Universität Bremen

AG Theoretische Grundlagen der KI, Fachbereich Informatik, Universität Bremen

Skript zu den Lehrveranstaltungen

Theoretische Informatik 1 + 2

Prof. Dr. Carsten Lutz

Inhaltsverzeichnis

Hinweis

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

Dieses Skript ist eine erweiterte und modifizierte Version eines Vorlesungsskriptes von Franz Baader.

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 Theoretische Informatik 1 / Teile I + 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 Programmes 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)
- Was für verschiedene Typen von Sprachen lassen sich unterscheiden?
- Was für Eigenschaften haben die verschiedenen Sprachtypen?

Berechenbarkeit und Komplexität

Behandelt in Theoretische Informatik 2 / Teile III + 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.:

- Was für Berechenbarkeitsmodelle 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 wir Pascal, Java, oder C++ ist eine formale Sprache.
- Die Menge aller wohlgeformten Eingaben für ein Programm oder eine Form 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 i.d.R. unendlich, wie in allen obigen Beispielen (es gibt zum Beispiel unendlich viele wohlgeformte Pascal-Programme). Wie beschreibt man derartige Sprachen mit endlichem Aufwand?

   - *Automaten* oder *Maschinen*, die genau die Elemente der Menge akzeptieren. Wir werden viele verschiedene Automatenmodelle kennenlernen, wie z.B. endliche Automaten, Kellerautomaten und Turingmaschinen.
   - *Grammatiken*, die genau die Elemente der Menge 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, die in der **Chomsky-Hierarchie** zusammengefasst sind:

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

Bei 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 Automat, Grammatik, 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 Durchschnitt, 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 (Durchschnitt).
- 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 "Theoretische Informatik 1" sind *Wörter* und *formale Sprachen*.

Wörter und Formale Sprachen

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

- $\Sigma_1 = \{a, b, c, \ldots, z\}$;
- $\Sigma_2 = \{0, 1\}$;
- $\Sigma_3 = \{0, \ldots, 9\} \cup \{,\}$;
- $\Sigma_4 =$ { program, const, var,label, procedure, function, type, begin, end, if, then, else, case, of, repeat, until, while, do, for, to } $\cup$ { 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 Schlüsselworte der Programmiersprache Pascal. Konkrete Variablennamen, Werte und Funktionsaufrufe sind zu den Schlüsselworten VAR, VALUE, FUNCTION abstrahiert, um Endlichkeit des Alphabetes 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 Pascalprogramm 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 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 bestehen, 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 Worte über Σ_4, die wohlgeformte Pascal-Programme beschreiben

Als Symbol Platzhalter für Sprachen verwenden wir meist L. Beachten Sie, 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 nichtleeren Wörter über Σ ein. Sowohl Σ^* als auch Σ^+ sind offensichtlich unendliche Sprachen.

Operationen auf Sprachen und Wörtern

Im folgenden werden wir sehr 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:

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

Die Präfixe von *aabbcc* sind also beispielsweise $a, aa, aab, aabb, aabbc, aabbcc$. Dieses Wort hat 21 Infixe (Teilwörter).

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* Worte aus den beteiligten Sprachen:

$$L_1 \cdot L_2 := \{u \cdot v \mid (u \in L_1) \wedge (v \in L_2)\}$$

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 $\emptyset \cdot L = L \cdot \emptyset = \emptyset$. 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 $L_1 \cdot L_2 = L_2 \cdot L_1$.

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:

$$\begin{array}{lrcl} \text{Vereinigung} & L_1 \cup L_2 & := & \{w \mid w \in L_1 \text{ oder } w \in L_2\} \\ \text{Durchschnitt} & L_1 \cap L_2 & := & \{w \mid w \in L_1 \text{ und } w \in L_2\} \\ \text{Komplement} & \overline{L_1} & := & \{w \mid w \in \Sigma^* \wedge w \notin L_1\} \end{array}$$

Manchmal verwenden wir zusätzlich die Differenz, also

$$L_1 \setminus L_2 := L_1 \cap \overline{L_2} = \{w \mid w \in L_1 \wedge w \notin L_2\}.$$

Vereinigung und Durchschnitt 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 induktive Sprachen $L^0, L^1, \ldots$ und darauf basierend dann die Anwendung des Kleene-Sterns erhaltene Sprache L^*:

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

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 das leere Wort:

$$L^+ := \bigcup_{n \geq 1} L^n = L^* \setminus \{\varepsilon\}.$$

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

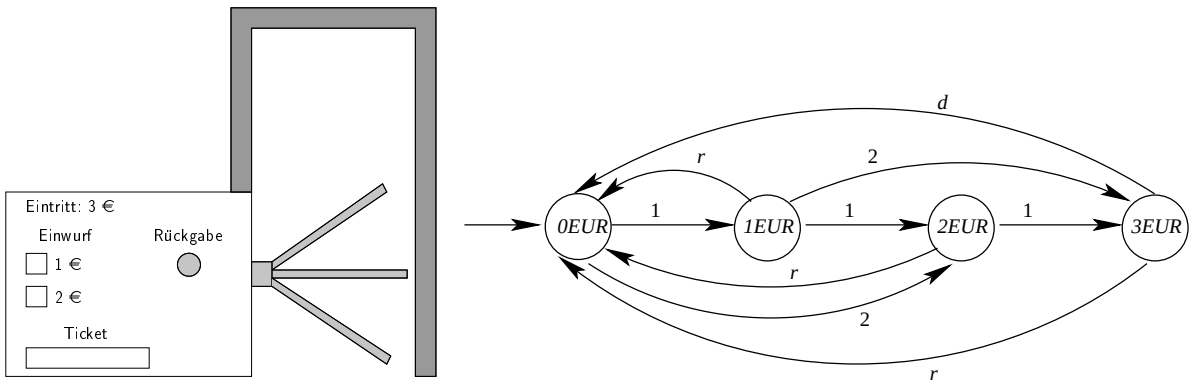
- eine endliche Menge von Zuständen, in denen sich der Automat befinden kann

  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).

- festen Übergängsregeln 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 Euro eingeworfen werden. Nach Einwurf von 3 Euro 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,- Euro 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 dem 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 formal.

Definition 1.1 (DEA)
Ein *deterministischer endlicher Automat (DEA)* ist von der Form $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, wobei

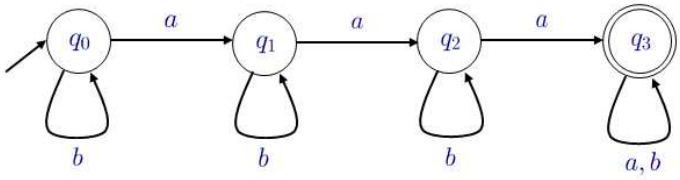
- Q eine endliche Menge von *Zuständen* ist,
- Σ ein *Eingabealphabet* ist,
- $q_0 \in Q$ der *Anfangszustand* ist,
- $\delta : Q \times \Sigma \to Q$ die *Übergangsfunktion* ist,
- $F \subseteq Q$ eine Menge von *Endzuständen* ist.

Beispiel 1.2
Der DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ mit den Komponenten

- $Q = \{q_0, q_1, q_2, q_3\}$,
- $\Sigma = \{a, b\}$,
- $\delta(q_0, a) = q_1, \quad \delta(q_1, a) = q_2, \quad \delta(q_2, a) = \delta(q_3, a) = q_3$

  $\delta(q_i, b) = q_i \quad$ 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 Alphabetssymbol). Der Startzustand wird durch einen Pfeil gekennzeichnet und die Endzustände durch einen Doppelkreis.

Intuitiv arbeitet der Automat, indem er ein Wort Symbol für Symbol von links nach rechts liest und dabei entsprechend der Übergangsfunktion den Zustand wechselt. Er beginnt im Startzustand und akzeptiert das Eingabewort wenn er sich am Ende in einem Endzustand befindet. Wir beschreiben dieses Verhalten nun formal.

Definition 1.3 (kanonische Fortsetzung von δ)
Die *kanonische Fortsetzung von* $\delta : Q \times \Sigma \to Q$ *von einzelnen Symbolen auf beliebige Wörter*, also auf eine Funktion $\delta : Q \times \Sigma^* \to Q$, wird induktiv (über die Wortlänge) definiert:

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

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

Als Beispiel für Definition 1.3 betrachte wieder den Automat $\mathcal{A}$ aus Beispiel 1.6. Es gilt $\delta(q_0, bbbabbbb) = q_1$ und $\delta(q_0, baaab) = q_3$.

Definition 1.4 (Akzeptiertes Wort, erkannte Sprache)
Ein DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ *akzeptiert* das Wort $w \in \Sigma^*$ wenn $\delta(q_0, w) \in F$. 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 1.6 die Sprache

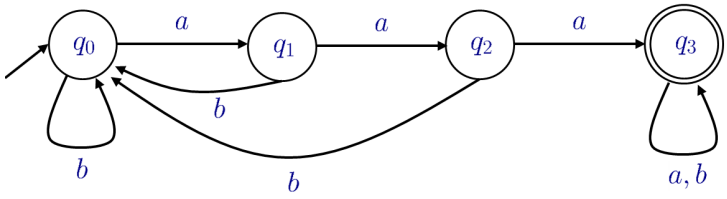
$$L(\mathcal{A}) = \{w \in \{a, b\}^* \mid \ |w|_a \geq 3\}$$

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

Definition 1.5 (Erkennbarkeit einer Sprache)
Eine Sprache $L \subseteq \Sigma^*$ heißt *erkennbar*, wenn es einen DEA $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

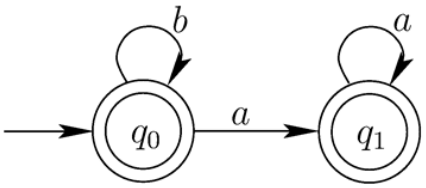
Wir haben also gerade gesehen, dass die Sprache $L = \{w \in \{a, b\}^* \mid \ |w|_a \geq 3\}$ erkennbar ist. Folgendes Beispiel liefert eine weitere erkennbare Sprache.

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.



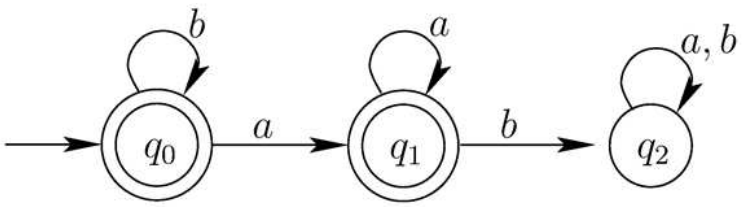
Beachte:
Die Übergangsfunktion eines DEAs ist eine *totale Funktion*, es muß also für jede mögliche Kombination von Zustand und Symbol ein Folgesymbol angegeben werden,

Beispiel 1.7
Folgendes ist 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* Endzustand ist):



Die im obigen Beispiel erkannte Sprache ist übrigens

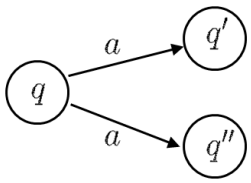
$$L = \{w \in \{a, b\}^* \mid ab \text{ ist nicht Infix von } w\}.$$

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) nicht als endlichen Automat indem man z.B. ausnutzt, dass der Arbeitsspeicher ganz genau 2GB 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 kennenlernen werden.

Von DEAs zu NEAs

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 Endzustand 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. Natürlich gibt es in der Realität keine nichtdeterministischen Maschinen. Dannoch 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 (Theoretische Informatik 2) 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 sogenannte "P vs. NP" Problem, das wichtigste ungelöste Problem der Informatik.

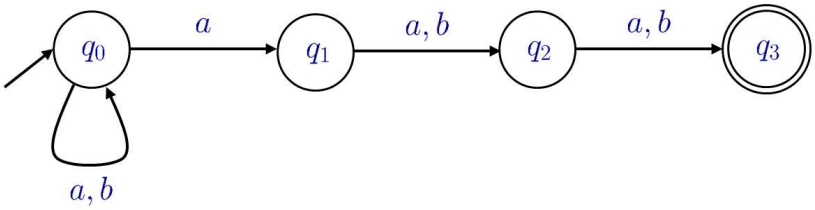
NEAs ergeben sich dadurch, dass man die Übergangs*funktion* von DEAs durch eine Übergangs*relation* ersetzt. Wir definieren DEAs der Vollständigkeit halber noch einmal als Ganzes.

Definition 1.8 (NEA)
Ein *Nichtdeterministischer endlicher Automat (NEA)* ist von der Form $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$, 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 Endzuständen 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

$$\pi = p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1 \xrightarrow{a_2}_{\mathcal{A}} p_2 \xrightarrow{a_3}_{\mathcal{A}} \cdots \xrightarrow{a_n}_{\mathcal{A}} p_n$$

so dass $(p_i, a_{i+1}, p_{i+1}) \in \Delta$ für $i = 0, \ldots, n-1$. Der Pfad 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*:

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

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 mindestens ein $q_f \in F$. 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 Endzustand endet. Dabei ist es irrelevant, dass der ebenfalls mögliche Pfad π_2 in einem nicht-Endzustand endet. Nicht akzeptiert wird beispielsweise die Eingabe *baa*, da keiner der möglich Pfade zu einem Endzustand 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 Endzustand nicht erreicht und der Automat akzeptiert 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 erkannten Sprachen interessieren, bezeichnen wir zwei NEAs als *äquivalent*, wenn sie dieselbe Sprache akzeptieren.

Ohne Nichtdeterminismus, also mittels eines DEA, ist es sehr viel schwieriger, die Sprache aus Beispiel 1.9 zu erkennen (Aufgabe!). 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 aber 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 folgendes klassische Resultat.

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

Effektiv bedeutet hier, dass es einen Algorithmus gibt, der als Eingabe einen NEA erhält uns als Ausgabe einen äqivalenten DEA liefert.

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 Endzustand enthält. Wir definieren also die Übergangsfunktion δ und Endzustandsmenge F' des DEAs so, dass für alle $w \in \Sigma^*$ gilt:

1. $\delta(\{q_0\}, w) = \{q \mid q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q\}$ und
2. $\{q \mid q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q\}$ ist DEA-Endzustand wenn mindestens ein Endzustand 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: $q' \in \delta(\{q_0\}, w)$ gdw. $q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q'$ $(\star)$

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

$$\begin{array}{lll} w \in L(\mathcal{A}) & \text{gdw.} & \exists q \in F : q_0 \overset{w}{\Longrightarrow}_{\mathcal{A}} q \qquad \text{(Def. } L(\mathcal{A})\text{)} \\ & \text{gdw.} & \exists q \in F : q \in \delta(\{q_0\}, w) \quad \text{(Hilfsaussage)} \\ & \text{gdw.} & \delta(\{q_0\}, w) \cap F \neq \emptyset \\ & \text{gdw.} & \delta(\{q_0\}, w) \in F' \qquad \text{(Def. } F'\text{)} \\ & \text{gdw.} & w \in L(\mathcal{A}') \end{array}$$

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

Induktionsanfang: $|w| = 0$

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

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

Induktionsschritt: $|w| = n + 1$

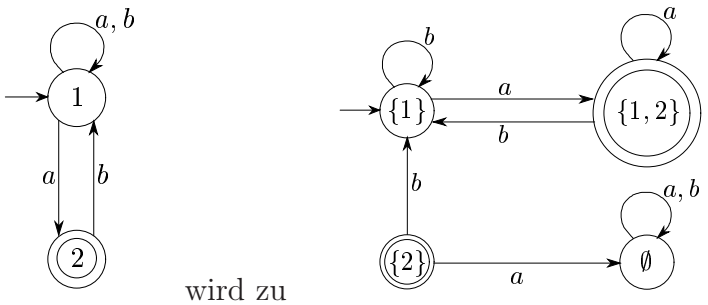
Sei $w = ua$ mit $u \in \Sigma^*$, $|u| = n$ und $a \in \Sigma$. Es gilt:

$$\begin{array}{rcll} \delta(\{q_0\}, ua) & = & \delta(\delta(\{q_0\}, u), a) & \text{(Def. 1.3)} \\ & = & \bigcup_{q' \in \delta(\{q_0\}, u)} \{q'' \mid (q', a, q'') \in \Delta\} & \text{(Def. } \delta\text{)} \\ & = & \bigcup_{q_0 \overset{u}{\Longrightarrow}_{\mathcal{A}} q'} \{q'' \mid (q', a, q'') \in \Delta\} & \text{(Ind.Voraus.)} \\ & = & \{q'' \mid q_0 \overset{ua}{\Longrightarrow}_{\mathcal{A}} q''\} & \text{(Def. Pfad)} \end{array}$$

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 *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 der Übung werden wir eine Methode kennenlernen, die Potenzmengenkonstruktion systematisch so anzuwenden, dass nicht erreichbare Zustände von vorn herein weggelassen werden. Dies reicht allerdings nicht aus, damit der erzeugte Automat so klein wie möglich ist!). Wir werden später eine allgemeine Methode kennenlernen, um zu einer gegebenen erkennbaren Sprachen den kleinstmöglichen DEA zu konstruieren.

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 wir 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 ε-Übergänge (im Unterschied zu $q \xrightarrow{a} p$).

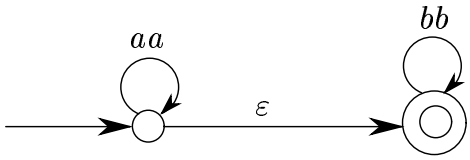
Satz 1.15
Zu jedem NEA mit Wortübergängen kann man effektiv einen äquivalenten NEA konstruieren.

Man zeigt Satz 1.15 mit Umweg über ε-NEAs.

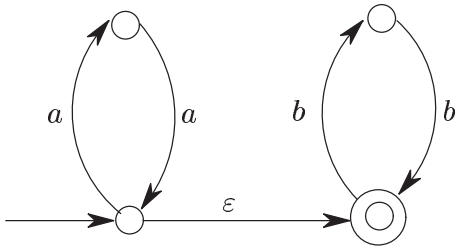
Lemma 1.16
Zu jedem NEA mit Wortübergängen kann man effektiv einen äquivalenten ε-NEA konstruieren.

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 Endzustandsmenge dazugenommen 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 effektiv 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

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

Noch zu zeigen: $L(\mathcal{A}) = L(\mathcal{A}')$ und $\mathcal{A}'$ kann effektiv bestimmt werden.

1. $L(\mathcal{A}') \subseteq L(\mathcal{A})$:
   Sei $w = a_1 \cdots a_n \in L(\mathcal{A}')$. Dann gibt es in $\mathcal{A}'$ 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 u.U. 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 $p_n = q_0 \xrightarrow{\varepsilon}_{\mathcal{A}} \hat{p}$ für ein $\hat{p} \in F$. Es gibt also in $\mathcal{A}$ einen Pfad von p_0 über q_0 nach $\hat{p} \in F$ mit Beschriftung w, daher $w \in L(\mathcal{A})$.

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

   $$\pi = p_0 \xrightarrow{a_1}_{\mathcal{A}} p_1 \xrightarrow{a_2}_{\mathcal{A}} \cdots \xrightarrow{a_{m-1}}_{\mathcal{A}} p_{m-1} \xrightarrow{a_m}_{\mathcal{A}} p_m$$

   Pfad in $\mathcal{A}$ mit $p_0 = q_0$, $p_m \in F$ und Beschriftung w, wobei $a_i \in \Sigma \cup \{\varepsilon\}$. Seien $i_1, \ldots, i_n$ die Indizes mit $a_{i_j} \neq \varepsilon$. Dann ist $w = a_{i_1} \cdots a_{i_n}$.

   1. Fall: $n > 0$, also $w \neq \varepsilon$
   Nach Definition von Δ' ist

   $$p_0 \xrightarrow{a_{i_1}}_{\mathcal{A}} p_{i_1} \xrightarrow{a_{i_2}}_{\mathcal{A}} \cdots \xrightarrow{a_{i_{n-1}}}_{\mathcal{A}} p_{i_{n-1}} \xrightarrow{a_{i_n}}_{\mathcal{A}} p_{i_n}$$

   ein nicht-leerer Pfad in $\mathcal{A}'$. Aus $p_m \in F$ folgt $p_{i_n} \in F'$, was $w \in L(\mathcal{A}')$ zeigt.

   2. Fall: $n = 0$, also $w = \varepsilon$
   Dann ist $a_1 = \cdots = a_m = \varepsilon$. Es gilt also $q_0 = p_0 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p_m \in F$, was $q_0 \in F'$ liefert. Also $w = \varepsilon \in L(\mathcal{A}')$.

3. Δ' und F' können effektiv bestimmt werden:

   - $p \overset{a}{\Longrightarrow}_{\mathcal{A}} q$ gilt genau dann, wenn es Zustände $p', q' \in Q$ gibt, für die gilt:

     $$p \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p', \quad (p', a, q') \in \Delta, \quad q' \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} q$$

     Man muss nur endlich viele p', q' prüfen. Ob „$(p', a, q') \in \Delta$?“ kann effektiv geprüft werden, da Δ endliche Menge. Weiterhin sind „$p \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} p'$?“ sowie „$q' \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} q$?“ Erreichbarkeitsprobleme in dem endlichen Graphen $G = (V, E)$ mit $V = Q$ und

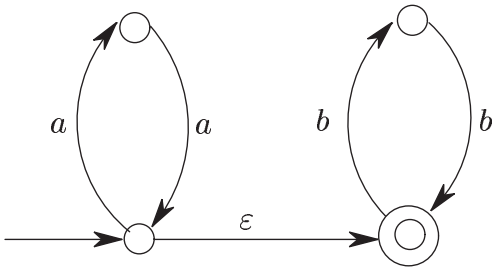
     $$E = \{(u, v) \mid (u, \varepsilon, v) \in \Delta\},$$

     können also effektiv entschieden werden (siehe Appendix A).

