliebigen Alphabeten Σ als natürliche Zahlen darstellen (und umgekehrt), d. h. bijektive Abbildungen $\pi : \Sigma^* \to \mathbb{N}$, so dass sowohl π als auch die inverse Funktion π^{-1} berechenbar sind.

15.1. LOOP-Programme

Wir betrachten zunächst eine sehr einfache imperative Programmiersprache.

LOOP-Programme sind aus den folgenden Komponenten aufgebaut:

- Variablen: x_0, x_1, x_2, ...
- Konstanten: 0, 1, 2, ... (also die Elemente von $\mathbb{N}$)
- Trennsymbole: `;` und `:=`
- Operationssymbole: $+$ und $\dot{-}$
- Schlüsselwörter: `LOOP`, `DO`, `END`

Die folgende Definition beschreibt die wohlgeformten LOOP-Programme im Detail.

Definition 15.1 (Syntax LOOP)

Die *Syntax von LOOP-Programmen* ist induktiv definiert:

1) Jede *Wertzuweisung*

$$x_i := x_j + c \text{ und}$$
$$x_i := x_j \dot{-} c$$

für $i, j \geq 0$ und $c \in \mathbb{N}$ ist ein LOOP-Programm.

2) Falls P_1 und P_2 LOOP-Programme sind, so ist auch

$$P_1 ; P_2 \qquad \textit{(Hintereinanderausführung)}$$

ein LOOP-Programm.

3) Falls P ein LOOP-Programm ist und $i \geq 0$, so ist auch

$$\texttt{LOOP } x_i \texttt{ DO } P \texttt{ END} \qquad \textit{(}\texttt{LOOP}\textit{-Schleife)}$$

ein LOOP-Programm.

Die *Semantik* dieser einfachen Sprache ist wie folgt definiert:

Bei einem LOOP-Programm, das eine k-stellige Funktion $f : \mathbb{N}^k \to \mathbb{N}$ berechnen soll,

- werden die *Variablen* $x_1, \ldots, x_k$ mit den *Eingabewerten* $n_1, \ldots, n_k$ vorbesetzt,
- erhalten alle anderen Variablen eingangs den Wert 0 und
- ist die *Ausgabe* der Wert der Variable x_0 nach Ausführung des Programms.

Die einzelnen Programmkonstrukte haben die folgende Bedeutung:

1) $x_i := x_j + c$
Der neue Wert der Variablen x_i ist die Summe des alten Wertes von x_j und c.

$x_i := x_j \dot{-} c$
Der neue Wert der Variablen x_i ist der Wert von x_j minus c, falls dieser Wert ≥ 0 ist, und 0 sonst (das nennt man auch *modifizierte Subtraktion*, angezeigt durch den Punkt über dem Minuszeichen).

2) $P_1 ; P_2$
Hier wird zunächst P_1 und dann P_2 ausgeführt.

3) `LOOP` x_i `DO` P `END`
Das Programm P wird so oft ausgeführt, wie der Wert von x_i zu Beginn angibt. Änderungen des Wertes von x_i während der Ausführung von P haben keinen Einfluss auf die Anzahl der Schleifendurchläufe.

LOOP-Programme lassen zunächst nur Addition und Subtraktion von Konstanten zu, aber nicht von Variablen. Wir werden aber sehen, dass man letzteres ebenfalls ausdrücken kann. `LOOP`-Schleifen entsprechen einer einfachen Variante der aus vielen Programmiersprachen bekannten `FOR`-Schleifen.

Definition 15.2 (LOOP-berechenbar)

Die Funktion

$$f : \mathbb{N}^k \to \mathbb{N}$$

heißt *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das f im folgenden Sinne berechnet:

- *Gestartet* mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen)
- *stoppt* P mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen x_0.

Beispiel:

Die Additionsfunktion ist LOOP-berechenbar:

x_0 `:=` $x_1 + 0$`;`
`LOOP` x_2 `DO` x_0 `:=` $x_0 + 1$ `END`

Basierend auf der Additionsfunktion ist auch die Multiplikationsfunktion LOOP-berechenbar. Folgendes Programm steht natürlich eigentlich für die Schachtelung zweier `LOOP`-Schleifen:

`LOOP` x_2 `DO` x_0 `:=` $x_0 + x_1$ `END`

Man nutzt hier aus, dass x_0 eingangs den Wert 0 hat.

Einige andere Programmkonstrukte typischer imperativer Programmiersprachen lassen sich mittels `LOOP`-Schleifen simulieren. Wir betrachten zwei Beispiele:

- Absolute Wertzuweisungen: $x_i := c$ wird simuliert durch

  `LOOP` x_i `DO` x_i `:=` $x_i \dot{-} 1$ `END`;
  x_i `:=` $x_i + c$

- `IF` $x = 0$ `THEN` P `END` kann simuliert werden durch:

  $y := 1$`;`
  `LOOP` x `DO` y `:=` 0 `END`;
  `LOOP` y `DO` P `END`

  wobei y eine neue Variable ist, die nicht in P vorkommt.

Im Folgenden verwenden wir diese zusätzlichen Konstrukte sowie LOOP-berechenbare arithmetische Operationen o. B. d. A. direkt in LOOP-Programmen. Gemeint ist dann natürlich das LOOP-Programm, das man durch Ersetzen der verwendeten Konstrukte und Operationen durch die definierenden LOOP-Programme erhält.

Beispiel:

Die ganzzahlige Divisionsfunktion ist LOOP-berechenbar. Genauer: wir betrachten die Funktion

$$x \operatorname{\mathsf{div}} y := \begin{cases} \left\lfloor \frac{x}{y} \right\rfloor & \text{falls } y \neq 0, \\ x & \text{falls } y = 0. \end{cases}$$

Der zweite Fall $x \operatorname{\mathsf{div}} 0 = x$ ist hier willkürlich festgelegt, damit der Funktionswert $f(x, y)$ für alle Argumente x, y definiert ist. div ist LOOP-berechenbar:

```
LOOP x1 DO
  x3 := x3 + x2;
  IF x3 ∸ x1 = 0 THEN x0 := x0 + 1 END
END
```

Man nutzt hier aus, dass sowohl x_0 als auch die Hilfsvariable x_3 eingangs den Wert 0 haben. Die Bedingung $x_3 \dot{-} x_1 = 0$ verwenden wir zum Testen von $x_1 \geq x_3$. Letzteres könnte man also auch o. B. d. A. als Bedingung innerhalb des `IF`-Konstruktes zulassen.

Im Gegensatz zu Turingmaschinen terminieren LOOP-Programme offensichtlich immer, da für jede Schleife eine feste Anzahl von Durchläufen durch den anfänglichen Wert der Schleifenvariablen festgelegt wird. Daher sind alle durch LOOP-Programme berechneten Funktionen total. Es folgt sofort, dass *nicht* jede Turing-berechenbare Funktion auch LOOP-berechenbar ist. Wie aber steht es mit der LOOP-Berechenbarkeit totaler Funktionen? Das folgende Resultat zeigt, dass LOOP-Programme auch bezüglich solcher Funktionen nicht berechnungsvollständig sind.

Satz 15.3

Es gibt totale berechenbare Funktionen, die nicht LOOP-berechenbar sind.

Beweis. Wir definieren die Länge $|P|$ eines LOOP-Programms P induktiv über dessen Aufbau:

1) $|x_i := x_j + c| := i + j + c + 1$
   $|x_i := x_j \dot{-} c| := i + j + c + 1$
2) $|P;Q| := |P| + |Q| + 1$
3) $|\texttt{LOOP } x_i \texttt{ DO } P \texttt{ END}| := |P| + i + 1$

Für eine gegebene Länge n gibt es nur endlich viele LOOP-Programme dieser Länge: neben der Programmlänge ist mit obiger Definition auch die Menge der möglichen vorkommenden Variablen und Konstantensymbole beschränkt. Deshalb ist die folgende Funktion total:

$$f(x, y) := 1 + \max\{g(y) \mid g : \mathbb{N} \to \mathbb{N} \text{ wird von einem LOOP-Programm der Länge } x \text{ berechnet}\}$$

Behauptung 1:

Die Funktion

$$d : \mathbb{N} \to \mathbb{N} \text{ mit } z \mapsto f(z, z)$$

ist nicht LOOP-berechenbar, denn:

Sei P ein LOOP-Programm, das d berechnet und sei $n = |P|$.

Wir betrachten $d(n) = f(n, n)$. Nach Definition ist $f(n, n)$ größer als jeder Funktionswert, den ein LOOP-Programm der Länge n bei Eingabe n berechnet. Dies widerspricht der Tatsache, dass d von P berechnet wird.

Behauptung 2:

Die Funktion d ist Turing-berechenbar, denn:

Bei Eingabe z kann eine TM die *endlich vielen* LOOP-Programme der Länge z aufzählen und jeweils auf die Eingabe z anwenden (wir werden später noch genau beweisen, dass eine TM jedes LOOP-Programm simulieren kann). Da alle diese Aufrufe terminieren, kann die TM in endlicher Zeit den maximalen so erhaltenen Funktionswert berechnen (und dann um eins erhöhen). ❑

Die im Beweis von Satz 15.3 angegebene Funktion ist unter Bezugnahme auf LOOP-Programme definiert und daher eher technischer Natur. Eine prominente Funktion, die ebenfalls nicht durch LOOP-Programme berechenbar ist, deren Definition sich aber nicht auf LOOP-Programme bezieht, ist die *Ackermannfunktion* A. Sie ist wie folgt definiert:

$$\begin{aligned} A(0, y) &:= y + 1 \\ A(x + 1, 0) &:= A(x, 1) \\ A(x + 1, y + 1) &:= A(x, A(x + 1, y)) \end{aligned}$$

Es handelt sich hier um eine geschachtelte Induktion: in der letzten Zeile wird der Wert für das zweite Argument durch die Funktion selbst berechnet. Intuitiv ist diese Funktion wohldefiniert, da in jedem Schritt entweder das erste Argument verringert wird oder gleich bleibt, wobei dann aber das zweite Argument verringert wird. Ein präziser, induktiver Beweis der Wohldefiniertheit ist nicht schwierig.

Man kann leicht zeigen, dass A Turing-berechenbar ist (zum Beispiel, indem man die Definition von A direkt in eine rekursive Funktion in einer beliebigen Programmiersprache „übersetzt“). Der Beweis, dass A nicht LOOP-berechenbar ist, beruht auf einem ähnlichen Argument wie der Beweis von Satz 15.3: die Funktionswerte der Ackermannfunktion *wachsen schneller* als bei jeder LOOP-berechenbaren Funktion. Die Beweisdetails sind allerdings etwas technischer, siehe [Sch08].

15.2. WHILE-Programme

Wir erweitern LOOP-Programme nun um eine `WHILE`-*Schleife* und erhalten so WHILE-Programme. Die `WHILE`-Schleife unterscheidet sich von der `LOOP`-Schleife dadurch, dass die Anzahl der Durchläufe nicht von Anfang an feststeht. Dieser Unterschied ist fundamental, denn im Gegensatz zu LOOP-Programmen stellen sich WHILE-Programme als berechnungsvollständig heraus. Intuitiv ist es also nicht möglich, eine berechnungsvollständige Programmiersprache allein auf `FOR`-Schleifen aufzubauen, wohingegen `WHILE`-Schleifen ausreichend sind (wir werden noch sehen, dass sich `FOR`-Schleifen mittels `WHILE`-Schleifen ausdrücken lassen).

Definition 15.4 (Syntax von WHILE-Programmen)
Die Syntax von WHILE-Programmen enthält alle Konstrukte in der Syntax von LOOP-Programmen (Definition 15.1) und zusätzlich

4) Falls P ein WHILE-Programm ist und $i \geq 0$, so ist auch

$$\texttt{WHILE } x_i \neq 0 \texttt{ DO } P \texttt{ END} \qquad \texttt{WHILE-}\textit{Schleife}$$

ein WHILE-Programm.

Die *Semantik* dieses Konstrukts ist wie folgt definiert:

- Das Programm P wird solange iteriert, bis x_i den Wert 0 erhält.
- Geschieht das nicht, so terminiert diese Schleife nicht.

Offensichtlich müssen WHILE-Programme im Gegensatz zu LOOP-Programmen nicht unbedingt terminieren. Man könnte bei der Definition der WHILE-Programme auf das `LOOP`-Konstrukt verzichten, da es durch `WHILE` simulierbar ist:

$$\texttt{LOOP } x \texttt{ DO } P \texttt{ END}$$

kann simuliert werden durch

$$y \texttt{ := } x + 0\texttt{;}$$
$$\texttt{WHILE } y \neq 0 \texttt{ DO } y \texttt{ := } y \dot{-} 1\texttt{;} P \texttt{ END}$$

wobei y eine neue Variable ist.

Definition 15.5 (WHILE-berechenbar)
Eine (partielle) k-stellige Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das f im folgenden Sinne berechnet:

- *Gestartet* mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen)
- *stoppt* P mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen x_0, falls dieser Wert definiert ist.
- Sonst stoppt P nicht.

Beispiel:
Die Funktion

$$f : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x, y) \mapsto \begin{cases} x - y & \text{falls} \quad x \geq y \\ \text{undefiniert} & \text{sonst} \end{cases}$$

ist WHILE-berechenbar durch das folgende Programm:

```
WHILE x_2 ≠ 0 DO
  IF x_1 ≠ 0 THEN
    x_1 := x_1 ∸ 1;
    x_2 := x_2 ∸ 1;
  END
END;
x_0 := x_1
```

Satz 15.6

Jede WHILE-berechenbare Funktion ist Turing-berechenbar.

Beweisskizze. Sei P ein WHILE-Programm, das eine k-stellige Funktion berechnet. Wir nehmen o. B. d. A. an, dass alle Variablen $x_0, \ldots, x_k$ auch wirklich in P vorkommen. Wir zeigen per Induktion über den Aufbau von Q, dass es zu jedem Teilprogramm Q von P eine Mehrband-DTM M_Q gibt, die im folgenden Sinne Q entspricht:

- Wenn ℓ der maximale Index ist, so dass Q die Variable x_ℓ verwendet und die Bänder 1 bis $\ell + 1$ von M_Q eingangs die Werte der Variablen $x_0, \ldots, x_\ell$ vor Ausführung des Teilprogramms Q enthalten,
- dann terminiert M_Q gdw. das Teilprogramm Q auf den gegebenen Variablenwerten terminiert und
- wenn M_Q terminiert, dann finden sich danach auf den Bändern 1 bis $\ell + 1$ die Werte der Variablen $x_0, \ldots, x_\ell$ nach Ausführung des Teilprogramms Q.

Weitere Arbeitsbänder werden nicht verwendet. Die Induktion liefert schließlich eine DTM M_P für P selbst. Man kann M_P nun leicht in eine DTM M umwandeln, die im Sinne von Definition 13.2 dieselbe Funktion wie P berechnet:

- Zunächst wird der Inhalt $w_1 \not\flat w_2 \not\flat \cdots \not\flat w_k$ des ersten Bandes gelesen und w_1 auf Band 2 geschrieben, w_2 auf Band 3 usw. Danach wird Band 1 gelöscht (entspricht der Initialisierung der Variablen x_0 mit 0).
- Nun wird M_P gestartet.
- Wenn M_P terminiert, wird der Schreib-Lese-Kopf auf Band 1 ganz an das linke Ende bewegt (denn die Ausgabe muss ja nach Definition 13.2 direkt beim Kopf beginnen).

Wir nehmen o. B. d. A. an, dass in P alle `LOOP`-Schleifen durch `WHILE`-Schleifen ersetzt wurden. Die Konstruktion der DTMs M_Q ist nun wie folgt:

- Sei Q von der Form $x_i := x_j + c$ oder $x_i := x_j \dot{-} c$. Es ist leicht, die benötigten Turingmaschinen M_Q zu konstruieren.
- Sei Q von der Form $Q_1; Q_2$. Man erhält die Maschine M_Q durch Hintereinanderausführung von M_{Q_1} und M_{Q_2}.
- Sei Q von der Form `WHILE` $x_i \neq 0$ `DO` Q' `END`. Die Turingmaschine M_Q prüft zunächst, ob Band $i + 1$ leer ist (was $x_i = 0$ entspricht). Ist dies der Fall, so terminiert M_Q. Andernfalls wird $M_{Q'}$ gestartet und danach von vorn begonnen.

❑

Satz 15.7

Jede Turing-berechenbare Funktion ist WHILE-berechenbar.

Beweisskizze. Um diesen Satz zu beweisen, kodieren wir Konfigurationen von Turingmaschinen, dargestellt als Wörter der Form

$$\alpha q \beta \text{ für } \alpha, \beta \in \Gamma^* \text{ und } q \in Q,$$

in drei natürliche Zahlen.

Diese werden dann in den drei Programmvariablen x_1, x_2, x_3 des WHILE-Programms gespeichert:

- x_1 repräsentiert α,
- x_2 repräsentiert q,
- x_3 repräsentiert β.

Es sei o. B. d. A. $\Gamma = \{a_1, \ldots, a_n\}$ und $Q = \{q_0, \ldots, q_k\}$ mit q_0 Startzustand, d. h. wir können Alphabetssymbole und Zustände über ihren Index als natürliche Zahl beschreiben.

Die Konfiguration

$$a_{i_1} \cdots a_{i_l} q_m a_{j_1} \cdots a_{j_r}$$

wird dargestellt als

$$\begin{aligned} x_1 = (i_1, \ldots, i_l)_d &:= \sum_{p=1}^{l} i_p \cdot d^{l-p} \\ x_2 = m & \\ x_3 = (j_r, \ldots, j_1)_d &:= \sum_{p=1}^{r} j_p \cdot d^{p-1}, \end{aligned}$$

wobei $d = |\Gamma| + 1$ ist, d. h.

- $a_{i_1} \ldots a_{i_l}$ repräsentiert x_1 in d-ärer Zahlendarstellung und
- $a_{j_r} \ldots a_{j_1}$ (Reihenfolge!) repräsentiert x_3 in d-ärer Zahlendarstellung.

Ist beispielsweise $d = |\Gamma| + 1 = 10$, also $\Gamma = \{a_1, \ldots, a_9\}$, so wird die Konfiguration

$$a_4 a_2 a_7 a_1 \, q_3 \, a_1 a_8 a_3 \quad \text{dargestellt durch} \quad x_1 = 4271,\ x_2 = 3,\ x_3 = 381.$$

Wir brauchen $d = |\Gamma| + 1$ statt $d = |\Gamma|$, da wir die Ziffer 0 nicht verwenden können: die unterschiedlichen Strings a_0 und $a_0 a_0$ hätten die Kodierungen 0 und 00, also dieselbe Zahl.

Die elementaren Operationen auf Konfigurationen, die zur Implementierung von Berechnungsschritten der simulierten TM benötigt werden, lassen sich mittels einfacher arithmetischer Operationen realisieren:

Herauslesen des aktuellen Symbols a_{j_1}
Ist $x_3 = (j_r, \ldots, j_1)_d$, so ist $j_1 = x_3 \bmod d$.

Ändern dieses Symbols zu a_j
Der neue Wert von x_3 ist $(j_r, \ldots, j_2, j)_d = \mathrm{div}((j_r, \ldots, j_2, j_1)_d, d) \cdot d + j$

Verschieben des Schreib-Lese-Kopfes
kann durch ähnliche arithmetische Operationen realisiert werden.

All diese Operationen sind offensichtlich WHILE-berechenbar (sogar LOOP!).

Das WHILE-Programm, welches die gegebene DTM simuliert, arbeitet wie folgt:

1) Aus der Eingabe wird die Kodierung der Startkonfiguration der DTM in den Variablen x_1, x_2, x_3 erzeugt.

   Wenn also beispielsweise eine binäre Funktion berechnet werden soll und die Eingabe $x_1 = 3$ und $x_2 = 5$ ist, so muss die Startkonfiguration $q_0aaa\not baaaaa$ erzeugt werden, repräsentiert durch $x_1 = 2$, $x_2 = 0$, $x_3 = 111112111$ wenn $a_1 = a$ und $a_2 = \not b$.

2) In einer WHILE-Schleife wird bei jedem Durchlauf ein Schritt der TM-Berechnung simuliert (wie oben angedeutet):

   - In Abhängigkeit vom aktuellen Zustand (Wert von x_2) und
   - dem gelesenen Symbol, d. h. von $x_3 \bmod d$
   - wird mit den oben dargestellten arithmetischen Operationen das aktuelle Symbol verändert und
   - der Schreib-Lese-Kopf bewegt.

   Die WHILE-Schleife terminiert, wenn der aktuelle Zustand zusammen mit dem gelesenen Symbol keinen Nachfolgezustand hat.

   All dies ist durch einfache (WHILE-berechenbare) arithmetische Operationen realisierbar.

3) Aus dem Wert von x_3 nach Terminierung der WHILE-Schleife wird der Ausgabewert herausgelesen und in die Variable x_0 geschrieben.

   Dazu braucht man eine weitere WHILE-Schleife: starte mit $x_0 = 0$; extrahiere wiederholt Symbole aus x_3; solange es sich dabei um a handelt, inkrementiere x_0; sobald ein anderes Symbol gefunden wird oder x_3 erschöpft ist, gib den Wert von x_0 zurück.

❑

16. Primitiv rekursive und μ-rekursive Funktionen

Sowohl die LOOP-berechenbaren Funktionen als auch die WHILE-berechenbaren Funktionen lassen sich auch auf gänzlich andere Weise definieren, wobei nicht der Mechanismus der Berechnung, sondern die Funktion selbst in den Mittelpunkt gestellt wird. Genauer gesagt betrachten wir die Klasse der sogenannten primitiv rekursiven Funktionen, die sich als identisch zu den LOOP-berechenbaren Funktionen herausstellen, und die Klasse der μ-rekursiven Funktionen, die sich als identisch zu den WHILE-berechenbaren Funktionen herausstellen. Beide Klassen waren historisch unter den ersten betrachteten Berechnungsmodellen und spielen in der Theorie der Berechenbarkeit und der Komplexitätstheorie noch heute eine wichtige Rolle. Insbesondere kann man sie als mathematisches Modell von funktionalen Programmiersprachen auffassen. Auch in diesem Kapitel betrachten wir ausschließlich k-stellige Funktionen $f : \mathbb{N}^k \to \mathbb{N}$.

16.1. Primitiv rekursive Funktionen

Die Klasse der *primitiv rekursiven Funktionen* besteht aus den Funktionen, die man aus *Grundfunktionen* durch Anwenden der Operationen *Komposition* und *primitive Rekursion* erhält. Diese sind wie folgt definiert.

Definition 16.1 (Grundfunktionen)
Die folgenden Funktionen sind *primitiv rekursive Grundfunktionen.*

1. die konstante *Nullfunktionen* null_n beliebiger Stelligkeit $n \geq 0$ mit $(x_1, \ldots, x_n) \mapsto 0$
2. die *Projektionsfunktionen* $\mathsf{proj}_{n \to i}$ beliebiger Stelligkeit n auf das i-te Argument, $1 \leq i \leq n$, mit $(x_1, \ldots, x_n) \mapsto x_i$.
3. die einstellige *Nachfolgerfunktion* succ

So ist also z. B. null_3 die dreistellige Funktion mit $\mathsf{null}_3(x_1, x_2, x_3) = 0$ für alle x_1, x_2, x_3, und $\mathsf{proj}_{3\to1}$ ist die dreistellige Funktion für die Projektion auf das erste Argument, also $\mathsf{proj}_{3\to1}(x_1, x_2, x_3) = x_1$ für alle x_1, x_2, x_3.

Definition 16.2 (Komposition)
Eine n-stellige Funktion f entsteht aus n-stelligen Funktionen $h_1, \ldots, h_k$ unter Anwendung einer k-stelligen Funktion g, wenn gilt:

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n))$$

Man sagt auch: f ist die *Komposition* von g mit $h_1, \ldots, h_k$, geschrieben $f = g(h_1, \ldots, h_k)$.

Zum Beispiel erhält man die Funktion $f(x) = x + 2$ durch Komposition von $g = \mathsf{succ}$ mit $h_1 = \mathsf{succ}$ $(n = k = 1)$.

Definition 16.3 (Primitive Rekursion)
Sei g eine n-stellige und h eine $(n+2)$-stellige Funktion. Die $(n+1)$-stellige Funktion f entsteht per *primitiver Rekursion* aus g und h, geschrieben $f = \mathsf{PR}(g, h)$, wenn gilt:

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y+1) &= h(f(x_1, \ldots, x_n, y), x_1, \ldots, x_n, y) \end{aligned}$$

Primitive Rekursion bedeutet hier einen rekursiven Abstieg über das letzte Funktionsargument: man darf $f(\ldots, n+1)$ definieren unter Verwendung von $f(\ldots, n)$. Die Funktion g legt also den Rekursionsanfang fest und h den Rekursionsschritt. Bei der Berechnung des Funktionswertes im Rekursionsschritt hat man zur Verfügung: alle Argumente inklusive Rekursionsargument sowie den per rekursivem Aufruf ermittelten Wert.

Beispiele.

- **Addition** kann durch primitive Rekursion wie folgt definiert werden:

$$\begin{aligned} \mathsf{add}(x, 0) &= x && = g(x) \\ \mathsf{add}(x, y+1) &= \mathsf{add}(x, y) + 1 && = h(\mathsf{add}(x, y), x, y) \end{aligned}$$

  Das heißt also: add entsteht durch primitive Rekursion aus den Funktionen
  - $g(x_1) = \mathsf{proj}_{1\to1}(x_1)$, also die Identitätsfunktion als Projektion;
  - $h(x_1, x_2, x_3) = \mathsf{succ}(\mathsf{proj}_{3\to1}(x_1, x_2, x_3))$.

  Man kann auch schreiben: $\mathsf{add} = \mathsf{PR}\Big(\mathsf{proj}_{1\to1}, \mathsf{succ}(\mathsf{proj}_{3\to1})\Big)$

- **Multiplikation** erhält man durch primitive Rekursion aus der Addition:

$$\begin{aligned} \mathsf{mult}(x, 0) &= 0 \\ \mathsf{mult}(x, y+1) &= \mathsf{add}(\mathsf{mult}(x, y), x) \end{aligned}$$

  Das heißt also: mult entsteht durch primitive Rekursion aus den Funktionen
  - $g(x_1) = \mathsf{null}_1(x_1)$;
  - $h(x_1, x_2, x_3) = \mathsf{add}(\mathsf{proj}_{3\to1}(x_1, x_2, x_3), \mathsf{proj}_{3\to2}(x_1, x_2, x_3)))$.

  Man kann auch schreiben: $\mathsf{mult} = \mathsf{PR}\Big(\mathsf{null}_1, \mathsf{add}(\mathsf{proj}_{3\to1}, \mathsf{proj}_{3\to2})\Big)$

- **Exponentiation** erhält man durch primitive Rekursion aus der Multiplikation:

$$\begin{aligned} \mathsf{exp}(x, 0) &= 1 \\ \mathsf{exp}(x, y+1) &= \mathsf{mult}(\mathsf{exp}(x, y), x) \end{aligned}$$

  Es ergibt sich die Exponentiation x^y (mit dem Sonderfall $0^0 = 1$). Analog zur Multiplikation erhält man: $\mathsf{exp} = \mathsf{PR}\Big(\mathsf{succ}(\mathsf{null}_1), \mathsf{mult}(\mathsf{proj}_{3\to1}, \mathsf{proj}_{3\to2})\Big)$

Definition 16.4 (Klasse der primitiv rekursiven Funktionen)
Die Klasse der *primitiv rekursiven Funktionen* besteht aus allen Funktionen, die man aus den Grundfunktionen durch Anwenden von Komposition und primitiver Rekursion erhält.

Man zeigt leicht per Induktion über den Aufbau primitiv rekursiver Funktionen, dass jede primitiv rekursive Funktion *total* ist. Es gilt der folgende Satz.

Satz 16.5
Die Klasse der primitiv rekursiven Funktionen stimmt mit der Klasse der LOOP-berechenbaren Funktionen überein.

Beweis.

(I) Alle primitiv rekursiven Funktionen sind LOOP-berechenbar. Wir zeigen dies per Induktion über den Aufbau der primitiv rekursiven Funktionen:

- Für die *Grundfunktionen* ist klar, dass sie LOOP-berechenbar sind.
- Komposition:

  $$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n))$$

  Es seien $P_1, \ldots, P_k, P$ LOOP-Programme für $h_1, \ldots, h_k, g$.

  Durch Speichern der Eingabewerte und der Zwischenergebnisse in unbenutzten Variablen kann man zunächst die Werte von $h_1, \ldots, h_k$ mittels $P_1, \ldots, P_k$ berechnen und dann auf diese Werte P anwenden. Genauer:

  $$\left[\begin{array}{l} y_1 := x_1;\ \cdots\ ;\ y_n := x_n; \\ \text{Führe } P_1 \text{ aus} \\ z_1 := x_0; \end{array}\right.$$

  $$\left[\begin{array}{l} x_i := 0 \text{ für alle in } P_1 \text{ verwendeten Variablen } x_i \\ x_1 := y_1;\ \cdots\ ;\ x_n := y_n; \\ \text{Führe } P_2 \text{ aus} \\ z_2 := x_0; \end{array}\right.$$

  $$\vdots$$

  $$\left[\begin{array}{l} \text{Führe } P_k \text{ aus} \\ z_k := x_0; \end{array}\right.$$

  $$\left[\begin{array}{l} x_i := 0 \text{ für alle in } P_k \text{ verwendeten Variablen } x_i \\ x_1 := z_1;\ \cdots\ ;\ x_k := z_k; \\ \text{Führe } P \text{ aus} \end{array}\right.$$

- primitive Rekursion:

  $$f(x_1, \ldots, x_n) = g(x_1, \ldots, x_{n-1}) \qquad \text{falls } x_n = 0$$
  $$f(x_1, \ldots, x_n) = h(f(x_1, \ldots, x_{n-1}, x_n - 1), x_1, \ldots, x_{n-1}, x_n - 1) \qquad \text{falls } x_n > 0$$

  Die Funktion f kann durch das folgende LOOP-Programm berechnet werden.

$z_1 := 0;$
$z_2 := g(x_1, \ldots, x_{n-1});$ $\quad(\star)$
`LOOP` x_n `DO`
$\quad z_2 := h(z_2, z_1, \ldots, x_{n-1}, z_1);$ $\quad(\star)$
$\quad z_1 := z_1 + 1$
`END`;
$x_0 := z_2$

Dabei sind die mit $(\star)$ gekennzeichneten Anweisungen Abkürzungen für Programme, welche Ein- und Ausgaben geeignet kopieren und die Programme für g und h anwenden.

Die Variablen z_1, z_2 sind neue Variablen, die in den Programmen für g und h *nicht* vorkommen.

(II) Alle LOOP-berechenbaren Funktionen sind primitiv rekursiv.

Da der Beweis technisch etwas aufwändiger ist, beschränken wir uns auf eine Skizze. Für ein gegebenes LOOP-Programm P zeigt man per Induktion über die Teilprogramme von P, dass es zu jedem Teilprogramm Q von P eine „äquivalente“ primitiv rekursive Funktion gibt.

Leider ist es nicht ausreichend, „äquivalent“ als „liefert dieselbe Ausgabe/denselben Funktionswert“ zu interpretieren, da P zusätzliche Hilfsvariablen verwenden kann, deren Werte zwischen den Teilprogrammen von P weitergegeben werden (siehe Beweis von Satz 15.6). Ein adäquater Äquivalenzbegriff muss darum die Ein- und Ausgabewerte *aller* in P verwendeten Variablen $x_0, \ldots, x_n$ berücksichtigen.

Da aber jede primitiv rekursive Funktion nur einen einzelnen Funktionswert zurückliefert und so etwas wie „Hilfsvariablen“ nicht zur Verfügung steht, ist es notwendig, n Variablenwerte als eine einzige Zahl zu kodieren.

Wir benötigen also eine primitiv rekursive Bijektion

$$c^{(n)} : \mathbb{N}^n \to \mathbb{N} \quad (n \geq 2)$$

sowie die zugehörigen Umkehrfunktionen

$$c_0^{(n)}, \ldots, c_{n-1}^{(n)} \quad \text{(jeweils } \mathbb{N} \to \mathbb{N}\text{)}.$$

Für $n = 2$ kann man beispielsweise die folgende Funktion verwenden:

$$c^{(2)} : \mathbb{N}^2 \to \mathbb{N} \qquad \text{mit} \qquad c(x, y) = \frac{(x+y)(x+y+1)}{2} + x$$

sowie ihre Umkehrfunktionen

$$c_0^{(2)}(z) = z - \mathsf{bisher}(\mathsf{diag}(z)) \qquad \text{und} \qquad c_1^{(2)}(z) = \mathsf{diag}(z) - c_0^{(2)}(z),$$

wobei

$$\mathsf{bisher}(d) = \frac{d(d+1)}{2} \qquad \text{und} \qquad \mathsf{diag}(z) = \mathsf{max}\{d \in \mathbb{N} \mid \mathsf{bisher}(d) \leq z\}.$$

Man kann zeigen, dass $c^{(2)}$ in der Tat eine Bijektion ist und dass sowohl $c^{(2)}$ als auch die Umkehrfunktionen primitiv rekursiv sind.

Für $n > 2$ erhält man $c^{(n)}$ durch wiederholtes Anwenden von $c^{(2)}$.

Man übersetzt nun jedes Teilprogramm Q von P in eine *einstellige* primitiv rekursive Funktion, indem man die Komposition folgender Schritte bildet: 1) aus dem Eingabewert die Werte aller Variablen dekodieren; 2) Q simulieren; und 3) alle Variablenwerte wieder in einen einzigen Wert kodieren. ❑

16.2. μ-rekursive Funktionen

Aus Satz 16.5 folgt natürlich, dass die primitiv rekursiven Funktionen nicht berechnungsvollständig sind. In diesem Abschnitt erweitern wir die primitiv rekursiven Funktionen zu den μ-rekursiven Funktionen. Dieser Schritt entspricht dem Übergang von LOOP-Programmen zu WHILE-Programmen und liefert insbesondere Berechnungsvollständigkeit.

Die Erweiterung der primitiv rekursiven Funktionen wird durch Hinzunahme des sogenannten μ-Operators erreicht. Die durch diesen Operator beschriebene Operation wird auch als *Minimalisierung* bezeichnet. Intuitiv ist das letzte Argument einer k-stelligen Funktion g als Minimalisierungsargument festgelegt, und das Resultat der Minimalisierung von g ist das kleinste x, für das $g(\ldots, x) = 0$ gilt. So ein x muss natürlich nicht existieren; somit lassen sich mit dem μ-Operator auch echt partielle Funktionen definieren.

Zunächst sieht man leicht, dass sich die Semantik von Komposition und primitiver Rekursion auf partielle Funktionen erweitern lassen. Zum Beispiel ist die Komposition $f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_k(x_1, \ldots, x_n))$ undefiniert, wenn

- eines der $h_i(x_1, \ldots, x_n)$ *undefiniert* ist oder
- alle h_i-Werte *definiert* sind, aber g von diesen Werten *undefiniert* ist.

Auch bei primitiver Rekursion setzen sich undefinierte Werte fort.

Wir wollen nun den μ-Operator präzise definieren.

Definition 16.6

Eine n-stellige Funktion f entsteht aus einer $(n+1)$-stelligen Funktion g *unbeschränkte Minimalisierung* (oder: Anwendung des *μ-Operators*), geschrieben $f = \mu g$, wenn

$$f(x_1, \ldots, x_n) = \mathsf{min}\{y \mid g(x_1, \ldots, x_k, y) = 0 \text{ und } g(x_1, \ldots, x_k, z) \text{ ist definiert für alle } z < y\}.$$

Dabei ist $\mathsf{min}\,\emptyset$ undefiniert.

Man beachte, dass das Ergebnis der Minimalisierung also in zwei Fällen undefiniert ist: (i) wenn kein y existiert, so dass $g(x_1, \ldots, x_k, y) = 0$ gilt und (ii) wenn für das kleinste y mit $g(x_1, \ldots, x_k, y) = 0$ gilt: es gibt ein z mit $0 \leq z < y$ und $g(x_1, \ldots, x_k, z)$ undefiniert.

Beispiele.

- Die 1-stellige, überall undefinierte Funktion undef kann definiert werden als

$$\mathsf{undef}(x) = \mu g(x) \qquad \text{mit} \qquad g(x, y) = \mathsf{succ}(y).$$

  $\mu g(x)$ liefert also unabhängig von der Eingabe x das kleinste y, so dass $y + 1 = 0$; so ein y existiert aber offensichtlich nicht.

- Um die Funktion

$$\mathsf{div}(x_1, x_2) = \begin{cases} \dfrac{x_1}{x_2} & \text{falls } \dfrac{x_1}{x_2} \in \mathbb{N} \\ 0 & \text{falls } x_1 = x_2 = 0 \\ \text{undefiniert} & \text{sonst} \end{cases}$$

  als μ-rekursive Funktion zu definieren, überlegt man sich zunächst leicht, dass die symmetrische Differenz $\mathsf{sdiff}(x_1, x_2) = (x_1 \dot{-} x_2) + (x_2 \dot{-} x_1)$ primitiv rekursiv ist. Dann definiert man

$$\mathsf{div}(x_1, x_2) = \mu g(x_1, x_2) \qquad \text{mit} \qquad g(x_1, x_2, y) = \mathsf{sdiff}(x_1, \mathsf{mult}(x_2, y)).$$

  $\mu g(x_1, x_2)$ liefert also das kleinste y, für das die symmetrische Differenz zwischen x_1 und $x_2 \cdot y$ gleich 0 ist. Offensichtlich ist y genau der gesuchte Teiler bzw. undefiniert, wenn dieser in $\mathbb{N}$ nicht existiert.

Definition 16.7 (Klasse der μ-rekursiven Funktionen)

Die Klasse der *μ-rekursiven Funktionen* besteht aus allen Funktionen, die man aus den primitiv-rekursiven Grundfunktionen durch Anwenden von Komposition, primitiver Rekursion und unbeschränkter Minimalisierung erhält.

Satz 16.8

Die Klasse der μ-rekursiven Funktionen stimmt mit der Klasse der WHILE-berechenbaren Funktionen überein.

Beweis. Wir müssen hierzu den Beweis von Satz 16.5 um die Behandlung des zusätzlichen μ-Operators/WHILE-Konstrukts ergänzen. Wir beschränken uns dabei auf den Nachweis, dass alle μ-rekursiven Funktionen auch WHILE-berechenbar sind.

Es sei $f = \mu g$ und P (nach Induktionsvoraussetzung) ein WHILE-Programm für g. Dann berechnet das folgende Programm die Funktion f:

$x_0 := 0;$
$y := g(x_1, \ldots, x_n, x_0);$ (realisierbar mittels P)
`WHILE` $y \neq 0$ `DO`
    $x_0 := x_0 + 1;$
    $y := g(x_1, \ldots, x_n, x_0);$ (realisierbar mittels P)
`END`

Dieses Programm terminiert nicht, wenn eine der Berechnungen der Funktion g mittels des Programms P nicht terminiert oder wenn y in der WHILE-Schleife niemals den Wert 0 annimmt. In beiden Fällen ist nach Definition von unbeschränkter Minimalisierung aber auch der Wert von μg nicht definiert. ❑

Satz 16.8 liefert einen weiteren Anhaltspunkt für die Gültigkeit der Church-Turing-These. Insgesamt haben wir in den vergangenen Abschnitten gezeigt:

Satz 16.9

Die folgenden Klassen von Funktionen stimmen überein.

1. *Turing-berechenbare Funktionen*
2. *WHILE-berechenbare Funktionen*
3. *μ-rekursive Funktionen*

Es gibt noch weitere äquivalente Berechnungsmodelle; siehe die Liste im Abschnitt „Einführung“ von Teil III.

17. Entscheidbare und unentscheidbare Probleme

Wir wechseln nun von Funktionen und dem mit ihnen verknüpften Berechenbarkeitsbegriff zu Entscheidungsproblemen und dem damit assoziierten Begriff der Entscheidbarkeit.

17.1. Entscheidungsprobleme und Entscheidbarkeit

In der Informatik erfordern viele Probleme nur eine ja-/nein-Antwort anstatt der Berechnung eines „echten“ Funktionswertes, z. B.:

- Das Leerheitsproblem für NEAs (Abschnitt 4): gegeben ein NEA $\mathcal{A}$, ist $L(\mathcal{A}) = \emptyset$?
- Das Wortproblem für kontextfreie Grammatiken (Abschnitt 9): gegeben eine kontextfreie Grammatik G und ein Wort w, ist $w \in L(G)$?
- Das Erreichbarkeitsproblem in gerichteten Graphen: gegeben ein gerichteter Graph G und zwei Knoten x, y, ist y von x aus in G erreichbar?
- etc.

Derartige Probleme nennen wir Entscheidungsprobleme.

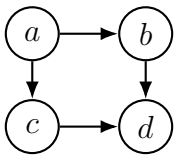
Wir schauen uns drei solche Entscheidungsprobleme genauer an. Das erste ist das bekannte Erreichbarkeitsproblem in gerichteten Graphen:

Definition 17.1

Das *Graph-Erreichbarkeitsproblem (REACH)*[1] ist wie folgt definiert.

Eingabe:	ein gerichteter Graph $G = (V, E)$ und zwei Knoten $s, t \in V$
Frage:	Gibt es einen *Pfad* von s nach t in G, d. h. eine Folge $s = v_0, v_1, v_2, \ldots, v_n = t$ mit $(v_i, v_{i+1}) \in E$ für alle i mit $0 \leq i < n$?

Beispiel.
Im nebenstehenden Graphen gibt es einen Pfad von a nach d (nämlich z. B. a, b, d), aber keinen Pfad von d nach a.



Es ist bekannt, dass REACH in Polynomialzeit lösbar ist, z. B. durch Breitensuche.

Die anderen beiden Entscheidungsprobleme sind Varianten eines anschaulich zu formulierenden Puzzle-Problems. Eine der zwei Varianten wird sich als deutlich schwieriger herausstellen als die andere.

In der einfachen Variante, die wir vorerst mit Pseudo-PKP bezeichnen, sind schwarze und weiße Typen von Spielsteinen gegeben, die mit Wörtern über dem Alphabet $\{a, b\}$

[1] Die Abkürzung steht für engl. *Reachability*.

beschriftet sind. Von jedem Steintyp stehen beliebig viele Steine zur Verfügung. Die Frage, die es zu entscheiden gilt, lautet: Gibt es ein Wort, das man sowohl aus schwarzen als auch weißen Steinen legen kann? Dieses Wort nennen wir auch ein *Lösungswort* des Puzzles.

Sind z. B. die schwarzen Steintypen $\boxed{\boldsymbol{ab}}$, $\boxed{\boldsymbol{ba}}$, $\boxed{\boldsymbol{abb}}$ und die weißen Steintypen $\boxed{bab}$, $\boxed{aa}$, $\boxed{bb}$ gegeben, dann hat das Puzzle ein Lösungswort, nämlich *babaabb*, denn:

$$\boxed{\boldsymbol{ba}} \cdot \boxed{\boldsymbol{ba}} \cdot \boxed{\boldsymbol{abb}} \;=\; babaabb \;=\; \boxed{bab} \cdot \boxed{aa} \cdot \boxed{bb}$$

Das Pseudo-PKP ist also folgendes Entscheidungsproblem.

Eingabe: eine Folge $P = (u_1, v_1), \ldots, (u_k, v_k)$ von Wortpaaren mit $u_i, v_i \in \Sigma^*$, für ein endliches Alphabet Σ

Frage: Gibt es Indexfolgen $i_1, \ldots, i_n$ und $j_1, \ldots, j_m$ mit $n, m > 0$, so dass $u_{i_1} \cdots u_{i_n} = v_{j_1} \cdots v_{j_m}$?

Es wird also nach einem *Lösungswort* $w \in \Sigma^*$ gesucht, das sowohl aus den u_i als auch aus den v_j zusammengesetzt werden kann.

Es ist leicht zu sehen, dass das Pseudo-PKP lösbar ist, wenn man endliche Automaten zu Hilfe nimmt. Folgendes Verfahren entscheidet das Pseudo-PKP in polynomieller Zeit:

1. Konstruiere einen NEA $\mathcal{A}$, der alle Wörter akzeptiert, die aus den u_i zusammengesetzt sind (nur $\mathcal{O}(|u_1| + \cdots + |u_k|)$ viele Zustände).
2. Konstruiere einen NEA $\mathcal{B}$, der alle Wörter akzeptiert, die aus den v_j zusammengesetzt sind (nur $\mathcal{O}(|v_1| + \cdots + |v_k|)$ viele Zustände).
3. Teste, ob $L(\mathcal{A}) \cap L(\mathcal{B}) \neq \emptyset$: konstruiere zunächst den Produktautomaten (Abschnitt 3) und teste dann auf Leerheit (Satz 4.5).

In der schwierigeren Variante ist auch wieder eine Menge von Typen von Spielsteinen gegeben; diesmal sind aber immer ein schwarzer (u_i) und und ein weißer (v_j) miteinander verbunden, ähnlich wie bei einem Dominostein, aber vertikal – oben schwarz, unten weiß. Von jedem Typ stehen wieder beliebig viele Steine zur Verfügung. Die Frage ist nun: Lassen sich die Steine so in einer Reihe auslegen, dass das schwarze (obere) Wort gleich dem weißen (unteren) ist.

Sind z. B. die Steintypen $P_1 = \left\{ \begin{array}{|c|}\hline \boldsymbol{a} \\ \hline aaa \\ \hline \end{array}, \begin{array}{|c|}\hline \boldsymbol{abaa} \\ \hline ab \\ \hline \end{array}, \begin{array}{|c|}\hline \boldsymbol{aab} \\ \hline b \\ \hline \end{array} \right\}$ gegeben, dann sind mögliche Lösungen:

$$\begin{array}{|c|c|}\hline \boldsymbol{a} & \boldsymbol{aab} \\ \hline aaa & b \\ \hline \end{array} \quad \text{oder} \quad \begin{array}{|c|c|}\hline \boldsymbol{abaa} & \boldsymbol{a} \\ \hline ab & aaa \\ \hline \end{array}$$

Andererseits gibt es für die Steintypen $P_2 = \left\{ \begin{array}{|c|}\hline \boldsymbol{ab} \\ \hline aba \\ \hline \end{array}, \begin{array}{|c|}\hline \boldsymbol{baa} \\ \hline aa \\ \hline \end{array}, \begin{array}{|c|}\hline \boldsymbol{aba} \\ \hline baa \\ \hline \end{array} \right\}$ keine Lösung (s. unten).

Dieses Problem nennt man auch das Postsche Korrespondenzproblem. Es wurde 1946 von Emil Post definiert. Wir werden später sehen, dass es unentscheidbar ist und man es

recht bequem verwenden kann, um mittels Reduktion die Unentscheidbarkeit weiterer Probleme zu zeigen. Es ist wie folgt definiert.

Definition 17.2 (Postsches Korrespondenzproblem)

Das *Postsche Korrespondenzproblem (PKP)* ist folgendes Entscheidungsproblem.

Eingabe: eine Folge $P = (u_1, v_1), \ldots, (u_k, v_k)$ von Wortpaaren mit $u_i, v_i \in \Sigma^+$, für ein endliches Alphabet Σ

Frage: Gibt es eine Indexfolge $i_1, \ldots, i_n$ mit $n > 0$, so dass $u_{i_1} \cdots u_{i_n} = v_{i_1} \cdots v_{i_n}$?

Wir nennen dann P eine *Instanz* des PKP und $i_1, \ldots, i_n$ eine *Lösung* von P.

Beispiel. Die Instanz P_1 von oben hat die Folgen $1, 3$ und $2, 1$ als Lösung und damit auch beliebige Konkatenationen dieser Folgen, also $1, 3, 2, 1,\ 1, 3, 1, 3, 1, 3$ usw.

Dass P_2 keine Lösung hat, kann man wie folgt begründen: jede Lösung müsste mit dem Index 1 beginnen, da in allen anderen Paaren das erste Symbol von u_i und v_i verschieden ist. Danach kann man nur mit dem Index 3 weitermachen (beliebig oft). Dabei bleibt die Konkatenation $v_{i_1} \cdots v_{i_n}$ stets länger als die Konkatenation $u_{i_1} \cdots u_{i_n}$.

Die kürzeste Lösung für die Instanz $P_3 = \left[\begin{smallmatrix}aab\\a\end{smallmatrix}\right], \left[\begin{smallmatrix}ab\\abb\end{smallmatrix}\right], \left[\begin{smallmatrix}ab\\bab\end{smallmatrix}\right], \left[\begin{smallmatrix}ba\\aab\end{smallmatrix}\right]$ hat die Länge 66.

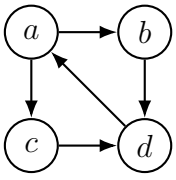
In diesem Abschnitt werden wir zeigen, dass das PKP und weitere Entscheidungsprobleme unentscheidbar sind. Wir betrachten dazu ein Entscheidungsproblem nicht als eine Funktion, sondern als Menge von Wörtern über einem geeigneten Alphabet Σ, also als formale Sprache $L \subseteq \Sigma^*$. Das assoziierte ja-/nein-Problem ist dann: ist ein gegebenes Wort $w \in \Sigma^*$ enthalten in L?

Zur Illustration betrachten wir das PKP, REACH sowie das Wortproblem für kontextfreie Grammatiken:

- Sei Σ ein festes Alphabet. Jede Instanz P des PKP kann als Wort $\mathsf{code}(P) = u_1 \# v_1 \# u_2 \# v_2 \# \cdots \# u_k \# v_k$ über dem Alphabet $\Sigma \cup \{\#\}$ repräsentiert werden, wobei $\# \notin \Sigma$ ein Trennsymbol ist, das es erlaubt, die einzelnen u_i und v_i zu unterscheiden. Das PKP ist dann die Sprache

$$\text{PKP} = \{\mathsf{code}(P) \mid P \text{ hat eine Lösung}\}.$$

- Gerichtete Graphen kodiert man mithilfe von *Adjazenzmatrizen*: z. B. entspricht dem untenstehenden Graphen G die nebenstehende Adjazenzmatrix, wenn man eine beliebige Knotenreihenfolge festlegt (hier a, b, c, d):



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Diese kann wiederum durch zeilenweises Hintereinanderschreiben ihrer Einträge als Wort über dem Alphabet $\{0,1\}$ kodiert werden: im obigen Beispiel ist $\mathsf{code}(G) = 0110000100011000$. Einzelne Knoten kodiert man unär durch ihre Nummer in der festgelegten Reihenfolge. Das Graph-Erreichbarkeitsproblem ist dann die Sprache

$$\text{REACH} = \{\mathsf{code}(G)\#\mathsf{code}(s)\#\mathsf{code}(t) \mid \text{es gibt Pfad von } s \text{ nach } t \text{ in } G\}$$

über dem Alphabet $\{0,1,\#\}$.

- Jede Grammatik $G = (N, \Sigma, P, S)$ kann als Wort $\mathsf{code}(G) \in \Gamma^*$ über einem festen (d. h. nicht von G abhängigen) Alphabet Γ aufgefasst werden (eine Kodierung erhält man analog zu der weiter unten für Turingmaschinen angegebenen). Ebenso kann jedes Wort $w \in \Sigma^*$ als $\mathsf{code}(w) \in \Gamma^*$ kodiert werden. Das Wortproblem für kontextfreie Grammatiken ist dann die Sprache

  $$\{\mathsf{code}(G)\#\mathsf{code}(w) \mid G \text{ kontextfrei}, w \in L(G)\}$$

  über dem Alphabet $\Gamma \cup \{\#\}$, mit $\# \notin \Gamma$.

In diesem Kapitel beweisen wir das fundamentale Resultat, dass es (natürliche und wohldefinierte) Entscheidungsprobleme gibt, die nicht entscheidbar sind, d. h. für die kein Algorithmus existiert, der das gewünschte ja/nein-Resultat berechnet. Wir verwenden hier wieder Turing-Maschinen als zugrunde liegendes Berechnungsmodell, begründet durch die Church-Turing-These und die Bemerkungen am Anfang von Abschnitt 13.

Definition 17.3 (entscheidbar)

Eine Sprache $L \subseteq \Sigma^*$ heißt *entscheidbar*, falls es eine DTM gibt, die bei Eingabe von $w \in \Sigma^*$

- in akzeptierender Stoppkonfiguration anhält, wenn $w \in L$
- in nicht-akzeptierender Stoppkonfiguration anhält, wenn $w \notin L$

Die Eingabe w entspricht dabei wieder der Startkonfiguration $q_0 w$. Wir sagen dann auch, dass die DTM die Sprache L *entscheidet* und ein *Entscheidungsverfahren* für L ist.

Beispiel.

1. Das Pseudo-PKP ist entscheidbar:

   Das obige Verfahren terminiert immer und gibt immer die richtige Antwort. Es kann auf einer DTM implementiert werden.

2. Das Graph-Erreichbarkeitsproblem ist entscheidbar:

   Der Breitensuche-Algorithmus terminiert immer, gibt immer die richtige Antwort und kann auf einer DTM implementiert werden.

3. Das Wortproblem für kontextfreie Grammatiken ist entscheidbar:

   Der CYK-Algorithmus (Algorithmus 2, Satz 9.17) terminiert immer, gibt immer die richtige Antwort und kann auf einer DTM implementiert werden.

Entscheidbarkeit entspricht intuitiv der Existenz eines Algorithmus, der das Entscheidungsproblem (in endlicher Zeit) löst. Ein Vergleich der Definitionen 13.2 (Punkt 2) und 17.3 zeigt, dass jede DTM, die eine Sprache L entscheidet, offensichtlich auch L erkennt. Die Umkehrung gilt jedoch nicht, da für das Entscheiden zusätzlich gefordert ist, dass die DTM auch für Eingaben $w \notin L$ anhält.