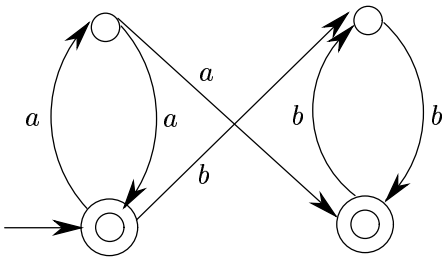
   - „$q_0 \overset{\varepsilon}{\Longrightarrow}_{\mathcal{A}} F$?“ ist ebenfalls Erreichbarkeitsproblem.

□

Beispiel: (zu Lemma 1.18)
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 allgemeine Eigenschaften, die von jeder erkennbaren Sprache erfüllt werden. 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, das in verschiedenen Versionen existiert.

Lemma 2.1 (Pumping-Lemma, einfache Version)
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

- $y \neq \varepsilon$
- $xy^k z \in L$ *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

$$(p_0, a_1, p_1)(p_1, a_2, p_2) \ldots (p_{m-1}, a_m, p_m)$$

in $\mathcal{A}$ mit $p_0 = q_0$ und $p_m \in F$. Wegen $m \geq n_0 = |Q|$ können die $m + 1$ Zustände $p_0, \ldots, p_m$ nicht alle verschieden sein. Es gibt also ein $i < j$ mit $p_i = p_j$. Für

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

gilt daher $y \neq \varepsilon$ (da $i < j$) und

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

Folglich gilt für alle $k \geq 0$ auch $p_i \overset{y^k}{\Longrightarrow}_{\mathcal{A}} p_i$, was $xy^k z \in L$ zeigt. □

Wir zeigen mit Hilfe dieses Lemmas, dass die Sprache $\{a^n b^n \mid n \geq 0\}$ nicht erkennbar ist.

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

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, L sei erkennbar. Es gibt also eine Zahl n_0 mit den in Lemma 2.1 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 $xy^kz \in L$ für alle $k \geq 0$.

1. Fall: y liegt ganz in a^{n_0}.
D.h. $x = a^{k_1},\ y = a^{k_2},\ z = a^{k_3}b^{n_0}$ mit $k_2 > 0$ und $n_0 = k_1 + k_2 + k_3$. Damit ist aber $xy^0z = xz = a^{k_1+k_3}b^{n_0} \notin L$, da $k_1 + k_3 < n_0$. Widerspruch.

2. Fall: y liegt ganz in b^{n_0}.
Führt entsprechend zu einem Widerspruch.

3. Fall: y enthält as und bs.
Dann ist xy^2z von der Form $a^{k_1}b^{k_2}a^{k_3}b^{k_4}$ wobei alle $k_i > 0$, also $xy^2z \notin L$. Widerspruch.

In allen drei Fällen haben wir also einen Widerspruch erhalten, d.h. die Annahme „L ist erkennbar“ war falsch. □

Als eine weitere Konsequenz von Lemma 2.1 erhält man, dass das Leerheitsproblem für erkennbare Sprachen entscheidbar ist.

Satz 2.2
Sei L eine erkennbare Sprache, gegeben durch NEA oder DEA. Dann kann man effektiv entscheiden, ob $L = \emptyset$.

Beweis. Sei L erkennbar und n_0 die zugehörige Zahl aus Lemma 2.1. Dann gilt:

$$(\star) \qquad L \neq \emptyset \ \ \underline{\text{gdw}}. \ \ \exists w \in L \text{ mit } |w| < n_0$$

„$\Leftarrow$“: trivial

„$\Rightarrow$“: Es sei $L \neq \emptyset$. Wähle ein Wort w kürzester Länge in L. Wäre $|w| \geq n_0$, so müsste es eine Zerlegung $w = xyz,\ y \neq \varepsilon$ geben mit $xy^0z = xz \in L$. Dies ist ein Widerspruch zur Minimalität von w. □

Um $L = \emptyset$ zu entscheiden, muss man also nur für die *endlich vielen* Wörter $w \in \Sigma^*$ der Länge $< n_0$ entscheiden, ob w zu L gehört. Dies kann man für jedes einzelne Wort (z.B. durch Eingabe in den zugehörigen DEA) einfach entscheiden (vgl. Wortproblem, Satz 3.4).

Mit Hilfe der einfachen Variante 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 2.1 beschriebene Pumping-Eigenschaft erfüllen. Anders ausgedrückt ist die Pumping-Eigenschaft aus Lemma 2.1 zwar *notwendig* für die Erkennbarkeit einer Sprache, aber nicht *hinreichend*.

Beispiel 2.3
Ist $L = \{a^n b^m \mid n \neq m\}$ erkennbar? Versucht man, Nichterkennbarkeit mit Lemma 2.1 zu zeigen, so scheitert man, da das Lemma für L zutrifft:
Wähle $n_0 := 3$. Es sei nun $w \in L$ mit $|w| \geq 3$, d.h. $w = a^n b^m$, $n \neq m$ und $n + m \geq 3$. Wir zeigen: w läßt sich zerlegen in $w = xyz$ mit $y \neq \varepsilon$ und $xy^k z \in L$ für alle $k \geq 0$.

1. Fall: $n > m$ (es gibt mehr as als bs)

1.1.: $n = m + 1$ (es gibt genau ein a mehr als bs)
Wegen $|w| \geq 3$ kann man dann w zerlegen in $x = \varepsilon$, $y = a^2$, $z = a^{n-2}b^m$. Es gilt:

a) $xy^0 z = a^{n-2}b^m$ hat ein a weniger als bs, ist also in L

b) $xy^k z = a^{(n-2)+2k}b^m$ hat mehr as also b für alle $k \geq 1$, ist also in L

1.2.: $n > m + 1$ (es gibt mind. 2 as mehr als bs)
Zerlege w in $x = \varepsilon$, $y = a$, $z = a^{n-1}b^m$. Dann hat jedes $xy^k z = a^{(n-1)+k}b^m$ mehr as als bs, ist also in L.

2. Fall: $n < m$ (es gibt mehr bs als as)
Symmetrisch zum 1. Fall.

Trotzdem ist $L = \{a^n b^m \mid n \neq m\}$ nicht erkennbar, was man mit der folgenden verschärften Variante des Pumping-Lemmas nachweisen kann.

Lemma 2.4 (Pumping-Lemma, verschärfte Variante)
Es sei L erkennbar. Dann gibt es eine natürliche Zahl $n_0 \geq 1$, so dass gilt:
Für alle Wörter $u, v, w \in \Sigma^*$ mit $uvw \in L$ und $|v| \geq n_0$ gibt es eine Zerlegung $v = xyz$ mit

- $y \neq \varepsilon$
- $uxy^k zw \in L$ *für alle* $k \geq 0$

Beweis. Es sei wieder $n_0 := |Q|$, wobei Q die Zustände eines NEA $\mathcal{A}$ für L sind. Ist $uvw \in L$, so gibt es Zustände $p, q, f \in Q$ mit

$$q_0 \overset{u}{\Longrightarrow}_{\mathcal{A}} p \overset{v}{\Longrightarrow}_{\mathcal{A}} q \overset{w}{\Longrightarrow}_{\mathcal{A}} f \in F$$

Auf dem Pfad von p nach q liegen $|v| + 1 > n_0$ Zustände, also müssen zwei davon gleich sein. Jetzt kann man wie im Beweis von Lemma 2.1 weitermachen. □

Im Vergleich mit Lemma 2.1 macht dieses Lemma eine stärkere Aussage: es ist nicht nur so, dass man jedes Wort w mit $|w| \geq n_0$ in drei Teile zerlegen und dann „pumpen“ kann, sondern das gilt sogar für *jedes Teilwort* v von w mit $|v| \geq n_0$. Beim Nachweis der Nichterkennbarkeit hat das den Vorteil, dass man das Teilwort v frei wählen kann, so wie es zum Herstellen eines Widerspruchs am bequemsten ist.

Beispiel 2.3 (Fortsetzung)
$L = \{a^n b^m \mid n \neq m\}$ ist *nicht* erkennbar.

Beweis. Angenommen, L ist doch erkennbar; dann gibt es $n_0 \geq 1$, das die in Lemma 2.4 geforderten Eigenschaften hat. Wähle des Wort

$$a^{n_0} b^{n_0!+n_0} \in L \text{ und die Zerlegung } u := \varepsilon,\ v := a^{n_0},\ w := b^{n_0!+n_0}$$

Dann gibt es eine Zerlegung $v = xyz$ mit $y \neq \varepsilon$ und $uxy^k zw \in L$ für alle $k \geq 0$. Sei

$$x = a^{k_1},\ y = a^{k_2},\ z = a^{k_3},\ k_1 + k_2 + k_3 = n_0,\ k_2 > 0$$

Da $0 < k_2 \leq n_0$ gibt es ein ℓ mit $k_2 \cdot \ell = n_0!$. Betrachte das Wort $uxy^{\ell+1}zw$, welches in L sein müsste. Die Anzahl von as ist

$$k_1 + (\ell + 1) \cdot k_2 + k_3 = k_1 + k_2 + k_3 + (\ell \cdot k_2) = n_0 + n_0!$$

und die Anzahl von bs (welche nur im Teilwort w auftreten) ebenso, also gilt $uxy^{\ell+1}zw \notin L$. Widerspruch. □

Auch Lemma 2.4 formuliert nur eine notwendige Eigenschaft für die Erkennbarkeit einer Sprache, aber keine hinreichende. In der Literatur findet man noch verschärftere (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 und Entscheidungsprobleme

Die Klasse der erkennbaren Sprachen ist unter den meisten natürlichen Operationen abgeschlossen: wenn man eine solche Operation auf eine (oder mehrere) erkennbare Sprachen anwendet, so erhält man wieder eine erkennbare Sprache. Diese Eigenschaft ist für viele verschiedene Zwecke sehr nützlich, von denen wir einige exemplarisch kennenlernen werden. 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

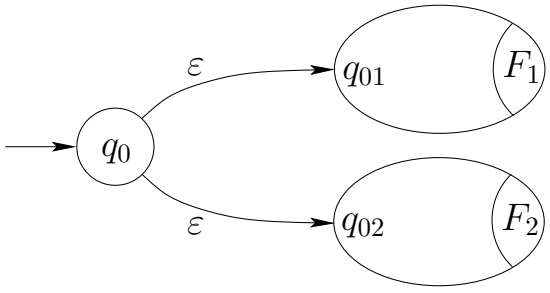
- $L_1 \cup L_2$ *(Vereinigung)*
- $\overline{L_1}$ *(Komplement)*
- $L_1 \cap L_2$ *(Durchschnitt)*
- $L_1 \setminus L_2$ *(Differenz)*
- $L_1 \cdot L_2$ *(Konkatenation)*
- L_1^* *(Kleene-Stern)*

erkennbar.

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 Endzustände mit den Nicht-Endzuständen:

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

Es gilt nämlich:

$$\begin{array}{lll} w \in \overline{L_1} & \underline{\text{gdw}}. & w \notin L(\mathcal{A}_1) \\ & \underline{\text{gdw}}. & w \notin L(\mathcal{A}) \\ & \underline{\text{gdw}}. & \delta(q_0, w) \notin F \\ & \underline{\text{gdw}}. & \delta(q_0, w) \in Q \setminus F \\ & \underline{\text{gdw}}. & w \in L(\overline{\mathcal{A}}) \end{array}$$

Beachte: Diese Konstruktion funktioniert nicht für NEAs.

3) **Abschluss unter Durchschnitt:**
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 aufwendig ist und exponentiell große Automaten liefert, kann es günstiger sein, direkt einen NEA für $L_1 \cap L_2$ zu konstruieren, den sogenannten *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 $w \in L(\mathcal{A})$ $\underline{\text{gdw}}.$ es gibt einen Pfad

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

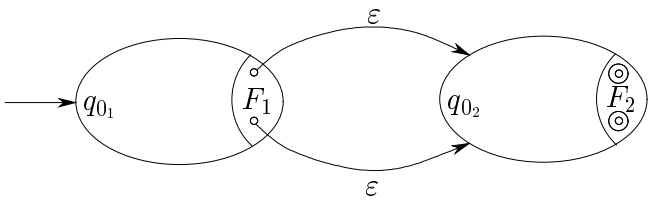
mit $(q_{1,0}, q_{2,0}) = (q_{01}, q_{02})$ und $(q_{1,n}, q_{2,n}) \in F_1 \times F_2$. Nach Konstruktion von $\mathcal{A}$ ist das der Fall $\underline{\text{gdw}}.$ für jedes $i \in \{1, 2\}$

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

ein Pfad ist mit q_{0i} und $q_{i,n} \in F_i$. Das ist der Fall $\underline{\text{gdw}}.$ $w \in L_1 \cap L_2$.

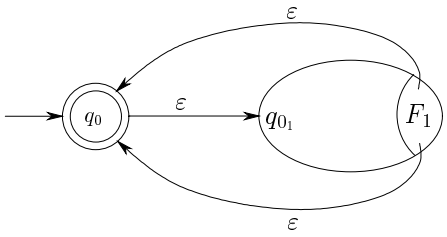
4) **Abschluss unter Differenz:**
Folgt aus 1) und 2), da $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$.

5) **Abschluss unter Konkatenation:**
Der folgende ε-NEA erkennt $L_1 \cdot L_2$:
$\mathcal{A} := (Q_1 \cup Q_2, \Sigma, q_{01}, \Delta, F_2)$, wobei
$\Delta := \Delta_1 \cup \Delta_2 \cup \{(f, \varepsilon, q_{02}) \mid f \in F_1\}$



6) **Abschluss unter Kleene-Stern:**
Der folgende ε-NEA erkennt L_1^*:
$\mathcal{A} := (Q_1 \cup \{q_0\}, \Sigma, q_0, \Delta, \{q_0\})$, wobei

- $q_0 \notin Q_1$
- $\Delta := \Delta_1 \cup \{(f, \varepsilon, q_0) \mid f \in F_1\} \cup \{(q_0, \varepsilon, q_{01})\}$.



□

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

Beachte:

Alle angegebenen Konstruktionen sind effektiv. Die Automaten für die Sprachen $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$ und L_1^* sind polynomiell in der Größe der Automaten für L_1, L_2. Beim Komplement kann der konstruierte Automat exponentiell groß sein, wenn man mit einem NEA beginnt.

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

Beispiel 3.2

$L := \{a^n b^m \mid n \neq m\}$ ist *nicht* erkennbar (vgl. Beispiel 2.3). Anstatt dies direkt mit Lemma 2.4 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.

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—formal werden wir diesen Begriff erst in Teil III einführen. Die drei betrachteten Probleme sind:

- das *Leerheitsproblem*
- das *Wortproblem* und
- das *Äquivalenzproblem.*

Wir werden jeweils *Entscheidbarkeit* und *Komplexität* untersuchen:

- Ein Problem ist entscheidbar, wenn es einen Algorithmus gibt, der das Problem löst. Wie wir in Teil III sehen werden, gibt es wohldefinierte (und praktisch relevante) Probleme, die nicht entscheidbar sind.
- Bei der Komplexität eines Problemes geht es um den minimalen Ressourcenverbrauch (insbesondere Laufzeit und Spreicherverbrauch), von Algorithmen, die das Probem lösen. Wir werden uns hier auf eine Laufzeitanalyse der betrachteten Algorithmen beschränken und das Thema Komplexität in Teil IV detailliert behandeln.

Bei den hier behandelten Problemen besteht die Eingabe aus einem DEA oder einem NEA. Bezüglich der Entscheidbarkeit macht das keinen Unterschied, da man aus einem NEA ja effektiv einen äquivalenten DEA konstruieren kann (Satz 1.12). Bezüglich der Komplexität kann es aber sehr wohl einen Unterschied geben, da der Übergang NEA $\longrightarrow$ DEA exponentiell sein kann.

Leerheitsproblem:

Geg.: erkennbare Sprache L (durch DEA oder NEA)
Frage: Ist $L \neq \emptyset$?

Wir wissen bereits (Satz 2.2), dass das Problem entscheidbar ist. Allerdings ist das im Beweis des Satzes beschriebene Entscheidungsverfahren viel zu aufwendig (*exponentiell* viele Wörter der Länge $< n_0$, falls $|\Sigma| > 1$).

Satz 3.3

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

Beweis. Man kann $\mathcal{A}$ als gerichteten Graphen $G = (Q, E)$ auffassen mit

$$E := \{(q_1, q_2) \mid (q_1, a, q_2) \in \Delta \text{ für ein } a \in \Sigma\}$$

Dann gilt: $L(\mathcal{A}) \neq \emptyset$ gdw. in der von q_0 aus erreichbaren Knotenmenge befindet sich ein Endzustand. Die von q_0 aus erreichbaren Knoten kann man mit Aufwand $O(|Q| + |E|)$ berechnen (siehe Appendix A). Wenn die Endzustandsmenge F geschickt repräsentiert ist,[1] kann man in Zeit $O(|Q|)$ prüfen, ob die Menge der erreichbaren Zustände einen

[1] Eine elegante Möglichkeit ist, die Zustände so zu benennen, dass man anhand des Zustandsnamens einfach entscheiden kann, ob der Zustand ein Endzustand ist oder nicht—beispielsweise könnten Endzustände durch positive Zahlen repräsentiert werden und Nicht-Endzustände durch negative.

Endzustand enthält. Damit ist das Leerheitsproblem in Zeit $O(|Q| + |E|)$ entscheidbar. □

Wortproblem:
Geg.: erkennbare Sprache L, Wort $w \in \Sigma^*$
Frage: Gilt $w \in L$?

Ist $L = L(\mathcal{A})$ für einen DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, so kann man einfach, beginnend mit q_0, durch Anwendung von δ berechnen, zu welchem Zustand man in $\mathcal{A}$ mit w kommt und prüfen, ob dies ein Endzustand ist. Man muss δ offensichtlich $|w|$ mal anwenden und jede Anwendung benötigt $|\delta| = |Q| \cdot |\Sigma|$ Schritte (Durchsuchen von δ nach dem richtigen Übergang). Diese Laufzeit lässt sich durch geschickte Repräsentation von δ weiter verbessern, aber selbst diese grobe Analyse liefert:

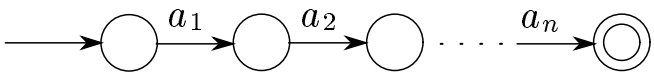
Satz 3.4
Das Wortproblem für DEAs ist in Zeit $O(|w| \cdot |\delta|)$ entscheidbar.

Für einen NEA ist dieser trivile Algorithmus nicht möglich, da es ja mehrere mit w beschriftete Pfade geben kann und man (im schlimmsten Fall) alle betrachten muss, um festzustellen, ob einer davon mit einem Endzustand aufhört. Das würde zu exponentieller Laufzeit führen. Das folgende Resultat zeigt, dass es auch (viel) besser geht.

Satz 3.5
Das Wortproblem für NEAs ist in Zeit $O(|w| \cdot |\delta|)$ entscheidbar.

Wir verwenden eine *Reduktion* des Wortproblems auf das Leerheitsproblem: der schon gefundene Algorithmus für das Leerheitsproblem wird verwendet, um das Wortproblem zu lösen (mehr zu Reduktionen findet sich in den Teilen III+IV).

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



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

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

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 3.3 prüfen, ob dieser eine nicht-leere Sprache erkennt.

Wir analysieren zunächst die Größe des Produktautomaten:

Zustände: $|Q| \cdot (|w| + 1)$

Übergänge: Da es in $\mathcal{A}_w$ genau $|w|$ viele Übergänge gibt, ist die Zahl Übergänge des Produktautomaten durch $|w| \cdot |\Delta|$ beschränkt.

Nach Satz 3.3 ist daher der Aufwand zum Testen von $L(\mathcal{A}) \cap L(\mathcal{A}_w) \neq \emptyset$ also:

$$O(|Q| \cdot (|w| + 1) + |w| \cdot |\Delta|) = O(|w| \cdot (|Q| + |\Delta|)) \qquad \square$$

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

$$2 \cdot O(|w| \cdot (|Q| + |\Delta|)) = O(|w| \cdot (|Q| + |\Delta|)).$$

Äquivalenzproblem:
Geg.: erkennbare Sprachen L_1, L_2
Frage: Gilt $L_1 = L_2$?

Wir verwenden wieder eine Reduktion auf das Leerheitsproblem:

$$L_1 = L_2 \quad \underline{\text{gdw}}. \quad (L_1 \setminus L_2) \cup (L_1 \setminus L_2) = \emptyset$$

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 3.6
Das Äquivalenzproblem für DEAs ist in polynomieller Zeit entscheidbar. Für NEAs ist es in exponentieller Zeit entscheidbar.

Beweis. Wir haben gesehen, dass die Automatenkonstruktionen, welche Abschluss unter Vereinigung und Differenz zeigen, effektiv sind. Daraus ergibt sich direkt die Entscheidbarkeit des Äquivalenzproblems für DEAs und NEAs. Die Konstruktion für Vereinigung ist sowohl für DEAs als auch für NEAs polynomiell. Bei der Differenz ist dies nur dann der Fall, wenn bereits DEAs vorliegen. Bei NEAs muss die Potenzmengenkonstruktion angewendet werden und der auf Leerheit zu testende Automat kann exponentiell groß sein. Damit ergibt sich exponentielle Laufzeit. $\square$

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.

4. 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 NEA $\mathcal{A}$.

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

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

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

Im folgenden betrachten wir eine weitere Charakterisierung mit Hilfe *regulärer Ausdrücke*. Diese Stellen eine natürliche „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 4.1 (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 $r, s \in Reg_\Sigma$, so auch $(r + s), (r \cdot s), r^* \in Reg_\Sigma$.

Beispiel 4.2
$((a \cdot b^*) + \emptyset^*)^* \in Reg_\Sigma$ für $\Sigma = \{a, b\}$

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 Ausdruck aus Beispiel 4.2 kann also geschrieben werden als $(ab^* + \emptyset^*)^*$.

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

Definition 4.3 (Semantik regulärer Ausdrücke)
Die durch den regulären Ausdruck r definierte Sprache $L(r)$ ist induktiv definiert:

- $L(\emptyset) := \emptyset, \quad L(\varepsilon) := \{\varepsilon\}, \quad L(a) := \{a\}$
- $L(r + s) := L(r) \cup L(s), \quad L(r \cdot s) := L(r) \cdot L(s), \quad L(r^*) := L(r)^*$

Eine Sprache $L \subseteq \Sigma^*$ heißt *regulär*, falls es ein $r \in Reg_\Sigma$ gibt mit $L = L(r)$.

Beispiel:

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

Bemerkung:

Statt $L(r)$ schreiben wir im folgenden häufig einfach r.
Dies ermöglicht es und 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 4.4 (Kleene)

Für eine Sprache $L \subseteq \Sigma^*$ sind äquivalent:

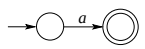
1) L ist regulär.

2) L ist erkennbar.

Beweis.

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

Anfang:

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

Schritt: Weiß man bereits, dass $L(r)$ und $L(s)$ erkennbar sind, so folgt mit Satz 3.1 (Abschlusseigenschaften), dass auch

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

erkennbar sind.

„$2 \to 1$“: 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.

Anfang: $X = \emptyset$.

- 1. Fall: $u \neq v$
  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: $u = v$
  Wie voriger Fall, ausser dass $L^{\emptyset}_{p,q}$ (und damit auch der konstruierte reguläre Ausdruck) zusätzlich ε enthält.

Schritt: $X \neq \emptyset$.

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

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

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.

$\subseteq$ 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 $w \in L^{X-\{\widehat{q}\}}_{p,q}$. Andernfalls seien $i_1, \ldots, i_k$ alle Indizes mit $p_{i_j} = \widehat{q}$ (und $i_1 < \cdots < i_k$). Offensichtlich gilt:

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

$\supseteq$ Wenn $w \in L^{X-\{\widehat{q}\}}_{p,q}$, dann offenbar $w \in L^X_{p,q}$. Wenn

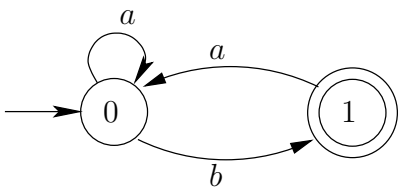
$$w \in \left(L^{X-\{\widehat{q}\}}_{p,\widehat{q}} \cdot \left(L^{X-\{\widehat{q}\}}_{\widehat{q},\widehat{q}}\right)^* \cdot L^{X-\{\widehat{q}\}}_{\widehat{q},q} \right),$$

dann $w = xyz$ mit $x \in L^{X-\{\widehat{q}\}}_{p,\widehat{q}}$, $y \in (L^{X-\{\widehat{q}\}}_{\widehat{q},\widehat{q}})^*$, und $z \in L^{X-\{\widehat{q}\}}_{\widehat{q},q}$. 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 $w \in L^X_{p,q}$.

Wenn man die Konstruktion aus „$2 \to 1$“ 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 4.5

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



Da 1 der einzige Endzustand 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 „$2 \to 1$“ 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 4.4 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 $L(t') = \Sigma^* \setminus L(r)$.

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

5. Minimale DEAs und die Nerode-Rechtskongruenz

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 5.1 (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 $\delta(q_0, w) = q$. Sonst heißt q *unerreichbar*.

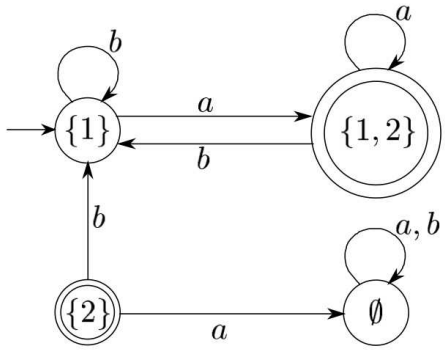
Da für die erkannte Sprache nur Zustände wichtig sind, welche von q_0 erreicht werden, erhält man durch Weglassen unerreichbarer Zustände einen äquivalenten Automaten:

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

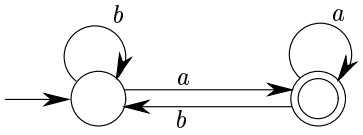
- $Q_0 = \{q \in Q \mid q \text{ ist erreichbar}\}$
- $\delta_0 = \delta \mid_{Q_0 \times \Sigma}$ Beachte: Für $q \in Q_0$ und $a \in \Sigma$ ist auch $\delta(q, a) \in Q_0$!
- $F_0 = F \cap Q_0$

Beispiel.

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 $\mathcal{A}'_0$:

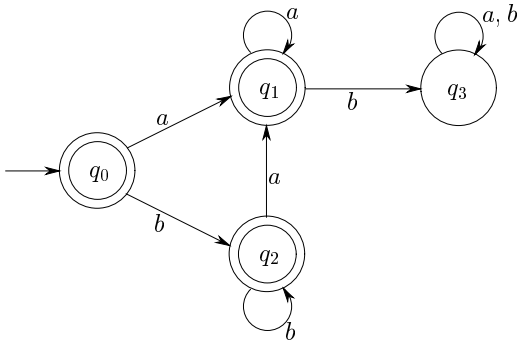


2. Schritt: Zusammenfassen äquivalenter Zustände

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 5.2

Im folgenden DEA 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 5.3 (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. $L(\mathcal{A}_q) = L(\mathcal{A}_{q'})$.

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 5.4

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.

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

Beweis.

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

2) lässt sich wie folgt herleiten:

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

□

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

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

Lemma 5.5
$\sim_{\mathcal{A}}$ kann effektiv berechnet werden.

Beweis. Wir definieren eine Folge von Relationen $\sim_0, \sim_1, \sim_2, \ldots$:

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

Diese sind (Über-)Approximationen von $\sim_{\mathcal{A}}$ im folgenden Sinn. Für alle $k \geq 0$ gilt:

$(*)$ $q \sim_k q' \quad \underline{\text{gdw}}.$ für alle $w \in \Sigma^*$ mit $|w| \leq n$: $w \in L(\mathcal{A}_q) \Leftrightarrow w \in L(\mathcal{A}_{q'})$.

Der Beweis von $(*)$ ist per Induktion über k:

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

Schritt:

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

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 $(*)$ 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 $(*)$ 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 5.2 (Fortsetzung)
Für den Automaten aus Beispiel 5.2 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 $\delta(q_1, b) \notin F$.
- $\sim_2 = \sim_1 = \sim_{\mathcal{A}}$.

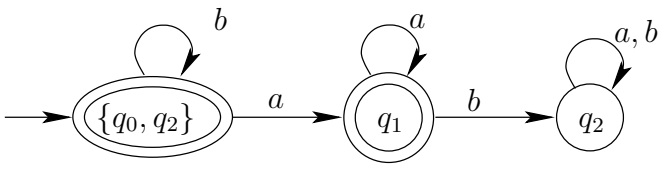
Die nachfolgende Konstruktion zeigt, wie äquivalente Zustände zusammengefasst werden können: da alle Zustände in einer Äquivalenzklasse $[q]_{\mathcal{A}}$ paarweise $\mathcal{A}$-äquivalent sind und Zustände aus verschiedenen Äquivalenzklassen niemals $\mathcal{A}$-äquivalent sein können, verwendet man die Äquivalenzklassen selbst als Zustände. Jede solche Klasse verhält sich dann genau wie die in ihr enthaltenen Zustände.

Definition 5.6 (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 5.4)
- $\widetilde{F} := \{[q]_{\mathcal{A}} \mid q \in F\}$

Nach Lemma 5.5 kann der Quotientenautomat effektiv konstruiert werden.

Beispiel 5.2 (Fortsetzung)
Für den Automaten aus Beispiel 5.2 ergibt sich der folgende Quotientenautomat:



Lemma 5.7
$\widetilde{\mathcal{A}}$ *ist äquivalent zu* $\mathcal{A}$.

Beweis. Es ist einfach, per Induktion über $|w|$ zu zeigen, dass die Erweiterung von $\widetilde{\delta}$ auf Wörter sich wie folgt verhält:

$$\widetilde{\delta}([q_0], w) \in \widetilde{F} \quad \underline{\text{gdw}}. \quad [\delta(q_0, w)]_{\mathcal{A}} \in \widetilde{F} \qquad (*)$$

Nun gilt:

$$\begin{array}{lll} w \in L(\mathcal{A}) & \underline{\text{gdw}}. & \delta(q_0, w) \in F \\ & \underline{\text{gdw}}. & [\delta(q_0, w)]_{\mathcal{A}} \in \widetilde{F} \qquad (\text{Def. } \widetilde{F}) \\ & \underline{\text{gdw}}. & \widetilde{\delta}([q_0]_{\mathcal{A}}, w) \in \widetilde{F} \qquad (*) \\ & \underline{\text{gdw}}. & w \in L(\widetilde{\mathcal{A}}) \end{array}$$

□

Die folgende Definition fasst die beiden Minimierungs-Schritte zusammen:

Definition 5.8 (reduzierter Automat zu einem DEA)
Für einen DEA $\mathcal{A}$ bezeichnet $\mathcal{A}_{red} := \widetilde{\mathcal{A}}_0$ den *reduzierten Automaten*, den man aus $\mathcal{A}$ durch Eliminieren unerreichbarer Zustände und nachfolgendes Zusammenfassen äquivalenter Zustände erhält.

Wir wollen zeigen, dass der reduzierte Automat nicht weiter vereinfacht werden kann: $\mathcal{A}_{red}$ 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 sogenannte Nerode-Rechtskongruenz.

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

Man beachte, dass das Wort w in Definition 5.9 auch gleich ε sein kann. Darum folgt aus $u \simeq_L v$, dass $u \in L \Leftrightarrow v \in L$.

Beispiel 5.10
Wir betrachten die Sprache

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

(vgl. Beispiele 1.7, 5.2)

- Es gilt:
  $\varepsilon \simeq_L b$: $\quad \forall w : \varepsilon w \in L \quad \underline{\text{gdw}}. \quad w \in L$
  $\underline{\text{gdw}}. \quad w$ enthält ab nicht
  $\underline{\text{gdw}}. \quad bw$ enthält ab nicht
  $\underline{\text{gdw}}. \quad bw \in L$
- $\varepsilon \not\simeq_L a$: $\quad \varepsilon b \in L$, aber $a \cdot b \notin L$

Wie 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* eine Äquivalenzrelation die Anzahl ihrer Klassen.

Lemma 5.11 (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 Zustände $\geq$ 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:
   $(\star)$ $\quad uww' \in L \Leftrightarrow vww' \in L$
   Wegen $ww' \in \Sigma^*$ folgt $(\star)$ aus $u \simeq_L v$.

3) Zeige $L = \bigcup_{u \in L} [u]_L$.

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

   „$\supseteq$“: Sei $u \in L$ und $v \in [u]_L$.
   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: $\delta(q_0, u) = \delta(q_0, v)$ impliziert $u \simeq_L v$:

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

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

□

Beispiel 5.10 (Fortsetzung)
$\simeq_L$ hat drei Klassen:

- $[\varepsilon]_L = \{b\}^*$
- $[a]_L = \{b\}^* \cdot \{a\}^+$
- $[ab]_L = \Sigma^* \cdot \{ab\} \cdot \Sigma^*$

Die vielleicht interessanteste 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 natürlich nur auf Sprachen L anwenden, so dass $\simeq_L$ nur endlich viele Äquivalenzklassen hat.

Definition 5.12 (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:

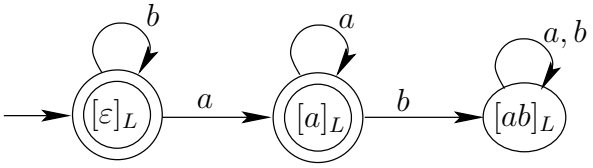
- $Q' := \{[u]_L \mid u \in \Sigma^*\}$
- $q'_0 := [\varepsilon]_L$
- $\delta'([u]_L, a) := [ua]_L$ (repräsentantenunabhängig wegen Lemma 5.11, Punkt 2)
- $F' := \{[u]_L \mid u \in L\}$

Man beachte, dass $\mathcal{A}_L$ mit Punkt 4 von Lemma 5.11 eine minimale Anzahl von Zuständen hat: es gibt keinen DEA, der $L(\mathcal{A}_L)$ erkennt und weniger Zustände hat.

Beispiel 5.10 (Fortsetzung)
Für die Sprache

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

ergibt sich damit folgender kanonischer Automat $\mathcal{A}_L$:



Lemma 5.13
Hat $\simeq_L$ endlichen Index, so ist $\mathcal{A}_L$ ein DEA mit $L = L(\mathcal{A}_L)$.

Beweis. Es gilt:

$$\begin{array}{lcll} L(\mathcal{A}_L) & = & \{u \mid \delta'(q'_0, u) \in F'\} & \\ & = & \{u \mid \delta'([\varepsilon]_L, u) \in F'\} & \text{(Def. } q'_0\text{)} \\ & = & \{u \mid [u]_L \in F'\} & \text{(wegen } \delta'([\varepsilon]_L, u) = [u]_L\text{)} \\ & = & \{u \mid u \in L\} & \text{(Def. } F'\text{)} \\ & = & L & \end{array}$$

□

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

Beweis.

„⇒“: Ergibt sich unmittelbar aus Lemma 5.11, 4).

„⇐“: Ergibt sich unmittelbar aus Lemma 5.13, da $\mathcal{A}_L$ DEA ist, der L erkennt.

□

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

Beispiel 5.15 (nichterkennbare 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

Satz 5.16 (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 $\mathcal{A}_{red}$.

Beweis. Sei $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ und $\mathcal{A}_{red} = (\widetilde{Q}, \Sigma, [q_0]_\mathcal{A}, \widetilde{\delta}, \widetilde{F})$. Wir definieren eine injektive Abbildung π, die jedem Zustand aus $\widetilde{Q}$ eine Äquivalenzklasse von $\simeq_L$ zuordnet. Es folgt, dass $\mathcal{A}_{red}$ *höchstens* so viele Zustände hat, wie $\simeq_L$ Äquivalenzklassen (Schubfachprinzip), also ist er nach Punkt 4 von Lemma 5.11 minimal.

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

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

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

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

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

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. Formal wird das durch den Begriff der Isomorphie beschrieben.

Definition 5.17 (isomorph)
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 5.18
$\mathcal{A} \simeq \mathcal{A}' \;\Rightarrow\; L(\mathcal{A}) = L(\mathcal{A}')$

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

$$\begin{array}{lll} w \in L(\mathcal{A}) & \underline{\text{gdw}}. & \delta(q_0, w) \in F \\ & \underline{\text{gdw}}. & \pi(\delta(q_0, w)) \in F' \qquad (\text{wegen } \pi(F) = F') \\ & \underline{\text{gdw}}. & \delta'(\pi(q_0), w) \in F' \\ & \underline{\text{gdw}}. & \delta'(q'_0, w) \in F' \qquad (\text{wegen } q'_0 = \pi(q_0)) \\ & \underline{\text{gdw}}. & w \in L(\mathcal{A}') \quad \square \end{array}$$

Wir können nun Minimalität und Eindeutigkeit des reduzierten Automaten zeigen.

Satz 5.19 (Isomorphie reduzierter und kanonischer Automat)
Es sei L eine erkennbare Sprache und $\mathcal{A}$ ein DEA mit $L(\mathcal{A}) = L$. Dann gilt: der reduzierte Automat $\mathcal{A}_{red} := \widetilde{\mathcal{A}}_0$ isomorph zum kanonischen Automaten $\mathcal{A}_L$.

Beweis. Es sei $\mathcal{A}_{red} = (Q, \Sigma, q_0, \delta, F)$ 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 $\delta(q_0, w_q) = q$, da in $\mathcal{A}_{red}$ alle Zustände erreichbar sind. O.B.d.A. sei $w_{q_0} = \varepsilon$. Wir definieren $\pi(q) := [w_q]_L$.

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

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

<u>nicht</u> gilt. Das heißt aber, dass

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

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 5.11).

III) $\pi(q_0) = q'_0$:
Da $w_{q_0} = \varepsilon$ und $q'_0 = [\varepsilon]_L$.

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

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

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

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

gilt wegen:

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

Mittels $(*)$ zeigen wir nun:

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

□

Dieser Satz zeigt auch folgende interessante Eigenschaften:

- der reduzierte Automat $\mathcal{A}_{red}$ ist unabhängig vom ursprünglichen DEA $\mathcal{A}$:
  wenn $L(\mathcal{A}) = L(\mathcal{B}) = L$, dann gilt wegen $\mathcal{A}_{red} \simeq \mathcal{A}_L \simeq \mathcal{B}_{red}$ auch $\mathcal{A}_{red} \simeq \mathcal{B}_{red}$ (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 $\mathcal{A} = \mathcal{A}_{red} \simeq \mathcal{A}_L \simeq \mathcal{B}_{red} = \mathcal{B}$, also auch $\mathcal{A} \simeq \mathcal{B}$.

Im Prinzip liefert Satz 5.19 zudem eine Methode, um von zwei Automaten zu entscheiden, ob sie dieselbe Sprache akzeptieren:

Korollar 5.20
Es seien $\mathcal{A}$ und $\mathcal{A}'$ DEAs. Dann gilt: $L(\mathcal{A}) = L(\mathcal{A}')$ gdw. $\mathcal{A}_{red} \simeq \mathcal{A}'_{red}$.

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 an Stelle von DEAs gegeben, so kann man diese zuerst deterministisch machen und dann das Korollar anwenden.

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 nicht-deterministischen 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. 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 Akzeptaz via Endzustand funktioniert hier natürlich nicht mehr und man studiert verschiedene Akzeptanzbedingungen wie z.B. Büchi-Akzeptanz und Rabin-Akzeptanz.

Baumautomaten:

Diese Automaten erhalten Bäume statt Wörter 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.

6. 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 6.1

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 6.2 (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 6.3

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 6.4 (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:*
   $x = x_1 u x_2$ und $y = x_1 v x_2$ mit $x \vdash_G y$ und $x_1, x_2 \in (N \cup \Sigma)^*$
2) *y aus x in n Schritten ableitbar:*
   $x \vdash_G^n y$ gdw. $x \vdash x_2 \vdash \cdots \vdash x_{n-1} \vdash y$ für $x_2, \ldots, x_{n-1} \in (N \cup \Sigma)^*$
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 6.3 (Fortsetzung)
Eine Beispielableitung in der Grammatik aus Beispiel 6.3:

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

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:

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

„$\subseteq$“: Sei $S \vdash_G^* w$ mit $w \in \Sigma^*$. Offenbar wird die Regel $S \longrightarrow abc$ in der Ableitung von w genau einmal angewendet. Ist das der erste Schritt, so $w = abc \in \{a^n b^n c^n \mid n \geq 1\}$. Andernfalls kann vor der Anwendung von $S \longrightarrow abc$ nur die Regel $S \longrightarrow aSBc$

angewendet werden, da keine b's generiert werden können. Die Anwendung von $S \longrightarrow abc$ hat damit die Form

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

Nun sind nur noch $cB \longrightarrow Bc$ und $bB \longrightarrow bb$ angewendbar. Man zeigt leicht per Induktion über die Anzahl der Regelanwendungen, dass alle dabei entstehenden Wörter die folgende Form haben:

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

Insbesondere ist also $w = a^n b^n c^n \in \{a^n b^n c^n \mid n \geq 1\}$.

□

Beispiel 6.5

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 6.5, 6.3 und 6.1 gehören zu unterschiedlichen Sprachklassen, die alle durch Grammatiken erzeugt werden können. Sie sind angeordnet in der Chomsky-Hierarchie.

Definition 6.6 (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 (kontextsensitiv)***, 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 *kontextsensitiven* 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, kontextfreie Grammatiken ausschließlich durch Regeln der obigen Form zu definieren.

Beachte auch: Bei allen Grammatiktypen ist es nach Definition erlaubt, dass ein Nichtterminal auf der linken Seite mehrerer Regeln verwendet wird.