Entscheidbarkeit kann als Spezialfall der Berechenbarkeit *totaler* Funktionen gesehen werden: eine Sprache L ist entscheidbar gdw. ihre *charakteristische Funktion*, also die totale Funktion χ_L, definiert durch

$$\chi_L(w) := \begin{cases} 1 & \text{wenn } w \in L \\ 0 & \text{sonst} \end{cases}$$

berechenbar ist.

Man sieht leicht, dass die Klasse der entscheidbaren Sprachen unter Komplement abgeschlossen ist. Diese Eigenschaft werden wird im Folgenden verwenden.

Satz 17.4

Für alle $L \subseteq \Sigma^*$ gilt: ist L entscheidbar, so ist auch das Komplement $\overline{L} = \Sigma^* \setminus L$ entscheidbar.

Beweis. Eine DTM M, die L berechnet, wird wie folgt zu einer DTM für $\overline{L}$ modifiziert: setze $F = Q \setminus F$ (tausche akzeptierende und nicht-akzeptierende Zustände). Diese Konstruktion liefert das gewünschte Resultat, weil M deterministisch ist und auf jeder Eingabe terminiert. ❑

17.2. Diagonalisierung und erste unentscheidbare Probleme

Entscheidbarkeitsbeweise haben wir im Prinzip schon viele geführt, dafür jedoch meist keine Turingmaschinen, sondern abstrakte algorithmische Beschreibungen oder Pseudocode verwendet, siehe oben oder z. B. das Wortproblem für monotone Grammatiken in Satz 14.7.

Aber wie zeigt man, dass ein Problem *nicht* entscheidbar ist? Wir werden zwei Verfahren kennen lernen: *Diagonalisierung* und *Reduktion*. Unser erstes Unentscheidbarkeitsresultat wird Diagonalisierung verwenden. Haben wir aber erst die Unentscheidbarkeit einiger Probleme bewiesen, so ist danach meist Reduktion der einfachere Weg, um die Unentscheidbarkeit weiterer Probleme nachzuweisen.

Für die Diagonalisierung müssen wir Turingmaschinen als Eingaben für Turingmaschinen verwenden. Zu diesem Zweck überzeugen wir uns zunächst, dass Turingmaschinen leicht als Wörter kodiert werden können.

Konventionen:

- *Arbeitsalphabete* der betrachteten Turingmaschinen sind endliche Teilmengen von $\{a_0, a_1, a_2, \ldots\}$, wobei a_0 das Blank-Symbol ist (wir schreiben wie gehabt $\not b$).
- *Zustandsmengen* sind endliche Teilmengen von $\{q_0, q_1, q_2, \ldots\}$, wobei q_0 stets der *Anfangszustand* ist.

Es sei $M = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ eine Turingmaschine, die obige Konventionen erfüllt.

1) Eine *Transition*
$$t = (q_i, a_j, a_{j'}, m, q_{i'})$$
wird *kodiert* durch
$$\mathsf{code}(t) = \mathsf{bin}(i)\#\mathsf{bin}(j)\#\mathsf{bin}(j')\#m\#\mathsf{bin}(i')$$
wobei $\mathsf{bin}(n)$ die Binärdarstellung einer Zahl ist.

2) Besteht Δ aus den Transitionen $t_1, \ldots, t_k$ und ist $F = \{q_{i_1}, \ldots, q_{i_r}\}$ und $\Sigma = \{a_{j_1}, \ldots, a_{j_s}\}$, so wird M *kodiert* durch $\mathsf{code}(M) =$
$$\mathsf{code}(t_1)\#\#\cdots\#\#\mathsf{code}(t_k)\#\#\#\mathsf{bin}(i_1)\#\cdots\#\mathsf{bin}(i_r)\#\#\#\mathsf{bin}(j_1)\#\cdots\#\mathsf{bin}(j_s)$$

Beachte, dass die einzelnen Transitionen durch ## getrennt sind und die Kodierung der Übergangsrelation durch ### abgeschlossen wird.

Mit dieser Kodierung gibt es jedoch Wörter $w \in \{0, 1, \ell, n, r, \#\}^*$, die keine DTM kodieren (z. B. ℓnr, 101010 oder #####). Um zu erreichen, dass *jedes* Wort eine DTM kodiert, wählen wir eine beliebige, aber feste DTM M_0. Jedem Wort, das gemäß des obigen Schemas nicht der Kodierung einer DTM entspricht, weisen wir *per Konvention* die DTM M_0 zu. Man beachte, dass sich aus einem gegebenen Wort $w \in \{0, 1, \ell, n, r, \#\}^*$ trotzdem leicht die kodierte DTM extrahieren lässt. Wir bezeichnen die durch ein Wort $w \in \{0, 1, \ell, n, r, \#\}^*$ kodierte DTM mit M_w.

Nun können wir uns der Existenz unentscheidbarer Probleme zuwenden. Es ist natürlich zunächst einmal gar nicht klar, ob es solche Probleme überhaupt gibt. Wir beginnen mit einer Variante des Wortproblems für DTMs, dem *speziellen Wortproblem* für DTMs:

Eingabe:	ein Wort $w \in \Sigma^*$ für ein Alphabet Σ
Frage:	Ist $w \in L(M_w)$?

Mit anderen Worten: akzeptiert eine DTM *ihre eigene Kodierung als Eingabe*? Wir verwenden eine Technik, die auf Alan Turing zurückgeht und auf Diagonalisierung nach Georg Cantor beruht.

Satz 17.5

Das spezielle Wortproblem für DTMs ist unentscheidbar.

Beweis. Mit Satz 17.4 genügt es zu zeigen, dass das Komplement des speziellen Wortproblems unentscheidbar ist, also die Sprache

$$L_D = \{w \mid w \notin L(M_w)\}.$$

Wir führen einen Widerspruchsbeweis. Angenommen, L_D wäre entscheidbar. Dann gibt es eine konkrete DTM M, die L_D entscheidet, und es gibt ein Wort $w \in \{0, 1, \ell, n, r, \#\}^*$ mit $M = M_w$. Es gilt nun:

$$\begin{array}{lll} w \in L(M_w) & \text{gdw. } w \in L_D & (\text{denn } L(M_w) = L_D) \\ & \text{gdw. } w \notin L(M_w) & (\text{nach Def. } L_D) \end{array}$$

Widerspruch. ❏

Die Sprache L_D im vorangehenden Beweis wird auch die *Diagonalisierungssprache* genannt. Dabei bezieht sich der Begriff „Diagonalisierung“ auf die Darstellung der Zugehörigkeit der Wörter $w_0, w_1, \ldots$ zu den Sprachen $L(M_{w_0}), L(M_{w_1}), \ldots$ als zweidimensionale Matrix: der Eintrag einer Matrixzelle ist 1, wenn das Wort zur Sprache gehört, und 0 sonst. L_D unterscheidet sich nun von *jeder* Sprache $L(M_{w_i})$ genau im Wort w_i, d. h. genau auf der Diagonalen der Matrix.

Damit ist auch offensichtlich, dass L_D nicht in der Liste $L(M_{w_0}), L(M_{w_1}), \ldots$ vorkommt, im Widerspruch dazu, dass L_D entscheidbar ist.

Aus der Unentscheidbarkeit des speziellen Wortproblems für DTMs kann man die Unentscheidbarkeit weiterer, natürlicherer Probleme herleiten. Wir betrachten zunächst das (allgemeine) Wortproblem für DTMs.

Satz 17.6

Das Wortproblem für DTM ist unentscheidbar, d. h. es gibt kein Entscheidungsverfahren dafür, ob eine gegebene DTM M ein gegebenes Eingabewort w akzeptiert.

Beweis. Wäre das Wortproblem entscheidbar, so auch das spezielle Wortproblem. Um für ein gegebenes Wort w zu entscheiden, ob $w \in L(M_w)$, müsste man dann nämlich nur die DTM für das (allgemeine) Wortproblem starten und ihr als Eingabe w (als Kodierung der DTM M_w) sowie nochmals w (als Eingabe für M_w) übergeben. ❏

Aus Satz 17.6 folgt auch die Unentscheidbarkeit des Wortproblems für Typ-0-Grammatiken, da die im Beweis von Satz 14.1 beschriebene Übersetzung derartiger Grammatiken in DTMs *effektiv* ist, d. h. mittels DTMs implementierbar. Dadurch könnte ein Entscheidungsverfahren für Typ-0-Grammatiken leicht in ein Entscheidungsverfahren für DTMs umgewandelt werden.

Als nächstes betrachten wir ein weiteres klassisches Problem für Turingmaschinen: das Halteproblem. Wenn man „DTM“ mit „Programm“ gleichsetzt, so entspricht das Halteproblem einer wichtigen Fragestellung aus der Programmanalyse: lässt sich automatisch

feststellen, ob ein Programm auf einer gegebenen Eingabe stoppt (oder in eine Endlosschleife gerät)? Das folgende Unentscheidbarkeitsresultat zeigt, dass das unmöglich ist.

Satz 17.7

Das Halteproblem für DTM ist unentscheidbar, d. h. es gibt kein Entscheidungsverfahren dafür, ob eine gegebene DTM M auf einer gegebenen Eingabe w anhält.

Beweis. Wir zeigen: wäre das Halteproblem entscheidbar, so auch das Wortproblem.

Um von einer gegebenen DTM M und einem gegebenen Wort w zu entscheiden, ob $w \in L(M)$, könnte man dann nämlich wie folgt vorgehen:

Modifiziere M zu einer TM $\widehat{M}$:

- $\widehat{M}$ verhält sich zunächst wie M.
- Stoppt M mit *akzeptierender* Stoppkonfiguration, so stoppt auch $\widehat{M}$.
  Stoppt Mmit *nicht-akzeptierender* Stoppkonfiguration, so geht $\widehat{M}$ in Endlosschleife.

Damit gilt:

$$M \text{ akzeptiert } w \quad \text{gdw.} \quad \widehat{M} \text{ hält auf Eingabe } w.$$

Mit dem Entscheidungsverfahren für das Halteproblem, angewandt auf $\widehat{M}$ und w, könnte man also das Wortproblem „ist w in $L(M)$“ entscheiden. ❑

17.3. Reduktionen

Das in den Beweisen der Sätze 17.6 und 17.7 gewählte Vorgehen nennt man *Reduktion*:

- Das Lösen eines Problems L_1 (z. B. Wortproblem) wird auf das Lösen eines Problems L_2 (z. B. Halteproblem) reduziert.
- Wäre daher L_2 entscheidbar, so auch L_1.
- Weiß man bereits, dass L_1 unentscheidbar ist, so folgt daher, dass auch L_2 unentscheidbar ist.

Reduktionen sind ein sehr wichtiges Hilfsmittel, um die Unentscheidbarkeit von Problemen nachzuweisen. In der Tat wird dieser Ansatz wesentlich häufiger verwendet als Diagonalisierung (die aber trotzdem unverzichtbar ist, um sich erstmal ein originäres unentscheidbares Problem zu schaffen, das man dann reduzieren kann). Genau definiert sind Reduktionen wie folgt.

Definition 17.8 (Reduktion)

1) Eine *Reduktion von* $L_1 \subseteq \Sigma^*$ *auf* $L_2 \subseteq \Sigma^*$ ist eine (totale) berechenbare Funktion

$$f : \Sigma^* \to \Sigma^*,$$

so dass für alle $w \in \Sigma^*$:

$$w \in L_1 \text{ gdw. } f(w) \in L_2. \qquad (*)$$

2) Wir schreiben

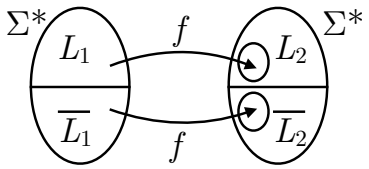
$$L_1 \leq L_2 \quad (L_1 \text{ ist auf } L_2 \textit{ reduzierbar}),$$

falls es eine Reduktion von L_1 nach L_2 gibt.

Die Eigenschaft $(*)$ lässt sich äquivalent auch so formulieren:

- Wenn $w \in L_1$, dann $f(w) \in L_2$ und
- wenn $w \notin L_1$, dann $f(w) \notin L_2$.

Man sagt auch, dass f alle ja-Instanzen des Problems L_1 auf ja-Instanzen von L_2 und alle nein-Instanzen auf nein-Instanzen abbildet. Anschaulich und informell kann man das so darstellen:



Beispiel 17.9

In Abschnitt 4 haben wir gesehen, dass sich das Leerheitsproblem für endliche Automaten im Grunde wie das Erreichbarkeitsproblem für gerichtete Graphen lösen lässt. Diesen Zusammenhang präzisieren wir jetzt, indem wir zeigen, dass das Komplement des Leerheitsproblems auf das Erreichbarkeitsproblem reduzierbar ist. Sei dazu $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein beliebiger NEA (als Wort $w = \mathsf{code}(\mathcal{A})$ kodiert, ganz analog zur Kodierung von NTMs). Die Reduktionsfunktion f bildet nun jedes Wort auf das Wort $\mathsf{code}(G_\mathcal{A})\#\mathsf{code}(s)\#\mathsf{code}(t)$ ab, das das Tripel $(G_\mathcal{A}, s, t)$ kodiert, wobei $G_\mathcal{A}$ den Übergangsgraphen von $\mathcal{A}$ repräsentiert und s, t zwei neue Knoten sind, die eine ausgehende Kante zum Startzustand bzw. eingehende Kanten von allen akzeptierenden Zuständen haben. Genauer:

$$\begin{aligned} G_\mathcal{A} &= (V_\mathcal{A}, E_\mathcal{A}) \\ V_\mathcal{A} &= Q \uplus \{s, t\} \\ E_\mathcal{A} &= \{(s, q_0)\} \cup \{(q, t) \mid q \in F\} \cup \{(q, q') \mid (q, a, q') \in \Delta,\, a \in \Sigma\} \end{aligned}$$

Es ist klar, dass f eine geeignete Reduktionsfunktion ist, denn:

- $L(\mathcal{A})$ ist nicht leer gdw. es gibt Pfad von s nach t in $G_\mathcal{A}$.
- f ist berechenbar (d. h. es gibt eine DTM, die bei Eingabe $w = \mathsf{code}(\mathcal{A})$ die den Funktionswert $f(w) = \mathsf{code}(G_\mathcal{A})\#\mathsf{code}(s)\#\mathsf{code}(t)$ berechnet).

Als ein weiteres Beispiel betrachten wir eine Reduktion vom PKP auf das Schnitt-Leerheitsproblem für kontextfreie Grammatiken, die uns später noch nützlich sein wird:

Definition 17.10

Das *Schnitt-Leerheitsproblem für kontextfreie Grammatiken* ist:

Eingabe: zwei kontextfreie Grammatiken G_1, G_2
Frage: Gilt $L(G_1) \cap L(G_2) = \emptyset$?

Lemma 17.11

Das PKP ist auf das Komplement des Schnitt-Leerheitsproblems für kontextfreie Grammatiken reduzierbar.

Beweis. Dazu müssen wir zeigen:

Zu jeder gegebenen Instanz P des PKP kann eine TM kontextfreie Grammatiken $G_P^{(\ell)}, G_P^{(r)}$ konstruieren, so dass gilt:

$$P \text{ hat Lösung} \quad \text{gdw.} \quad L(G_P^{(\ell)}) \cap L(G_P^{(r)}) \neq \emptyset$$

Es sei $P = (u_1, v_1), \ldots, (u_k, v_k)$. Wir definieren $G_P^{(\ell)} = (N_\ell, \Sigma_\ell, P_\ell, S_\ell)$ mit

- $N_\ell = \{S_\ell\}$,
- $\Sigma_\ell = \Sigma \cup \{1, \ldots, k\}$ und
- $P_\ell = \{S_\ell \longrightarrow u_i S_\ell i, S_\ell \longrightarrow u_i i \mid 1 \leq i \leq k\}$.

$G_P^{(r)}$ wird entsprechend definiert. Es gilt:

$$L(G_P^{(\ell)}) = \{u_{i_1} \ldots u_{i_m} i_m \ldots i_1 \mid m \geq 1, i_j \in \{1, \ldots, k\}\}$$

$$L(G_P^{(r)}) = \{v_{i_1} \ldots v_{i_m} i_m \ldots i_1 \mid m \geq 1, i_j \in \{1, \ldots, k\}\}$$

Daraus folgt nun unmittelbar:

$$\begin{aligned} & L(G_P^{(\ell)}) \cap L(G_P^{(r)}) \neq \emptyset \\ \text{gdw.} \quad & \exists m \geq 1 \; \exists i_1, \ldots, i_m \in \{1, \ldots, k\} : u_{i_1} \ldots u_{i_m} i_m \ldots i_1 = v_{i_1} \ldots v_{i_m} i_m \ldots i_1 \\ \text{gdw.} \quad & P \text{ hat Lösung.} \end{aligned}$$

❑

Offensichtlich ist die Komposition zweier berechenbarer Funktionen wieder eine berechenbare Funktion. Daher ist „$\leq$“ eine *Ordnungsrelation*:

Lemma 17.12

Wenn $L_1 \leq L_2$ und $L_2 \leq L_3$, dann $L_1 \leq L_3$.

Die intuitive Bedeutung von „$L_1 \leq L_2$“ ist, dass L_1 *höchstens so schwer* ist wie L_2 oder – im Umkehrschluss – dass L_2 *mindestens* so schwer ist wie L_1. Mit dieser Intuition sind auch die wichtigsten Zusammenhänge zwischen Reduzierbarkeit und (Un)entscheidbarkeit im Einklang, die im folgenden Lemma angegeben sind.

Lemma 17.13

1) $L_1 \leq L_2$ *und* L_2 *entscheidbar* $\Rightarrow$ L_1 *entscheidbar.*

2) $L_1 \leq L_2$ *und* L_1 *unentscheidbar* $\Rightarrow$ L_2 *unentscheidbar.*

Beweis.

1) Um „$w \in L_1$“ zu entscheiden,
   - berechnet man $f(w)$ und
   - entscheidet „$f(w) \in L_2$“.
2) Folgt unmittelbar aus 1).

❑

Beispiel 17.9, Fortsetzung. Aus Lemma 17.13 (1) und der Entscheidbarkeit des Graph-Erreichbarkeitsproblems folgt, dass auch das Komplement des Leerheitsproblems für NEAs entscheidbar ist und – wegen Satz 17.4 – auch das Leerheitsproblem selbst.

Man kann auch anders herum das Graph-Erreichbarkeitsproblem auf das Komplement des Leerheitsproblems für NEAs reduzieren. Intuitiv gesprochen sind dann beide Probleme *gleich schwer.*

Als ein Beispiel für die Anwendung von Lemma 17.13 (2) kann man die Konstruktion im Beweis von Satz 17.7 betrachten: die Konstruktion von $\widehat{M}$ beschreibt eine Funktion f mit $f(\mathsf{code}(M)\#\mathsf{code}(w)) = \widehat{M}$ für jede DTM M und jedes Eingabewort w. Diese Funktion ist eine Reduktion gemäß Definition 17.8 vom bereits als unentscheidbar nachgewiesenen Wortproblem für DTM auf das Halteproblem.

Wir werden im Folgenden noch weitere Beispiele für Reduktionen sehen. Zunächst zeigen wir jedoch mit Hilfe einer Reduktion des Halteproblems folgendes sehr starke Resultat.

17.4. Der Satz von Rice

Jede *nichttriviale semantische Eigenschaft* von Programmen (DTM) ist *unentscheidbar.*

- *Semantisch* heißt hier: Die Eigenschaft hängt nicht von der syntaktischen Form des Programms (der DTM), sondern nur von der erkannten Sprache ab.
- *Nichttrivial:* Es gibt Turing-erkennbare Sprachen, die die Eigenschaft erfüllen, aber nicht alle Turing-erkennbaren Sprachen erfüllen sie.

Zum Beispiel ist eine semantische und nichttriviale Eigenschaft, ob die von einer DTM erkannte Sprache leer ist. Also ist die Unentscheidbarkeit des Leerheitsproblems eine konkrete Instanz des hier bewiesenen, sehr allgemeinen Resultates. Da nach der Church-Turing-These DTMs äquivalent zu anderen Berechnungsmodellen wie etwa WHILE-Programmen sind, kann man diese Aussage intuitiv so verstehen, dass *alle interessanten Eigenschaften*, die das *Verhalten von Programmen* betreffen, unentscheidbar sind. Dies hat weitreichende Konsequenzen im Gebiet der Programmverifikation, wo man Programme automatisch auf ihre Korrektheit prüfen möchte.

Ein Beispiel für eine *nicht* semantische Eigenschaft ist zum Beispiel: die DTM macht auf dem leeren Wort mehr als 100 Schritte.

Wir setzen im Folgenden semantische Eigenschaften von DTMs mit Eigenschaften der von ihnen erkannten Sprachen gleich: eine *Eigenschaft Turing-erkennbarer Sprachen* ist eine Menge

$$E \subseteq \{L \subseteq \Sigma^* \mid L \text{ ist Turing-erkennbar}\}.$$

Eine Sprache L *erfüllt* E, falls $L \in E$.

Das angekündigte Resultat lautet wie folgt.

Satz 17.14 (Satz von Rice)
Es sei E eine Eigenschaft Turing-erkennbarer Sprachen, so dass gilt:

$$\emptyset \subsetneq E \subsetneq \{L \subseteq \Sigma^* \mid L \textit{ ist Turing-erkennbar}\}.$$

Dann ist die Sprache

$$L(E) := \{\mathsf{code}(M) \mid M \textit{ DTM mit } L(M) \in E\}$$

unentscheidbar.

Beweis. Sei E wie in Satz 17.14. Wir geben eine Reduktion des Halteproblems auf die Sprache $L(E)$ an, müssen also Folgendes zeigen.

Gegeben eine DTM M und ein Eingabewort w für M, kann man eine DTM $\widehat{M}$ konstruieren, so dass gilt:

$$M \text{ hält auf } w \quad \text{gdw.} \quad L(\widehat{M}) \in E.$$

Die Reduktionsfunktion f aus Definition 17.8 bildet also die Eingabe (M, w) für das Halteproblem auf $f(M, w) = \widehat{M}$ ab. Beachte, dass die Funktion f berechenbar sein muss, d. h. die Konstruktion von $\widehat{M}$ aus M muss effektiv mittels einer DTM realisierbar sein.

Wir unterscheiden 2 Fälle.

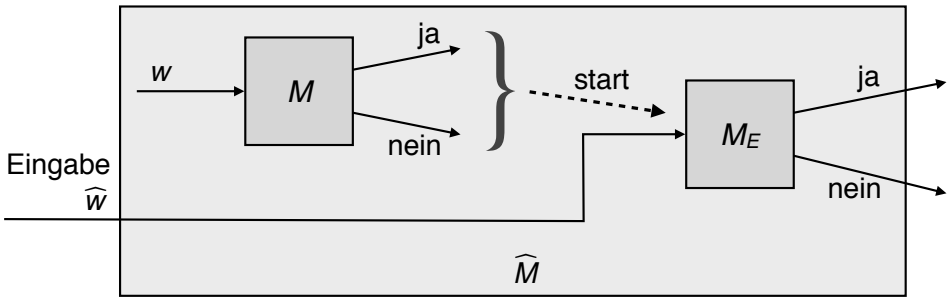
1. Fall: die leere Sprache erfüllt E *nicht.*

Zur Konstruktion von $\widehat{M}$ verwenden wir eine Sprache $L_E \in E$ und eine dazugehörige DTM M_E mit $L(M_E) = L_E$ (so eine Sprache und DTM existieren, da E nichttrivial).

Konstruiere nun zu gegebener DTM M und Eingabe w für M die DTM $\widehat{M}$ wie folgt:

- $\widehat{M}$ speichert zunächst die Eingabe $\widehat{w}$ zur späteren Verwendung.
- Danach schreibt $\widehat{M}$ das Wort w auf das Eingabeband und verhält sich genau wie M. Das heißt insbesondere: wenn M nicht terminiert, so terminiert auch $\widehat{M}$ nicht.
- Wenn M terminiert, so wird die ursprüngliche Eingabe $\widehat{w}$ auf dem Eingabeband wiederhergestellt; dann verhält sich $\widehat{M}$ wie die Maschine M_E.

Anschaulich lässt sich die Arbeitsweise von $\widehat{M}$ so darstellen:



Man kann prüfen, dass $\widehat{M}$ sich wie gewünscht verhält:

1. Wenn M auf w anhält, dann erkennt $\widehat{M}$ die Sprache L_E, also $L(\widehat{M}) \in E$.
2. Wenn M auf w nicht anhält, dann erkennt $\widehat{M}$ die leere Sprache, also $L(\widehat{M}) \notin E$.

2. Fall: die leere Sprache erfüllt E. Dann betrachten wir stattdessen die Komplementäreigenschaft

$$\overline{E} = \{L \subseteq \Sigma^* \mid L \text{ ist Turing-erkennbar}\} \setminus E.$$

Diese wird dann nicht von der leeren Sprache erfüllt und wir können wie im 1. Fall zeigen, dass $L(\overline{E})$ unentscheidbar ist. Unentscheidbarkeit von $L(E)$ folgt dann mit Satz 17.4 und der Beobachtung, dass

$$L(\overline{E}) = \overline{L(E)}.$$

❑

Man kann den Satz von Rice und dessen Beweis sehr bequem benutzen, um für viele Probleme die Unentscheidbarkeit zu zeigen und, falls gewünscht, sogar eine konkrete Reduktion vom Halteproblem zu finden. Dazu kann man folgende Vorgehensweise nutzen:

1. Ist die Eigenschaft E trivial?

   Wenn ja, dann ist sie entscheidbar.

2. Ist E semantisch?

   Wenn nein, dann hilft der Satz von Rice nicht.
   Wenn ja, ist E unentscheidbar; eine Reduktion findet man wie folgt.

3. Erfüllt die leere Sprache E?

   Wenn nein, dann reduziere das Halteproblem auf E wie im Beweis beschrieben.
   Wenn ja, reduziere das Halteproblem auf das Komplement $\overline{E}$.

Diese Vorgehensweise sei am Beispiel des Leerheitsproblems für DTMs demonstriert.

Satz 17.15

Das Leerheitsproblem für DTM ist unentscheidbar.

Beweis. Folgt direkt aus dem Satz von Rice, denn die zugehörige Eigenschaft $E = \{\emptyset\}$ (nur die leere Sprache!) ist

1. nichttrivial (manche Turing-erkennbaren Sprachen sind leer, manche nicht)
2. und semantisch (Leerheit ist eine Eigenschaft einer Sprache, unabhängig von der konkreten Form der DTM).

Außerdem ist das Leerheitsproblem genau die Sprache

$$L(E) = \{\mathsf{code}(M) \mid M \text{ DTM mit } L(M) = \emptyset\},$$

welche also laut Satz 17.14 unentscheidbar ist. ❑

Punkt 3 der obigen Vorgehensweise liefert uns auch noch eine konkrete Reduktion vom Halteproblem: Da die leere Sprache E erfüllt (hier ist die leere Sprache sogar das einzige Element in E), kann man aus dem Beweis von Satz 17.14 eine Reduktion vom Halteproblem auf das *Komplement* des Leerheitsproblems ablesen:

Als Sprache $L_{\overline{E}} \in \overline{E}$ laut Punkt (a) müssen wir eine nichtleere Sprache wählen – am einfachsten ist hier die Sprache Σ^*. Also können wir die zugehörige DTM $M_{\overline{E}}$ so wählen, dass sie immer sofort akzeptiert (es gibt nur einen Zustand, der gleichzeitig Anfangs- und akzeptierender Zustand ist und keinen Folgezustand hat).

Um bei gegebener DTM M zu entscheiden, ob M auf Eingabe w *nicht* hält, konstruiert man die DTM $\widehat{M}$ wie folgt:

- $\widehat{M}$ speichert ihre Eingabe $\widehat{w}$ (hier könnte man $\widehat{w}$ auch löschen, weil der letzte Schritt nicht von $\widehat{w}$ abhängt).
- Danach schreibt $\widehat{M}$ das Wort w aufs Eingabeband und verhält sich wie M.
- Stoppt die Berechnung, so wird $M_{\overline{E}}$ gestartet – d. h. $\widehat{M}$ akzeptiert direkt.

Dann gilt: M hält auf Eingabe w gdw. $L(\widehat{M}) \neq \emptyset$.

Um zu zeigen, dass das Äquivalenzproblem für DTM unentscheidbar ist, kann man den Satz von Rice leider nicht verwenden, denn es handelt sich um eine (nichttriviale semantische) Eigenschaft *zweier* Turing-erkennbarer Sprachen. Es lässt sich aber bequem eine Reduktion vom Leerheitsproblem finden.

Satz 17.16

Das Äquivalenzproblem für DTM ist unentscheidbar.

Beweis. Eine Reduktion des Leerheitsproblems auf das Äquivalenzproblem ist wie folgt:

Gegeben eine DTM M (Eingabe für das Leerheitsproblem), konstruiere zusätzlich eine DTM $M_\emptyset$ mit $L(M_\emptyset) = \emptyset$ (so dass M und $M_\emptyset$ Eingaben für das Äquivalenzproblem sind). Dann gilt: $L(M) = \emptyset$ gdw. $L(M) = L(M_\emptyset)$. ❑

17.5. Weitere unentscheidbare Probleme

Die bisher als unentscheidbar nachgewiesenen Probleme beziehen sich allesamt auf (semantische) Eigenschaften von Turingmaschinen bzw. von Programmen. Derartige Probleme spielen im Gebiet der automatischen Programmverifikation eine wichtige Rolle. Es gibt aber auch eine große Zahl von unentscheidbaren Problemen, die (direkt) nichts mit Programmen zu tun haben. In diesem Teilabschnitt betrachten wir einige ausgewählte Probleme dieser Art.

Wir wollen zuerst zeigen, dass das PKP unentscheidbar ist. Dazu führen wir zunächst ein Zwischenproblem ein, das *modifizierte PKP (MPKP)*:

Hier muss für die Lösung zusätzlich $i_1 = 1$ gelten, d. h. das Wortpaar, mit dem die Lösung beginnen muss, ist festgelegt.

Lemma 17.17

Das MPKP ist auf das PKP reduzierbar.

Beweis. Es sei $P = (u_1, v_1), \ldots, (u_k, v_k)$ eine Instanz des MPKP über dem Alphabet Σ. Es seien $\#, \$$ Symbole, die nicht in Σ vorkommen.

Wir definieren die Instanz $\widehat{P}$ des PKP über $\widehat{\Sigma} = \Sigma \cup \{\#, \$\}$ wie folgt:

$$\widehat{P} := (u'_0, v'_0), (u'_1, v'_1), \ldots, (u'_k, v'_k), (u'_{k+1}, v'_{k+1}),$$

wobei gilt:

- Für $1 \leq i \leq k$ entsteht u'_i aus u_i, indem man *hinter jedem* Symbol ein $\#$ einfügt. Ist z. B. $u_i = abb$, so ist $u'_i = a\#b\#b\#$.
- Für $1 \leq i \leq k$ entsteht v'_i aus v_i, indem man *vor jedem* Symbol ein $\#$ einfügt.
- $u'_0 := \#u'_1$ und $v'_0 := v'_1$
- $u'_{k+1} := \$$ und $v'_{k+1} := \#\$$

Offenbar ist die Reduktion berechenbar. Es bleibt zu zeigen, dass gilt:

$$\text{Das MPKP } P \text{ hat eine Lösung} \quad \text{gdw.} \quad \text{das PKP } \widehat{P} \text{ hat eine Lösung.}$$

„$\Rightarrow$“. Wenn $i_1, \ldots, i_m$ Lösung für das MPKP P, dann muss $i_1 = 1$ sein. Man überprüft leicht, dass dann $1, i_2 + 1, \ldots, i_m + 1, k + 2$ eine Lösung für das PKP $\widehat{P}$ ist (beachte: wir beginnen die Zählung der Paare immer bei 1; der Index 1 bezeichnet im Fall von $\widehat{P}$ also das Paar (u'_0, v'_0)).

„$\Leftarrow$“. Sei $i_1, \ldots, i_m$ Lösung für das PKP $\widehat{P}$. Dann ist $i_1 = 1$, weil in allen Paaren außer dem ersten die Wörter u'_i, v'_i mit unterschiedlichen Symbolen beginnen. Außerdem muss es ein $\ell \in \{1, \ldots, m\}$ geben mit $i_\ell = k + 2$, weil vor Anwendung des Paares $k + 2$ die linke Konkatenation $u'_{i_1} u'_{i_2} \cdots u'_{i_j}$ immer mit $\#$ endet, die rechte Konkatenation

$v'_{i_1} v'_{i_2} \cdots v'_{i_j}$ jedoch nicht. Sei ℓ minimal mit dieser Eigenschaft. Man prüft leicht, dass $1, i_2 - 1, \ldots, i_{\ell-1} - 1$ eine Lösung für das MPKP P ist. ❑

Beispiel:

$P = (a, aba), (bab, b)$ ist als MPKP lösbar mit Lösung $1, 2$. Die Sequenz $2, 1$ liefert zwar identische Konkatenationen, ist aber im MPKP nicht zulässig. Die Konstruktion liefert:

$$\begin{array}{rc} & \begin{array}{cccc} (u'_0, v'_0) & (u'_1, v'_1) & (u'_2, v'_2) & (u'_3, v'_3) \end{array} \\ \widehat{P} = & (\#a\#, \#a\#b\#a), (a\#, \#a\#b\#a), (b\#a\#b\#, \#b), (\$, \#\$) \end{array}$$

Die Lösung $1, 2$ von P liefert die Lösung $0, 2, 3$ von $\widehat{P}$:

$\#a\#|\, b\#a\#b\#|\$$
$\#a\#\, b\#a|\#b|\#\$$

Die Sequenz $2, 1$ liefert keine Lösung von $\widehat{P}$: wegen der Verwendung des #-Symbols muss jede Lösung von $\widehat{P}$ mit Index 0 anfangen. Dies entspricht wie gewünscht Lösungen von P, die mit Index 1 beginnen.

Wäre daher das PKP entscheidbar, so auch das MPKP. Um die Unentscheidbarkeit des PKP zu zeigen, genügt es also zu zeigen, dass das MPKP unentscheidbar ist.

Lemma 17.18

Das Halteproblem ist auf das MPKP reduzierbar.

Beweis. Gegeben sei eine DTM $M = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ und ein Eingabewort $w \in \Sigma^*$.

Wir müssen zeigen, wie eine TM eine gegebene Eingabe M, w in eine Instanz $P_{M,w}$ des MPKP überführen kann, so dass gilt:

$$M \text{ hält auf Eingabe } w \;\text{ gdw. }\; P_{M,w} \text{ hat eine Lösung.}$$

Wir verwenden für das MPKP $P_{M,w}$ das Alphabet $\Gamma \cup Q \cup \{\#\}$ mit $\# \notin \Gamma \cup Q$. Das MPKP $P_{M,w}$ besteht aus den folgenden Wortpaaren:

1) Anfangsregel:
$$(\#,\ \#q_0 w\#)$$

2) Kopierregeln:
$$(a,\ a) \text{ für alle } a \in \Gamma \cup \{\#\}$$

3) Übergangsregeln:

$$\begin{aligned}
(qa,\ q'a') &\quad \text{falls } (q,a,a',n,q') \in \Delta \\
(qa,\ a'q') &\quad \text{falls } (q,a,a',r,q') \in \Delta \\
(bqa,\ q'ba') &\quad \text{falls } (q,a,a',l,q') \in \Delta \text{ und } b \in \Gamma \\
(\#qa,\ \#q'\not\flat\, a') &\quad \text{falls } (q,a,a',l,q') \in \Delta \\
(q\#,\ q'a'\#) &\quad \text{falls } (q,\not\flat,a',n,q') \in \Delta \\
(q\#,\ a'q'\#) &\quad \text{falls } (q,\not\flat,a',r,q') \in \Delta \\
(bq\#,\ q'ba'\#) &\quad \text{falls } (q,\not\flat,a',l,q') \in \Delta \\
(\#q\#,\ \#q'\not\flat\, a'\#) &\quad \text{falls } (q,\not\flat,a',l,q') \in \Delta
\end{aligned}$$

4) Löschregeln:

(aq, q) und (qa, q) für alle $a \in \Gamma$ und $q \in Q$ Stoppzustand

(O.B.d.A. hänge in M das Stoppen nur vom erreichten Zustand, aber nicht vom gerade gelesenen Bandsymbol ab; ein *Stoppzustand* ist dann ein Zustand q, so dass die TM in q bei jedem gelesenen Symbol anhält.)

5) Abschlussregel:

$(q\#\#, \#)$ für alle $q \in Q$ mit q Stoppzustand

Falls M bei Eingabe w hält, so gibt es eine Berechnung

$$k_0 \vdash_M k_1 \vdash_M \ldots \vdash_M k_t$$

mit $k_0 = q_0 w$ und $k_t = u\hat{q}v$ mit $\hat{q}$ Endzustand.

Daraus kann man eine Lösung des MPKP bauen. Zunächst erzeugt man

$$\begin{aligned}
&\#k_0\#k_1\#k_2\#\ldots\# \\
&\#k_0\#k_1\#k_2\#\ldots\#k_t\#
\end{aligned}$$

- Dabei beginnt man mit $(\#, \#k_0\#)$.
  - Durch Kopierregeln erzeugt man die Teile von k_0 und k_1, die sich nicht unterscheiden.
  - Der Teil, der sich unterscheidet, wird durch die entsprechende Übergangsregel realisiert.

  z. B. $(q_0, a, a', r, q_1) \in \Delta$ und $k_0 = q_0ab$

  $\#|q_0\,a|b|\#$
  $\#\ q_0\ a\ b\ \#\,|\,a'q_1|\,b\,|\,\#$

  Man erhält so:

  $\#k_0\#$
  $\#k_0\#k_1\#$

- Nun macht man dies so weiter, bis die Stoppkonfiguration k_t mit Stoppzustand $\hat{q}$ erreicht ist. Durch Verwenden von Löschregeln und Kopierregeln löscht man nacheinander die dem Stoppzustand benachbarten Symbole von k_t, z. B.:

  $\ldots \#a\hat{q}|\, b|\#|\hat{q}\, b|\#$
  $\ldots \#a\, \hat{q}\, b\ \#|\hat{q}|b|\#|\hat{q}|\#$

- Danach wendet man die Abschlussregel an:

  $\ldots \#|\hat{q}\#\ \#$
  $\ldots \#\ \hat{q}\#|\#$

Umgekehrt zeigt man leicht, dass jede Lösung des MPKP einer haltenden Folge von Konfigurationsübergängen entspricht, welche mit k_0 beginnt:

- Die Konfigurationsfolge muss mit k_0 beginnen, da wir das MPKP betrachten.
- Die Kopier- und Übergangsregeln können nur so angewendet werden, dass die mit k_0 beginnende Berechnung von M entsteht.
- Die so erzeugten Wörter können nicht gleich lang werden.
- Daher muss ein Stoppzustand erreicht werden, damit Lösch- und Abschlussregeln eingesetzt werden können.

❑

Da das Halteproblem unentscheidbar ist (Satz 17.7), folgt die Unentscheidbarkeit des MPKP und damit (wegen Lemma 17.17) die Unentscheidbarkeit des PKP.

Satz 17.19

Das PKP ist unentscheidbar.

Wir verwenden dieses Resultat, um Unentscheidbarkeit von Problemen für kontextfreie und kontextsensitive Sprachen nachzuweisen. Zunächst folgt direkt aus Satz 17.19 und Lemma 17.11:

Lemma 17.20

Das Schnitt-Leerheitsproblem für kontextfreie Grammatiken ist unentscheidbar.

Beachte, dass man das Schnitt-Leerheitsproblem für kontextfreie Sprachen nicht einfach auf das Leerheitsproblem für kontextfreie Sprachen reduzieren kann, denn die kontextfreien Sprachen sind nicht unter Schnitt abgeschlossen (Korollar 10.6).

Wir wissen jedoch bereits, dass jede kontextfreie Sprache auch kontextsensitiv ist und dass die kontextsensitiven Sprachen unter Schnitt abgeschlossen sind (Satz 14.6). Daraus folgt der folgende Satz. Zur Erinnerung: monotone Grammatiken erzeugen die kontextsensitiven Sprachen.

Satz 17.21

Für monotone Grammatiken sind das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.

Beweis. Es existiert eine einfache Reduktion des Schnitt-Leerheitsproblems kontextfreier Grammatiken auf das Leerheitsproblem monotoner Grammatiken: gegeben kontextfreie Grammatiken G_1 und G_2, konstruiere monotone Grammatik G mit $L(G) = L(G_1) \cap L(G_2)$ (zum Beispiel mittels Umweg über linear beschränkte Automaten), entscheide dann ob $L(G) = \emptyset$.

Das Leerheitsproblem ist ein Spezialfall des Äquivalenzproblems, da

$$L(G) = \emptyset \text{ gdw. } L(G) = L(G_\emptyset) \quad (G_\emptyset : \text{ monotone Grammatik mit } L(G_\emptyset) = \emptyset).$$

❑

Folgendes Resultat eine direkte Konsequenz aus den bisherigen Unentscheidbarkeitsresultaten in diesem Abschnitt sowie der Korrespondenz zwischen Turingmaschinen und Typ-0-Grammatiken:

Satz 17.22

Für Typ-0-Grammatiken sind das Wortproblem, das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.

Beweis. Für das Leerheitsproblem und das Äquivalenzproblem folgt Unentscheidbarkeit aus Satz 17.21.

Die Unentscheidbarkeit des Wortproblems folgt aus dem Zusammenhang zwischen Typ-0-Grammatiken und DTMs: Nach Satz 17.6 ist das Wortproblem für DTMs unentscheidbar. Die Beweisrichtung „⇒“ von Satz 14.1 liefert eine effektive (d. h. durch eine DTM implementierbare) Umwandlung einer DTM in eine Typ-0-Grammatik und damit Reduktion zum Wortproblem für Typ-0-Grammatiken. ❑

Zur Erinnerung: im Gegensatz zum Leerheitsproblem für kontextsensitive Sprachen hatten wir gezeigt, dass das Leerheitsproblem für kontext*freie* Sprachen entscheidbar ist. Das Äquivalenzproblem ist allerdings bereits für kontextfreie Sprachen unentscheidbar. Die hier gezeigten Unentscheidbarkeitsresultate gelten natürlich auch für linear beschränkte Automaten bzw. Kellerautomaten, da man mittels einer TM Grammatiken in das entsprechende Automatenmodell übersetzen kann (und umgekehrt).

Satz 17.23

Für kontextfreie Grammatiken ist das Äquivalenzproblem unentscheidbar.

Beweis.

1. Man kann sich leicht überlegen, dass die Sprachen $L(G_P^{(\ell)})$ und $L(G_P^{(r)})$ aus dem Beweis von Lemma 17.11 durch deterministische Kellerautomaten akzeptiert werden können.
2. Die von deterministischen Kellerautomaten akzeptierten kontextfreien Sprachen sind (im Unterschied zu den kontextfreien Sprachen selbst) unter Komplement abgeschlossen.

   D. h. es gibt auch einen deterministischen Kellerautomaten und damit eine kontextfreie Grammatik für
   $$\overline{L(G_P^{(\ell)})}$$
   (siehe z.B. [Weg05], Satz 8.1.3).
3. Es sei $\overline{G}$ die kontextfreie Grammatik mit $L(\overline{G}) = \overline{L(G_P^{(\ell)})}$. Nun gilt:
   $$\begin{aligned} L(G_P^{(\ell)}) \cap L(G_P^{(r)}) = \emptyset \quad & \text{gdw.} \quad L(G_P^{(r)}) \subseteq L(\overline{G}) \\ & \text{gdw.} \quad L(G_P^{(r)}) \cup L(\overline{G}) = L(\overline{G}) \\ & \text{gdw.} \quad L(G_\cup) = L(\overline{G}), \end{aligned}$$
   wobei $G_\cup$ die kontextfreie Grammatik für $L(G_P^{(r)}) \cup L(\overline{G})$ ist.
4. Wäre also das Äquivalenzproblem für kontextfreie Grammatiken entscheidbar, so auch
   $$L(G_P^{(\ell)}) \cap L(G_P^{(r)}) \neq \emptyset$$
   und damit das PKP.

❑

An dieser Stelle haben wir die Entscheidbarkeit von Wortproblem, Leerheitsproblem und Äquivalenzproblem für die verschiedenen in dieser Vorlesung behandelten Maschinenmodelle und Grammatiktypen vollständig geklärt. Eine zusammenfassende Tabelle findet sich in Anhang **??**.

17.6. Eine weitere Anwendung von Diagonalisierung

Im Folgenden wenden wir nochmals Diagonalisierung an, um eine Lücke in unserem Wissen über formale Sprachen zu schließen. Wir zeigen, dass die Typ-1-Sprachen eine *echte* Teilmenge der Typ-0-Sprachen sind.

Satz 17.24
$\mathcal{L}_1 \subsetneq \mathcal{L}_0$.

Beweis. Es sei

- $G_0, G_1, \ldots$ eine effektive (d. h. mit TM machbare) Aufzählung aller monotonen Grammatiken mit Terminalalphabet $\Sigma = \{a, b\}$
- und $w_0, w_1, \ldots$ eine effektive Aufzählung aller Wörter über Σ.

Wir definieren nun $L \subseteq \{a, b\}^*$ als

$$L = \{w_i \mid i \geq 0 \text{ und } w_i \notin L(G_i)\} \quad \text{(Diagonalisierung!)}.$$

- L ist Turing-erkennbar und damit in $\mathcal{L}_0$. In der Tat ist L sogar entscheidbar: bei Eingabe w kann eine DTM
  - durch Aufzählen der $w_0, w_1, \ldots$ den Index i mit $w = w_i$ bestimmen
  - durch Aufzählen der $G_0, G_1, \ldots$ dann auch die Grammatik G_i konstruieren
  - dann das Wortproblem für $w_i \in L(G_i)$ für Typ 1-Sprachen entscheiden (siehe Satz 14.7)
- L ist nicht kontextsensitiv. Anderenfalls gäbe es einen Index k mit $L = L(G_k)$. Nun ist aber

$$\begin{array}{lll} w_k \in L(G_k) & \text{gdw. } w_k \in L & (\text{denn } L(G_k) = L) \\ & \text{gdw. } w_k \notin L(G_k) & (\text{nach Def. } L) \end{array}$$

  Widerspruch.

❑

18. Semi-Entscheidbarkeit und rekursive Aufzählbarkeit

Wir betrachten die Semi-Entscheidbarkeit von Entscheidungsproblemen, die eine natürliche Abschwächung von Entscheidbarkeit darstellt, sowie den eng verwandten Begriff der rekursiven Aufzählbarkeit. Dabei geht es um die Frage, ob man, wenn ein Problem L schon nicht entscheidbar ist, wenigstens einen Algorithmus finden kann, der

- auf *positiven* Eingaben stets terminiert (Semi-Entscheidbarkeit) bzw.
- alle (im Allgemeinen unendlich viele) positiven Eingaben systematisch aufzählt und ausgibt (rekursive Aufzählbarkeit).

Wir werden sehen, dass dies für manche unentscheidbare Probleme möglich ist, für andere aber nicht. Dadurch ergeben sich verschiedene Grade der Unlösbarkeit. Wir werden auch einen Zusammenhang zur Erkennbarkeit von Sprachen mittels Turingmaschinen herstellen und einige verbliebene Fragen bezüglich Typ-0- und Typ-1-Sprachen klären.

Definition 18.1 (Semi-entscheidbar, rekursiv aufzählbar)
Eine Sprache $L \subseteq \Sigma^*$ heißt

1) *semi-entscheidbar*, falls es eine DTM gibt, die bei Eingabe $w \in \Sigma^*$
   - terminiert, falls $w \in L$ ist, und
   - nicht terminiert sonst.

   Eine solche DTM nennen wir ein *Semi-Entscheidungsverfahren* für L.

2) *rekursiv aufzählbar*, falls L von einer Aufzähl-Turingmaschine aufgezählt wird. Diese ist wie folgt definiert:

   Eine *Aufzähl-Turingmaschine* M ist eine DTM, die einen speziellen Ausgabezustand q_{Ausgabe} hat.

   Eine *Ausgabekonfiguration* ist von der Form

   $$uq_{\text{Ausgabe}}\, wav \text{ mit } u, v \in \Gamma^*, w \in \Sigma^* \text{ und } a \in \Gamma \setminus \Sigma.$$

   Diese Konfiguration *erzeugt die Ausgabe* w.

   Die durch M *aufgezählte Sprache* ist

   $$L = \{w \in \Sigma^* \mid w \text{ ist Ausgabe einer Ausgabekonfiguration, die von } M \text{ ausgehend von Startkonfiguration } q_0\not\emptyset \text{ erreicht wird}\}.$$

   Wir nennen M dann ein *Aufzählverfahren* für L.

Im Gegensatz zu einem Entscheidungsverfahren terminiert ein Semi-Entscheidungsverfahren also nur auf positiven Eingaben. In der Tat ist Terminierung ja sogar die verwendete Akzeptanzbedingung. Man überlegt sich leicht, dass eine Sprache L semi-entscheidbar ist gdw. L der Definitionsbereich einer berechenbaren einstelligen partiellen

Funktion ist. Man beachte auch, dass es bei Aufzähl-TMs erlaubt ist, dass dasselbe Wort mehrfach aufgezählt wird.

Der Begriff „rekursiv“ in „rekursiv aufzählbar“ steht hier synonym für „durch eine TM zu bewerkstelligen“. Das kommt daher, dass man die mit TM berechenbaren Funktionen auch durch bestimmte Rekursionsvorschriften definieren kann. Die Klasse der entscheidbaren Sprachen wird dann oft auch REC genannt (zur Erinnerung: Entscheidbarkeit ist nur ein Spezialfall von Berechenbarkeit). Die Klasse der rekursiv aufzählbaren Sprachen wird entsprechend RE genannt.

Beispiel. Wir überlegen uns kurz, dass jede Typ-0-Sprache sowohl semi-entscheidbar als auch rekursiv aufzählbar ist. Sei G eine Typ-0-Grammatik. Eine DTM M, die ein Semi-Entscheidungsverfahren für $L(G)$ ist, arbeitet bei Eingabe von w wie folgt:

- Verwalte eine Menge W von bereits abgeleiteten Wörtern (die nicht unbedingt Terminalwörter sein müssen).
- Starte mit $W = \{S\}$, wobei S das Startsymbol von G ist.
- Führe dann immer wieder folgenden Schritt aus:

  Erzeuge alle Wörter, die sich aus Wörtern in W in einem Schritt mit einer Regel aus G herleiten lassen, und füge alle diese Wörter zu W hinzu.

  Terminiere, sobald das Eingabewort w in W gefunden wird.

Die beschriebene DTM lässt sich leicht in ein Aufzählverfahren für $L(G)$ umwandeln. Jedes neue Wort, das aus den Wörtern in W in einem Schritt hergeleitet werden kann, wird (per Ausgabezustand) ausgegeben, wenn es ein Terminalwort ist; da es kein Eingabewort w mehr gibt, entfällt der Test, ob dieses sich in W befindet.

Es ist kein Zufall, dass wir dasselbe Beispiel für Semi-Entscheidbarkeit und rekursive Aufzählbarkeit verwenden. Das folgende Resultat zeigt den engen Zusammenhang beider Begriffe und stellt zudem eine Verbindung zwischen Entscheidbarkeit und Semi-Entscheidbarkeit her.

Satz 18.2

Für alle $L \subseteq \Sigma^*$ gilt:

1) L ist rekursiv aufzählbar gdw. L ist semi-entscheidbar.

2) Ist L entscheidbar, so auch semi-entscheidbar.

3) L ist entscheidbar gdw. L und $\overline{L}$ semi-entscheidbar sind.

Beweis.

1) **„⇒“:** Es sei L rekursiv aufzählbar und M eine Aufzähl-DTM für L. Die Maschine M', die ein Semi-Entscheidungsverfahren für L ist, arbeitet wie folgt:

   - Sie speichert die Eingabe w auf einem zusätzlichen Band.

- Sie beginnt mit der Aufzählung von L.
- Bei jeder Ausgabekonfiguration überprüft sie, ob die entsprechende Ausgabe mit w übereinstimmt. Wenn ja, so terminiert M'. Sonst sucht sie die nächste Ausgabekonfiguration von M.
- Terminiert M, ohne dass w ausgegeben wurde, so geht M' in eine Endlosschleife.

M' terminiert daher genau dann nicht, wenn w nicht in der Aufzählung vorkommt.

„⇐“: Es sei M ein Semi-Entscheidungsverfahren für L und

$$\Sigma^* = \{w_1, w_2, w_3, \ldots\}$$

Die Maschine M' arbeitet wie folgt:

(1) Führe einen Schritt der Berechnung von M auf Eingabe w_1 aus.

(2) Führe zwei Schritte der Berechnung von M auf Eingaben w_1 und w_2 aus.
⋮

(n) Führe n Schritte der Berechnung von M auf Eingaben $w_1, \ldots, w_n$ aus.
⋮

Terminiert M für eine dieser Eingaben, so gebe diese Eingabe aus und mache weiter.

Beachte: Das hier angewendete Verfahren der Verzahnung zweier Aufzählungen (der Wörter w_i und der Anzahl der Berechnungsschritte) heißt auch *Dovetailing* und ist eine Art Diagonalisierung. Es ist notwendig, weil man nicht z. B. in Schritt (1) einfach M auf der Eingabe w_1 zu Ende laufen lassen kann, denn M muss ja auf w_1 nicht unbedingt terminieren.

2) Eine DTM M, die L entscheidet, wird wie folgt modifiziert:
   - hält M in nicht-akzeptierender Stoppkonfiguration, so gehe in Endlosschleife.
   - keine Änderung, wenn M in akzeptierender Stoppkonfiguration stoppt.

3) **„⇒“:** Ergibt sich aus 2) und Satz 17.4.

   „⇐“: Sind L und $\overline{L}$ semi-entscheidbar, so mit 1) auch rekursiv aufzählbar.