Eine sehr wichtige Eigenschaft von kontextsensitiven 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 6.7

Die Grammatik aus Beispiel 6.1 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 6.3 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 6.5 ist vom Typ 2.

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

Definition 6.8 (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 vom 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 6.1 vom 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 vom Typ 2 und jede Grammatik von Typ 1 auch eine vom Typ 0. Da Grammatiken vom Typ 2 und 3 das Verkürzen des abgeleiteten Wortes erlauben, sind solche Grammatiken nicht notwendigerweise vom Typ 1. Wir werden jedoch später sehen, dass jede Typ 2 Grammatik effektiv in eine Typ 1 Grammatik gewandelt werden kann, die dieselbe Sprache erzeugt. Daher bilden die assoziierten Sprachtypen eine Hierarchie. Diese ist sogar strikt.

Lemma 6.9
$\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$

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 (Satz 8.12). Auch die Striktheit der Inklusionen werden wir erst später beweisen. □

7. Rechtslineare Grammatiken und reguläre Sprachen

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

Satz 7.1
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

   S $L \in \mathcal{L}_3$, 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 gibt eine Ableitung

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

   für Produktionen $B_{i-1} \longrightarrow w_i B_i \in P \quad (i = 1, \ldots, n)$ 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 Worttransitionen*, der die Nichtterminalsymbole von G als Zustände hat:

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

   - $\Omega \notin N$ Endzustand 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 $(\star)$ entsprechen nun genau Pfaden in $\mathcal{A}$:

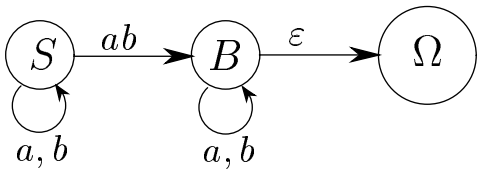
   $$(\star\star) \quad (S, w_1, B_1)(B_1, w_2, B_2) \ldots (B_{n-2}, w_{n-1}, B_{n-1})(B_{n-1}, w_n, \Omega)$$

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

   Beispiel 6.5 (Fortsetzung)
   Die Grammatik

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

   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:

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

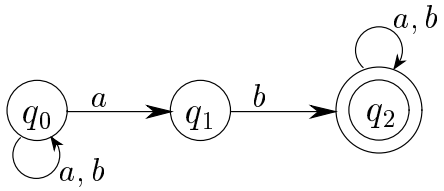
   Ein Pfad in $\mathcal{A}$ der Form

$$(q_0, a_1, q_1)(q_1, a_2, q_2) \ldots (q_{n-1}, a_n, q_n)$$

   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:
   Der folgende NEA



   liefert die Grammatik mit den rechtslinearen Produktionen

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

□

Korollar 7.2
$\mathcal{L}_3 \subset \mathcal{L}_2$.

Beweis. Wir wissen bereits, dass $\mathcal{L}_3 \subseteq \mathcal{L}_2$ gilt. Außerdem haben wir mit Beispiel 6.1 $L := \{a^n b^n \mid n \geq 0\} \in \mathcal{L}_2$. Wir haben bereits gezeigt, dass L nicht erkennbar/regulär ist, d.h. mit Satz 7.1 folgt $L \notin \mathcal{L}_3$. □

Beispiel 7.3

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 2.3, 3.2) Man kann diese Sprache mit folgender 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:

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

woraus sich ergibt:

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

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

8. 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* kennenlernen 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.

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

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 8.1 (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 8.2
Zu einer kontextfreien Grammatik $G = (N, \Sigma, P, S)$ kann man effektiv die Menge der terminierenden Symbole bestimmen.

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

$$T_1 \subseteq T_2 \subseteq \ldots \subseteq N.$$

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, $i \geq 1$, terminierend:

- $i = 1$: $\quad A \in T_1 \Rightarrow A \vdash_G w \in \Sigma^*$, also A terminierend.
- $i \to i+1$: $A \in T_{i+1} \Rightarrow A \vdash_G u_1 B_1 u_2 B_2 \dots u_n B_n u_{n+1}$ mit $u_j \in \Sigma^*$ und $B_j \in T_j$, also (Induktion) $B_j \vdash_G^* w_j \in \Sigma^*$. Es folgt, dass A terminierend.

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

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

- $i = 1$: $\quad A \vdash_G^{\leq 1} w \in \Sigma^* \Rightarrow A \longrightarrow w \in P \Rightarrow A \in T_1$
- $i \to i+1$: $A \vdash_G^{\leq i+1} w \in \Sigma^*$
  $\Rightarrow A \vdash_G u_1 B_1 \dots u_n B_n u_{n+1} \vdash_G^{\leq i} u_1 w_1 \dots u_n w_n u_{n+1} = w$,
  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 8.3

$$P = \{ \quad \begin{array}{ll} S \longrightarrow A, & S \longrightarrow aCbBa, \\ A \longrightarrow aBAc, & B \longrightarrow Cab, \\ C \longrightarrow AB, & C \longrightarrow aa, \end{array}$$

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

Es ist also A das einzige nichtterminierende Symbol.

Seitenbemerkung: Lemma 8.2 hat folgende interessante Konsequenz.

Satz 8.4

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 später sehen, dass das Leerheitsproblem für Grammatiken der Typen 0 und 1 nicht entscheidbar ist.

Lemma 8.5

Zu einer kontextfreien Grammatik $G = (N, \Sigma, P, S)$ kann man effektiv die Menge der erreichbaren Nichtterminalsymbole bestimmen.

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

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

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 8.6

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

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

Es ist also A das einzige nichterreichbare Symbol.

Lemma 8.2 und 8.5 zusammen zeigen, wie man unerreichbare und nichtterminierende Symbole eliminieren kann.

Satz 8.7
Zu jeder kontextfreien Grammatik G mit $L(G) \neq \emptyset$ kann man effektiv eine äquivalente reduzierte kontextfreie Grammatik erzeugen.

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

Erster Schritt: Eliminieren nicht terminierender Symbole.

Definiere $G' := (N', \Sigma, P', S)$, wobei

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

Beachte: Weil $L(G) \neq 0$ ist S terminierend, also $S \in N'$!

Zweiter Schritt: Eliminieren unerreichbarer Symbole.

Wir definieren $G'' := (N'', \Sigma, P'', S)$, wobei

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

Beachte: Ist $A \in N''$ und $A \longrightarrow u_1 B u_2 \in P'$, so ist $B \in N''$.

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 nicht terminierender 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 zuerst die unerreichbaren Symbole, so ändert sich die Grammatik also nicht. Das einzige nicht terminierende 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.

Wie zeigen nun, dass jede reduzierte Grammatik effektiv 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 sogennante *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 ε nicht auf der rechten Seite von Produktionen vorkommt.

Wir beginnen mit dem Eliminieren von ε-Regeln.

Definition 8.8 (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 8.9
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 $A \vdash_G^* \varepsilon$:

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

Es gibt ein k mit $N_k = N_{k+1} = \bigcup_{i \geq 1} N_i$. Für dieses k gilt: $A \in N_k$ gdw. $A \vdash_G^* \varepsilon$.

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

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

die Regeln

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

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

Beispiel:

$$\begin{aligned}
P = \quad & \{S \longrightarrow aS,\ S \longrightarrow SS,\ S \longrightarrow bA,\\
& A \longrightarrow BB,\\
& B \longrightarrow CC,\ B \longrightarrow aAbC,\\
& C \longrightarrow \varepsilon\}\\
N_0 = \quad & \{C\},\\
N_1 = \quad & \{C, B\},\\
N_2 = \quad & \{C, B, A\} = N_3\\
P' = \quad & \{S \longrightarrow aS,\ S \longrightarrow SS,\ S \longrightarrow bA,\ S \longrightarrow b,\\
& A \longrightarrow BB,\ A \longrightarrow B,\\
& B \longrightarrow CC,\ B \longrightarrow C,\\
& B \longrightarrow aAbC,\ B \longrightarrow abC,\ B \longrightarrow aAb, B \longrightarrow ab\}
\end{aligned}$$

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

Satz 8.10

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

Beweis. Konstruiere G' wie im Beweis von Lemma 8.9 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 8.11

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

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 ja aber wie auch bei Typ-1 gefordert auf keiner rechten Regelseite auftritt). □

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

Satz 8.12
Zu jeder kontextfreien Grammatik kann man effektiv eine äquivalente kontextfreie Grammatik konstruieren, die keine Kettenregeln enthält.

Beweisskizze.

1) Bestimme zu jedem $A \in N$ die Menge $N(A) := \{B \in N \mid A \vdash_G^* B\}$ (effektiv machbar).
2) $P' = \{A \longrightarrow w \mid B \longrightarrow w \in P,\ B \in N(A) \text{ und } w \notin N\}$ □

Beispiel:

$$\begin{aligned} P &= \{S \longrightarrow A,\ A \longrightarrow B,\ B \longrightarrow aA,\ B \longrightarrow b\} \\ N(S) &= \{S, A, B\}, \\ N(A) &= \{A, B\}, \\ N(B) &= \{B\} \\ P' &= \{B \longrightarrow aA,\ A \longrightarrow aA,\ S \longrightarrow aA, \\ &\qquad B \longrightarrow b, \quad A \longrightarrow b, \quad S \longrightarrow b\} \end{aligned}$$

Wir etablieren nun die Chomsky-Normalform.

Satz 8.13 (Chomsky-Normalform)
Jede kontextfreie Grammatik lässt sich umformen in eine äquivalente Grammatik, die nur Regeln der Form

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

enthält. 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 $A \longrightarrow B_1 \dots B_n$ für $n > 2$ werden ersetzt durch

$$A \longrightarrow B_1 C_1,\ C_1 \longrightarrow B_2 C_2,\ \dots,\ C_{n-2} \longrightarrow B_{n-1} B_n$$

wobei die C_i jeweils neue Symbole sind.

□

Wir betrachten nun das *Wortproblem* für kontextfrei 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 den meisten Anwendungen durchaus sinnvoll: wenn man z.B. einen Parser für eine Programmiersprache erstellt, so tut man dies i.d.R. 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 beginnen damit, kurz den Grund für die Entscheidbarkeit des Wortproblem für *reguläre* Sprachen zu rekapitulieren. Wir haben zwar eine Reduktion auf das Leerheitsproblem verwendet (um eine gute Komplexität zu erreichen), aber ein viel einfacherer Grund für Entscheidbarkeit ist der folgende: ein akzeptierender Pfad im NEA $\mathcal{A}$, der mit w beschriftet ist, muss genau die Länge $|w|$ haben. Es gibt offensichtlich nur endlich viele Kandidaten für Pfade dieser Länge. Man kann also alle Kandidaten *aufzählen* und dann *prüfen*, ob einer davon ein akzeptierender Pfad für w ist. Analog dazu muss jede Ableitung von w in einer rechtslinearen Grammatik genau die Länge $|w|$ haben, so dass man hier dasselbe Argument wie für Pfade in einem Automaten verwenden kann.

Im allgemeinen kann man bei kontextfreien Grammatiken keine Schranke für die Länge einer Ableitung von w angeben. Dies liegt an den Regeln $A \longrightarrow \varepsilon$ und an Kettenregeln: man kann u.U. mit Kettenregeln beliebig oft zwei Nichtterminale hin- und hertauschen erzeugen bevor man mit "sinnvollen" Ableitungsschritten weitermacht. Bei Grammatiken in Chomsky-Normalform existiert eine solche Schranke aber sehr wohl:

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

Es folgt: $w \in L(G)$ wird durch eine Ableitung der Länge $\leq 2|w|+1$ erzeugt ("+1" wegen des leeren Wortes, das Länge 0 hat, aber einen Ableitungsschritt benötigt).

Da es aber i.a. $\geq 2^n$ Ableitungen der Länge n geben kann, liefert dies ein *exponentielles Verfahren* zur Entscheidung des Wortproblems.

Ein besseres Verfahren (*kubisch*) liefert die folgende Überlegung:

Definition 8.14

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

- $w_{ij} := a_i \dots a_j \quad$ (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.	$A \vdash_G^* a_i \dots a_j$
		gdw.	$\exists A \longrightarrow BC \in P$ und ein k mit $i \leq k < j$ mit
			$B \vdash_G^* a_i \dots a_k$ und $C \vdash_G^* a_{k+1} \dots a_j$
		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 sogenannten "Divide & Conquer" ("Teile und Herrsche") Methode. Die generelle Idee dabei ist, das eigentliche Problem in Teilprobleme zu uerlegen 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.

Algorithmus 8.15 (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)
      $N_{ij} := N_{ij} \cup \{A \mid \exists A \longrightarrow BC \in P$ mit $B \in N_{ik}$ und $C \in N_{(k+1)j}\}$

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 8.16

Für eine gegebene Grammatik in Chomsky-Normalform entscheidet Algorithmus 8.15 die Frage „$w \in L(G)$?" in der Zeit $O(|w|^3)$.

Beweis. Drei geschachtelte Schleifen, die jeweils $\leq |w| = n$ Schritte machen, daraus folgt: $|w|^3$ Schritte in der innersten Schleife. □

Beachte:
Die Größe von G ist hier als konstant angenommen (fest vorgegebenes G). Daher braucht die Suche nach den Produktionen $A \longrightarrow BC$ und $A \longrightarrow a_i$ auch nur konstante Zeit.

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

und $w = abacba$:

$i\backslash j$	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 „Theoretische Informatik II“ beweisen, dass das Äquivalenzproblem für kontextfreie Sprachen unentscheidbar ist, siehe Satz 17.7 dieses Skriptes. Es gibt also keinen Algorithmus, der für zwei gegebene kontextfreie Grammatiken G_1 und G_2 entscheidet, ob $L(G_1) = L(G_2)$.

Eine weitere interessante Normalform für kontextfreie Grammatiken ist die *Greibach-Normalform*, bei der jede Produktion mindestens ein Terminalsymbol erzeugt. Wir geben den folgenden Satz ohne Beweis an.

Satz 8.17 (Greibach-Normalform)
Jede kontextfreie Grammatik lässt sich effektiv 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.

9. Abschlusseigenschaften und Pumping Lemma

Die kontextfreien Sprachen verhalten sich bezüglich Abschlusseigenschaften nicht ganz so gut wie die regulären Sprachen. Wir beginnen mit positiven Resultaten.

Satz 9.1
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 $G_i = (N_i, \Sigma, P_i, S_i) \quad (i = 1, 2)$. O.B.d.A. nehmen wir 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^* □

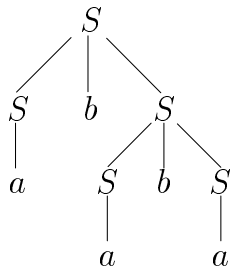
Wir werden zeigen, dass Abschluss unter *Durchschnitt* und *Komplement nicht* gilt. Dazu benötigen wir zunächst eine geeignete Methode, von einer Sprache nachzuweisen, dass sie *nicht* kontextfrei ist. Dies gelingt wieder mit Hilfe eines Pumping-Lemmas. Um dieses zu zeigen, stellt man Ableitungen als Bäume dar, sogenannte *Ableitungsbäume*.

Beispiel:
$P = \{S \longrightarrow SbS, S \longrightarrow a\}$
Drei *Ableitungen* des Wortes *ababa*:

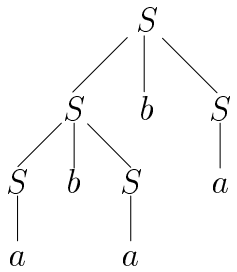
1) $S \vdash SbS \vdash abS \vdash abSbS \vdash ababS \vdash ababa$
2) $S \vdash SbS \vdash abS \vdash abSbS \vdash abSba \vdash ababa$
3) $S \vdash SbS \vdash Sba \vdash SbSba \vdash Sbaba \vdash ababa$

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 $A \longrightarrow \alpha_1 \ldots \alpha_n \in P$ 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 $A \vdash_G^* \alpha_1 \ldots \alpha_n$.

Lemma 9.2 (Pumping-Lemma für kontextfreie Sprachen)
Für jede kontextfreie Sprache L gibt es ein $n_0 \geq 0$ so dass gilt:
für jedes $z \in L$ mit $|z| \geq n_0$ existiert eine Zerlegung $z = uvwxy$ mit:

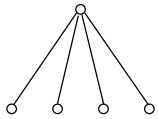
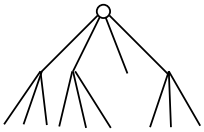
- $vx \neq \varepsilon$ *und* $|vwx| \leq n_0$
- $uv^iwx^iy \in L$ *für alle* $i \geq 0$.

Beweis. Sei G eine kontextfreie Grammatik mit $L(G) = L$. Nach Satz 8.10 und 8.12 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 eine Schranke auf die Länge der rechten Regelseiten in G;
- $n_0 = k^{m+1}$.

Wir verfahren nun wir folgt.

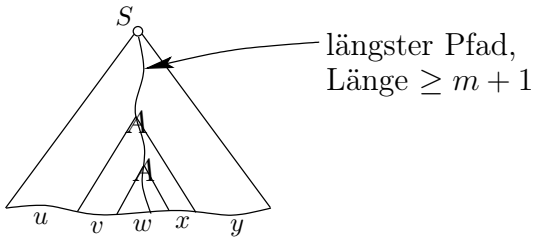
1) Ein Baum der Tiefe $\leq t$ und der Verzweigungszahl $\leq k$ hat maximal k^t viele Blätter:

eine Ebene: $\leq k$ Blätter	zwei Ebenen: $\leq k^2$ Blätter
	

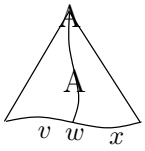
   , etc.

   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 Anzahl 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 Pumpinglemma, um beispielhaft von einer Sprache nachzuweisen, dass sie nicht kontextfrei ist.

Lemma 9.3
$L = \{a^nb^nc^n \mid n \geq 1\} \notin \mathcal{L}_2$.

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 9.2. Wir betrachten $z = a^{n_0}b^{n_0}c^{n_0} \in L = L(G)$. Mit Satz 9.2 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 einen Symbole aus $\{a, b, c\}$, der in xv nicht vorkommt. Daher kommt dieser 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 kontextsensitiven Sprachen sind.

Satz 9.4
$\mathcal{L}_2 \subset \mathcal{L}_1$.

Beweis. Wir haben bereits gezeigt, dass $\mathcal{L}_2 \subseteq \mathcal{L}_1$ gilt (Korollar 8.11). 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 9.3 ist $L \notin \mathcal{L}_2$. Nach Beispiel 6.3 gilt aber $L \in \mathcal{L}_1$. □

Ausserdem können wir nun zeigen, dass die kontextfreien Sprachen unter zwei wichtigen Operationen nicht abgeschlossen sind.

Korollar 9.5
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 9.4.
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).

□

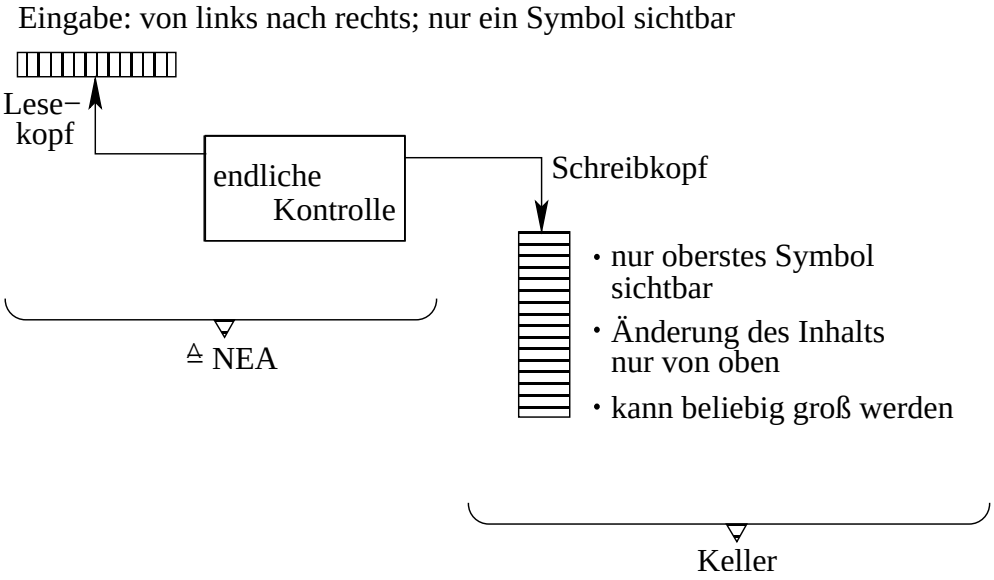
Beachte:
Man kann daher das Äquivalenzproblem für kontextfreie Sprachen nicht einfach auf das Leerheitsproblem reduzieren (dazu braucht man sowohl Schnitt als auch Komplement). Wie bereits erwähnt werden wir später sogar sehen, dass das Äquivalenzproblem für kontextfreie Sprachen *unentscheidbar* ist.

10. 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 sogenannten *Keller (engl. 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 10.1 (Kellerautomat)
Ein *Kellerautomat (pushdown automaton, kurz PDA)* hat die Form
$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$, 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 und
- $\Delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times \Gamma^* \times Q$ eine endliche *Übergangsrelation* 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).

Im Gegensatz zu den ε-Übergängen von ε-NEAs können bei PDAs Zustände der zweiten Form im allgemeinen nicht eliminiert werden (z.B. kann ohne solche Übergänge das leere Wort nicht eliminert werden). Man beachte, dass ein Kellerautomat nicht über Endzustände verfügt. Wie wir im folgenden sehen werden ist Akzeptanz stattdessen über den *leeren Keller* definiert.

Um die Sprache zu definieren, die von einem Kellerautomaten erkannt wird, brauchen wir den Begriff der *Konfiguration*, die den aktuellen Stand einer Kellerautomatenberechnet 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* $\gamma \in \Gamma^*$ (Schreiblesekopf steht auf dem ersten Symbol von γ)

Definition 10.2
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

$$\mathcal{K}_0 = \mathcal{K},\ \mathcal{K}_n = \mathcal{K}' \text{ und } \mathcal{K}_i \vdash_{\mathcal{A}} \mathcal{K}_{i+1} \text{ für } 0 \leq i < n.$$

Der Automat $\mathcal{A}$ *akzeptiert* das Wort $w \in \Sigma^*$ gdw.

$$(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon) \quad \text{(Eingabewort ganz gelesen und Keller leer).}$$

Die von $\mathcal{A}$ *erkannte Sprache* ist

$$L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}.$$

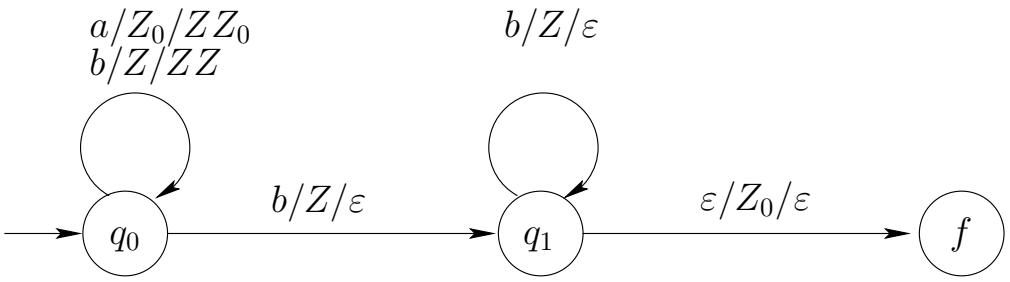
Im folgenden zwei einfach Beispiele für Kellerautomaten.

Beispiel 10.3

Ein PDA für $\{a^n b^n \mid n \geq 1\}$.

- $Q = \{q_0, q_1, f\}$,
- $\Gamma = \{Z, Z_0\}$,
- $\Sigma = \{a, b\}$ und
- $\Delta = \{(q_0, a, Z_0, ZZ_0, q_0)$, (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)

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 kann und Z durch γ ersetzen.



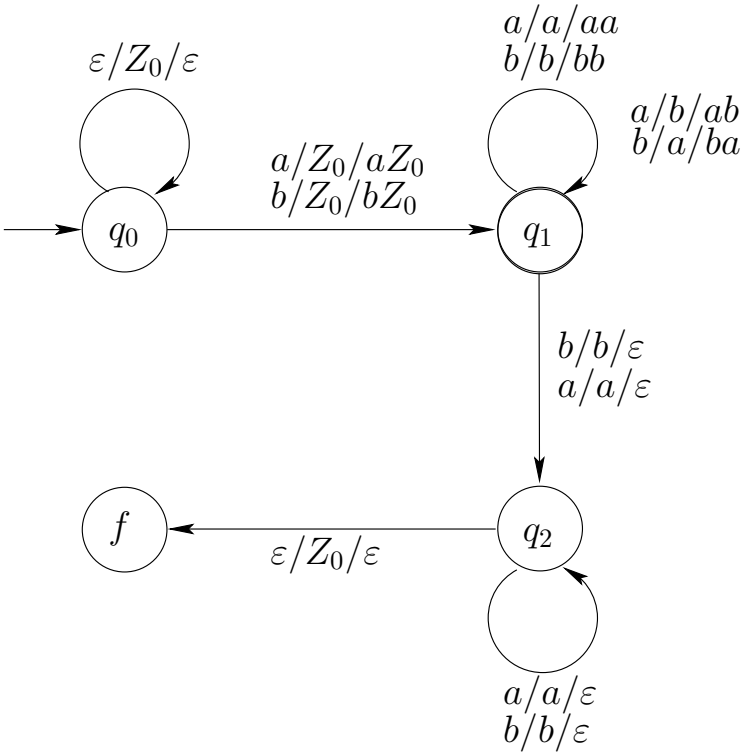
Betrachten wir beispielhaft einige Konfigurationsübergänge:

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

Beispiel 10.4

Ein PDA für $L = \{w \overleftarrow{w} \mid w \in \{a, b\}^*\}$ (wobei für $w = a_1 \dots a_n$ gilt $\overleftarrow{w} = a_n \dots a_1$).

- $Q = \{q_0, q_1, q_2, f\}$,
- $\Gamma = \{a, b, Z_0\}$,
- $\Sigma = \{a, b\}$, und
- $\Delta =$



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 10.1 und 10.2 eingeführte Version von Kellerautomaten akzeptiert *per leerem Keller*. Man kann stattdessen auch Akzeptanz *per Endzustand* definieren.

Definition 10.5

Ein *Kellerautomat (PDA) mit Endzuständen* ist ein Tupel

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F),$$

wobei alle Komponenten bis auf F wie in Definition 10.1 sind und $F \subseteq Q$ eine *Endzustandsmenge* ist. Ein solcher PDA *akzeptiert* ein Eingabewort $w \in \Sigma^*$ gdw. $(q_0, w, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \gamma)$ für ein $q \in F$ und $\gamma \in \Gamma^*$.

Es ist nicht schwer, zu zeigen, dass PDAs mit Akzeptanz per leerem Keller und solche mit Akzeptanz per Endzustand dieselbe Sprachklasse definieren. Der Beweis wird als Übung gelassen.

Satz 10.6

Jeder Sprache, die von einem PDA erkannt wird, wird auch von einem PDA mit Endzuständen erkannt und umgekehrt.

Wir werden nun zeigen, dass man mit Kellerautomaten genau die kontextfreien Sprachen erkennen kann.

Satz 10.7

Für eine formale Sprache L sind äquivalent:

1) $L = L(G)$ für eine kontextfreie Grammatik G.

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

Beweis.

„$1 \longrightarrow 2$“: Es sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik. Der zugehörige PDA simuliert Linksableitungen von G, d.h. Ableitungen, bei denen stets das am weitesten links stehende Nichtterminalsymbol zuerst ersetzt wird.

Leicht zu zeigen:

Da G kontextfrei ist, kann jedes Wort in $L(G)$ kann durch eine Linksableitung erzeugt werden.

Wir definieren $\mathcal{A} = (\{q\}, \Sigma, \Sigma \cup N, q, S, \Delta)$ mit $\Delta :=$

$\{(q, \varepsilon, A, \gamma, q) \mid A \longrightarrow \gamma \in P\} \cup$	(Anwenden einer Produktion auf oberstes Kellersymbol) $(\star)$
$\{(q, a, a, \varepsilon, q) \mid a \in \Sigma\}$	(Entfernen bereits erzeugter Terminalsymbole von der Kellerspitze, wenn sie in der Eingabe vorhanden sind) $(\star\star)$

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*

$$(q, abba, S) \vdash_{\mathcal{A}} (q, abba, aSa) \vdash_{\mathcal{A}} (q, bba, Sa) \vdash_{\mathcal{A}} (q, bba, bSba) \vdash_{\mathcal{A}} (q, ba, Sba) \vdash_{\mathcal{A}} (q, ba, ba) \vdash_{\mathcal{A}} (q, a, a) \vdash_{\mathcal{A}} (q, \varepsilon, \varepsilon)$$

Behauptung:

Für $u, v \in \Sigma^*$, $\{\varepsilon\} \cup N \cdot (\Sigma \cup N)^*$ gilt:

$$S \vdash_G^* u\alpha \text{ mit Linksableitung} \quad \underline{\text{gdw}}. \quad (q, uv, S) \vdash_{\mathcal{A}}^* (q, v, \alpha).$$

Beachte:

Für $\alpha = \varepsilon = v$ folgt:

$$S \vdash_G^* u \quad \underline{\text{gdw}}. \quad (q, u, S) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$$

d.h. $L(G) = L(\mathcal{A})$.

Beweis der Behauptung.

„⇐“: Beweis durch Induktion über die Anzahl k der Übergänge mit Transitionen der Form $(\star) : (q, \varepsilon, A, \gamma, q)$.

$k = 0$:
Da im Keller S steht, sind auch keine Transitionen der Form $(\star\star) : (q, a, a, \varepsilon, q)$ möglich, d.h. $u = \varepsilon$ und $\alpha = S$. Offenbar gilt $S \vdash^* S$.

$k \to k + 1$: Sei $(q, \varepsilon, A, \gamma, q)$ die letzte angewendete Transition dieser Art (also $A \longrightarrow \gamma \in P$), u_1 das Präfix von u, das bis zu dieser letzten Transition schon gelesen wurde und $u = u_1 u_2$. Es gilt:

$$(q, u_1u_2v, S) \vdash^*_{\mathcal{A}} (q, u_2v, A\alpha') \overset{(q,\varepsilon,A,\gamma,q)}{\vdash_{\mathcal{A}}} (q, u_2v, \gamma\alpha') \overset{\text{Trans. } (\star\star)}{\vdash^*_{\mathcal{A}}} (q, v, \alpha)$$

Induktion liefert $S \vdash^{*_G} u_1 A\alpha'$; die Regel $A \longrightarrow \gamma \in P$ liefert also

$$S \vdash^*_G u_1\gamma\alpha'.$$

Da sich (q, v, a) aus $(q, u_2, v, \gamma\alpha')$ durch Anwenden von Regeln der Form $(\star\star)$ ergibt, gilt $\gamma\alpha' = u_2\alpha$ und wir erhalten

$$S \vdash^*_G u_1u_2\alpha = u\alpha.$$

„⇒“: Beweis durch Induktion über die Länge der Linksableitung

$k = 0$:
Dann ist $u = \varepsilon,\ \alpha = S$ und $(q, v, S) \vdash^0_{\mathcal{A}} (q, v, S)$

$k \to k + 1$: Sei $S \vdash^{k+1}_G u\alpha$ und $A \to \gamma$ die letzte angewendete Regel, also

$$S \vdash^k_G u'A\beta \vdash_G u'\gamma\beta = u\alpha.$$

Wegen Linksableitung ist $u' \in \Sigma^*$.

Es sei u'' das längste Anfangsstück von $\gamma\beta$, das in Σ^* liegt. Da A mit Nichtterminal beginnt ist $u'u'' = u$ und damit $\gamma\beta = u''\alpha$. Wähle $v \in \Sigma^*$. Zu zeigen:

$$(q, u'u''v, S) \vdash^*_{\mathcal{A}} (q, v, \alpha).$$

Wähle $v' = u''v$. Induktion liefert:

$$(q, u'u''v, S) = (q, u'v', S) \vdash^*_{\mathcal{A}} (q, u''v, A\beta)$$

und wegen $A \to \gamma \in P$ liefert die Transition $(q, \varepsilon, A, \gamma, q)$

$$(q, u''v, A\beta) \vdash_{\mathcal{A}} (q, u''v, \gamma\beta) = (q, u''v, u''\alpha).$$

Da $u'' \in \Sigma^*$ gibt es Übergänge $(q, u''v, u''\alpha) \vdash^*_{\mathcal{A}} (q, v, \alpha)$. □

„$2 \longrightarrow 1$“: Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ ein PDA. Die Nichtterminalsymbole der zugehörigen Grammatik 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 eine Transition $(p, a, Z, X_1 \cdots X_n)$, 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).

Formale Definition:

$$\begin{aligned}
G &:= (N, \Sigma, P, S) \text{ mit} \\
N &:= \{S\} \cup \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \\
P &:= \{S \longrightarrow [q_0, Z_0, q] \mid q \in Q\} \cup \\
&\quad \{[p, Z, q] \longrightarrow a \mid (p, a, Z, \varepsilon, q) \in \Delta \text{ mit } a \in \Sigma \cup \{\varepsilon\}\ \} \cup \\
&\quad \{[p, Z, q] \longrightarrow a[p_0, X_1, p_1][p_1, X_2, p_2] \ldots [p_{n-1}, X_n, q] \mid \\
&\qquad (p, a, Z, X_1 \ldots X_n, p_0) \in \Delta \text{ und} \\
&\qquad a \in \Sigma \cup \{\varepsilon\}, \\
&\qquad p_1, \ldots, p_{n-1} \in Q, \\
&\qquad n \geq 1\}
\end{aligned}$$

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:

$$(\star) \qquad [p, Z, q] \vdash_G^* u \quad \underline{\text{gdw}}. \quad (p, u, Z) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$$

Für $p = p_0$ und $Z = Z_0$ folgt daraus:

$$\vdash_G [q_0, Z_0, q] \vdash_G^* u \quad \underline{\text{gdw}}. \quad S(q_0, u, Z_0) \vdash_{\mathcal{A}}^* (q, \varepsilon, \varepsilon)$$

d.h. $u \in L(G) \quad \underline{\text{gdw}}. \quad u \in L(\mathcal{A})$.

Der Beweis dieser Behauptung kann durch Induktion über die Länge der Konfigurationsfolge („⇒“) bzw. über die Länge der Ableitung („⇐“) geführt werden.

□

Beispiel:
Gegeben sei der PDA für $\{a^n b^n \mid n \geq 1\}$ aus Beispiel 10.3. Die Berechnung des 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 10.7 ergibt sich leicht folgendes Korollar. Wir nennen zwei PDAs $\mathcal{A}$ und $\mathcal{A}'$ *äquivalent* wenn $L(\mathcal{A}) = L(\mathcal{A}')$.

Korollar 10.8
Jeder PDA $\mathcal{A}$ kann effektiv in einen PDA $\mathcal{A}'$ gewandelt werden, 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 „$2 \longrightarrow 1$“ des Beweises von Satz 10.7 anwenden und dann die aus dem Teil „$1 \longrightarrow 2$“. Man erhält einen äquivalenten PDA der nach Konstruktion nur einen einzigen Zustand enthält. Alle Konstruktionen sind effektiv. □

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 10.9
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 $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F)$ (o.B.d.A. mit Endzuständen) 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).

Deterministische Kellerautomaten

Analog zu endlichen Automaten kann man auch bei Kellerautomaten eine deterministische Variante betrachten. Intuitiv ist der Kellerautomat aus Beispiel 10.3 deterministisch, da es zu jeder Konfiguration höchstens eine Folgekonfiguration gibt. Der Kellerautomat aus Beispiel 10.4 ist hingegen nicht-deterministisch, 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 nicht-deterministische. Daher definieren die deterministischen PDAs eine eigene Sprachklasse, die *deterministisch kontextfreien Sprachen*.

Deterministische DEAs akzeptieren immer per Endzustand (aus gutem Grund, wie wir noch sehen werden).

Definition 10.10 (deterministischer Kellerautomat)
Ein *deterministischer Kellerautomat (dPDA)* ist ein PDA mit Endzuständen

$$\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta, F)$$

der folgende Eigenschaften erfüllt:

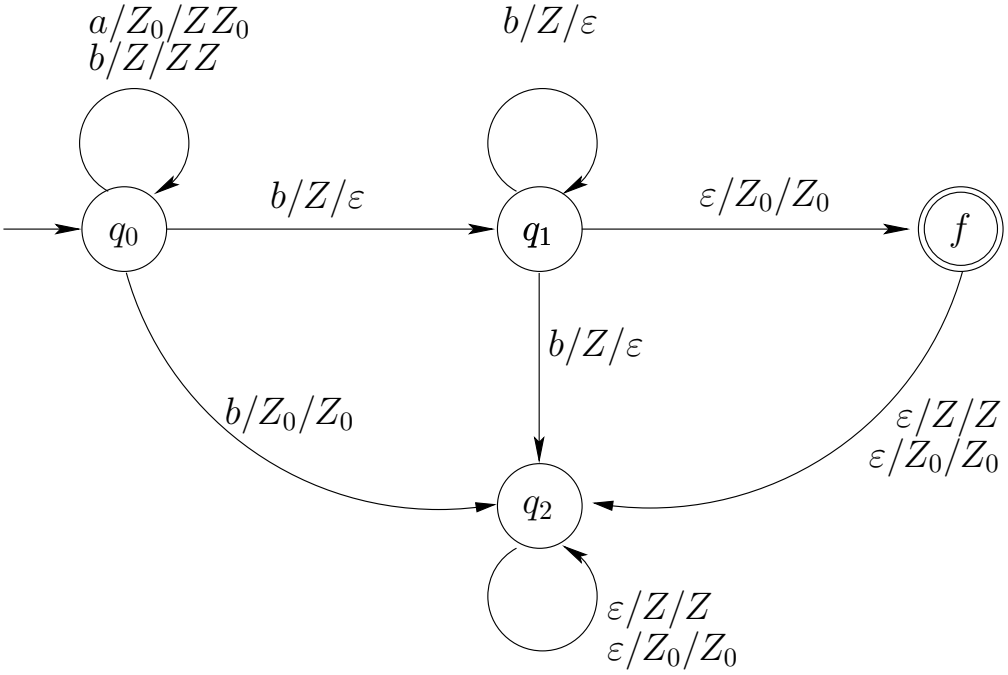
1. Für alle $q \in Q$, $a \in \Sigma$ und $Z \in \Gamma$ gibt es genau eine Transition der Form (q, a, Z, γ, q') oder $(q, \varepsilon, Z, \gamma, q')$;
2. Wenn eine Transition das Kellerstartsymbol Z_0 entfernt, so muss sie es direkt wieder zurückschreiben; alle Transitionen, in denen Z_0 vorkommt, müssen also die Form (q, a, Z_0, Z_0, q') haben.

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 normaler PDA mit Endzuständen 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 10.11

Als Beispiel für einen dPDA betrachte die folgende Variante des PDAs aus Beispiel 10.3, die ebenfalls $L = \{a^n b^n \mid n \geq 1\}$ erkennt:

- $Q = \{q_0, q_1, q_2, f\}$;
- $\Gamma = \{Z, Z_0\}$;
- $\Sigma = \{a, b\}$;
- $\Delta =$



Im Unterschied zum Automaten aus Beispiel 10.3 ist f ein Endzustand geworden, der Übergang von q_1 nach f entfernt Z_0 nicht mehr 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 vorangegangenen Beispiel:

- Auf manchen Eingaben gibt es mehr als eine Berechnung. Als Beispiel betrachte die Eingabe *aabb*. Nach diese gelesen wurde, befindet sich der PDA im Zustand q_1, der kein Endzustand ist. Man kann jedoch den Endzustand f in einem weiteren Schritt erreichen und darum wird die Eingabe *aabb* akzeptiert. Danach kann man im Prizip noch den Nichtendzustand q_2 erreichen und in diesem beliebig oft loopen (was aber nicht sinnvoll ist).

- Auf anderen Eingaben wie z.B. *aa* gibt es nur eine einzige Berechnung.
- 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* (also sehr effizient) entschieden werden kann. Aus diesem Grund spielen dPDAs im Gebiet des Compilerbaus eine wichtige Rolle.

Definition 10.12
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 bezeichnen wir mit $\mathcal{L}_2^d$.

Folgende Einordnung der deterministisch kontextfreien Sprachen ist leicht vorzunehmen.

Satz 10.13
$\mathcal{L}_3 \subset \mathcal{L}_2^d \subseteq \mathcal{L}_2$.

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 die Transition (q, a, Z_0, Z_0, q') des dPDA. Die Inklusion ist echt, da mit Beispiel 10.11 $L = \{a^n b^n \mid n \geq 1\} \in \mathcal{L}_2^d$, wohingegen $L \notin \mathcal{L}_3$. Die Inklusion $\mathcal{L}_2^d \subseteq \mathcal{L}_2$ gilt wegen Satz 10.6. □

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 aufwendigen Beweis und verweisen z.B. auf [Koz06].

Satz 10.14
$\mathcal{L}_2^d$ *ist unter Komplement abgeschlossen.*

Zur Erinnerung: die kontextfreien Sprachen selbst sind mit Korollar 9.5 nicht unter Komplement abgeschlossen. Man kann zeigen, dass die deterministisch kontextfreien Sprachen nicht unter Schnitt, Vereinigung, Konkatenation und Kleene-Stern abgeschlossen sind.

Satz 10.15
$\mathcal{L}_2^d \subset \mathcal{L}_2$.

Beweis. Mit Satz 10.14 ist der folgende sehr einfache Beweis möglich: wäre $\mathcal{L}_2^d = \mathcal{L}_2$, so wäre mit Satz 10.14 $\mathcal{L}_2$ unter Komplement abgeschlossen, was jedoch ein Widerspruch zu Korollar 9.5 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 10.14 auch $\overline{L} \in \mathcal{L}_2^d \subseteq \mathcal{L}_2$, womit ein Widerspruch hergestellt ist. □

Auch die Sprache $\{w \overleftarrow{w} \mid w \in \{a,b\}^*\}$ aus Beispiel 10.4 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 $\{w\, c\, \overleftarrow{w} \mid w \in \{a,b\}^*\}$, 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 das akzeptieren per leerem Keller bei dPDAs zu Problemen führt.