   Für Eingabe w lässt man die Aufzähl-DTMs M und M' für L und $\overline{L}$ parallel laufen (d. h. jeweils abwechselnd ein Schritt von M auf einem Band gefolgt von einem Schritt von M' auf dem anderen).

   Die Eingabe w kommt in einer der beiden Aufzählungen vor:

$$w \in \Sigma^* = L \cup \overline{L}$$

   Kommt w bei M vor, so stoppe in akzeptierendem Zustand; sonst stoppe in nicht-akzeptierendem Zustand. ❑

Wir analysieren nun beispielhaft die Semi-Entscheidbarkeit einiger relevanter Probleme. Ein wichtiges Hilfsmittel ist dabei eine *universelle Turingmaschine* U, die wie ein Interpreter für Programmiersprachen funktioniert und damit *jede* Turingmaschine simulieren kann. U erhält als Eingabe

- die Kodierung der zu simulierenden Turingmaschine M als Wort;
- das Eingabewort w, auf dem M simuliert werden soll.

Genauer gesagt erhält U als Eingabe das Wort $\mathsf{code}(M)\#\#\#w$, wobei $\mathsf{code}(M)$ die Kodierung von M aus Kapitel 17 ist. Man kann leicht die kodierte DTM von ihrer Eingabe w trennen: letztere findet sich nach dem dritten ###-Block (denn die ersten zwei solchen Blöcke befinden sich innerhalb von $\mathsf{code}(M)$). Die universelle Turingmaschine akzeptiert die Eingabe $\mathsf{code}(M)\#\#\#w$ gdw. w von M akzeptiert wird.

Wir betrachten hier der Einfachheit halber Turingmaschinen über dem Eingabealphabet $\Sigma = \{a, b\}$. Diese Konvention erspart es uns, die Wörter w in der Eingabe für U zu kodieren.

Satz 18.3

Es gibt eine universelle DTM U über $\Sigma \cup \{0, 1, \ell, n, r, \#\}$, d. h. eine DTM mit der folgenden Eigenschaft. Für alle DTM M und alle $w \in \Sigma^*$ gilt:

$$U \text{ akzeptiert } \mathsf{code}(M)\#\#\#w \quad \text{gdw.} \quad M \text{ akzeptiert } w$$

Beweis. U führt bei Eingabe $\mathsf{code}(M)\#\#\#w$ die M-Berechnung

$$q_0 w = k_0 \vdash_M k_1 \vdash_M k_2 \vdash_M \dots$$

in kodierter Form aus, d. h. U erzeugt sukzessive Bandbeschriftungen

$$\mathsf{code}(M)\#\#\#\mathsf{code}(k_0), \quad \mathsf{code}(M)\#\#\#\mathsf{code}(k_1), \quad \mathsf{code}(M)\#\#\#\mathsf{code}(k_2), \quad \dots$$

Wir brauchen also noch eine *Kodierung von Konfigurationen als Wörter*:

$$\mathsf{code}(a_{i_1} \dots a_{i_m} q_j a_{i_{m+1}} \dots a_{i_p}) = \mathsf{bin}(i_1)\# \dots \#\mathsf{bin}(i_m)\#\mathsf{bin}(j)\#\#\mathsf{bin}(i_{m+1})\# \dots \#\mathsf{bin}(i_p)$$

Beachte, dass die Kopfposition durch ## gekennzeichnet ist und dass der aktuelle Zustand unmittelbar links vom ##-Marker repräsentiert ist.

Arbeitsweise von U bei Eingabe $\mathsf{code}(M)\#\#\#w$:

- Wandle w in die Kodierung der Anfangskonfiguration, erzeuge also

$$\mathsf{code}(M)\#\#\#\mathsf{code}(q_0 w).$$

- Simuliere die Schritte von M, ausgehend vom jeweiligen Konfigurationskode

$$\dots\#\mathsf{bin}(j)\#\#\mathsf{bin}(i)\#\dots \text{ (Zustand } q_j\text{, gelesenes Symbol } a_i\text{)}.$$

- Suche eine Transitionskodierung $\mathsf{code}(t)$ in $\mathsf{code}(M)$, die mit $\mathsf{bin}(j)\#\mathsf{bin}(i)$ beginnt.
- Falls es so ein $\mathsf{code}(t)$ gibt, ändere die Konfigurationskodierung entsprechend t.
- Sonst geht U in Stoppkonfiguration. Diese ist akzeptierend gdw. $\mathsf{bin}(j)$ in der Kodierung der Menge der akzeptierenden Zustände vorkommt.

Es ist nicht schwer, die Schritte im Detail auszuarbeiten. Man beachte allerdings, dass jeder dieser Schritte viele Einzelschritte von U erfordert. Um z. B. ein $\mathsf{code}(t)$ zu finden, das mit $\mathsf{bin}(j)\#\mathsf{bin}(i)$ beginnt, muss man die aktuelle Konfiguration sukzessive mit *jeder* Transitionskodierung vergleichen. Jeder einzelne solche Vergleich erfordert wiederholtes Hin- und Herlaufen zwischen der aktuellen Konfiguration und dem gerade betrachteten $\mathsf{code}(t)$ (Vergleich Symbol für Symbol). ❑

Wir werden nun die Semi-Entscheidbarkeit einiger Probleme und ihrer Komplemente betrachten, beginnend mit dem Wortproblem.

Satz 18.4

Das Wortproblem für DTMs ist semi-entscheidbar; sein Komplement ist nicht semi-entscheidbar.

Beweis.

1) Das Wortproblem ist semi-entscheidbar: Bei Eingabe von M und w starte die universelle Turingmaschine U auf $\mathsf{code}(M)\#\#\#w$. Wenn U akzeptiert, dann stoppe. Wenn U anhält und verwirft, gehe in Endlosschleife.
2) Das Komplement des Wortproblems ist nicht semi-entscheidbar: Folgt aus 1), Punkt 3 von Satz 18.2 und der Unentscheidbarkeit des Wortproblems. ❑

Satz 18.4 erlaubt uns, die letzte verbliebene Abschlusseigenschaft von Typ 0-Sprachen zu klären: den Abschluss unter Komplement. Zunächst beobachten wir, dass die Semi-Entscheidbarkeit von Sprachen und deren Erkennbarkeit durch Turingmaschinen äquivalent zueinander sind.

Bemerkung 18.5

Eine Sprache L ist semi-entscheidbar gdw. sie Turing-erkennbar ist:

„⇐“ Wir können mit Satz 13.6 o. B. d. A. annehmen, dass es eine DTM M gibt, die L erkennt. Bei $w \notin L$ kann M in einem nicht-akzeptierenden Zustand halten, wohingegen ein Semi-Entscheidungsverfahren nicht terminieren darf.

Modifikation: wenn M bei $w \notin L$ in einem nicht-akzeptierenden Zustand anhält, dann gehe in Endlosschleife.

„⇒“ Ein Semi-Entscheidungsverfahren M für L kann bei $w \in L$ in beliebigem Zustand anhalten, wohingegen es bei Turing-Erkennbarkeit ein akzeptierender Zustand sein muss.

Modifikation: wenn M bei $w \in L$ in einem nicht-akzeptierenden Zustand anhält, dann wechsle in einem Schritt in einen akzeptierenden Zustand (der keine Folgezustände hat).

Aus Satz 18.4 und Bemerkung 18.5 folgt nun direkt:

Satz 18.6

$\mathcal{L}_0$ ist nicht unter Komplement abgeschlossen.

An dieser Stelle haben wie die Abschlusseigenschaften aller in dieser Vorlesung behandelter Sprachklassen vollständig geklärt. Eine Übersicht findet sich in Anhang **??**.

Wir untersuchen nun das Leerheitsproblem in Bezug auf Semi-Entscheidbarkeit. Es stellt sich heraus, dass es sich genau umgekehrt zum Wortproblem verhält: das Problem selbst ist nicht semi-entscheidbar, sein Komplement aber schon.

Satz 18.7

Das Leerheitsproblem für DTMs ist nicht semi-entscheidbar; sein Komplement ist semi-entscheidbar.

Beweis. Wie im Beweis von Satz 18.4 genügt es, zu zeigen, dass das Komplement des Leerheitsproblems semi-entscheidbar ist. Mit Satz 18.2 und der Unentscheidbarkeit des Leerheitsproblems folgt dann, dass sein Komplement nicht semi-entscheidbar ist.

Das Semi-Entscheidungsverfahren für das Komplement des Leerheitsproblems verwendet die universelle Turingmaschine U. Als zusätzliche Ingredienz ist es nicht schwer, eine Turingmaschine zu konstruieren, die bei Eingabe von $\mathsf{code}(M)$ eine Aufzählung $w_1, w_2, \ldots$ aller Wörter über dem Eingabealphabet von M ausgibt.

Das Semi-Entscheidungsverfahren für das Komplement des Leerheitsproblems verwendet wieder Dovetailing (siehe Beweis von Satz 18.2) und geht bei Eingabe M wie folgt vor:

(1) Führe einen Schritt der Berechnung von M auf Eingabe w_1 aus.

(2) Führe zwei Schritte der Berechnung von M auf Eingaben w_1 und w_2 aus.

$\vdots$

(n) Führe n Schritte der Berechnung von M auf Eingaben $w_1, \ldots, w_n$ aus.

$\vdots$

Akzeptiert M eine der Eingaben, so terminiere (denn dann ist $L(M) \neq \emptyset$). Sonst fahre fort. ❑

Das Wortproblem und das Leerheitsproblem für DTMs verhalten sich also unterschiedlich bezüglich Semi-Entscheidbarkeit. Das Halteproblem wiederum verhält sich genauso wie das Wortproblem, was man bequem zeigen kann, wenn man folgende Beobachtung nutzt, deren Beweis vollständig analog zu Lemma 17.13 ist.

Lemma 18.8

1) $L_1 \leq L_2$ *und* L_2 *semi-entscheidbar* $\Rightarrow$ L_1 *semi-entscheidbar.*

2) $L_1 \leq L_2$ *und* L_1 *nicht semi-entscheidbar* $\Rightarrow$ L_2 *nicht semi-entscheidbar.*

Nun kann man ganz leicht das Halteproblem behandeln:

Korollar 18.9

Das Halteproblem für DTMs ist semi-entscheidbar; sein Komplement ist nicht semi-entscheidbar.

Beweis.

1) Nach Satz 18.7 ist das Komplement des Leerheitsproblems semi-entscheidbar. Der Beweis des Satzes von Rice 17.14 liefert eine Reduktion vom Halteproblem auf das Komplement des Leerheitsproblems. Mit Lemma 18.8 ist damit auch das Halteproblem semi-entscheidbar.

2) Nach Satz 18.4 ist das Komplement des Wortproblems nicht semi-entscheidbar. Der Beweis von Satz 17.7 liefert eine Reduktion vom Komplement des Wortproblems auf das Komplement des Halteproblems. ❑

Das Äquivalenzproblem zeigt wieder ein anderes Verhalten als die drei vorigen Probleme: weder das Problem selbst noch sein Komplement sind semi-entscheidbar. In diesem Fall kann man natürlich nicht mehr wie in den Beweisen der Sätze 18.4 und 18.7 argumentieren. Stattdessen nutzt man wieder Lemma 18.8 und zwei Reduktionen vom Halteproblem.

Satz 18.10

Das Äquivalenzproblem für DTMs ist nicht semi-entscheidbar, sein Komplement ist ebenfalls nicht semi-entscheidbar.

Beweis: siehe Übung.

Vergleicht man Satz 18.10 mit Satz 18.4, Satz 18.7 oder Korollar 18.9, so zeigt sich: obwohl alle vier Probleme unentscheidbar sind, ist das Äquivalenzproblem echt schwerer als das Wortproblem bzw. das Leerheitsproblem bzw. das Halteproblem. In der Tat kann man im Beweis von Satz 18.10 das Halteproblem auf das Äquivalenzproblem reduzieren (siehe Übung), aber wegen der übrigen Resultate (Semi-Entscheidbarkeit des Halteproblems, Nicht-Semi-Entscheidbarkeit des Äquivalenzproblems und Lemma 18.8, Teil 1) ist die umgekehrte Reduktion nicht möglich. Das ist es, was hier mit „echt schwerer“ gemeint ist. Es gibt sogar unendliche Folgen von (unentscheidbaren) Problemen $L_1, L_2, \ldots$ so dass

$$L_1 \leq L_2 \leq L_3 \leq \cdots \qquad \text{und} \qquad \cdots \not\leq L_3 \not\leq L_2 \not\leq L_1.$$

Man spricht hier auch vom *Grad der Unlösbarkeit* oder vom *Turing-Grad.*

Wir werfen im Folgenden noch einen kurzen Blick auf dieses Thema. Dazu führen wir das interessante Konzept der *Orakel-Turingmaschine* ein, das sowohl in der Theorie der Berechenbarkeit als auch in der Komplexitätstheorie von Nutzen ist.

Orakel-Turingmaschinen

Definition 18.11 (Orakel-Turingmaschine)

Eine *Orakel-Turingmaschine (OTM)* M ist eine DTM, die mit einem *Orakel* $O \subseteq \Sigma^*$ ausgestattet ist. M hat

- ein zusätzliches Orakelband sowie
- drei spezielle Zustände $q_?$, q_+, q_- .

Der Folgezustand von $q_?$ ist q_+ , wenn das momentane Wort auf dem Orakelband in der Orakelsprache O enthalten ist, und q_- sonst. Kopfposition und Bandinhalte bleiben dabei unverändert. Ansonsten verhalten sich q_+ und q_- wie normale Zustände.

Man beachte, dass die Orakelsprache O auch eine unentscheidbare Sprache sein kann. Das Orakel liefert trotzdem in einem Schritt die gewünschte Antwort. Eine OTM gleicht also einem Gedankenexperiment: was wäre möglich, wenn z. B. das Wortproblem für DTMs entscheidbar wäre? Orakel-Turingmaschinen liefern auch einen natürlichen alternativen Reduktionsbegriff.

Definition 18.12 (Turing-reduzierbar)

$L_1 \subseteq \Sigma^*$ ist *Turing-reduzierbar* auf $L_2 \subseteq \Sigma^*$, wenn es eine OTM M mit Orakel L_2 gibt, die L_1 in folgendem Sinne *löst*:

$$L(M) = L_1 \text{ und } M \text{ stoppt auf jeder Eingabe.}$$

Wir schreiben dann $L_1 \leq_T L_2$.

Man beachte den Unterschied zwischen Reduzierbarkeit im Sinne von Definition 17.8 und Turing-Reduzierbarkeit: Turing-Reduzierbarkeit von L_1 auf L_2 bedeutet intuitiv, dass es ein Entscheidungsverfahren für L_1 gibt, wenn man ein Entscheidungsverfahren für L_2 hat und dieses beliebig oft und in beliebiger Weise als „Subroutine“ verwenden darf. Bei Reduzierbarkeit gemäß Definition 17.8, die wir zwecks Unterscheidung von nun an *many-one-Reduzierbarkeit* nennen, darf die Subroutine nur einmal am Schluss aufgerufen werden, und ihr Ergebnis muss direkt zurückgegeben werden (und darf z. B. nicht komplementiert werden). Wir sprechen in Definition 18.12 von *Lösbarkeit* statt von *Entscheidbarkeit*, weil Lösbarkeit mit OTMs natürlich nicht mit Entscheidbarkeit (mit DTMs) übereinstimmt, denn die Orakelsprache kann ja unentscheidbar sein.

Es stellt sich die Frage, wie beide Reduktionsbegriffe zusammenhängen. Many-one-Reduzierbarkeit ist der *feinere* Reduktionsbegriff, in folgendem Sinne.

Lemma 18.13

1) Für alle L_1, L_2 gilt: wenn $L_1 \leq L_2$, dann $L_1 \leq_T L_2$

2) Die Umkehrung von Punkt 1 gilt nicht.

Beweis. Für Punkt 1 nehmen wir an, dass $L_1 \leq L_2$. Dann gibt es nach Definition von many-one-Reduzierbarkeit eine DTM M, die zu jeder Eingabe w eine Ausgabe $f(w)$ liefert, so dass $w \in L_1$ gdw. $f(w) \in L_2$. Modifiziere M so, dass $f(w)$ stattdessen auf das Orakelband geschrieben und dann ein Orakel L_2 aufgerufen wird. Wir erhalten eine OTM mit Orakel L_2, die offensichtlich L_1 löst, also $L_1 \leq_T L_2$.

Für Punkt 2 überlegen wir uns zunächst, dass *jede* Sprache L auf ihr Komplement $\overline{L}$ Turing-reduzierbar ist: gegeben Eingabe w für L, starte Orakel $\overline{L}$ auf w, komplementiere dann das Ergebnis. Wir wissen aber, dass beispielsweise das Leerheitsproblem nicht many-one-reduzierbar auf sein Komplement ist, denn sonst wäre es mit Lemma 18.8 semi-entscheidbar. ❑

In der Theorie der Berechenbarkeit wurde der Raum unentscheidbarer Probleme sowohl bezüglich Turing-Reduktionen als auch bezüglich many-one-Reduktionen intensiv studiert. Hier beschränken wir uns darauf, eine unendliche Familie von Sprachen zu identifizieren, die immer schwerer werden (bzgl. beider Arten von Reduktionen):

- W_0 das Wortproblem für DTMs (als formale Sprache)
- W_1 das Wortproblem für DTMs mit Orakel W_0
- W_2 das Wortproblem für DTMs mit Orakel W_1
- usw.

Diese Probleme mögen recht esoterisch erscheinen. Sie sind jedoch vollkommen wohldefiniert und auch nicht „irrealer“ als jedes andere wohldefinierte Problem.

Satz 18.14

Für alle $i \geq 0$ gilt: $W_i \leq W_{i+1}$ *aber* $W_{i+1} \not\leq_T W_i$

Beweis. Wir zeigen zunächst $W_i \leq W_{i+1}$. Um W_i zu lösen, können wir bei Eingabe von (M, w)

1. eine OTM M' mit Orakel W_i konstruieren, die ihre Eingabe auf das Orakelband schreibt, das Orakel aufruft und das Ergebnis einfach zurückgibt;
2. dann M' auf Eingabe (M, w) starten.

Das liefert offensichtlich eine many-one-Reduktion von W_i auf W_{i+1}: gegeben Eingabe (M, w) für W_i, konstruiere Eingabe $f(M, w) = (M', (M, w))$ für W_{i+1}.

Für den Beweis von $W_{i+1} \not\leq_T W_i$ beschränken wir uns auf eine Skizze. Sei W^s_{i+1} das *spezielle Wortproblem für OTMs mit Orakel* W_i: gegeben eine (Kodierung von) OTM M mit Orakel W_i, entscheide, ob M die eigene Kodierung als Eingabe akzeptiert.

Es genügt zu zeigen: keine OTM mit Orakel W_i löst W^s_{i+1}. Denn wenn $W_{i+1} \not\leq_T W_i$ gelten würde, dann gäbe es per Definition von Turing-Reduzierbarkeit eine OTM M mit Orakel W_i, die W_{i+1} löst; offensichtlich ließe sich M leicht auf W^s_{i+1} anpassen.

Der Beweis, dass W^s_{i+1} von keiner OTM mit Orakel W_i gelöst wird, ist *exakt derselbe* wie für die Unentscheidbarkeit der ursprünglichen Wortproblems W_0. Wir geben hier nur eine Übersicht und fordern die Lesenden auf, die Details nochmal selbständig nachzuvollziehen:

- Man zeigt zunächst: das *Komplement* von W^s_{i+1} wird nicht von einer OTM mit Orakel W_i gelöst (die Komplementsprache implementiert Diagonalisierung).
- Dazu verwendet man denselben Widerspruchsbeweis wie vorher (wenn es eine lösende OTM gäbe, so führt ihr Start auf der eigenen Kodierung zu einem Widerspruch).
- Also gibt es auch keine OTM mit Orakel W_i, die W^s_{i+1} löst, denn daraus könnte man wie gehabt durch Vertauschen der akzeptierenden mit den nicht-akzeptierenden Zuständen eine OTM konstruieren, die das Komplement von W^s_{i+1} löst. ❑

Der Beweis von Satz 18.14 zeigt, dass die Beweismethode Diagonalisierung auf Orakel *relativiert*, also auch in Gegenwart von Orakeln funktioniert. Für den obigen Beweis ist das von Vorteil. Es zeigt aber auch die Grenzen der Diagonalisierung auf. Man kann zum Beispiel beweisen, dass das wichtige P-vs.-NP-Problem der Komplexitätstheorie (siehe Abschnitt 19) *nicht* mittels Diagonalisierung gelöst werden kann. Grund dafür ist genau die beschriebene Relativierung auf Orakel.

IV. Komplexität

Einführung

In der Praxis genügt es nicht zu wissen, dass eine Funktion berechenbar ist. Man interessiert sich auch dafür, wie groß der *Aufwand* zur Berechnung ist. Aufwand bezieht sich hierbei auf den Ressourcenverbrauch, wobei folgende Ressourcen die wichtigste Rolle spielen:

Rechenzeit
(bei TM: Anzahl der Übergänge bis zum Halten)

Speicherplatz
(bei TM: Anzahl der benutzten Felder)

Beides soll abgeschätzt werden als *Funktion in der Größe der Eingabe*. Dem liegt die Idee zugrunde, dass mit größeren Eingaben üblicherweise auch der Ressourcenbedarf zum Verarbeiten der Eingabe wächst.

Es ist wichtig, sauber zwischen der Komplexität von Algorithmen und der von Problemen zu unterscheiden.

Komplexität eines konkreten Algorithmus/Programms. Wieviel Ressourcen verbaucht dieser Algorithmus? Derartige Fragestellungen gehören in das Gebiet der Algorithmentheorie und werden hier nicht primär betrachtet.

Komplexität eines Entscheidungsproblems: Wieviel Aufwand benötigt der „beste“ Algorithmus, der ein gegebenes Problem löst? Dies ist die Fragestellung, mit der sich die Komplexitätstheorie beschäftigt. Wir setzen dabei wieder „Algorithmus“ mit Turingmaschine gleich.

Die Komplexitätstheorie liefert äußerst wichtige Anhaltspunkte dafür, welche Probleme effizient lösbar sind und welche nicht. Wir betrachten die klassische Komplexitätstheorie, die sich immer am „schlimmsten Fall (worst case)“ orientiert, also an Eingaben mit maximalem Ressourcenbedarf. Die Worst-Case-Annahme ist für viele praktische Anwendungen relevant, für andere aber deutlich zu pessimistisch, da dort der schlimmste Fall selten oder niemals vorkommt. Man kann auch den „durchschnittlichen Fall (average case)“ analysieren. Dies ist jedoch technisch anspruchsvoller und erfordert ein genaues Verständnis davon, was ein durchschnittlicher Fall eigentlich ist. Auch in praktischen Anwendungen ist es oft nicht einfach, eine formale Beschreibung davon anzugeben.

19. Einige Komplexitätsklassen

Eine Komplexitätsklasse ist eine Klasse von Entscheidungsproblemen, die mit einer bestimmten „Menge“ einer Ressource gelöst werden können. Wir führen zunächst ein allgemeines Schema zur Definition von *Komplexitätsklassen* ein und fixieren dann einige fundamentale *Zeit-* und *Platzkomplexitätsklassen*. Im Gegensatz zur Berechenbarkeit ist es in der Komplexitätstheorie sehr wichtig, zwischen deterministischen und nichtdeterministischen Turingmaschinen zu unterscheiden.

Zunächst führen wir formal ein, was es heißt, dass der Aufwand durch eine Funktion der Größe der Eingabe *beschränkt* ist.

Definition 19.1 ($f(n)$-zeitbeschränkt, $f(n)$-platzbeschränkt)
Es sei $f : \mathbb{N} \to \mathbb{N}$ eine Funktion und M eine DTM über Σ.

1) M heißt $f(n)$-*zeitbeschränkt*, falls für alle $w \in \Sigma^*$ die Maschine M bei Eingabe w nach $\leq f(|w|)$ Schritten anhält.
2) M heißt $f(n)$-*platzbeschränkt*, falls für alle $w \in \Sigma^*$ die Maschine M bei Eingabe w $\leq f(|w|)$ viele Felder des Bandes benutzt.

Auf *NTM* kann man die Definition dadurch übertragen, dass die Aufwandsbeschränkung für *alle* bei der gegebenen Eingabe *möglichen Berechnungen* zutreffen muss. Beachte, dass eine $f(n)$-zeitbeschränkte TM auf jeder Eingabe terminiert; für eine $f(n)$-platzbeschränkte TM muss das nicht unbedingt der Fall sein.

Definition 19.2 (Komplexitätsklassen)

$$\mathsf{DTIME}(f(n)) := \{L \mid \text{es gibt eine } f(n)\text{-}\textbf{zeit}\text{beschränkte } \textbf{D}\text{TM } M \text{ mit } L(M) = L\}$$
$$\mathsf{NTIME}(f(n)) := \{L \mid \text{es gibt eine } f(n)\text{-}\textbf{zeit}\text{beschränkte } \textbf{N}\text{TM } M \text{ mit } L(M) = L\}$$
$$\mathsf{DSPACE}(f(n)) := \{L \mid \text{es gibt eine } f(n)\text{-}\textbf{platz}\text{beschränkte } \textbf{D}\text{TM } M \text{ mit } L(M) = L\}$$
$$\mathsf{NSPACE}(f(n)) := \{L \mid \text{es gibt eine } f(n)\text{-}\textbf{platz}\text{beschränkte } \textbf{N}\text{TM } M \text{ mit } L(M) = L\}$$

Wegen der Bemerkung vor Definition 19.2 enthalten alle Komplexitätsklassen der Form $\mathsf{DTIME}(f(n))$ und $\mathsf{NTIME}(f(n))$ nur entscheidbare Probleme. Man kann zeigen, dass das auch für alle Klassen der Form $\mathsf{DSPACE}(f(n))$ und $\mathsf{NSPACE}(f(n))$ gilt.

Im Folgenden beobachten wir einige elementare Zusammenhänge zwischen Komplexitätsklassen.

Satz 19.3

1) $\mathsf{DTIME}(f(n)) \subseteq \mathsf{DSPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$
2) $\mathsf{DTIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$

3) $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(f(n))})$ *wenn* $f(n)$ *von einer* $f(n)$*-zeitbeschränkten DTM berechnet werden kann.*[1]

In Punkt 3) von Satz 19.3 steht $\mathsf{DTIME}(2^{\mathcal{O}(f(n))})$ als abkürzende Schreibweise für $\bigcup_{g(n)\in\mathcal{O}(f(n))} \mathsf{DTIME}(2^{g(n)})$. Die Einschränkung, dass $f(n)$ von einer $f(n)$-zeitbeschränkten DTM berechnet werden können muss, schließt extreme Fälle aus, zum Beispiel Funktionen $f(n)$, die nicht berechenbar sind. Alle für uns interessanten Funktionen wie Polynome und Exponentialfunktionen erfüllen diese Eigenschaft.

Beweis. Teile 1) und 2) folgen unmittelbar daraus, dass man in k Schritten höchstens k Felder benutzen kann und jede DTM auch eine NTM ist.

Im Folgenden skizzieren wir den Beweis von Teil 3). Sei $L \in \mathsf{NSPACE}(f(n))$. Dann gibt es eine $f(n)$-platzbeschränkte NTM M mit $L(M) = L$. Mittels der Konstruktion aus dem Beweis von Satz 13.6 könnten wir M in eine DTM M' wandeln, so dass $L(M) = L(M')$. Allerdings terminiert M' nicht auf jeder Eingabe und ist daher nicht $2^{\mathcal{O}(f(n))}$-zeitbeschränkt. Wir brauchen also eine bessere Konstruktion.

Wir stellen dazu die Berechnungen von M auf Eingaben w der Länge n als gerichteten Graphen dar, den so genannten *Konfigurationsgraphen* $G_{M,n} = (V_{M,n}, E_{M,n})$, wobei

- $V_{M,n}$ die Menge der Konfigurationen von M mit Länge höchstens $f(n)$ ist und
- $E_{M,n}$ diejenigen Kanten (k, k') enthält, so dass k' von k in einem Berechnungsschritt erreicht werden kann.

Es ist nun offensichtlich, dass w von M akzeptiert wird gdw. in $G_{M,n}$ vom Knoten q_0w aus eine akzeptierende Stoppkonfiguration erreichbar ist (wenn also ein Pfad von M zu einer solchen Konfiguration existiert).

Eine terminierende DTM M' mit $L(M') = L(M)$ kann also wie folgt vorgehen:

- Bei Eingabe w der Länge n konstruiere den Konfigurationsgraphen $G_{M,n}$.[2]
- Überprüfe, ob von q_0w aus eine akzeptierende Stoppkonfiguration erreichbar ist.

Wir analysieren nun den Zeitbedarf von M'. Zunächst bestimmen wir die Größe von $G_{M,n}$: die Anzahl der Konfigurationen von M der Länge $\leq f(n)$ beträgt

$$\mathsf{akonf}_M(n) := |Q| \cdot f(n) \cdot |\Gamma|^{f(n)},$$

wobei $|Q|$ die Anzahl der möglichen Zustände ist, $f(n)$ die Anzahl der möglichen Kopfpositionen und $|\Gamma|^{f(n)}$ die Anzahl der möglichen Beschriftungen von Bändern der Länge $f(n)$.

Man sieht nun leicht, dass $|V_{M,n}| = \mathsf{akonf}_M(n) \in 2^{\mathcal{O}(f(n))}$ (beachte, dass $|Q|$ und $|\Gamma|$ Konstanten sind, da sie nicht von der Eingabe abhängen) und damit auch $|E_{M,n}| \in 2^{\mathcal{O}(f(n))}$.

[1] Wobei sowohl die Eingabe als auch die Ausgabe unär kodiert sind.

[2] Hierzu muss man zunächst $f(n)$ berechnen; da $f(n)$ von einer $f(n)$-zeitbeschränkten DTM berechnet werden kann, ist dies in der zur Verfügung stehenden Zeit möglich.

Zudem ist es für eine DTM einfach, den Graph $G_{M,n}$ in Zeit $2^{\mathcal{O}(f(n))}$ zu konstruieren. Es müssen danach noch $\leq 2^{\mathcal{O}(f(n))}$ Erreichbarkeitstests gemacht werden (einer für jede akzeptierende Stoppkonfiguration in $G_{M,n}$), von denen jeder lineare Zeit (in der Größe des Graphen $G_{M,n}$) benötigt. Insgesamt ergibt sich ein Zeitbedarf von $2^{\mathcal{O}(f(n))}$. ❑

Wir betrachten die folgenden *fundamentalen* Komplexitätsklassen:

Definition 19.4 (P, NP, PSpace, NPSpace, ExpTime)

$$\begin{aligned}
\mathsf{P} &:= \bigcup_{p \text{ Polynom in } n} \mathsf{DTIME}(p(n)) \\
\mathsf{NP} &:= \bigcup_{p \text{ Polynom in } n} \mathsf{NTIME}(p(n)) \\
\mathsf{PSpace} &:= \bigcup_{p \text{ Polynom in } n} \mathsf{DSPACE}(p(n)) \\
\mathsf{NPSpace} &:= \bigcup_{p \text{ Polynom in } n} \mathsf{NSPACE}(p(n)) \\
\mathsf{ExpTime} &:= \bigcup_{p \text{ Polynom in } n} \mathsf{DTIME}(2^{p(n)})
\end{aligned}$$

Aus Satz 19.3 ergibt sich sofort der folgende Zusammenhang zwischen diesen Klassen:

Korollar 19.5

$$\begin{array}{ccccc}
\mathsf{P} & \subseteq & \mathsf{PSpace} & & \\
\cap\mathrm{I} & & \cap\mathrm{I} & & \\
\mathsf{NP} & \subseteq & \mathsf{NPSpace} & \subseteq & \mathsf{ExpTime}
\end{array}$$

Wir werden später noch sehen, dass PSpace und NPSpace dieselbe Klasse sind, also PSpace = NPSpace gilt. Daher betrachtet man die Klasse NPSpace in der Regel nicht explizit. Mit obigem Korollar ergibt sich also folgendes Bild:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{ExpTime}.$$

Besonders wichtig sind die Komplexitätsklassen P und NP:

- Die Probleme in P werden im Allgemeinen als die *effizient lösbaren* Probleme angesehen (engl. *tractable*, d. h. machbar), siehe auch Appendix **??**.
- Im Gegensatz dazu nimmt man an, dass NP viele Probleme enthält, die *nicht effizient lösbar* sind (engl. *intractable*).

Die Bedeutung der Komplexitätsklasse P ist dadurch hinreichend motiviert. Im Gegensatz dazu ist die Bedeutung von NP etwas schwieriger einzusehen, da nichtdeterministische Maschinen in der Praxis ja nicht existieren. Auch diese Klasse ist aber von großer Bedeutung, da

1. sehr viele natürliche Probleme aus der Informatik in NP enthalten sind, von denen
2. angenommen wird, dass sie nicht in P (also nicht effizient lösbar) sind.

Wir betrachten hier drei typische Beispiele für Probleme in NP, die wir später noch genauer analysieren werden. Das erste solche Problem ist das Erfüllbarkeitsproblem der Aussagenlogik, das in Appendix **??** definiert ist. Dieses Problem repräsentieren wir als Menge SAT aller erfüllbaren aussagenlogischen Formeln. Genauer gesagt enthält SAT nicht die Formeln selbst, sondern wieder eine geeignete Kodierung als Wörter, siehe Abschnitt 17.1. So muss man beispielsweise die potentiell unendlich vielen Aussagenvariablen $x_1, x_2, \ldots$ mittels eines endlichen Alphabets darstellen. Von solchen Details, die leicht auszuarbeiten sind, werden wir im Folgenden aber abstrahieren.

Lemma 19.6
SAT $\in$ NP.

Beweis. Die NTM, die SAT in polynomieller Zeit entscheidet, arbeitet wie folgt:

- Die Variablen in der Eingabeformel φ seien $x_1, \ldots, x_n$. Schreibe nichtdeterministisch ein beliebiges Wort $u \in \{0,1\}^*$ der Länge n hinter die Eingabe auf das Band.
- Betrachte u als Belegung V mit $V(x_i) = 1$ gdw. das i-te Symbol von u eine 1 ist.

  Überprüfe nun deterministisch und in Polynomialzeit, ob V die Formel φ erfüllt (man überlegt sich leicht, dass dies tatsächlich möglich ist). Akzeptiere, wenn das der Fall ist; sonst verwirf.

Die NTM akzeptiert ihre Eingabe gdw. es eine erfolgreiche Berechnung gibt gdw. die Eingabe eine erfüllbare aussagenlogische Formel ist.

Man beachte, dass die Belegungen für φ genau den Berechnungen der NTM entsprechen. Von beiden gibt es exponentiell viele; jede einzelne ist aber nur polynomiell groß bzw. lang. Daher ist polynomielle Zeit ausreichend. ❑

Wir betrachten ein weiteres Problem in NP.

Definition 19.7 (CLIQUE)

Gegeben: Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$.

Frage: Besitzt G eine k-*Clique*, d.h. eine Teilmenge $C \subseteq V$ mit

- für alle $u \neq v$ in C gilt: $\{u, v\} \in E$ und
- $|C| = k$.

Also besteht CLIQUE aus all denjenigen Paaren (G, k), so dass der ungerichtete Graph G eine k-Clique enthält, geeignet repräsentiert als formale Sprache.

Lemma 19.8
CLIQUE $\in$ NP.

Beweis. Eine NTM, die CLIQUE in polynomieller Zeit entscheidet, konstruiert man analog zum Beweis von Lemma 19.6: gegeben (G, k),

- schreibe nichtdeterministisch eine Knotenmenge C der Größe k auf das Band und
- überprüfe deterministisch und in Polynomialzeit, ob C eine Clique in G ist. ❑

Als drittes Beispiel für ein Problem in NP sei das bekannte *Traveling-Salesman-Problem* erwähnt.

Definition 19.9 (Traveling-Salesman-Problem, TSP)

Gegeben: Ein ungerichter Graph $G = (V, E)$, eine Kostenzuweisung $f : E \to \mathbb{N}$ zu den Kanten und Zielkosten $k \in \mathbb{N}$.

Frage: Gibt es eine Tour durch G mit Kosten maximal k, also eine Aufzählung $u_1, \ldots, u_n$ aller Knoten in V so dass

1. $\{u_i, u_{i+1}\} \in E$ für $1 \leq i < n$ sowie $\{u_n, u_1\} \in E$ und
2. $\sum_{1 \leq i \leq n} f(\{u_i, u_{i+1}\}) \leq k$.

Intuitiv beschreiben die Knoten in G Städte, die eine Handlungsreisende besuchen will. Die Kanten in G beschreiben Verbindungen zwischen Städten, und die Funktion f gibt die Kosten dieser Verbindungen an. Gesucht ist nun eine günstige Rundreise, bei der jede Stadt genau einmal besucht wird und die Reisende zum Schluss wieder am Ausgangspunkt ankommt.

Lemma 19.10

TSP $\in$ NP.

Beweis. Eine NTM, die TSP in polynomieller Zeit entscheidet, konstruiert man wieder analog zum Beweis von Lemma 19.6: gegeben $G = (V, E)$, f und k,

- schreibe nichtdeterministisch eine Aufzählung von V auf das Band und
- überprüfe deterministisch und in polynomieller Zeit, ob die Bedingungen 1–2 aus Definition 19.9 erfüllt sind. ❑

Algorithmen, wie sie in den Beweisen der Lemmas 19.6, 19.8 und 19.10 verwendet wurden, formuliert man üblicherweise mittels der Metapher des *Ratens*, zum Beispiel für SAT:

- Bei Eingabe von Formel φ mit Variablen $x_1, \ldots, x_n$ *rate* Belegung V für $x_1, \ldots, x_n$.
- Überprüfe deterministisch und in Polynomialzeit, ob V die Formel φ erfüllt.

Die NTM akzeptiert also die Eingabe, wenn es *möglich* ist, so zu raten, dass die Berechnung erfolgreich ist (in akzeptierender Stoppkonfiguration endet). Man kann sich vereinfachend vorstellen, dass die Maschine *richtig rät*, sofern dies überhaupt möglich ist. Man beachte aber, dass eine polyzeitbeschränkte NTM nur ein polynomiell großes Wort raten kann

und dass man deterministisch in Polyzeit prüfen können muss, ob richtig geraten wurde. So ist es zum Beispiel möglich zu raten, ob eine gegebene TM auf dem leeren Band anhält; aber es ist dann nicht möglich zu überprüfen, ob richtig geraten wurde.

SAT, CLIQUE und TSP zählen zu den Problem in NP, von denen vermutet wird, dass sie *nicht* in polynomieller Zeit lösbar sind. Bisher war jedoch trotz verschiedenster Ansätze noch niemand in der Lage, auch nur zu beweisen, dass $\mathsf{P} \neq \mathsf{NP}$ gilt. Dieses *P-versus-NP-Problem* wird als das wichtigste (und schwierigste) offene Problem der Informatik angesehen. Es gehört zu den sieben Millenniumsproblemen, die im Jahr 2000 durch das *Clay Mathematics Institute (CMI)* zur Förderung der Mathematik in Cambridge (Massachusetts) vorgestellt wurde. Für eine veröffentlichte Lösung lobte das CMI ein Preisgeld von jeweils einer Million US-Dollar aus. Bisher wurde nur eines dieser sieben Probleme gelöst, und zwar die Poincaré-Vermutung 2002 von Grigoriy Perelman.

Man mag sich fragen, ob es sinnvoll ist, die Komplexitätsklassen P und NP basierend auf *Turingmaschinen* zu definieren. Könnte es nicht sein, dass man ein Problem in einer modernen Programmiersprache wie Java mit viel weniger Berechnungsschritten lösen kann als auf einer TM? Interessanterweise scheint das nicht der Fall zu sein: für alle bekannten Berechnungsmodelle gilt, dass sie einander mit *höchstens polynomiellem Mehraufwand* simulieren können. Dies führt zu folgender Verschärfung der Church-Turing-These.

Erweiterte Church-Turing-These:

Für jedes „vernünftige und vollständige“ Berechnungsmodell gibt es ein Polynom p, so dass gilt: jedes Problem, das in diesem Modell von einem $f(n)$-zeitbeschränkten Algorithmus entschieden werden kann, kann mittels einer $p(f(n))$-zeitbeschränkten Turingmaschine entschieden werden, und umgekehrt.

Die erweiterte Church-Turing-These hat denselben Status wie die ursprüngliche Church-Turing-These: sie bezieht sich auf formal nicht definierte Begriffe (wie „vernünftiges Berechnungsmodell“) und kann daher nicht bewiesen werden. Sie wird dennoch als gültig angenommen, da bis heute kein Gegenbeispiel gefunden werden konnte.

20. NP-vollständige Probleme

Wegen $\mathsf{P} \subseteq \mathsf{NP}$ enthält NP auch einfach lösbare Probleme. Es stellt sich also die Frage: Welche Probleme in NP sind schwierig, also nicht in polynomieller Zeit entscheidbar? Bemerkenswerterweise hat auf diese Frage noch niemand eine wasserdichte Antwort gefunden:

- Es gibt sehr viele Probleme in NP, von denen man annimmt, dass sie nicht in polynomieller Zeit zu lösen sind (z. B. SAT, CLIQUE und TSP).
- Dennoch konnte man bisher von keinem einzigen Problem in NP *beweisen*, dass es nicht in P ist.

Es ist also im Prinzip nicht einmal ausgeschlossen, dass die als schwierig vermuteten Probleme in NP sich doch als einfach herausstellen und daher gilt: $\mathsf{P} = \mathsf{NP}$. Daran glaubt jedoch kaum jemand. Ein Beweis von $\mathsf{P} \neq \mathsf{NP}$ steht aber nach wie vor aus. In der Tat handelt es sich dabei um das berühmteste offene Problem der theoretischen Informatik (siehe Ende von Abschnitt 19).

Um die schwierigen Probleme in NP von den einfachen abzugrenzen, obwohl nicht einmal $\mathsf{P} \neq \mathsf{NP}$ bekannt ist, behilft man sich mit der fundamentalen Idee der NP-*Vollständigkeit*. Intuitiv gehört jedes NP-vollständige Problem L zu den schwersten Problemen in NP, in folgendem Sinne:

> Für jedes Problem $L' \in \mathsf{NP}$ gilt: das Lösen von L' erfordert *nur polynomiell mehr Zeitaufwand* als das Lösen von L.

Insbesondere gilt für jedes NP-vollständige Problem L: wenn $L \in \mathsf{P}$, dann ist *jedes* Problem aus NP auch in P; es gilt also $\mathsf{P} = \mathsf{NP}$ (was vermutlich nicht der Fall ist).

Um *polynomiellen Mehraufwand* präzise zu definieren, verfeinern wir in geeigneter Weise den Begriff der Reduktion.

Definition 20.1 (Polynomialzeitreduktion, $\leq_{\mathsf{p}}$)

1) Eine Reduktion f von $L \subseteq \Sigma^*$ auf $L' \subseteq \Sigma^*$ heißt *Polynomialzeitreduktion*, wenn es ein Polynom $p(n)$ und eine $p(n)$-zeitbeschränkte DTM gibt, die f berechnet.
2) Wenn es eine Polynomialzeitreduktion von L auf L' gibt, dann schreiben wir $L \leq_{\mathsf{p}} L'$.

Wir definieren nun die zentralen Begriffe dieses Kapitels.

Definition 20.2 (NP-hart, NP-vollständig)

1) Eine Sprache L heißt NP-*hart*, wenn für *alle* $L' \in \mathsf{NP}$ gilt: $L' \leq_{\mathsf{p}} L$.
2) L heißt *NP-vollständig*, wenn $L \in \mathsf{NP}$ und L NP-hart ist.

Das folgende Lemma ist der Grund, warum die Aussage „L ist NP-vollständig“ ein guter Ersatz für die Aussage „$L \notin \mathsf{P}$“ ist, solange $\mathsf{P} \neq \mathsf{NP}$ nicht bewiesen ist.

Lemma 20.3

Für jede NP*-vollständige Sprache L gilt: wenn $L \in \mathsf{P}$, dann $\mathsf{P} = \mathsf{NP}$.*

Beweis. Sei L NP-vollständig und $L \in \mathsf{P}$. Zu zeigen ist: $\mathsf{NP} \subseteq \mathsf{P}$. Wegen der NP-Vollständigkeit von L gilt $L' \leq_\mathsf{p} L$ für jedes $L' \in \mathsf{NP}$. Es genügt also zu zeigen:

$$L' \leq_\mathsf{p} L \text{ und } L \in \mathsf{P} \text{ impliziert } L' \in \mathsf{P}.$$

Sei f eine Reduktion von L' auf L, berechenbar in Zeit $p(n)$, und sei M eine $q(n)$-zeitbeschränkte DTM, die L_2 entscheidet, wobei p und q Polynome sind. Dann wird L' von einer DTM M' entschieden, die zunächst $f(w)$ berechnet und dann M auf $f(w)$ startet. M' arbeitet in polynomieller Zeit:

- Zum Berechnen von $f(w)$ werden $p(|w|)$ Schritte benötigt (siehe oben).
- Wir können o. B. d. A. annehmen, dass $|f(w)| \leq p(|w|)$.[3] Der Lauf von M benötigt dann Zeit $q(p(|w|))$.

Der Zeitbedarf von M' ist insgesamt also $p(|w|) + q(p(|w|))$. ❑

Es ist zunächst nicht unmittelbar klar, warum NP-vollständige Probleme überhaupt existieren sollten. Überraschenderweise stellt sich aber heraus, dass es viele solche Probleme gibt. Ein besonders wichtiges ist das Erfüllbarkeitsproblem der Aussagenlogik. Das folgende weithin bekannte Resultat wurde unabhängig voneinander von Cook und Levin bewiesen.

Satz 20.4

SAT *ist* NP*-vollständig.*

Beweis. Wir haben bereits gezeigt, dass $\mathsf{SAT} \in \mathsf{NP}$. Es bleibt also zu beweisen, dass SAT NP-hart ist. Mit anderen Worten: wir müssen zeigen, dass *jedes* Problem $L \in \mathsf{NP}$ polynomiell auf SAT reduzierbar ist. Allen diesen Problemen gemeinsam ist, dass sie (per Definition von NP) als das Wortproblem einer polynomiell zeitbeschränkten NTM aufgefasst werden können.

Sei also M eine $p(n)$-zeitbeschränkte NTM, mit $p(n)$ Polynom. Unser Ziel ist, für jede Eingabe w eine aussagenlogische Formel φ_w zu finden, so dass gilt:

1. w wird von M akzeptiert gdw. φ_w erfüllbar ist.
2. φ_w kann in polynomieller Zeit (in $|w|$) konstruiert werden.

Die Konstruktion von φ_w beruht auf den folgenden Ideen. Jede Berechnung von M auf Eingabe $w = a_0 \cdots a_{n-1}$ kann man wie folgt als Matrix darstellen:

[3] Im Allgemeinen muss das nicht der Fall sein. Zum Beispiel könnte f die Identität sein und $p(n)$ die konstante Nullfunktion. Man wählt dann einfach ein größeres Polynom p.

0	$\not\flat$	$\cdots$	$\not\flat$	a_0, q_0	a_1	$\cdots$	a_{n-1}	$\not\flat$	$\cdots$	$\not\flat$
1	$\not\flat$	$\cdots$	$\not\flat$	b	a_1, q	$\cdots$	a_{n-1}	$\not\flat$	$\cdots$	$\not\flat$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$p(n)$										
	$-p(n)$	$\cdots$	-1	0	1	$\cdots$			$\cdots$	$p(n)$

Jede Zeile der Matrix repräsentiert eine Konfiguration, wobei die oberste Zeile der Startkonfiguration entspricht und jede weitere Zeile der Folgekonfiguration der jeweiligen Vorgängerzeile. Die Anzahl der Zeilen ist also durch $p(n)+1$ beschränkt; wir nummerieren sie von 0 bis $p(n)$. Zum Nummerieren der Spalten weisen wir der Spalte, in der in der ersten Zeile der Schreib-Lese-Kopf steht, die Nummer 0 zu. Aufgrund der Zeitbeschränkung von $p(n)$ genügen die Spaltennummern $-p(n), \ldots, p(n)$.

Diese Matrix lässt sich nun mittels polynomiell vieler Aussagenvariablen darstellen:

- $B_{a,i,t}$: „Zum Zeitpunkt t ist Zelle i mit a beschriftet."
- $K_{i,t}$: „Zum Zeitpunkt t ist der Kopf über Zelle i."
- $Z_{q,t}$: „Zum Zeitpunkt t ist q der aktuelle Zustand."

Dabei ist $0 \leq t \leq p(n)$, $-p(n) \leq i \leq p(n)$, $a \in \Gamma$ und $q \in Q$. Wenn beispielsweise in der Startkonfiguration die Zelle 3 mit dem Symbol a beschriftet ist, so wird dies dadurch repräsentiert, dass $B_{a,3,0}$ mit 1 belegt wird und $B_{b,3,0}$ mit 0, für alle $b \in \Gamma \setminus \{a\}$.

Die für die Eingabe w konstruierte Formel φ_w verwendet die obigen Variablen. Wir werden φ_w so konstruieren, dass erfüllende Belegungen von φ_w genau den akzeptierenden Berechnungen von M auf w entsprechen. Genauer gesagt ist φ_w eine Konjunktion, die aus den folgenden Konjunkten besteht:

(i) Die Berechnung beginnt mit der Startkonfiguration für $w = a_1 \cdots a_n$:

$$\psi_{\mathsf{ini}} := Z_{q_0,0} \wedge K_{0,0} \wedge \bigwedge_{0 \leq i < n} B_{a_i,i,0} \wedge \bigwedge_{n \leq i \leq p(n)} B_{\not\flat,i,0} \wedge \bigwedge_{-p(n) \leq i < 0} B_{\not\flat,i,0}$$

(ii) Die Übergangsrelation wird respektiert:

$$\psi_{\mathsf{move}} := \bigwedge_{0 \leq t < p(n)} \bigwedge_{-p(n) \leq i \leq p(n)} \bigwedge_{\substack{a \in \Gamma \\ q \in Q \text{ kein Stoppzustand}}} \Big(\big(B_{a,i,t} \wedge K_{i,t} \wedge Z_{p,t}\big) \rightarrow \bigvee_{\substack{(q,a,a',m,q') \in \Delta \\ \text{so dass } -p(n) \leq m(i) \leq p(n)}} \big(B_{a',m(i),t+1} \wedge K_{m(i),t+1} \wedge Z_{q',t+1}\big) \Big),$$

wobei $m \in \{r, \ell, n\}$, und $m(i)$ ist definiert durch $r(i) = i + 1$, $\ell(i) = i - 1$ und $n(i) = i$.

Wir nehmen hier wieder o. B. d. A. an, dass das Stoppen nur vom Zustand, aber nicht vom gelesenen Symbol abhängt (es macht daher Sinn, von Stopp*zuständen* zu sprechen).

(iii) Zellen, die nicht unter dem Schreib-Lese-Kopf sind, ändern sich nicht:

$$\psi_{\mathsf{keep}} := \bigwedge_{0\le t<p(n)} \bigwedge_{-p(n)\le i\le p(n)} \bigwedge_{a\in\Gamma} \Big(\big(\neg K_{i,t} \wedge B_{a,i,t}\big) \to B_{a,i,t+1} \Big)$$

(iv) Die Eingabe wird akzeptiert:

$$\psi_{\mathsf{acc}} := \bigwedge_{q\in Q\setminus F \text{ Stoppzustand}} \bigwedge_{0\le t\le p(n)} \neg Z_{q,t}$$

Es genügt hier nicht zu fordern, dass ein akzeptierender Stoppzustand vorkommt, da sonst unerwünschte Belegungen der folgenden Form möglich wären: eine verwerfende Berechnung von M, die weniger als die maximal möglichen $p(n)$ Schritte macht, gefolgt von einer Konfiguration mit akzeptierendem Stoppzustand, obwohl M gar keine weiteren Schritte macht.

(v) Bandbeschriftung, Kopfposition, Zustand sind eindeutig und definiert:

$$\psi_{\mathsf{aux}} := \bigwedge_{t,q,q',q\neq q'} \neg(Z_{q,t} \wedge Z_{q',t}) \wedge \bigwedge_{t,i,a,a',a\neq a'} \neg(B_{a,i,t} \wedge B_{a',i,t}) \wedge \bigwedge_{t,i,j,i\neq j} \neg(K_{i,t} \wedge K_{j,t})$$

$$\bigwedge_t \bigvee_q Z_{q,t} \wedge \bigwedge_t \bigvee_i K_{i,t} \wedge \bigwedge_t \bigwedge_i \bigvee_a B_{a,i,t}$$

Hierbei läuft t jeweils implizit über $0,\dots,p(n)$, q und q' laufen über q, i und j über $-p(n),\dots,p(n)$, sowie a und a' über Γ.

Setze nun:

$$\varphi_w = \varphi_{\mathsf{ini}} \wedge \varphi_{\mathsf{move}} \wedge \varphi_{\mathsf{keep}} \wedge \varphi_{\mathsf{acc}} \wedge \varphi_{\mathsf{aux}}$$

Man kann nun überprüfen, dass M die Eingabe w akzeptiert gdw. φ_w erfüllbar ist. Idee:

„⇐“ Wenn w von M akzeptiert wird, dann gibt es eine akzeptierende Berechnung (Konfigurationsfolge)

$$k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_m$$

von M auf w. Erzeuge daraus in der offensichtlichen Weise eine Belegung V für die Variablen $B_{a,i,t}$, $K_{i,t}$ und $Z_{q,t}$, zum Beispiel:

$V(B_{a,i,t}) = 1$ gdw. wenn die i-te Zelle in k_t mit a beschriftet ist.

Wenn $m < p(n)$ (die Berechnung endet vor der maximal möglichen Anzahl von Schritten), dann verlängere vorher die Folge $k_0, k_1, \dots, k_m$ zu $k_0, k_1, \dots, k_{p(n)}$ durch $(p(n)-m)$-maliges Wiederholen der Konfiguration k_m.