Lemma 10.16
Es gibt keinen dPDA, der die reguläre Sprache $L = \{a^n \mid n \geq 0\}$ per leerem Keller erkennt.

Beweis. Angenommen, der dPDA $\mathcal{A}$ erkennt L per leerem Keller. 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 $w_n = \gamma_n = \varepsilon$. Wegen $aa \in L$ gibt es auch eine solche Folge Ω' für aa. Wegen des Determinismus von $\mathcal{A}$ und da a ein echtes Präfix von aa ist, ist auch Ω ein echtes Präfix von Ω'. Das bedeutet aber, dass es eine Folgekonfiguration für (q_n, w_n, γ_n) gibt, was aufgrund des leeren Kellers unmöglich ist. □

Man kann diese Probleme beheben, indem man ein explizites Symbol für das Wortende einführt, das auch in Transitionen von dPDAs verwendet werden kann. Mit einem solchen Symbol sind Akzeptanz per Endzustand und Akzeptanz per leerem Keller auch für dPDAs äquivalent.

III. Berechenbarkeit

Einführung

Aus der Sicht der Theorie der *formalen Sprachen* (Teil 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?

Es gibt aber auch eine zweite, ganz eigene Perspektive auf den Inhalt von Teil III, nämlich als Einführung in die *Theorie der Berechenbarkeit*. Hierbei handelt es sich um eine 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) gleich mächtig?
- Welche Ausdrucksmittel von Programmiersprachen sind verzichtbar, weil sie zwar der Benutzbarkeit dienen, aber die Berechnungsstärke nicht erhöhen?

In diesem Zusammenhang interessieren wir uns für

- (partielle oder totale) 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 $f(x, y) = x + y$;
  - Die binäre Konkatenationsfunktion $f : (\Sigma^*)^2 \to \Sigma^*$ mit $f(v, w) = vw$.

- Entscheidungsprobleme, formalisiert als Mengen $P \subseteq \Sigma^*$ für ein geeignetes Alphabet Σ (also als formale Sprachen).

  Als Beispiel mag das Leerheitsproblem für kontextfreie Grammatiken dienen: dieses ist formalisierbar als Menge

  $$\{\text{code}(G) \mid G \text{ ist kontextfreie Grammatik mit } L(G) = \emptyset\}$$

  wobei code(G) eine 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 $P \subseteq \Sigma^*$ ist entscheidbar wenn es einen Algorithmus gibt, der bei Eingabe w nach endlicher Zeit 1 zurückgibt wenn $w \in P$ und 0 sonst. Wir werden diese Begriffe später formal definieren.

Um sich klarzumachen, dass eine gegebene Funktion berechenbar oder ein Problem entscheidbar ist, genügt es also, einen entsprechenden Algorithmus für die Funktion bzw. das Problem anzugeben. Dies kann in Form eines Pascal-, Java- oder C-*Programmes* oder in Form einer *abstrakten Beschreibung* der Vorgehensweise bei der Berechnung geschehen. Zum Beispiel hatten 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.

Eine fundamentale Einsicht der Theorie der Berechenbarkeit ist, dass es wohldefinierte (und für die Informatik relevante!) Funktionen gibt, die nicht berechenbar sind, und analog dazu Entscheidungsprobleme, die nicht entscheidbar sind. Beim Nachweis der Nichtberechenbarkeit/entscheidbarkeit ist es nicht mehr ausreichend, einen intuitiven und nicht näher spezifizierten Berechenbarkeitsbegriff zu verwenden: die Aussage, dass es *kein Berechnungsverfahren gibt*, bezieht sich implizit auf *alle* Berechnungsverfahren (jedes Verfahren berechnet etwas anderes als die betrachtete Funktion). Aus diesem Grund benötigt man eine formale Definition dessen, was man unter einem Berechnungsverfahren/Algorithmus versteht.

Man benötigt ein *Berechnungsmodell*, das

1) **einfach** ist, damit formale Beweise erleichtert werden (z.B. nicht Programmiersprache Java),
2) **berechnungsuniversell** ist, d.h. alle intuitiv berechenbaren Funktionen damit berechnet werden können (bzw. alle intuitiv entscheidbaren Mengen entschieden werden können)—also keine endlichen Automaten, denn deren Berechnungsstärke ist viel zu schwach.

Wir werden drei Berechnungsmodelle betrachten:

- Turingmaschinen als besonders einfaches aber dennoch berechnungsuniverselles Modell
- WHILE-Programme als Abstraktion imperativer Programmiersprachen (im Prinzip handelt es sich um eine möglichst einfache, aber immernoch berechnungsuniverselle solche Sprache)
- μ-rekursive Funktionen als funktionsbasiertes, mathematisches Berechnungsmodell.

Es gibt noch eine Vielzahl anderer Modelle: Registermaschinen, GOTO-Programme, k-Zählermaschinen mit $k \geq 2$, Java-Programme, usw. Es hat sich aber herausgestellt, dass all diese Modelle *äquivalent* sind, d.h. die gleiche Klasse von Funktionen berechnen (bzw. Problemen entscheiden). Außerdem ist es bisher nicht gelungen, ein formales Berechnungsmodell zu finden, so dass

- die berechneten Funktionen noch intuitiv berechenbar erscheinen,
- Funktionen berechnet werden können, die nicht in den oben genannten Modellen ebenfalls 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. Man spricht hier von einer *These* und nicht von einem *Satz*, da es nicht möglich ist, diese 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 große Vielzahl existierender Berechnungsmodelle, die sich als äquivalent herausgestellt haben.

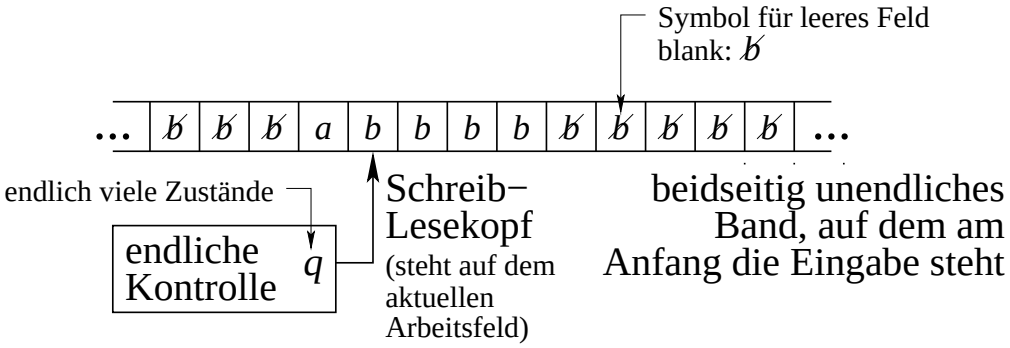
Im Teil III betrachten wir:

- Turingmaschinen
- Zusammenhang zwischen Turingmaschinen und Grammatiken
- Primitiv rekursive Funktionen und LOOP-Programme
- μ-rekursive Funktionen und WHILE-Programme
- (Partielle) Entscheidbarkeit, Aufzählbarkeit und deren Zusammenhänge
- Universelle Turingmaschinen und unentscheidbare Probleme
- Weitere unentscheidbare Probleme

11. Turingmaschinen

Turingmaschinen wurden um 1936 von dem englischen Mathematiker und Informatiker Alan Turing als besonders einfaches Berechnungsmodell vorgeschlagen. In den Worten von 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 Turingmaschine 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 ß. Das Verhalten der Turingmaschine hängt ab vom aktuellen Zustand und von Alphabetssymbol, das sich unter dem Schreib-Lesekopf findet. Ein Schritt der Maschine besteht darin, das Zeichen unter dem Schreib-Lesekopf zu ersetzen und dann den Kopf nach rechts oder links (oder gar nicht) zu bewegen.

Definition 11.1 (Turingmaschine)
Eine *Turingmaschine über dem Eingabealphabet* Σ hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, wobei

- Q endliche *Zustandsmenge* ist,
- Σ das *Eingabealphabet* ist,
- Γ das *Arbeitsalphabet* ist mit $\Sigma \subseteq \Gamma$, ß $\in \Gamma \setminus \Sigma$,
- $q_0 \in Q$ der *Anfangszustand* ist,
- $F \subseteq Q$ die *Endzustandsmenge* 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 $\mathcal{A}$

- das Symbol a durch a' ersetzen,
- in den Zustand q' gehen und
- den Schreib-Lesekopf entweder um ein Feld nach rechts (r), links (l) oder nicht (n) bewegen.

Die Maschine $\mathcal{A}$ heißt *deterministisch*, falls es für jedes Tupel $(q, a) \in Q \times \Gamma$ höchstens ein Tupel der Form $(q, a, \ldots, \ldots) \in \Delta$ 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 $\alpha,\ \beta$ 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 zu Beginn des Abschnitts wird durch die Konfiguration *aqbbbb*, aber auch durch $\not b \not b aqbbbb \not b$ beschrieben.

Formal werden Zustandsübergänge durch die Relation "$\vdash_{\mathcal{A}}$" 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

$$\begin{array}{rcll}
\alpha q a \beta & \vdash_{\mathcal{A}} & \alpha a' q' \beta & \text{falls} \quad (q, a, a', r, q') \in \Delta \\
\alpha q & \vdash_{\mathcal{A}} & \alpha a' q' & \text{falls} \quad (q, \not b, a', r, q') \in \Delta \\
\alpha b q a \beta & \vdash_{\mathcal{A}} & \alpha q' b a' \beta & \text{falls} \quad (q, a, a', l, q') \in \Delta \\
q a \beta & \vdash_{\mathcal{A}} & q' b a' \beta & \text{falls} \quad (q, \not b, a', l, q') \in \Delta \\
\alpha q a \beta & \vdash_{\mathcal{A}} & \alpha q' a' \beta' & \text{falls} \quad (q, a, a', n, q') \in \Delta
\end{array}$$

Weitere Bezeichnungen:

- Gilt $k \vdash_{\mathcal{A}} k'$, 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 $\mathcal{A}$ ist eine endliche oder unendliche Konfigurationsfolge

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} k_2 \vdash_{\mathcal{A}} \cdots$$

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 formalisiert beide Anwendung von Turingmaschinen: das Berechnen von Funktionen und das Erkennen von Sprachen. Im ersten Fall steht die Eingabe $(w_1, \ldots, w_n)$ in Form des Wortes $w_1 \not b w_2 \not b \cdots \not b w_n$ auf dem Band, wobei sich der Kopf zu Anfang auf dem ersten (linkeststehenden) Symbol von w_1 befindet. Nachdem die Maschine eine Stoppkonfiguration erreicht hat, findet sich die Ausgabe ab der Kopfposition bis zum ersten Symbol aus $\Gamma \setminus \Sigma$. Beim Erkennen von Sprachen steht das Eingabewort w auf dem Band und der Kopf befindet sich anfangs über dem ersten Symbol von w.

Definition 11.2 (Turing-berechenbar, Turing-erkennbar)

1) Die partielle Funktion $f : (\Sigma^*)^n \to \Sigma^*$ heißt *Turing-berechenbar*, falls es eine DTM $\mathcal{A}$ gibt mit
   a) der Definitionsbereich $\mathrm{dom}(f)$ von f besteht aus genau den Tupeln $(w_1, \ldots, w_n) \in (\Sigma^*)^n$ so dass $\mathcal{A}$ ab der Konfiguration
   $$k_0 = q_0 w_1 \not b w_2 \not b \ldots \not b\, w_n$$
   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 $\mathcal{A}$ *erkannte Sprache* ist
   $$L(\mathcal{A}) = \{w \in \Sigma^* \mid q_0 w \vdash_{\mathcal{A}}^* k, \text{ wobei } k \textit{ akzeptierende Stoppkonfiguration} \text{ ist}\}.$$
   Eine Sprache $L \subseteq \Sigma^*$ heißt *Turing-erkennbar*, falls es eine NTM $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

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 $\Sigma = \{a, b, c\}$. Endzustände werden 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 definiertheit*.
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-Endzustand oder sie stoppt 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:

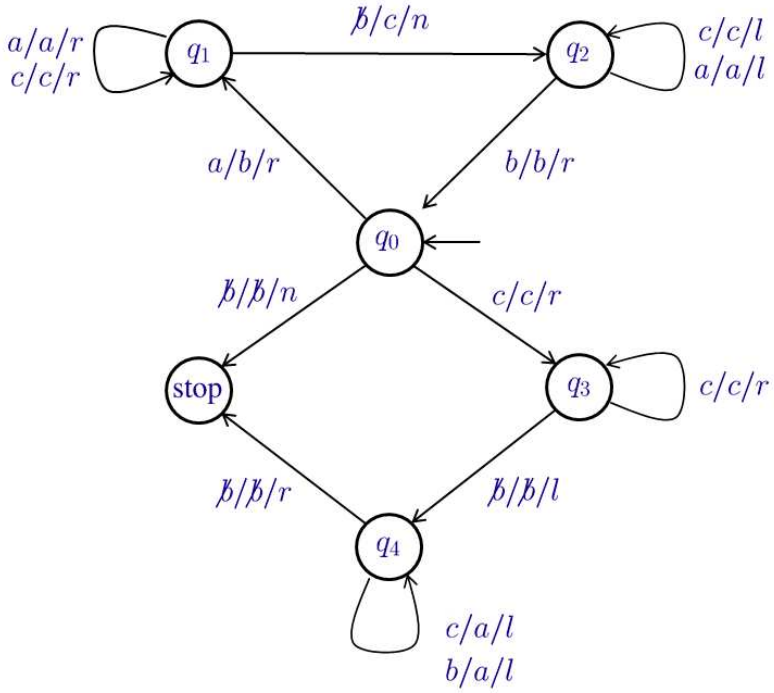
- Ersetze das erste a durch b,
- laufe nach rechts bis zum ersten blank, ersetze dieses durch c,
- laufe zurück bis zum zweiten a (unmittelbar rechts vom b), ersetze dieses durch b,
- laufe nach rechts bis zum ersten blank etc.
- Sind alle a's aufgebraucht, so ersetze noch die b's und c's wieder durch a's.

Dies wird durch die folgende *Übergangstafel* realisiert:

$(q_0, \not b, \not b, n, \text{stop})$,	$2 \cdot 0 = 0$
(q_0, a, b, r, q_1),	ersetze a durch b $\quad(\star)$
(q_1, a, a, r, q_1), (q_1, c, c, r, q_1),	laufe nach rechts über a's und bereits geschriebene c's
$(q_1, \not b, c, n, q_2)$,	schreibe weiteres c
(q_2, c, c, l, q_2), (q_2, a, a, l, q_2),	laufe zurück über c's und a's
(q_2, b, b, r, q_0), (q_0, c, c, r, q_3),	bei erstem b eins nach rechts und weiter wie $(\star)$ oder alle a's bereits ersetzt
(q_3, c, c, r, q_3), $(q_3, \not b, \not b, l, q_4)$,	laufe nach rechts bis Ende der c's letztes c erreicht
(q_4, c, a, l, q_4), (q_4, b, a, l, q_4),	ersetze c's und b's durch a's
$(q_4, \not b, \not b, r, \text{stop})$	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 $\alpha\text{stop}\beta$ eine Stoppkonfiguration.

In graphischer Darstellung stellen wir obige Turingmaschine wie folgt dar. Hierbei 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 Endzustände durch einen Doppelkreis. Da obige DTM eine Funktion berechnet (im Gegensatz zu: eine Sprache erkennt), spielen die Endzustände 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 Übergangstafel voll 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';
- 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 $\not b$ 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.

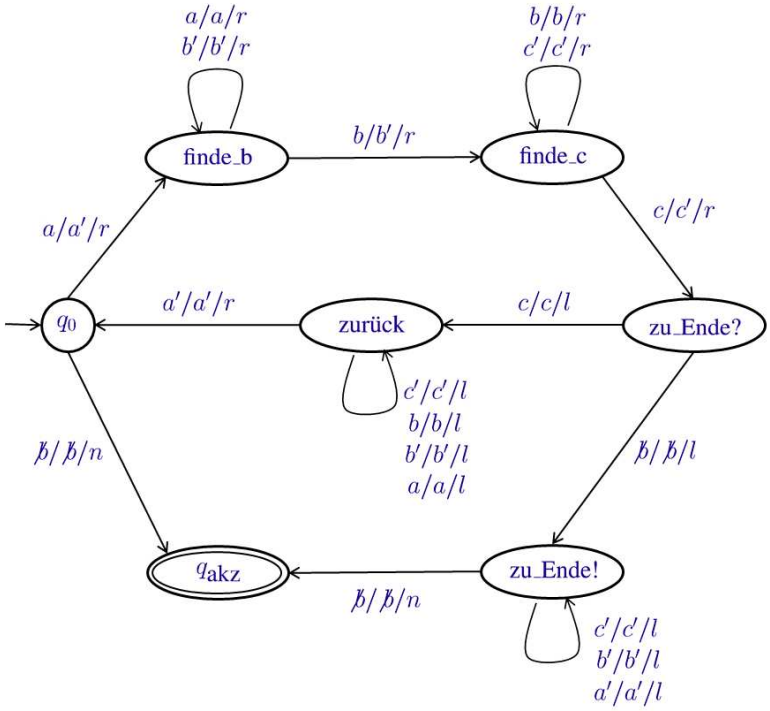
1. die Eingabe dieselbe Anzahl a's wie b's wie c's hat (denn es wurde jeweils dieselbe Anzahl ersetzt und danach waren keine a's, b' und c's mehr übrig);
2. in der Eingabe alle a's vor b's vor c's stehen.

Im Detail definiert man die Turingmaschine $\mathcal{A}$ wie folgt:
$\mathcal{A} = (\{q_0, q_{akz}, finde\_b, finde\_c, zu\_Ende\_?, zu\_Ende\_!, zurück\},$
$\{a, b, c\},$
$\{a, a', b, b', c, c', \not b\},$
$q_0, \Delta, \{q_{akz}\})$ mit $\Delta =$

$\{(q_0,$	$\not b,$	$\not b,$	$N,$	$q_{akz}),$
$(q_0,$	$a,$	$a',$	$R,$	$finde\_b),$
$(finde\_b,$	$a,$	$a,$	$R,$	$finde\_b),$
$(finde\_b,$	$b',$	$b',$	$R,$	$finde\_b),$
$(finde\_b,$	$b,$	$b',$	$R,$	$finde\_c),$
$(finde\_c,$	$b,$	$b,$	$R,$	$finde\_c),$
$(finde\_c,$	$c',$	$c',$	$R,$	$finde\_c),$
$(finde\_c,$	$c,$	$c',$	$R,$	$zu\_Ende\_?),$
$(zu\_Ende\_?,$	$c,$	$c,$	$L,$	$zurück),$
$(zurück,$	$c',$	$c',$	$L,$	$zurück),$
$(zurück,$	$b,$	$b,$	$L,$	$zurück),$
$(zurück,$	$b',$	$b',$	$L,$	$zurück),$
$(zurück,$	$a,$	$a,$	$L,$	$zurück),$
$(zurück,$	$a',$	$a',$	$R,$	$q_0),$
$(zu\_Ende\_?,$	$\not b,$	$\not b,$	L	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$c',$	$c',$	$L,$	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$b',$	$b',$	$L,$	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$a',$	$a',$	$L,$	$zu\_Ende\_!),$
$(zu\_Ende\_!,$	$\not b,$	$\not b,$	$N,$	$q_{akz})\}$

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 Endzustand 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 Turingmaschine 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-Leseköpfen

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 11.1 eingeführten 1-Band-TM.

Definition 11.3 (k-Band-TM)
Eine *k-Band-NTM* hat die Form $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ mit

- $Q, \Sigma, \Gamma, q_0, F$ wie in Definition 11.1 und
- $\Delta \subseteq Q \times \Gamma^k \times \Gamma^k \times \{r, l, n\}^k \times Q$.

Dabei bedeutet $(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$:

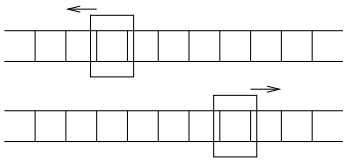
- Vom Zustand q aus
- mit $a_1, \ldots, a_k$ auf den Arbeitsfeldern der k Bänder

kann $\mathcal{A}$

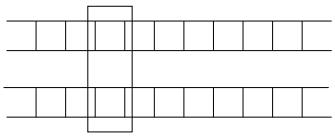
- das Symbol a_i auf dem i-ten Band durch b_i ersetzen,
- in den Zustand q' gehen und
- die Schreib-Leseköpfe der Bänder entsprechend d_i bewegen.

Das erste Band wird (o.B.d.A.) als Ein- und Ausgabeband verwendet.

Beachte:
Wichtig ist hier, dass sich die Köpfe der verschiedenen Bänder unabhängig bewegen können:



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 11.1) verwendet, die als Bandalphabet Γ^k statt Γ hat.

Offenbar kann man jede 1-Band-NTM (nach Definition 11.1) durch eine k-Band-NTM ($k > 1$) simulieren, indem man nur das erste Band wirklich verwendet. Der nächste Satz zeigt, dass auch die Umkehrung gilt:

Satz 11.4
Wird die Sprache L durch eine k-Band-NTM erkannt, so auch durch eine 1-Band-NTM.