Indem man alle Konjunkte $\varphi_{\mathsf{ini}}, \dots, \varphi_{\mathsf{aux}}$ von φ_w durchgeht, überprüft man, dass die so gebildete Belegung φ_w erfüllt. Also ist φ_w erfüllbar.

„⇒“ Aus einer Belegung V der Variablen $B_{a,i,t}$, $K_{i,t}$, $Z_{q,t}$, die φ_w erfüllt, liest man eine Konfigurationsfolge $k_0 \vdash_M k_1 \vdash_M \cdots \vdash_M k_m$ ab, zum Beispiel:

Die i-te Zelle in k_t wird mit a beschriftet, wenn $V(B_{a,i,t}) = 1$.

Die Konfigurationsfolge endet, sobald ein Stoppzustand abgelesen wurde. Unter Verwendung der Tatsache, dass die Belegung alle Konjunkte von φ_w erfüllt, kann man nun zeigen, dass es sich bei der abgelesenen Konfigurationsfolge um eine akzeptierende Berechnung von M auf w handelt.

Abschließend müssen wir noch begründen, dass φ_w in polynomieller Zeit konstruiert werden kann. Wir hatten bereits oben festgestellt, dass in φ_w nur polynomiell viele Aussagenvariablen verwendet werden. Um φ_w zu erzeugen, muss eine DTM nacheinander jedes der Konjunkte (i)–(v) aufs Band schreiben und dazwischen jeweils ein Zeichen „$\wedge$“. Jedes der Konjunkte wiederum kann sie durch eine oder mehrere ineinander verschachtelte for-Schleifen erzeugen, die jeweils nur polynomiell oft durchlaufen werden. Beispielsweise kann ψ_{ini} erzeugt werden, indem die DTM den String „$Z_{q_0,0} \wedge K_{0,0} \wedge$“ aufs Band schreibt, dann in einer for-Schleife für alle $i < n$ den String „$B_{a_i,i,0} \wedge$“ und dann in zwei weiteren for-Schleifen die restlichen Konjunkte von ψ_{ini}. Zur Erzeugung von ψ_{move} müssen Schleifen für t, i mit $0 \leq t < p(n)$ und $-p(n) \leq i \leq p(n)$ sowie für alle Transitionen $(q, a, a', m, q) \in \Delta$ entsprechend ineinander geschachtelt werden. Ähnliches gilt für die Konjunkte (iii)–(v). Dabei sind die verwendeten Parameter t, i, j durch $p(n) = p(|w|)$ beschränkt, und die übrigen Parameter a, a', p, q, q' sowie die Transitionen aus M durch die Größe von M, welche eine Konstante ist (d. h. sie ist nicht von $|w|$ abhängig). Jede for-Schleife wird also nur polynomiell oft durchlaufen, und die Schachtelungstiefe ist konstant. ❏

Analog zu unserem Vorgehen bei der Unentscheidbarkeit erhalten wir weitere NP-harte Probleme durch polynomielle Reduktion von bereits als NP-hart nachgewiesenen Problemen wie SAT. Um das zu zeigen, beobachten wir zunächst:

Lemma 20.5

Wenn $L_1 \leq_{\mathsf{p}} L_2$ und $L_2 \leq_{\mathsf{p}} L_3$, dann $L_1 \leq_{\mathsf{p}} L_3$.

Beweis. Man erhält eine Polynomialzeitreduktion von L_1 auf L_3, indem man Polynomialzeitreduktionen f_1 von L_1 auf L_2 und f_2 von L_2 auf L_3 komponiert, also $f_2(f_1(w))$ berechnet; vergl. Beweis von Lemma 20.3. ❏

Die erwähnten NP-Härtebeweise mittels Reduktion beruhen auf Punkt 2) des folgenden Satzes, der die wichtigsten Zusammenhänge von NP und Polynomialzeitreduktionen zusammenfasst.

Satz 20.6

1) Ist $L_2 \in \mathsf{NP}$ und gilt $L_1 \leq_{\mathsf{p}} L_2$, so ist auch L_1 in NP.

2) Ist L_1 NP-hart und gilt $L_1 \leq_{\mathsf{p}} L_2$, so ist auch L_2 NP-hart.

Beweis.

1) Wegen $L_2 \in \mathsf{NP}$ gibt es eine polynomialzeitbeschränkte NTM M, die L_2 akzeptiert. Wegen $L_1 \leq_{\mathsf{p}} L_2$ gibt es eine Polynomialzeitreduktion f von L_1 auf L_2. Die polynomialzeitbeschränkte NTM für L_1 berechnet zunächst $f(w)$ und startet dann M.

2) Sei L_1 NP-hart und $L_1 \leq_\mathsf{p} L_2$. Wähle ein $L \in \mathsf{NP}$. Wir müssen zeigen, dass $L \leq_\mathsf{p} L_2$.

   Da L_1 NP-hart, gilt $L \leq_\mathsf{p} L_1$. Mit $L_1 \leq_\mathsf{p} L_2$ und Lemma 20.5 folgt wie gewünscht $L \leq_\mathsf{p} L_2$. ❑

Mit Punkt 2) von Satz 20.6 kann man also die NP-Härte eines Problems L nachweisen, indem man eine Polynomialzeitreduktion eines bereits als NP-vollständig bekannten Problems wie SAT auf L findet. Dies wollen wir im Folgenden an einigen Beispielen illustrieren. Wie beginnen mit einem Spezialfall von SAT, bei dem nur aussagenlogische Formeln einer ganz speziellen Form als Eingabe zugelassen sind. Das dadurch entstehende Problem 3SAT spielt eine wichtige Rolle, da es oft einfacher ist, eine Reduktion von 3SAT auf ein gegebenes Problem L zu finden als eine Reduktion von SAT.

Definition 20.7 (3SAT)

Eine *3-Klausel* ist von der Form $\ell_1 \vee \ell_2 \vee \ell_3$, wobei ℓ_i eine Variable oder eine negierte Variable ist. Eine *3-Formel* ist eine endliche Konjunktion von 3-Klauseln.

3SAT ist das folgende Problem:

Gegeben:	eine 3-Formel φ
Frage:	Ist φ erfüllbar?

Satz 20.8

3SAT *ist* NP*-vollständig.*

Beweis.

1) 3SAT $\in$ NP folgt unmittelbar aus SAT $\in$ NP, da jede 3-Formel eine aussagenlogische Formel ist.

2) Es sei φ eine beliebige aussagenlogische Formel. Wir geben ein polynomielles Verfahren an, das φ in eine 3-Formel φ' umwandelt, so dass gilt:

$$\varphi \text{ erfüllbar} \quad \text{gdw.} \quad \varphi' \text{ erfüllbar}$$

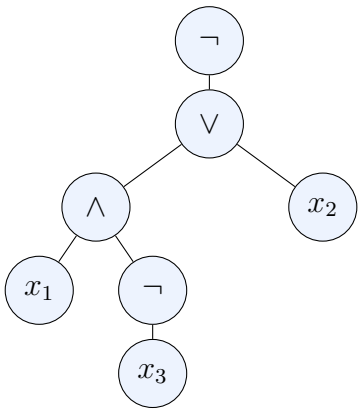
   Beachte: Es ist nicht gefordert, dass φ und φ' im logischen Sinne äquivalent sind, also von denselben Belegungen erfüllt werden.

   Die Umformung erfolgt in mehreren Schritten, die wir am Beispiel der Formel

$$\neg((x_1 \wedge \neg x_3) \vee x_2)$$

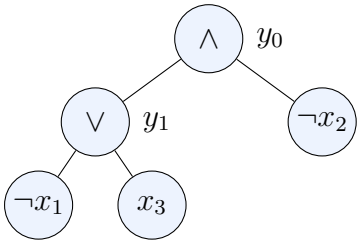
   veranschaulichen.

   Wir stellen diese Formel als Baum dar:



1. Schritt: Wende wiederholt die *De Morganschen Regeln* an und eliminiere doppelte Negationen, um die Negationszeichen zu den Blättern des Baumes zu schieben.

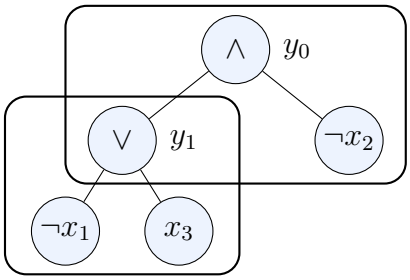
Dies ergibt den folgenden Baum (die Beschriftung y_0, y_1 wird im nächsten Schritt erklärt):



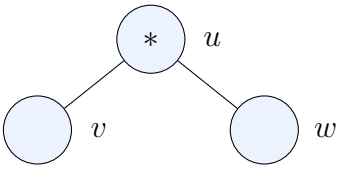
2. Schritt: Ordne jedem inneren Knoten eine neue Variable aus $\{y_0, y_1, \ldots\}$ zu, wobei die Wurzel y_0 erhält.

Intuitiv repräsentiert jede Variable y_i die Teilformel von φ, an deren Wurzel sie steht. Zum Beispiel repräsentiert y_1 die Teilformel $\neg x_1 \vee x_3$.

3. Schritt: Fasse jede Verzweigung (gedanklich) zu einer Dreiergruppe zusammen:



Jeder Verzweigung der Form



mit $* \in \{\wedge, \vee\}$ ordnen wir eine Formel der folgenden Form zu:

$$(u \leftrightarrow (v * w))$$

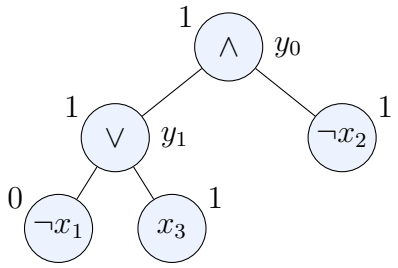
Diese Formeln werden nun konjunktiv mit y_0 verknüpft, was die Formel φ_1 liefert.

Im Beispiel ist φ_1 :

$$y_0 \wedge (y_0 \leftrightarrow (y_1 \wedge \neg x_2)) \wedge (y_1 \leftrightarrow (\neg x_1 \vee x_3))$$

Die Ausdrücke φ und φ_1 sind *erfüllbarkeitsäquivalent*, denn:

Eine erfüllende Belegung für φ kann zu einer für φ_1 erweitert werden, indem man die Werte für die Variablen y_i durch die Auswertung der entsprechenden Teilformel bestimmt, z. B.:



$x_1 = 1$
$x_2 = 0$
$x_3 = 1$
$y_1 = 1$
$y_0 = 1$

Umgekehrt ist jede erfüllende Belegung für φ_1 auch eine für φ. Genauer gesagt kann man von den Blättern zur Wurzel alle Variablen y_i betrachten und jeweils zeigen: der Wahrheitswert von y_i stimmt mit dem Wahrheitswert der von y_i repräsentierten Teilformel überein. Da jede Belegung, die φ_1 erfüllt, y_0 wahr machen muss, erfüllt jede solche Belegung dann auch φ.

4. Schritt: Jedes Konjunkt von φ_1 wird separat in eine 3-Formel umgeformt:

$$\begin{aligned} a \leftrightarrow (b \vee c) &\rightsquigarrow (\neg a \vee (b \vee c)) \wedge (\neg(b \vee c) \vee a) \\ &\rightsquigarrow (\neg a \vee b \vee c) \wedge (\neg b \vee a) \wedge (\neg c \vee a) \\ &\rightsquigarrow (\neg a \vee b \vee c) \wedge (\neg b \vee a \vee a) \wedge (\neg c \vee a \vee a) \end{aligned}$$

$$\begin{aligned} a \leftrightarrow (b \wedge c) &\rightsquigarrow (\neg a \vee (b \wedge c)) \wedge (\neg(b \wedge c) \vee a) \\ &\rightsquigarrow (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee a) \\ &\rightsquigarrow (\neg a \vee b \vee b) \wedge (\neg a \vee c \vee c) \wedge (\neg b \vee \neg c \vee a) \end{aligned}$$

Insgesamt erhält man eine 3-Formel, die äquivalent zu φ_1 und damit wie gewünscht erfüllbarkeitsäquivalent zu φ ist.

Beachte: Jeder Schritt kann offensichtlich deterministisch in polynomieller Zeit durchgeführt werden. ❏

Wir verwenden nun 3SAT, um zwei weitere Probleme als NP-vollständig nachzuweisen. Dabei wählen wir exemplarisch ein graphentheoretisches Problem und ein kombinatorisches Problem auf Mengen. Bei dem graphentheoretischen Problem handelt es sich im CLIQUE, von dem wir ja bereits nachgewiesen haben, dass es in NP liegt (siehe Def. 19.7 und Lemma 19.8).

Satz 20.9

CLIQUE *ist* NP*-vollständig.*

Beweis. Es bleibt, zu zeigen, dass CLIQUE NP-hart ist. Zu diesem Zweck reduzieren wir 3SAT in polynomieller Zeit auf CLIQUE.

Sei also

$$\varphi \;=\; \overbrace{(\ell_{11} \vee \ell_{12} \vee \ell_{13})}^{K_1} \wedge \ldots \wedge \overbrace{(\ell_{m1} \vee \ell_{m2} \vee \ell_{m3})}^{K_m}$$

eine 3-Formel, mit $\ell_{ij} \in \{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$.

Wir müssen zeigen, wie man in polynomieller Zeit aus φ einen Graph G und eine Cliquengröße k erzeugt, so dass φ erfüllbar ist gdw. G eine k-Clique hat.

Dies macht man wie folgt:

- $V := \{\langle i, j\rangle \mid 1 \leq i \leq m \text{ und } 1 \leq j \leq 3\}$
- $E := \{\{\langle i, j\rangle, \langle i', j'\rangle\} \mid i \neq i' \text{ und } \ell_{ij} \neq \overline{\ell}_{i'j'}\}$, wobei

$$\overline{\ell} = \begin{cases} \neg x & \text{falls} \quad \ell = x \\ x & \text{falls} \quad \ell = \neg x \end{cases}$$

- $k := m$

Die Knoten von G entsprechen also den *Vorkommen von Literalen* in φ (wenn ein Literal an mehreren Positionen in φ auftritt, so werden dafür mehrere Knoten erzeugt). Zwei Literalvorkommen werden durch eine (ungerichtete) Kante verbunden, wenn sie sich auf *unterschiedliche* Klauseln beziehen und *nicht* komplementäre Literale betreffen.

Es gilt:

	φ ist erfüllbar mit einer Belegung B
gdw.	es gibt in jeder Klausel K_i ein Literal ℓ_{ij_i} mit Wert 1
gdw.	es gibt Literale $\ell_{1j_1}, \ldots, \ell_{mj_m}$, die paarweise nicht komplementär sind (wobei ℓ_{ij_i} in Klausel i vorkommt)
gdw.	es gibt Knoten $\langle 1, j_1\rangle, \ldots, \langle m, j_m\rangle$, die paarweise miteinander verbunden sind
gdw.	es gibt eine m-Clique in G

❏

Übung: Betrachte die 3-Formel

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee \neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_1 \vee x_4).$$

Konstruiere den Graphen G wie im Beweis von Satz 20.9. Gib eine 3-Clique in G an und die dazugehörige Belegung, die φ erfüllt.

Es folgt die angekündigte Reduktion von 3SAT auf ein mengentheoretisches Problem. Dieses Problem heißt *Mengenüberdeckung*.

Definition 20.10 (Mengenüberdeckung)

Gegeben: Ein *Mengensystem* über einer endlichen Grundmenge M, d. h.

$$T_1, \ldots, T_k \subseteq M$$

sowie eine Zahl $n \geq 0$.

Frage: Gibt es eine Auswahl von n Mengen, die ganz M überdecken, d. h.

$$\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\} \text{ mit } T_{i_1} \cup \ldots \cup T_{i_n} = M\,?$$

Satz 20.11

Mengenüberdeckung ist NP*-vollständig.*

Beweis.

1) Mengenüberdeckung ist in NP:
   - Wähle nichtdeterministisch Indizes $i_1, \ldots, i_n$.
   - Überprüfe, ob $T_{i_1} \cup \ldots \cup T_{i_n} = M$ gilt.
2) Um NP-Härte zu zeigen, reduzieren wir 3SAT in Polyzeit auf Mengenüberdeckung.

   Sei also $\varphi = K_1 \wedge \ldots \wedge K_m$ eine 3-Formel, die die Variablen $x_1, \ldots, x_n$ enthält.

   Wir definieren $M := \{K_1, \ldots, K_m, x_1, \ldots, x_n\}$.

   Für jedes $i \in \{1, \ldots, n\}$ sei

   $$T_i := \{K_j \mid x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{x_i\}$$
   $$T_i' := \{K_j \mid \neg x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{x_i\}$$

   Wir betrachten das Mengensystem:

   $$T_1, \ldots, T_n, T_1', \ldots, T_n'$$

   Zu zeigen:
   φ ist erfüllbar gdw. es eine Mengenüberdeckung von M mit n Mengen gibt.

„⇒“ Sei φ erfüllbar mit Belegung V. Wähle

- T_i, falls $V(x_i) = 1$
- T'_i, falls $V(x_i) = 0$

Dies liefert n Mengen, in denen jedes Element von M vorkommt:

- $K_1, \ldots, K_m$, da V jede Klausel K_i erfüllt
- $x_1, \ldots, x_n$, da für jedes i entweder T_i oder T'_i gewählt wird

„⇐“ Sei umgekehrt

$$\{U_1, \ldots, U_n\} \subseteq \{T_1, \ldots, T_n, T'_1, \ldots, T'_n\} \text{ mit } U_1 \cup \ldots \cup U_n = M.$$

Da jede Variable x_i in $U_1 \cup \ldots \cup U_n$ ist, kommt T_i oder T'_i in $\{U_1, \ldots, U_n\}$ vor. Da wir nur n verschiedene Mengen haben, kommt sogar jeweils *genau eines* von beiden vor.

Definiere Belegung V wie folgt:

$$V(x_i) = \begin{cases} 1 & \text{falls} \quad T_i \in \{U_1, \ldots, U_n\} \\ 0 & \text{falls} \quad T'_i \in \{U_1, \ldots, U_n\} \end{cases}$$

V erfüllt jede Klausel K_j, da K_j in $U_1 \cup \ldots \cup U_n$ vorkommt. ❑

Übung: Betrachte die 3-Formel

$$\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_4 \vee \neg x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_1 \vee x_4).$$

Konstruiere die Mengen $M, T_1, \ldots, T_4, T'_1, \ldots, T'_4$ wie im Beweis von Satz 20.8. Finde Mengen $\{U_1, \ldots, U_4\} \subseteq \{T_1, \ldots, T_4, T'_1, \ldots, T'_4\}$, so dass $U_1 \cup \cdots \cup U_4 = M$, und gib die zugehörige Belegung an, die φ erfüllt.

Es gibt eine Vielzahl von NP-vollständigen Problemen in allen Teilgebieten der Informatik. Eine klassische Referenz ist das Buch „Computers and Intractability: A Guide to the Theory of NP-Completeness“ [GJ79], welches zu den meistzitierten in der Informatik überhaupt gehört. Es enthält eine hervorragende Einführung in das Gebiet der NP-Vollständigkeit und einen Katalog von ca. 300 NP-vollständigen Problemen. Seit der Publikation des Buches wurden tausende weitere Probleme als NP-vollständig identifiziert.

Komplemente und coNP

Wir befassen uns noch mit zwei weiteren Themen, die mit der Klasse NP und dem Begriff der NP-Vollständigkeit zu tun haben. In diesem Abschnitt geht es zunächst um Komplemente von NP-Problemen. Es ist offensichtlich, dass jede mittels deterministischen TMs definierte Komplexitätsklasse C unter Komplement abgeschlossen ist: wenn $L \in$ C durch eine DTM M bezeugt wird, dann wird $\overline{L} \in$ C durch die DTM bezeugt, die man aus

M' erhält, indem man die akzeptierenden und nicht-akzeptierenden Zustände vertauscht (vgl. Satz 17.4). Für nichtdeterministische Klassen funktioniert dieses Argument nicht, da die beschriebene Zustandsvertauschung nicht der Komplementbildung entspricht.

Man betrachte zum Beispiel das Komplement $\overline{\mathsf{SAT}}$ von SAT, also das folgende Problem: gegeben eine aussagenlogische Formel φ, entscheide ob φ *un*erfüllbar ist. Die Lesenden sind aufgefordert, sich zu vergegenwärtigen, dass ein typischer NP-Algorithmus der Form „rate und überprüfe“ hier nicht naheliegend ist, da es nicht um die *Existenz* einer bestimmten Belegung geht, sondern um *alle* Belegungen (sie müssen φ falsch machen). In der Tat ist nicht bekannt, ob $\overline{\mathsf{SAT}} \in \mathsf{NP}$, und es wird vermutet, dass das nicht der Fall ist (und damit natürlich auch, dass NP nicht unter Komplement abgeschlossen ist).

Um Probleme wie $\overline{\mathsf{SAT}}$ komplexitätstheoretisch zu erfassen, führt man eine Komplexitätsklasse namens coNP ein:

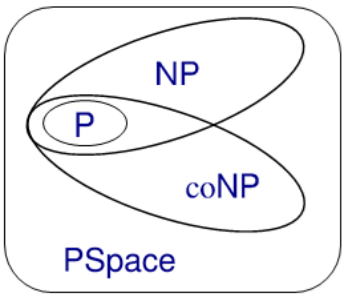
$$\mathsf{coNP} := \{\overline{L} \mid L \in \mathrm{NP}\}.$$

Mit anderen Worten: die Probleme in coNP sind genau die Komplemente der Probleme in NP. Es gilt also z. B. $\overline{\mathsf{SAT}} \in \mathsf{coNP}$. Es gibt aber auch „natürliche“ Probleme (deren Formulierung keine explizite Komplementierung beinhaltet) in coNP. Ein Beispiel ist das Gültigkeitsproblem der Aussagenlogik. Die Lesenden mögen überprüfen, dass dieses Problem wirklich in coNP liegt; mit anderen Worten also: dass das Nicht-Gültigkeitsproblem in NP ist (verwende dazu Lemma **??**).

Man sieht leicht, dass $\mathsf{P} \subseteq \mathsf{coNP}$:

$$L \in \mathsf{P} \quad \Rightarrow \quad \overline{L} \in \mathsf{P} \quad \Rightarrow \quad \overline{L} \in \mathsf{NP} \quad \Rightarrow \quad L \in \mathsf{coNP}$$

Damit gilt auch $\mathsf{P} \subseteq \mathsf{NP} \cap \mathsf{coNP}$ und wir erhalten folgendes Bild:



Man nimmt an, dass alle dargestellten Teilmengenbeziehungen echt sind, dass also gilt $\mathsf{P} \neq \mathsf{NP} \cap \mathsf{coNP}$ und $\mathsf{NP} \subsetneq \mathsf{coNP}$. Deshalb ist es sinnvoll, auch für coNP Begriffe von Härte und Vollständigkeit zu definieren. Dies geschieht vollkommen analog zu NP:

Definition 20.12 (coNP-hart, coNP-vollständig)

1) Eine Sprache L heißt coNP-hart, wenn für *alle* $L' \in \mathsf{coNP}$ gilt: $L' \leq_{\mathsf{p}} L$.
2) L heißt coNP-vollständig, wenn $L \in \mathsf{coNP}$ und L coNP-hart ist.

Man kann beispielsweise mit einfachen Argumenten zeigen, dass das Gültigkeitsproblem der Aussagenlogik coNP-vollständig ist.

Approximation

Viele NP-vollständige Probleme sind in ihrer natürlichsten Formulierung *Optimierungsprobleme* und keine Entscheidungsprobleme, zum Beispiel:

- CLIQUE: Gegeben ungerichteter Graph G, finde die größte Clique in G.
- Mengenüberdeckung: Gegeben Grundmenge M und Mengensystem $T_1, \ldots, T_k$, finde kleinstes $I \subseteq \{1, \ldots, k\}$ (bzgl. Kardinalität), so dass $\bigcup_{i \in I} T_i = M$.

Man überprüft leicht, dass ein Polynomialzeitalgorithmus für das Berechnen einer optimalen Lösung auch einen Polynomialzeitalgorithmus für das entsprechende Entscheidungsproblem liefert. Mit der Existenz solcher Algorithmen ist also nicht zu rechnen. Aber wie steht es mit effizienter Approximation, also mit Polynomialzeitalgorithmen, die zwar keine optimale, aber dennoch eine „gute“ Lösung berechnen?

Im Fall von *Minimierungsproblemen* wie Mengenüberdeckung wünscht man sich einen Approximationsalgorithmus, der in polynomieller Zeit läuft und so dass

$$\frac{\mathsf{APP}}{\mathsf{OPT}} \leq f(|w|)$$

für eine möglichst langsam wachsende Funktion f (und alle Eingaben w) gilt, wobei OPT die Kosten einer optimalen Lösung bei Eingabe w bezeichnet und APP die Kosten der approximativ gefundenen Lösung bei Eingabe w. Der obige Bruch beschreibt den *Approximationsfaktor* und ist stets ≥ 1; ein größerer Faktor bedeutet eine schlechtere Approximation. Man beachte, dass der Faktor mit steigender Eingabelänge größer werden darf, die Approximationsqualität also schlechter.

Wir betrachten Approximation am konkreten Beispiel von Mengenüberdeckung. Eine natürliche Idee ist folgender Greedy-Algorithmus:

Algorithmus 3 : Greedy-Algorithmus für Mengenüberdeckung

Procedure `greedy`$(M, T_1, \ldots, T_k)$**:**
- **input** : Menge M und Mengensystem $T_1, \ldots, T_k$
- **output** : Indexmenge $I \subseteq \{1, \ldots, k\}$, so dass $\bigcup_{i \in I} T_i = M$
- $C := I := \emptyset$
- **while** $C \neq M$ **do**
  - wähle $i \in \{1, \ldots, k\} \setminus I$, so dass $|T_i \setminus C|$ maximal
  - $C := C \cup T_i$
  - $I := I \cup \{i\}$
- **return** I

Der Algorithmus ist *greedy* (also gierig) in dem Sinne, dass er in jedem Schritt versucht, eine möglichst große Anzahl von Elementen aus M zu überdecken. Die Kosten der berechneten Lösung sind $\mathsf{APP} := |I|$.

Wir wollen nun den realisierten Approximationsfaktor abschätzen. Sei $e_1, \ldots, e_n$ eine Aufzählung von M in der Reihenfolge des Hinzufügens der Knoten zu C (bei gleichzeitigem

Hinzufügen wird beliebig aufgelöst). Wir assoziieren wie folgt Kosten mit Mengen T_i und Elementen e_j:

- $c(T_i) = \dfrac{1}{|T_i \setminus C|}$ (gemeint ist die Menge C zum Zeitpunkt des Hinzufügens von i zu I).
- $c(e_\ell) = c(T_i)$, wenn e_ℓ durch Auswahl von T_i zu C hinzugefügt wurde.

Zentral ist die folgende Kostenabschätzung.

Lemma 20.13

$c(e_\ell) \leq \dfrac{\mathsf{OPT}}{(n - \ell + 1)}$ *für jedes Element* e_ℓ .