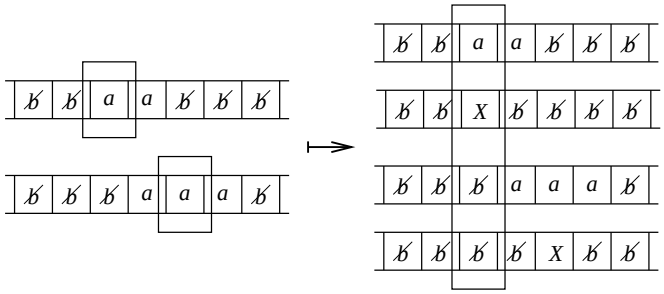
Beweis. Es sei $\mathcal{A}$ eine k-Band-NTM. Gesucht ist eine 1-Band-NTM $\mathcal{A}'$ mit $L(\mathcal{A}) = L(\mathcal{A}')$.

Arbeitsalphabet von $\mathcal{A}'$: $\Gamma^{2k} \cup \Sigma \cup \{\not b\}$;

- $\Sigma \cup \{\not b\}$ wird für Eingabe benötigt;
- Γ^{2k} sorgt dafür, dass $2k$ Spuren auf dem Band von $\mathcal{A}'$ zur Verfügung stehen.

Idee: Jeweils 2 Spuren kodieren ein Band von $\mathcal{A}$.

- 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 TM $\mathcal{A}'$ macht jeweils mehrere Schritte, um einen einzelnen Schritt von $\mathcal{A}$ zu simulieren. Im Detail arbeitet $\mathcal{A}'$ wie folgt.

Initialisierung: Zunächst wird die Anfangskonfiguration

$$q_0 a_1 \dots a_m$$

von $\mathcal{A}'$ in die Repräsentation der entsprechenden Anfangskonfiguration von $\mathcal{A}$ umgewandelt (durch geeignete Übergänge):

$\not b$	$\not b$	a_1	a_2	$\dots$	a_m	$\not b$	Spur 1 und 2 kodieren
$\not b$	$\not b$	X	$\not b$	$\dots$	$\not b$	$\not b$	Band 1
$\not b$	$\not b$	$\not b$	$\not b$	$\dots$	$\not b$	$\not b$	Spur 3 und 4 kodieren
$\not b$	$\not b$	X	$\not b$	$\dots$	$\not b$	$\not b$	Band 2
		$\vdots$					

Simulation eines Überganges von $\mathcal{A}'$:

- Von links nach rechts suche die mit X markierten Felder. Dabei merke man sich (im Zustand von $\mathcal{A}'$) die Symbole, die jeweils über dem X stehen. Außerdem zählt man (im Zustand von $\mathcal{A}'$), wie viele X man schon gelesen hat, um festzustellen, wann das k-te (und letzte) X erreicht ist. Man ermittelt so das aktuelle Tupel $(a_1, \dots, a_k)$.
- entscheide nichtdeterministisch, welcher Übergang

  $$(q, (a_1, \dots, a_k), (b_1, \dots, b_k), (d_1, \dots, d_k), q') \in \Delta$$

  von $\mathcal{A}$ 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 $\mathcal{A}'$ im Prinzip dieselben Zustände wir $\mathcal{A}$. Also stoppt $\mathcal{A}$ auf Eingabe w in akzeptierender Stoppkonfiguration gdw. $\mathcal{A}'$ in akzeptierender Stoppkonfiguration stoppt. □

Bemerkung 11.5.

1) War $\mathcal{A}$ 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 $\mathcal{A}$ in Stoppkonfiguration ist) in der Maschine $\mathcal{A}'$ noch die Ausgabe geeignet aufbereiten.

Wegen Satz 11.4 und Bemerkung 11.5 können wir von nun an ohne Beschränkung der Allgmeinheit 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 bei Turing-erkennbaren Sprachen machen kann, ohne an Ausdrucksstärke zu verlieren.

Satz 11.6

Zu jeder NTM gibt es eine DTM, die dieselbe Sprache erkennt.

Beweis. Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ eine NTM. Wegen Satz 11.4 und Bemerkung 11.5 genügt es, eine deterministische 3-Band-Turingmaschine $\mathcal{A}'$ zu konstruieren, so dass $L(A) = L(\mathcal{A}')$.

Die Maschine $\mathcal{A}'$ soll für wachsendes n auf dem dritten Band jeweils alle Berechnungen

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} k_2 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_n$$

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

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_n$$

(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 $\{1, \ldots, r\}$ auf und erzeugt zu jedem Wort $i_1 \ldots i_n$ die zugehörige Berechnung, so erhält man eine Aufzählung aller endlichen Berechnungen.

$\mathcal{A}'$ realisiert dies auf den drei Bändern wie folgt:

- Auf Band 1 bleibt die Eingabe gespeichert.
- Auf dem zweiten Band werden sukzessive alle Wörter $i_1 \dots i_n \in \{1, \dots, r\}^*$ erzeugt (z.B. durch normales Zählen zur Basis r).
- Für jedes dieser Wörter wird auf dem dritten Band die zugehörige Berechnung erzeugt (wenn sie existiert). Erreicht man hierbei eine akzeptierende Stoppkonfiguration von $\mathcal{A}$, so geht auch $\mathcal{A}'$ in eine akzeptierende Stoppkonfiguration.

□

12. Zusammenhang zwischen Turingmaschinen und Grammatiken

Wir zeigen zunächst den Zusammenhang zwischen *Typ-0-Sprachen* und *Turing-erkennbaren Sprachen*. Dieser 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 12.1

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 geben eine 2-Band-NTM an, die $L(G)$ erkennt (und nach Satz 11.4 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 man in akzeptierende Stoppkonfiguration, sonst werden weitere Ableitungsschritte vorgenommen.

Die Maschine geht dabei wie folgt vor:

1) Schreibe S auf Band 2, gehe nach links auf das $\not b$ 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 $\alpha \longrightarrow \beta$ aus P aus, die angewendet werden soll

4) Überprüfe, ob α tatsächlich die Beschriftung des Bandstücks der Länge $|\alpha|$ ab der gewählten Bandstelle ist.

5) Falls der Test erfolgreich war, so ersetze α durch β.

   Vorher müssen bei $|\alpha| < |\beta|$ die Symbole rechts von α um $|\beta| - |\alpha|$ Positionen nach rechts

   bzw. bei $|\alpha| > |\beta|$ um $|\alpha| - |\beta|$ Positionen nach links geschoben werden.

6) Gehe nach links bis zum ersten $\not b$ und vergleiche, ob die Beschriftung auf dem Band 1 mit der auf Band 2 übereinstimmt.

7) Wenn ja, so gehe in akzeptierenden Stoppzustand. Sonst fahre fort bei 2).

„$\Leftarrow$“. Es sei $L = L(\mathcal{A})$ für eine NTM $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$.

Wir konstruieren eine Grammatik G, die jedes Wort $w \in L(\mathcal{A})$ wie folgt erzeugt:

1. Phase: Erst wird w mit „genügend vielen“ $\not b$-Symbolen links und rechts davon erzeugt (dies passiert für jedes w, auch für $w \notin L(\mathcal{A})$).

„Genügend viele“ bedeutet dabei: so viele, wie $\mathcal{A}$ beim Akzeptieren von w vom Arbeitsband benötigt.

2. Phase: Auf dem so erzeugten Arbeitsband simuliert G die Berechnung von $\mathcal{A}$ 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 Tupel aus

$$(\Sigma \cup \{\not b\}) \times \Gamma$$

wobei das Tupel $[a, b]$ zwei Zwecken dient:

- in der ersten Komponente merkt man sich die ursprüngliche Eingabe w und
- in der zweiten Komponente simuliert man die Berechnung (die die Eingabe ja überschreiben kann).

Formale Definition:

$$N = \{S, A, B, E\} \cup Q \cup ((\Sigma \cup \{\not b\}) \times \Gamma),$$

wobei

- S, A, B zum Aufbau des Rechenbandes am Anfang,
- E zum Löschen am Schluss,
- Q zur Darstellung des aktuellen Zustandes,
- $\Sigma \cup \{\not b\}$ zum Speichern von w und
- Γ zur $\mathcal{A}$-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 $a_1 \dots a_n \in \Sigma^*, k, l, \geq 0$:

$$S \vdash_G^* [\not b, \not b]^k q_0 [a_1, a_1] \dots [a_n, a_n] [\not b, \not b]^l$$

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 die TM 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 $(q,b,\dots,\dots) \in \Delta$ 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:

$$w \in L(G) \text{ gdw. } \mathcal{A} \text{ akzeptiert } w. \qquad \square$$

Für Typ-0-Sprachen gelten die folgenden Abschlusseigenschaften:

Satz 12.2

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 den 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 Turing-erkennbaren Sprachen nicht unter Komplement abgeschlossen sind (Satz 16.10). $\square$

Wir werden später außerdem zeigen, dass für Turing-erkennbaren Sprachen (und damit für $\mathcal{L}_0$) alle bisher betrachteten Entscheidungsprobleme unentscheidbar sind (Sätze 16.6, 16.8, 16.9). Die Begriffe "entscheidbar" und "unentscheidbar" werden wir in Kürze formal definieren. Intuitiv bedeutet Unentscheidbarkeit, dass es keinen Algorithmus gibt, der das Problem löst.

Satz 12.3

Für $\mathcal{L}_0$ sind das Leerheitsproblem, das Wortproblem und das Äquivalenzproblem unentscheidbar.

Von den Sprachklassen aus der Chomsky-Hierarchie sind nun alle bis auf $\mathcal{L}_1$ (kontextsensitiv) durch geeignete Automaten/Maschinenmodelle charakterisiert. Nach Definition enthalten kontextsensitive 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 wie 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 12.4 (linear beschränkter Automat)

Ein *linear beschränkter Automat (LBA)* ist eine NTM $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$, so dass

- $\$, \not\in \in \Gamma \setminus \Sigma$
- Übergänge $(q, \not\in, \ldots)$ sind nur in der Form $(q, \not\in, \not\in, r, q')$ erlaubt (linker Rand darf nicht überschritten werden).
- Übergänge $(q, \$, \ldots)$ 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 ein LBA nicht unbedingt terminieren. Wie bei Turingmaschinen gilt nach der obigen Definition: terminiert ein LBA $\mathcal{A}$ auf einer gegebenen Eingabe w nicht, so ist $w \notin L(\mathcal{A})$.

Korollar 12.5

Eine Sprache L gehört zu $\mathcal{L}_1$ gdw. sie von einem LBA erkannt wird.

Beweis.

„⇒“: Verwende die Konstruktion aus dem Beweis von Satz 12.1.

Da alle Produktionen von kontextsensitiven Grammatiken nichtkürzend 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, vergleiche Beweis von Satz 11.4.

„⇐“: Durch Modifikation der Beweisidee von Satz 12.1 gelingt es, zu einem LBA eine Grammatik zu konstruieren, die nur nichtkürzende Regeln hat.

Idee:

Da man mit $|w|$ Arbeitsfeldern auskommt, muss man keine $[\not b, \not b]$ links und rechts von w erzeugen. Dadurch fallen dann auch die folgenden kürzenden Regeln weg:

$$E[\not b, \not b] \longrightarrow E$$

$$[\not b, \not b]E \longrightarrow E$$

Es gibt allerdings noch einige technische Probleme:

- Man muss die Randmarker $\not\in$ und \$ einführen und am Schluss löschen.
- Man muss das Hilfssymbol E und den Zustand q löschen.

Lösung:

Führe die Symbole $\not\in$, \$ sowie E und den Zustand q nicht als zusätzliche Symbole ein, sondern kodiere sie in die anderen Symbole hinein.

z.B. statt $[a, b]q[a', b'][a'', b'']$ verwende $[a, b][q, a', b'][a'', b'']$.

Basierend auf dieser Idee kann man die Konstruktion aus dem Beweis von Satz 12.1 so modifizieren, dass eine kontextsensitive Grammatik erzeugt wird.

□

Satz 12.6

$\mathcal{L}_1$ 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). □

Für LBAs ist bisher nicht bekannt, ob deterministische LBAs genauso stark wie nichtdeterministische LBAs sind.

Satz 12.7

Für $\mathcal{L}_1$ sind

1. *das Wortproblem entscheidbar*
2. *das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.*

Beweis. (1) Da kontextsensitive Produktionen (bis auf Spezialfall $S \longrightarrow \varepsilon$) nichtkürzend sind, muss man zur Entscheidung „$w \in L(G)$?“ nur die Menge aller aus S ableitbaren Wörter aus $(N \cup \Sigma)^*$ 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 17.6).

□

13. Primitiv rekursive Funktionen und Loop-Programme

In diesem und dem folgenden Abschnitt betrachten wir zwei weitere Arten von Berechnungsmodellen:

- Funktionale Modelle, bei denen nicht der Mechanismus der Berechnung, sondern die Funktionen selbst im Mittelpunkt stehen
- Berechnungsmodelle, die als Abstraktion von imperativen Programmiersprachen verstanden werden können.

Dies dient zwei Zwecken: erstens werden wir Berechnungsmodelle dieser Art identifizieren, die identische Berechnungsstärke haben (also dieselben Funktionen berechnen können) und zudem auch dieselbe Berechnungsstärke wie Turingmaschinen aufweisen. Dies ist ein gutes Indiz für die Gültigkeit der Church-Turing These. Zweitens erlauben uns die erzielten Resultate, sehr präzise diejenigen Operationen auf Funktionen bzw. Elemente von imperativen Programmiersprachen zu identifizieren, die für die Berechnungsvollständigkeit (Äquivalenz zu Turingmaschinen) verantwortlich sind. Wir trennen sie damit von bloßem "Beiwerk" zu trennen, das zwar angenehm für die Programmierung ist, aber nicht wirklich zur Berechnungsstärke beiträgt.

Wir gehen in zwei Schritten vor. Die in diesem Abschnitt eingeführten Berechnungsmodelle sind *nicht* Berechnungsvollständig, also nicht stark genug, um alle (Turing-)berechenbaren Funktionen zu erfassen. Wir werden im Abschnitt 14 zeigen, wie man die Modelle erweitern muss, um berechnungsvollständige Modelle zu erhalten.

Wir betrachten hier nur Funktionen von

$$\mathbb{N}^n \to \mathbb{N}.$$

Dies entspricht dem Spezialfall $|\Sigma| = 1$ bei Wortfunktionen, ist aber keine echte Einschränkung, da es berechenbare Kodierungsfunktionen gibt, d.h.

$$\pi : \Sigma^* \to \mathbb{N} \quad \text{bijektiv}$$

mit π und π^{-1} berechenbar.

Definition 13.1 (Grundfunktionen)
Die folgenden Funktionen sind *primitiv rekursive Grundfunktionen*:

1) $\mathrm{s} : \mathbb{N} \to \mathbb{N}$ mit $x \mapsto x + 1$ *(Nachfolgerfunktion)*
2) Für alle $n \geq 0$ und i, $1 \leq i \leq n$:
   - $\pi_i^{(n)} : \mathbb{N}^n \to \mathbb{N}$ mit $(x_1, \ldots, x_n) \mapsto x_i$ *(Projektion)*
   - $\mathrm{null}^{(n)} : \mathbb{N}^n \to \mathbb{N}$ mit $(x_1, \ldots, x_n) \mapsto 0$ *(Nullfunktion)*

Beachte, dass es die Projektion und die Nullfunktion jeweils mit beliebiger Stelligkeit gibt. Aus diesen einfachen Funktionen kann man mit Hilfe von Operatoren komplexere Funktionen aufbauen.

Definition 13.2 (Komposition)
Die Funktion $\mathrm{f} : \mathbb{N}^n \to \mathbb{N}$ entsteht aus

$$\begin{aligned} \mathrm{g} &: \mathbb{N}^m \to \mathbb{N} \text{ und} \\ \mathrm{h}_1, \ldots, \mathrm{h}_m &: \mathbb{N}^n \to \mathbb{N} \end{aligned}$$

durch *Komposition*, falls für alle $(x_1, \ldots, x_n) \in \mathbb{N}^n$ gilt:

$$\mathrm{f}(x_1, \ldots, x_n) = \mathrm{g}(\mathrm{h}_1(x_1, \ldots, x_n), \ldots, \mathrm{h}_m(x_1, \ldots, x_n)).$$

Wendet man Komposition auf echt partielle Funktionen an, so gilt:
$\mathrm{f}(x_1, \ldots, x_n)$ ist *undefiniert* gdw.

- eines der $\mathrm{h}_i(x_1, \ldots, x_n)$ ist *undefiniert* oder
- alle h_i-Werte sind *definiert*, aber g von diesen Werten ist *undefiniert*.

Beispiel:
$\mathrm{g}(x, y) = x$, $\mathrm{h}_1(0) = 0$, $\mathrm{h}_2(0)$ undefiniert.
Dann ist $\mathrm{g}(\mathrm{h}_1(0), \mathrm{h}_2(0))$ undefiniert Dies entspricht der call-by-value-Auswertung von Programmiersprachen: der Wert von $\mathrm{h}_2(0)$ muss als Wert übergeben werden (und darum auch existieren), auch wenn er dann gar nicht verwendet wird.

Definition 13.3 (primitive Rekursion)
Es sei $n \geq 0$. Die Funktion $\mathrm{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ entsteht aus

$$\begin{aligned} \mathrm{g} &: \mathbb{N}^n \to \mathbb{N} \text{ und} \\ \mathrm{h} &: \mathbb{N}^{n+2} \to \mathbb{N} \end{aligned}$$

durch *primitive Rekursion*, falls gilt:

- $\mathrm{f}(x_1, \ldots, x_n, 0) = \mathrm{g}(x_1, \ldots, x_n)$
- $\mathrm{f}(x_1, \ldots, x_n, y+1) = \mathrm{h}(x_1, \ldots, x_n, \mathrm{f}(x_1, \ldots, x_n, y), y)$

Beachte, dass das letzte Argument per Definition das "Rekursionsargument" ist. Die Funktion g definiert den Rekursionsanfang, h definiert den Rekursionsschritt. Bei der Berechnung des Rekursionsschrittes hat man zur Verfügung: alle Argumente inklusive Rekursionsargument sowie den per rekursivem Aufruf ermittelten Wert. Auch bei der primitiven Rekursion setzen sich wie bei der Komposition undefinierte Werte fort.

Beispiel 13.4 (Addition)
Die Addition natürlicher Zahlen kann durch primitive Rekursion wie folgt definiert werden:

$$\begin{aligned} \mathrm{add}(x, 0) &= x && = \mathrm{g}(x) \\ \mathrm{add}(x, y+1) &= \mathrm{add}(x, y) + 1 && = \mathrm{h}(x, \mathrm{add}(x, y), y) \end{aligned}$$

Das heißt also: add entsteht durch primitive Rekursion aus den Funktionen

- $g : \mathbb{N} \to \mathbb{N}$ mit $x \mapsto x$,
  d.h. $g = \pi_1^{(1)}$ ist Grundfunktion,
- $h : \mathbb{N}^3 \to \mathbb{N}$ mit $(x, z, y) \mapsto z + 1$,
  d.h. $h(x, z, y) = s(\pi_2^{(3)}(x, z, y))$.
  Also entsteht h durch Komposition aus Grundfunktionen.

Definition 13.5 (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.

Offenbar sind die Grundfunktionen *total* und die Operationen Komposition und primitive Rekursion erzeugen aus totalen Funktionen wieder totale.

Satz 13.6
Die Klasse der primitiv rekursiven Funktionen enthält nur totale Funktionen.

Trotzdem sind aber die Operationen primitive Rekursion und Komposition auch für partielle Funktionen definiert, und wir werden sie später auch auf partielle Funktionen anwenden.

Beispiel 13.4 zeigt, dass add zur Klasse der primitiv rekursiven Funktionen gehört. Wir betrachten nun weitere Beispiele für primitiv rekursive Funktionen.

Multiplikation erhält man durch primitive Rekursion aus der Addition:

$$\begin{aligned} \text{mult}(x, 0) &= 0 && = \text{null}^{(1)}(x) \\ \text{mult}(x, y+1) &= \text{add}(x, \text{mult}(x, y)) && = \text{add}(\pi_1^{(3)}, \pi_2^{(3)})(x, \text{mult}(x, y), y) \end{aligned}$$

Exponentiation erhält man durch primitive Rekursion aus der Multiplikation:

$$\begin{aligned} \exp(x, 0) &= 1 && = s(\text{null}^{(1)}(x)) \\ \exp(x, y+1) &= \text{mult}(x, \exp(x, y)) \end{aligned}$$

Betrachte die Vorgänger-Funktion

$$\text{pred} : \mathbb{N} \to \mathbb{N} \text{ mit } x \mapsto x \dot{-} 1 := \begin{cases} x - 1 & x > 0 \\ 0 & x = 0 \end{cases}$$

Diese Funktion "schneidet bei 0 ab", um den Zahlenraum $\mathbb{N}$ nicht zu verlassen. Die Funktion pred ist ebenfalls primitiv rekursiv, denn sie kann mittels primitiver Rekursion definiert werden (das einzige Argument ist das Rekursionsargument):

$$\begin{aligned} \text{pred}(0) &= 0 && = \text{null}^{(0)}(), \\ \text{pred}(y+1) &= y && = \pi_2^{(2)}(\text{pred}(y), y) \end{aligned}$$

In obiger Definition verwenden wir die primitive Rekursion, um eine Fallunterscheidung zu realisieren. Wir werden das später noch genauer betrachten.