Beweis. Wir argumentieren wie folgt:

1. In jeder Iteration liefern die verbliebenen (noch nicht ausgewählten) Mengen der *optimalen Lösung* eine Überdeckung der verbliebenen Elemente $\overline{C} = M \setminus C$ mit Kosten $\leq \mathsf{OPT}$.
2. Also gibt es ein verbliebenes T_i mit $c(T_i) \leq \dfrac{\mathsf{OPT}}{|\overline{C}|}$.

   (Denn $c(T_i)$ kann in späteren Iterationen höchstens größer, aber niemals kleiner werden; gälte $c(T_i) > \dfrac{\mathsf{OPT}}{|\overline{C}|}$ für alle T_i , dann ließe sich keine Überdeckung der verbliebenen Elemente mit Kosten $\leq \mathsf{OPT}$ finden).
3. Also $c(e_\ell) \leq \dfrac{\mathsf{OPT}}{|\overline{C}|}$.

   (Denn greedy-Auswahl von i mit $|T_i \setminus C|$ maximal entspricht Auswahl von i mit $c(T_i)$ minimal – per Definition von $c(T_i)$.)
4. Man sieht außerdem leicht: in der Iteration, in der e_ℓ zu M hinzugefügt wurde, enthält $\overline{C}$ mindestens $n - \ell + 1$ Elemente.

Aus den Punkten 3 und 4 erhalten wir also:

$$c(e_\ell) \leq \frac{\mathsf{OPT}}{|\overline{C}|} \leq \frac{\mathsf{OPT}}{n - \ell + 1}$$

❑

Wir können nun den Approximationsfaktor von Algorithmus 3 bestimmen. Er ist (additiv) logarithmisch und damit relativ gut. Insbesondere lässt sich beweisen, dass es für Mengenüberdeckung keine wesentlich besseren Approximationalgorithmen gibt, was angesichts der Einfachheit von Algorithmus 3 überraschend ist (und keineswegs einfach zu beweisen).

Satz 20.14

Für Algorithmus 3 gilt:

$$\frac{\mathsf{APP}}{\mathsf{OPT}} \leq 1 + \ln(|w|)$$

Beweis. Per Definition der Kosten von Elementen ist

$$c(e_1) + \cdots + c(e_n)$$

genau die Anzahl der ausgewählten Mengen, also APP.

Mit Lemma 20.13 folgt

$$\mathsf{APP} \;=\; c(e_1) + \cdots + c(e_n) \;\leq\; \left(1 + \frac{1}{2} + \cdots + \frac{1}{n}\right) \cdot \mathsf{OPT}.$$

Nun ist $1 + \frac{1}{2} + \cdots + \frac{1}{n}$ genau die n-te harmonische Zahl H_n und es ist bekannt, dass $H_n \leq 1 + \ln(n)$. ❑

Wir beenden an dieser Stelle unseren kurzen Ausflug zur Approximation und erwähnen noch, dass es natürlich viele andere Approximationstechniken gibt als Greedy-Algorithmen und dass sich NP-vollständige Probleme äußerst unterschiedlich bezüglich Approximierbarkeit verhalten. Hier einige Beispiele:

- Knotenüberdeckung in Graphen ist ein (immer noch NP-vollständiger) Spezialfall von Mengenüberdeckung, der Approximation mit konstantem Faktor 2 erlaubt. Die Approximationsgüte ist hier also nicht von der Eingabegröße abhängig.
- Mengenüberdeckung ist approximierbar mit logarithmisch-polynomiellem Faktor (Polynom von Logarithmus der Eingabegröße). Dies ist noch als verhältnismäßig gute Approximation aufzufassen.
- CLIQUE ist lediglich mit linearem Faktor approximierbar. Das ist ein schlechter Faktor, da die Qualität der Approximation mit steigender Eingabegröße rapide abnimmt.
- Das Traveling-Salesman-Problem ist gar nicht approximierbar in dem Sinne, dass nicht garantiert werden kann, dass überhaupt eine Lösung gefunden wird, wenn es eine gibt.

21. Jenseits von NP

Wie erwähnt, geht eine überwältigende Zahl von Forschern davon aus, dass P eine *echte* Teilmenge von NP ist. In der Tat kann man die Theorie der NP-Vollständigkeit als einen Anhaltspunkt werten, dass dies wirklich der Fall ist. Im Gegensatz dazu stellen sich interessanterweise die *Platz*komplexitätsklassen PSpace und NPSpace, die ja komplett analog zu P und NP definiert sind, als identisch heraus! Das folgende Resultat ist als der *Satz von Savitch* (auf Englisch: Savitch's Theorem) bekannt und wurde um 1970 bewiesen.

Satz 21.1

PSpace = NPSpace.

Beweis. Wir müssen zeigen, dass jede durch ein Polynom p platzbeschränkte NTM M durch eine polynomiell platzbeschränkte DTM M' simuliert werden kann. Ein erster Versuch könnte zum Beispiel darin bestehen, M' bei gegebener Eingabe w zunächst den Konfigurationsgraphen $G_{M,p(|w|)}$ (siehe Beweis von Satz 19.3) konstruieren zu lassen und dann nur noch zu überprüfen, ob eine akzeptierende Stoppkonfiguration von der Startkonfiguration aus erreichbar ist (alles deterministisch!). Dieser Ansatz scheitert jedoch daran, dass $G_{M,p(|w|)}$ exponentiell groß ist und der beschriebene Ansatz also unzulässig viel Platz verwendet.

Der Beweis des Satzes von Savitch beruht auf der Idee, dass man den eben beschriebenen Ansatz auch realisieren kann, ohne den Graphen $G_{M,p(|w|)}$ jemals komplett im Speicher zu halten. Das funktioniert wie folgt.

Wir nehmen o. B. d. A. an, dass es nur eine einzige akzeptierende Stoppkonfiguration k_{akz} gibt. Dies kann zum Beispiel dadurch erreicht werden, dass M vor dem Akzeptieren den Bandinhalt löscht und den Schreib-Lese-Kopf an eine definierte Stelle bewegt (z. B. vormaliger Beginn der Eingabe). Definiere ein Prädikat $\mathsf{Pfad}(k, k', i)$ wie folgt:

> $\mathsf{Pfad}(k, k', i)$ ist wahr gdw. es in $G_{M,p(|w|)}$ einen Pfad von Konfiguration k nach Konfiguration k' gibt, dessen Länge durch 2^i beschränkt ist.

Sei N die Anzahl der Knoten im Graph $G_{M,p(|w|)}$, also die Anzahl der Konfigurationen von M der Länge $p(|w|)$. Wir können die Aufgabe von M' nun wie folgt beschreiben: Entscheide, ob Folgendes gilt:

$$\mathsf{Pfad}(q_0 w, k_{\mathsf{akz}}, \lceil \log(N) \rceil).$$

Man beachte, dass der Logarithmus durch die Exponentiation in der Definition des Pfad-Prädikates wieder aufgehoben wird – man interessiert sich also für Pfade der Länge N, was offensichtlich ausreichend ist. Wir hatten uns bereits im Beweis von Satz 19.3 überlegt, dass $N \leq 2^{\mathcal{O}(p(|w|))}$. Damit ist aber $\lceil \log(N) \rceil$ polynomiell in $|w|$. Dies legt den folgenden „Teile und Herrsche"-Ansatz nahe:

> Um $\mathsf{Pfad}(k, k', i)$ zu entscheiden, iteriere über alle möglichen Mittelpunkte $\widehat{k}$ auf dem gesuchten Pfad von k zu k', entscheide rekursiv $\mathsf{Pfad}(k, \widehat{k}, i-1)$ und $\mathsf{Pfad}(\widehat{k}, k', i-1)$.

Beachte: wegen der Exponentiation in der Definition des Pfad-Prädikates entspricht hier „-1“ dem Teilen der Pfadlänge durch zwei!

Insgesamt erhalten wir den folgenden Algorithmus, der auf $\mathsf{Pfad}(q_0 w, k_{\mathsf{akz}}, \lceil \log(N) \rceil)$ gestartet wird:

Algorithmus 4 : Berechnung des Pfad-Prädikats

```
Procedure path(k, k', i):
    input  : Konfigurationen k, k' als Wörter kodiert; i ∈ ℕ
    output : true, falls Pfad(k, k', i); false sonst
    if i = 0 then
        if k = k' or k ⊢_M k' then return true
        else return false
    else
        foreach k̂ in G_{M,p(|w|)} do
            if path(k, k̂, i − 1) and path(k̂, k', i − 1) then return true
        return false
```

Mit den bereits gelieferten Argumenten ist dieser Algorithmus offensichtlich korrekt, d. h. eine DTM M', die diesen Algorithmus ausführt, entscheidet dieselbe Sprache wie die ursprüngliche NTM M. Es verbleibt, den Platzbedarf von M' zu analysieren:

- In jedem Aufruf sind die Werte von k, k', $\widehat{k}$ und i zu speichern. Die Größe der Konfigurationen ist $\mathcal{O}(p(|w|))$, da ihre Länge ja nur $p(|w|)$ beträgt. Da wir den Algorithmus mit $i = \lceil \log(N) \rceil$ starten, sind auch alle Werte von i in $\mathcal{O}(p(|w|))$.
- Auch die Größe des Rekursionsstacks trägt zum Speicherbedarf bei. Wegen des schon erwähnten Startwerts für i ist die Rekursionstiefe offensichtlich durch $\mathcal{O}(p(|w|))$ beschränkt.

Insgesamt ergibt sich damit ein Speicherplatzbedarf von $\mathcal{O}(p(|w|)^2)$. ❑

Aus dem Satz von Savitch ergibt sich zusammen mit Korollar 19.5:

Korollar 21.2

$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{ExpTime}$

Der Satz von Savitch hat unter anderem folgende Konsequenzen:

- Die Komplexitätsklasse NPSpace wird im Allgemeinen nicht explizit betrachtet.
- Wenn man nachweisen will, dass ein Problem in PSpace ist, dann kann man o. B. d. A. eine nichtdeterministische Turingmaschine verwenden, was oft praktisch ist (man kann also *raten*).

Den zweiten Punkt wollen wir illustrieren, indem wir folgendes Resultat beweisen.

Satz 21.3

Das Wortproblem für monotone Grammatiken ist in PSpace.

Beweis. Wegen Satz 21.1 genügt es zu zeigen: Es gibt eine polynomiell platzbeschränkte NTM, die das Wortproblem für monotone Grammatiken löst.

Sei als Eingabe die monotone Grammatik $G = (N, \Sigma, P, S)$ und das Wort $w \in \Sigma^*$ gegeben. Die NTM muss prüfen, ob G eine Ableitung von w erlaubt. Wir schätzen zunächst die Länge der zu betrachtenden Ableitungen ab:

- Wir können uns offensichtlich auf Ableitungen konzentrieren, in denen kein Wort doppelt vorkommt.
- Da G monoton ist, kommen in einer erfolgreichen Ableitung von w nur Wörter der Länge $\leq |w|$ vor.
- Es gibt nicht mehr als $\ell_{\mathsf{max}} := (|N \cup \Sigma| + 1)^{|w|}$ viele solche Wörter.

Die zu betrachtenden Ableitungen können also exponentiell lang sein und passen damit nicht in den polynomiell großen Speicher. Um die Existenz einer Ableitung

$$S = \alpha_0 \vdash_G \alpha_1 \vdash_G \cdots \vdash_G \alpha_k = w$$

zu überprüfen, geht die NTM daher wie folgt vor: Rate die Ableitung Schritt für Schritt; behalte stets nur zwei Wörter gleichzeitig im Speicher. Es wird also zunächst das Wort α_0 erzeugt. Danach wird α_1 geraten und $\alpha_0 \vdash_G \alpha_1$ überprüft. Dann wird α_0 aus dem Speicher gelöscht und α_2 geraten usw.

Es ist nun aber möglich so zu raten, dass die NTM in eine Endlosschleife gerät. Um dies zu verhindern, lässt sie zudem einen binären Schrittzähler mitlaufen. Wird w erzeugt, so akzeptiert die NTM. Wird w nach ℓ_{max} Schritten nicht erreicht, so verwirft sie.

Der Platzbedarf zum Speichern des Zählers beträgt $\lceil \log(\ell_{\mathsf{max}}) \rceil$, ist also polynomiell, da ℓ_{max} nur einfach exponentiell groß ist. Insgesamt wird damit nur polynomiell viel Speicher benötigt.

Man kann sich nun noch von der Korrektheit des Algorithmus überzeugen: Ist $w \in L(G)$, dann erzeugt eine der NTM-Berechnungen eine Ableitung von w der Länge $\leq \ell_{\mathsf{max}}$ und akzeptiert dann. Ist hingegen $w \notin L(G)$, dann erzeugt keine der NTM-Berechnungen eine Ableitung von w; also sind alle Berechnungen verwerfend. ❑

Entsprechend zur Definition von NP-Vollständigkeit eines Problems kann man auch Vollständigkeit für andere Komplexitätsklassen definieren, zum Beispiel für PSpace. Man beachte die Ähnlichkeit zu Definition 20.2.

Definition 21.4 (PSpace-hart, PSpace-vollständig)

1) Eine Sprache L heißt PSpace-*hart*, wenn für *alle* $L' \in \mathsf{PSpace}$ gilt: $L' \leq_p L$.
2) L heißt PSpace-*vollständig*, wenn $L \in \mathsf{PSpace}$ und L PSpace-hart ist.

Es ist leicht zu sehen, dass jedes PSpace-vollständige Problem auch NP-hart ist. Man nimmt an, dass PSpace-vollständige Probleme echt schwieriger sind als NP-vollständige

(dass also NP eine echte Teilmenge von PSpace ist), hat aber bisher keinen Beweis dafür gefunden.

Man kann beweisen, dass folgende Probleme aus dieser Vorlesung PSpace-vollständig sind:

- das Äquivalenzproblem für NEAs und für reguläre Ausdrücke
- das Wortproblem für monotone Grammatiken

ExpTime

Mit einem raffinierten Diagonalisierungsargument, das wird hier nicht im Detail betrachten wollen, kann man Folgendes zeigen.

Satz 21.5
$\mathsf{P} \neq \mathsf{ExpTime}$

Man kann mit einer DTM in exponentieller Zeit also beweisbar mehr Probleme lösen als in polynomieller Zeit. Das bedeutet natürlich, dass auch mindestens eine der drei Inklusionen aus Korollar 21.2

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{ExpTime}$$

echt sein muss. Leider weiß man aber nicht, welche Inklusion das ist. Analog zu NP-Härte und PSpace-Härte kann man den Begriff der ExpTime-Härte definieren. Es ist leicht, zu sehen, dass ein ExpTime-hartes Problem wegen Satz 21.5 nicht in P sein kann. Solche Probleme findet man zum Beispiel in der Logik oder in Form von (generalisierten) Spielen wie Dame und Schach.

Es gibt in der Komplexitätstheorie also durchaus Resultate, die Komplexitätsklassen separieren; Satz 21.5 ist nur eines von mehreren Beispielen. Man mag sich fragen, ob ein cleveres Diagonalisierungsargument auch verwendet werden kann, um das berühmte P-vs.-NP-Problem zu lösen, indem man per Diagonalisierung $\mathsf{P} \neq \mathsf{NP}$ zeigt. Leider ist dies nicht möglich: wir haben in Kapitel 18 gesehen, dass Diagonalisierung eine Beweistechnik ist, die auf Orakel relativiert. Man kann zeigen, dass keine Beweistechnik mit dieser Eigenschaft geeignet ist, um $\mathsf{P} \neq \mathsf{NP}$ zu beweisen (Satz von Gill, Baker, Solovay). Insgesamt weiß man leider deutlich mehr darüber, wie man $\mathsf{P} \neq \mathsf{NP}$ *nicht* beweist, als es aussichtsreiche Kandidaten für einen solchen Beweis gibt.

Es soll abschließend noch betont werden, dass es neben den hier diskutierten Komplexitätsklassen noch viele weitere wichtige Klassen gibt. Dies betrifft sowohl Klassen unterhalb von P und Klassen oberhalb von ExpTime als auch Klassen, die „zwischen“ den hier dargestellten liegen oder deren genauer Zusammenhang zu den hier behandelten Klassen bisher unbekannt ist. Beispielsweise ergeben sich die wichtigen Klassen LogSpace und

NLogSpace durch die Verwendung von *logarithmisch viel Platz* (wobei die Eingabe auf einem Extraband liegt und nicht zum Platzverbrauch beiträgt) und liegen unterhalb von P. Im Unterschied zu PSpace und NPSpace ist nicht bekannt, ob diese Klassen identisch sind. Weitere wichtige Klassen wie etwa RP, BPP und PP ergeben sich, wenn man Turingmaschinen um *probabilistische Fähigkeiten* ergänzt. Intuitiv darf eine solche Maschine „Münzen werfen“ und muss dann nur mit einer gewissen Wahrscheinlichkeit die richtige Antwort geben.

A. Anhang: Abkürzungsverzeichnis etc.

bzw. ..... beziehungsweise
DEA ..... deterministischer endlicher Automat
d. h. ..... das heißt
DTM ..... deterministische Turingmaschine
EBNF ..... erweiterte Backus-Naur-Form
etc. ..... et cetera
gdw. ..... genau dann wenn
LBA ..... linear beschränkter Automat
MPKP ..... modifiziertes Postsches Korrespondenzproblem
NEA ..... nichtdeterministischer endlicher Automat
NP ..... nichtdeterministisch polynomiell
NTM ..... nichtdeterministische Turingmaschine
o. B. d. A. ..... ohne Beschränkung der Allgemeinheit
PDA ..... pushdown automaton (Kellerautomat)
dPDA ..... Deterministischer Kellerautomat
PKP ..... Postsches Korrespondenzproblem
SAT ..... satisfiability problem (Erfüllbarkeitstest der Aussagenlogik)
TM ..... Turingmaschine (allgemein)
usw. ..... und so weiter
vgl. ..... vergleiche
z. B. ..... zum Beispiel
□ ..... was zu beweisen war (q. e. d.)

Mathematische Symbole und Notation

Allgemeines

$\{\ \}$ ..... Mengenklammern

$x \in M$ ..... x ist Element der Menge M

$M_1 \subseteq M_2$ ..... M_1 Teilmenge von M_2

$M_1 \subset M_2$ oder $M_1 \subsetneq M_2$ ..... M_1 *echte* Teilmenge von M_2

$\emptyset$ ..... leere Menge

$M_1 \cup M_2$ ..... Vereinigung von Mengen M_1 und M_2

$M_1 \cap M_2$ ..... Schnitt von Mengen M_1 und M_2

$M_1 \setminus M_2$ ..... Mengendifferenz: M_1 ohne die Elemente von M_2

$\overline{M}$ ..... Komplement der Menge M
(bezüglich einer als bekannt angenommenen „Gesamtmenge“)

$M_1 \times M_2$ ..... Kreuzprodukt der Mengen M_1 und M_2

$|M|$ ..... Kardinalität der Menge M (Anzahl Elemente)

2^M ..... Potenzmenge von M (Menge aller Teilmengen)

$\wedge$ ..... logisches „und“

$\vee$ ..... logisches „oder“

$\neg$ ..... logisches „nicht“

$\Rightarrow$ (auch $\rightarrow$) ..... logisches „impliziert“

$\Leftrightarrow$ (auch $\leftrightarrow$) ..... logisches „genau dann, wenn“

$\exists$ ..... „es existiert“

$\forall$ ..... „für alle“

$\mathbb{N}$ ..... Menge der natürlichen Zahlen (einschließlich der Null)

$f : M \rightarrow M'$ ..... Funktion f bildet von Menge M in Menge M' ab

$n!$ ..... n Fakultät ($n! = 1 \cdot 2 \cdot \ldots \cdot n$)

$f \in O(g)$ ..... Funktion f wächst schließlich nicht schneller als Funktion g

$f|_D$ ..... Einschränkung der Funktion f auf Definitionsbereich D

$\sim$ ..... Äquivalenzrelation

$:=$ ..... „ist definiert als“

Wörter und Sprachen

Σ ..... Alphabet (oft auch: Eingabealphabet)
a, b, c ..... Alphabetsymbole
w, u, v, x, y, z ..... Wörter
ε ..... leeres Wort
a^n ..... Wort, das aus n mal dem Symbol a besteht
$|w|$ ..... Länge des Wortes w
$|w|_a$ ..... Anzahl Vorkommen des Symbols a im Wort w
$w_1 \cdot w_2$ ..... Konkatenation der Wörter w_1 und w_2
L ..... formale Sprache
$L_1 \cdot L_2$ ..... Konkatenation der Sprachen L_1 und L_2
L^i ..... i-fache Konkatenation der Sprache L mit sich selbst
L^* ..... Kleene-Stern, angewendet auf die Sprache L
L^+ ..... $= L \cdot L^*$

Endliche Automaten

$\mathcal{A}$ ..... Automat (DEA, NEA, Kellerautomat, LBA, Turingmaschine)
Q ..... Zustandsmenge
F ..... Endzustandsmenge
q, p ..... Zustände von Automaten
q_0 ..... Startzustand von Automat
δ ..... Übergangsfunktion von deterministischen Automaten
$\hat{\delta}$ ..... Kanonische Fortsetzung der Übergangsfunktion
Δ ..... Übergangsrelation von nichtdeterministischen Automaten / Turingmaschinen
$L(\mathcal{A})$ ..... Vom Automaten $\mathcal{A}$ erkannte Sprache
$p \xrightarrow{a}_{\mathcal{A}} q$ ..... NEA $\mathcal{A}$ kann in einem Schritt unter Lesen von Symbol a von Zustand p in Zustand q übergehen
$p \xRightarrow{w}_{\mathcal{A}} q$ ..... NEA $\mathcal{A}$ kann unter Lesen von Wort w von Zustand p in Zustand q übergehen
$p \Longrightarrow^i_{\mathcal{A}} q$ ..... NEA $\mathcal{A}$ kann in i Schritten von Zustand p in Zustand q übergehen

Reguläre Ausdrücke

r, s ..... reguläre Ausdrücke
$r \cdot s$ ..... regulärer Ausdruck für Konkatenation
$r + s$ ..... regulärer Ausdruck für Vereinigung
r^* ..... regulärer Ausdruck für Kleene-Stern
Reg_Σ ..... Menge aller regulärer Ausdrücke über Σ

Nerode-Rechtskongruenz

$q \sim_{\mathcal{A}} q'$ ..... Zustände q und q' sind $\mathcal{A}$-äquivalent
$[q]_{\mathcal{A}}$ ..... Äquivalenzklasse von Zustand q bzgl. $\sim_{\mathcal{A}}$
$\simeq_L$ ..... Nerode-Rechtskongruenz für Sprache L
$[u]_L$ ..... Äquivalenzklasse von Wort u bzgl. $\simeq_L$
$\mathcal{A} \simeq \mathcal{A}'$ ..... Die DEAs $\mathcal{A}$ und $\mathcal{A}'$ sind isomorph

Grammatiken und Chomsky-Hierarchie

G ..... Grammatik
N ..... Menge der nichtterminalen Symbole einer Grammatik
P ..... Menge der Regeln (Produktionen) einer Grammatik
A, B, C ..... Nichtterminale Symbole
S ..... Startsymbol
$u \longrightarrow v$ ..... Produktion einer Grammatik
$x \vdash_G y$ ..... Wort y aus Wort x in Grammatik G in einem Schritt ableitbar
$x \vdash_G^i y$ ..... Wort y aus Wort x in Grammatik G in i Schritten ableitbar
$x \vdash_G^* y$ ..... Wort y aus Wort x in Grammatik G ableitbar
$L(G)$ ..... die von Grammatik G erzeugte Sprache
$\mathcal{L}_0$ ..... Klasse der Sprachen vom Typ 0
$\mathcal{L}_1$ ..... Klasse der kontextsensitiven Sprachen (Typ 1)
$\mathcal{L}_2$ ..... Klasse der kontextfreien Sprachen (Typ 2)
$\mathcal{L}_3$ ..... Klasse der regulären Sprachen (Typ 3)

Kellerautomaten und Turingmaschinen

Γ ..... Kelleralphabet (Kellerautomat), Bandalphabet (Turingmaschine)
Z_0 ..... Kellerstartsymbol
$k \vdash_{\mathcal{A}} k'$ ..... Kellerautomat/Turingmaschine $\mathcal{A}$ kann in einem Schritt von Konfiguration k in Konfiguration k' übergehen
$k \vdash_{\mathcal{A}}^i k'$ ..... Kellerautomat/Turingmaschine $\mathcal{A}$ kann in i Schritten von Konfiguration k in Konfiguration k' übergehen
$k \vdash_{\mathcal{A}}^* k'$ ..... Kellerautomat/Turingmaschine $\mathcal{A}$ kann von Konfiguration k in Konfiguration k' übergehen
$\mathcal{L}_2^d$ ..... Klasse der deterministisch kontextfreien Sprachen

$\not\flat$ ..... Blank-Symbol (Turingmaschine)
$\$, \not\in$ ..... Bandbegrenzungssymbole (LBA)

Griechische Buchstaben

Kleinbuchstaben und einige typische Verwendungen im Skript

α ..... alpha
β ..... beta
γ ..... gamma
δ (bezeichnet meist Übergangsfunktion) ..... delta
ε (bezeichnet meist das leere Wort) ..... epsilon
ζ ..... zeta
η ..... eta
ϑ, θ ..... theta
ι ..... iota
κ ..... kappa
λ ..... lambda
μ ..... my
ν ..... ny
ξ xi
o ..... omikron
π (bezeichnet meist Pfad in NEA) ..... pi
ρ ..... rho
σ, ς ..... sigma
τ ..... tau
υ ..... ypsilon
φ, ϕ ..... phi
χ ..... chi
ψ ..... psi
ω ..... omega

Großbuchstaben und einige typische Verwendungen im Skript

Γ (bezeichnet meist Keller- bzw. Bandalphabet) ..... Gamma
Δ (bezeichnet meist Übergangsrelation) ..... Delta
Θ ..... Theta
Λ ..... Lambda
Π ..... Pi
Ξ Xi
Σ (bezeichnet meist Alphabet) ..... Sigma
Υ ..... Ypsilon
Φ ..... Phi
Ψ ..... Psi
Ω ..... Omega

Die restlichen griechischen Großbuchstaben werden genauso geschrieben wie die entsprechenden lateinischen.

Literaturverzeichnis

[CLRS01] Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest und Clifford Stein: *Introduction to Algorithms*. The MIT Press, 2. Auflage, 2001.

[GJ79] Garey, Michael R. und David S. Johnson: *Computers and Intractability: A Guide to the Theory of* NP*-Completeness*. Freeman, 1979.

[GKP94] Graham, Ronald L., Donald E. Knuth und Oren Patashnik: *Concrete Mathematics*. Addison-Wesley, 2. Auflage, 1994.

[Koz07] Kozen, Dexter: *Automata and Computability*. Springer Verlag, 2007.

[Sch08] Schöning, Uwe: *Theoretische Informatik – kurzgefasst*. Spektrum Akademischer Verlag, 5. Auflage, 2008.

[Weg05] Wegener, Ingo: *Theoretische Informatik*. Teubner-Verlag, 2005.