Übung:
Zeige, dass

$$\text{sub} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x, y) \mapsto x \dot{-} y := \begin{cases} x - y & x \geq y \\ 0 & \text{sonst} \end{cases}$$

primitiv rekursiv ist.

Beispiel 13.7
Aus den bisher betrachteten Funktionen erhält man damit die Funktion

$$\mathrm{c} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x, y) \mapsto 2^x \cdot (2y + 1) \dot{-} 1$$

durch Komposition, d.h. c ist primitiv rekursiv. Diese Funktion ist interessant, da sie eine Bijektion von $\mathbb{N}^2 \to \mathbb{N}$ ist, d.h. man kann mit ihr Tupel natürlicher Zahlen in eine natürliche Zahl kodieren. Derartige Kodierungen werden sich später in Beweisen als sehr nützlich erweisen.

Lemma 13.8
Die Funktion c *aus Beispiel 13.7 ist eine Bijektion.*

Beweis.

Surjektivität: Es sei $z \in \mathbb{N}$. Dann betrachten wir die größte Zweierpotenz 2^x, die $z + 1$ teilt. Offenbar ist dann $\frac{z+1}{2^x}$ eine ungerade Zahl, d.h. es gibt ein y mit

$$\frac{z+1}{2^x} = 2y + 1.$$

Damit ist $z = 2^x \cdot (2y + 1) \dot{-} 1 = \mathrm{c}(x, y)$.

Injektivität: Offenbar ist die größte Zweierpotenz, die $z + 1$ teilt, eindeutig, d.h. x ist eindeutig durch z bestimmt. Damit ist aber auch y eindeutig bestimmt. □

Da c eine Bijektion ist, gibt es die Umkehrfunktionen c_0 und c_1 mit der Eigenschaft $\mathrm{c}_0(\mathrm{c}(x, y)) = x$ und $\mathrm{c}_1(\mathrm{c}(x, y)) = y$. Diese ergeben sich im Prinzip aus dem Beweis von Lemma 13.8, d.h.

$$\mathrm{c}_0(z) = \max\{x \in \mathbb{N} \mid 2^x | (z + 1)\}$$
$$\mathrm{c}_1(z) = ((\frac{z+1}{2^{\mathrm{c}_1(z)}}) \dot{-} 1)/2$$

Um zu zeigen, dass c_0, c_1 ebenfalls primitiv rekursiv sind, benötigen wir noch etwas Vorarbeit. Insbesondere wollen wir zeigen, dass mittels primitiver Rekursion Fallunterscheidungen und eine sogenannte beschränkte Minimalisierung möglich sind.

Die Funktionen $\text{sign} : \mathbb{N} \to \mathbb{N}$ und $\overline{\text{sign}} : \mathbb{N} \to \mathbb{N}$ mit

$$\text{sign}(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0 \end{cases} \qquad \overline{\text{sign}}(x) = \begin{cases} 1 & x = 0 \\ 0 & x > 0 \end{cases}$$

lassen sich per primitiver Rekursion definieren, dabei ist das einzige Argument auch das Rekursionsargument:

$$\text{sign}(0) = 0 \qquad \text{sign}(y+1) = 1$$
$$\overline{\text{sign}}(0) = 1 \qquad \overline{\text{sign}}(1) = 0$$

Definition 13.9 (Fallunterscheidung)
Es seien $\text{g}_1, \text{g}_2, \text{h} : \mathbb{N}^n \to \mathbb{N}$ gegeben.

Die Funktion $\text{f} : \mathbb{N}^n \to \mathbb{N}$ entsteht daraus durch *Fallunterscheidung*, falls für alle $\underline{x} \in \mathbb{N}^n$ gilt:

$$\text{f}(\underline{x}) = \begin{cases} \text{g}_1(\underline{x}) & \text{falls} \quad \text{h}(\underline{x}) = 0 \\ \text{g}_2(\underline{x}) & \text{falls} \quad \text{h}(\underline{x}) > 0 \end{cases}$$

Lemma 13.10
Sind $\text{g}_1, \text{g}_2, \text{h}$ *primitiv rekursiv, so auch* f.

Beweis.

$$\text{f}(\underline{x}) = \text{g}_1(\underline{x}) \cdot \overline{\text{sign}}(\text{h}(\underline{x})) + \text{g}_2(\underline{x}) \cdot \text{sign}(\text{h}(\underline{x})).$$

□

Definition 13.11 (beschränkte Minimalisierung)
Es sei $n \geq 0$. Die Funktion $\text{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ entsteht aus $\text{g} : \mathbb{N}^{n+1} \to \mathbb{N}$ durch *beschränkte Minimalisierung*, falls gilt:

$$\text{f}(\underline{x}, y) = \begin{cases} j & \text{falls} \quad j = \min\{i \leq y \mid \text{g}(\underline{x}, i) = 0\} \text{ existiert} \\ y+1 & \text{sonst} \end{cases}$$

Wir schreiben dann $\text{f} = \overline{\mu}\text{g}$.

Die beschränkte Minimalisierung gibt also den kleinsten Wert $j \leq y$ zurück, so dass $\text{g}(x, j) = 0$. Der Zusatz "beschränkt" bezieht sich auf die Tatsache, dass wir nur Werte $j \leq y$, mit y gegeben, in Betracht ziehen. Wir werden später auch unbeschränkte Minimalisierung kennenlernen und sehen, dass sie nicht primitiv rekursiv ist, sondern im Gegenteil für Turing-Vollständigkeit sorgt.

Lemma 13.12
Ist g *primitiv rekursiv, so auch* $\overline{\mu}$g.

Beweis. Wir definieren $\overline{\mu}$g mittels primitiver Rekursion wie folgt:

1) $\overline{\mu}\text{g}(\underline{x}, 0) = \begin{cases} 0 & \text{falls} \quad \text{g}(\underline{x}, 0) = 0 \\ 1 & \text{sonst} \end{cases}$

   D.h. $\overline{\mu}\text{g}(\underline{x}, 0) = \text{sign}(\text{g}(\underline{x}, 0))$

2) $\overline{\mu}\mathrm{g}(\underline{x}, y+1) = \begin{cases} \overline{\mu}\mathrm{g}(\underline{x}, y) & \text{falls} \quad \overline{\mu}\mathrm{g}(\underline{x}, y) \leq y \text{ oder } \mathrm{g}(\underline{x}, y+1) = 0 \\ y+2 & \text{sonst} \end{cases}$

Die im Rekursionsschritt verwendete Funktion h ist also eine Fallunterscheidung, genauer gesagt

$$\mathrm{h}(\underline{x}, z, y) = \begin{cases} 0 & \text{falls} \quad z \leq y \text{ oder } \mathrm{g}(\underline{x}, y+1) = 0 \\ 1 & \text{sonst} \end{cases}$$

Es gilt

$$\mathrm{h}(\underline{x}, z, y) = \mathrm{sign}((z \dot{-} y) \cdot \mathrm{g}(\underline{x}, y+1))$$

also ist h und damit auch $\overline{\mu}\mathrm{g}$ primitiv rekursiv. □

Wir kommen nun zurück zu den Umkehrfunktionen der Bijektion aus Beispiel 13.7. Betrachten wir zunächst die Teilbarkeitsrelation in der Definition von c_0.

Lemma 13.13
Die Funktion

$$\mathrm{teilt} : \mathbb{N}^2 \to \mathbb{N} \textit{ mit } (x, y) \mapsto \begin{cases} 0 & x|y \\ 1 & \textit{sonst} \end{cases}$$

ist primitiv rekursiv.

Beweis. Betrachte die Funktion

$$\mathrm{h}(x, z, y) = \begin{cases} 0 & x \cdot z = y \\ 1 & \text{sonst.} \end{cases}$$

h ist primitiv rekursiv, da

$$\mathrm{h}(x, y, z) = \mathrm{sign}((x \cdot z \dot{-} y) + (y \dot{-} x \cdot z)).$$

Damit ist auch $\overline{\mu}\mathrm{h}$ primitiv rekursiv und somit auch $g(x, y) = \overline{\mu}h(x, y, y)$. Es gilt

$$\mathrm{g}(x, y) = \begin{cases} j & \text{falls} \quad j = \min\{i \leq y \mid x \cdot i = y\} \text{ existiert} \\ y+1 & \text{sonst} \end{cases}$$

Damit ist aber $\mathrm{teilt}(x, y) = \mathrm{g}(x, y) \dot{-} y$, denn

- wenn $x|y$, dann ist $\mathrm{g}(x, y)$ ein Teiler von y, der ist $\leq y$, also $\mathrm{g}(x, y) \dot{-} y = 0$
- wenn $x|y$ nicht gilt, dann $\mathrm{g}(x, y) = y + 1$, also $\mathrm{g}(x, y) \dot{-} y = 1$.

□

Lemma 13.14
Die Umkehrfunktion c_0 *der Funktion* c *aus Beispiel 13.7 ist primitiv rekursiv.*

Beweis. $\mathrm{c}_0(z) = \max\{x \in \mathbb{N} \mid 2^x | (z+1)\}$.

Wir suchen nach dem *kleinsten* $i \leq z+1$ mit

$$2^{(z+1)\dot{-}i} \mid (z+1)$$

d.h. nach dem kleinsten $i \leq z+1$ mit

$$\mathrm{f}(z,i) := \mathrm{teilt}(2^{(z+1)\dot{-}i}, z+1) = 0.$$

Dies gelingt durch beschränkte Minimalisierung, zusammenfassend:

$$c_0(z) = (z+1)\dot{-}\overline{\mu}\mathrm{f}(z, z+1).$$ □

Um zu zeigen, dass auch c_1 primitiv rekursiv ist, betrachten wir die ganzzahlige Division

$$\mathrm{div} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x,y) \mapsto \begin{cases} \lfloor x/y \rfloor & \text{falls} \quad y > 0 \\ x & \text{sonst} \end{cases}$$

wobei $\lfloor x/y \rfloor$ die größte natürliche Zahl unterhalb von x/y ist.

Beachte:
Ist $y > 0$ und x durch y teilbar, so ist $\mathrm{div}(x,y) = x/y$. In der Definition von c_1 sind diese Bedingungen erfüllt. Es ist daher

$$\mathrm{c}_2(z) = \mathrm{div}(\mathrm{div}(z+1, 2^{\mathrm{c}_1(z)})\dot{-}1, 2).$$

Lemma 13.15
Die Umkehrfunktion c_1 *der Funktion* c *aus Beispiel 13.7 ist primitiv rekursiv.*

Beweis. Es genügt zu zeigen, dass div primitiv rekursiv ist. Betrachte

$$\mathrm{f}(x,y,z) = \begin{cases} 0 & \text{falls} \quad z \cdot y > x \\ 1 & \text{falls} \quad z \cdot y \leq x \end{cases}$$

Diese ist primitiv rekursiv, da

$$\mathrm{f}(x,y,z) = \overline{\mathrm{sign}}(z \cdot y \dot{-} x).$$

Damit ist auch $\overline{\mu}\mathrm{f}$ primitiv rekursiv und somit auch $\mathrm{g}(x,y) := \overline{\mu}\mathrm{f}(x,y,x)$. Es gilt

$$\mathrm{g}(x,y) = \begin{cases} \text{das kleinste } i \leq x \text{ mit } i \cdot y > x & \text{falls existent} \\ x+1 & \text{sonst} \end{cases}$$

Daraus folgt $\mathrm{div}(x,y) = \mathrm{g}(x,y)\dot{-}1$, denn

- ist $y > 1$, so existiert so ein i stets (nämlich $i = x$) und es ist $\mathrm{div}(x,y) = i - 1$.
- ist $y = 1$ oder $y = 0$, so existiert so ein i nicht, d.h. $x+1$ wird ausgegeben. In diesen Fällen ist aber auch $\mathrm{div}(x,y) = x$. □

Wir betrachten nun eine einfache imperative Programmiersprache, die genau die primitiv rekursiven Funktionen berechnen kann.

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`

Definition 13.16 (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

$$P_1 \texttt{;} P_2 \quad \textit{(Hintereinanderausführung)}$$

ein LOOP-Programm.

3) Falls P ein LOOP-Programm ist und $i \geq 0$, so ist auch

$$\texttt{LOOP } x_i \texttt{ DO } P \texttt{ END}$$

ein LOOP-Programm.

Die *Semantik* dieser einfachen Sprache ist wie folgt definiert:

Bei einem LOOP-Programm, das eine 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.
- Alle anderen Variablen erhalten den Wert 0.
- *Ausgabe* ist der Wert der Variablen 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.

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 sooft 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.

Definition 13.17 (LOOP-berechenbar)
Die Funktion

$$\mathrm{f} : \mathbb{N}^k \to \mathbb{N}$$

heißt *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das f in dem 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 $\mathrm{f}(n_1, \ldots, n_k)$ in der Variablen x_0.

Offensichtlich terminieren LOOP-Programme stets, 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. Das LOOP-Konstrukt entspricht einer einfachen variante von FOR-Schleifen.

Beispiel:
Die Additionsfunktion ist LOOP-berechenbar:

x_0 `:=` $x_1 + 0$`;`
`LOOP` x_2 `DO` x_0 `:=` $x_0 + 1$ `END`

Dieses Beispiel zeigt, dass die Addition von Variablen ebenfalls mittels LOOP-Programmen ausdrückbar ist. Man kann auch verschiedene andere Programmkonstrukte typischer imperativer Programmiersprachen 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 `:=` $x_i + c$ `END`

- `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 ist.

Satz 13.18

Die Klasse der primitiv rekursiven Funktionen stimmt mit der 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:

$$\mathrm{f}(x_1, \ldots, x_n) = \mathrm{g}(\mathrm{h}_1(x_1, \ldots, x_k), \ldots, \mathrm{h}_m(x_1, \ldots, x_k))$$

  Es seien $P_1, \ldots, P_m, P$ LOOP-Programme für $\mathrm{h}_1, \ldots, \mathrm{h}_m, \mathrm{g}$.

  Durch Speichern der Eingabewerte und der Zwischenergebnisse in unbenutzten Variablen kann man zunächst die Werte von $\mathrm{h}_1, \ldots, \mathrm{h}_m$ mittels $P_1, \ldots, P_m$ berechnen und dann auf diese Werte P anwenden. Genauer:

  y_1 := x_1; $\cdots$; y_n := x_n;
  Führe P_1 aus
  z_1 := x_0;
  x_i := 0; für alle in P_1 verwendeten Variablen x_i
  x_1 := y_1; $\cdots$; x_n := y_n;
  Führe P_2 aus
  $\vdots$
  z_m := x_m;
  x_i := 0; für alle in P_m verwendeten Variablen x_i
  x_1 := z_1; $\cdots$; x_m := z_m;
  Führe P aus

- primitive Rekursion:

$$\mathrm{f}(x_1, \ldots, x_n, 0) = \mathrm{g}(x_1, \ldots, x_n)$$
$$\mathrm{f}(x_1, \ldots, x_n, x_{n+1} + 1) = \mathrm{h}(x_1, \ldots, x_n, \mathrm{f}(x_1, \ldots, x_n, x_{n+1}), x_{n+1})$$

  Die Funktion f kann durch das LOOP-Programm

  z_1 := $\mathrm{g}(x_1, \ldots, x_n)$; $(\star)$
  z_2 := 0;
  `LOOP` x_{n+1} `DO`
  z_1 := $\mathrm{h}(x_1, \ldots, x_n, z_1, z_2)$; $(\star)$
  z_2 := $z_2 + 1$
  `END`
  $x_0 := z_1$

  berechnet werden.

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.

Dazu beschaffen wir uns zunächst eine primitiv rekursive Bijektion

$$\mathrm{c}^{(n)} : \mathbb{N}^n \to \mathbb{N} \quad (n \geq 2)$$

sowie die zugehörigen Umkehrfunktionen

$$\mathrm{c}_0^{(n)}, \ldots, \mathrm{c}_{n-1}^{(n)}.$$

Wir definieren dazu

$$\begin{aligned}
\mathrm{c}^{(n)}(x_0, \ldots, x_{n-1}) &:= \mathrm{c}(x_0, \mathrm{c}(x_1, \ldots, \mathrm{c}(x_{n-2}, x_{n-1}) \ldots)) \\
\mathrm{c}_0^{(n)}(z) &:= \mathrm{c}_0(z) \\
\mathrm{c}_1^{(n)}(z) &:= \mathrm{c}_1(\mathrm{c}_2(z)) \\
&\vdots \\
\mathrm{c}_{n-2}^{(n)}(z) &:= \mathrm{c}_1(\mathrm{c}_2^{(n-2)}(z)) \\
\mathrm{c}_{n-1}^{(n)}(z) &:= \mathrm{c}_2^{n-1}(z)
\end{aligned}$$

Da c und c_1, c_2 primitiv rekursiv sind, sind auch $\mathrm{c}^{(n)}$ und $\mathrm{c}_0^{(n)}, \ldots, \mathrm{c}_{n-1}^{(n)}$ primitiv rekursiv.

Es sei nun P_0 ein LOOP-Programm, das die Funktion $\mathrm{f}_0 : \mathbb{N}^r \to \mathbb{N}$ berechnet. Es sei ℓ der maximale Index der in P_0 vorkommenden Variablen und $k := \max\{r, \ell\}$.

Wir zeigen durch Induktion über den Aufbau von Teilprogrammen P von P_0, dass die Funktion $\mathrm{g}_P : \mathbb{N} \to \mathbb{N}$ primitiv rekursiv ist, wobei

$$\mathrm{g}_P(z) = \mathrm{c}^{(k+1)}(b_0, \ldots, b_k)$$

wenn

- $b_0, \ldots, b_k$ die Werte von $x_0, \ldots, x_k$ nach Ausführung des Programms P
- bei Startwerten $a_0 = \mathrm{c}_0^{(k+1)}(z), \ldots, a_k = \mathrm{c}_k^{(k+1)}(z)$ sind.

Das Kodieren ist notwendig, da das Resultat von primitiv rekursiven Funktionen stehts eine einzige natürliche Zahl ist, diese aber die Werte *aller* für P_0 relevanter Variablen $x_0, \ldots, x_k$ repräsentieren muß. Wir führen nun die Induktion durch:

1) Hat P die Form x_i `:=` $x_j + c$, so ist

$$\mathrm{g}_P(z) = \mathrm{c}^{(k+1)}(\mathrm{c}_0^{(k+1)}(z), \ldots, \mathrm{c}_{i-1}^{(k+1)}(z), \mathrm{c}_j^{(k+1)}(z) + c, \mathrm{c}_{i+1}^{(k+1)}(z), \ldots).$$

Primitiv rekursiv.

Entsprechend kann die andere Zuweisung behandelt werden.

2) $P = Q\,;R$
Dann ist

$$\mathrm{g}_P(z) = \mathrm{g}_R(\mathrm{g}_Q(z)).$$

Da

- $\mathrm{g}_Q, \mathrm{g}_R$ nach Induktionsvoraussetzung primitiv rekursiv sind und
- g_P durch Komposition daraus entsteht

ist auch g_P primitiv rekursiv.

3) $P =$ `LOOP` x_i `DO` Q `END`
Wir definieren zunächst die zweistellige Funktion h durch primitive Rekursion aus g_Q:

$$\begin{aligned} \mathrm{h}(x, 0) &= x, \\ \mathrm{h}(x, y+1) &= \mathrm{g}_Q(\mathrm{h}(x, y)) \end{aligned}$$

Offenbar liefert $\mathrm{h}(z, n)$

- die Kodierung der Werte der Variablen $x_0, \ldots, x_k$,
- nachdem man, beginnend mit $x_0 = \mathrm{c}_0^{(k+1)}(z), \ldots, x_k = \mathrm{c}_k^{(k+1)}(z)$,
- das Programm Q n-mal ausgeführt hat.

Daher gilt:

$$\mathrm{g}_P(z) = \mathrm{h}(z, \mathrm{c}_i^{(k+1)}(z)).$$

Dies schließt den Induktionsbeweis ab, dass die Funktion g_P für jedes Teilprogramm P von P_0 (einschließlich P_0 selbst) primitiv rekursiv ist.

Ist

$$\mathrm{f}_0 : \mathbb{N}^r \to \mathbb{N}$$

die von P_0 berechnete Funktion, so gilt also:

$$\mathrm{f}_0(z_1, \ldots, z_r) = \mathrm{c}_0^{(k+1)}(\mathrm{g}_{P_0}(\mathrm{c}^{(k+1)}(0, z_1, \ldots, z_r, \underbrace{0, \ldots, 0}_{k-r}))). \qquad \square$$

Da wir durch LOOP-berechenbare/primitiv rekursive Funktionen nur totale Funktionen erhalten, ist *nicht* jede Turing-berechenbare Funktion in dieser Klasse enthalten. Aber was ist mit den totalen berechenbaren Funktionen?

Satz 13.19

Es gibt totale berechenbare Funktionen, die nicht LOOP-berechenbar sind.

Beweis. Wir definieren die Länge von LOOP-Programmen induktiv über deren Aufbau:

1) $|x_i \texttt{ := } x_j + c| := i + j + c + 1$
   $|x_i \texttt{ := } x_j \dot{-} c| := i + j + c + 1$
2) $|P\texttt{;}Q| := |P| + |Q| + 1$
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:

$$\mathrm{f}(x, y) := 1 + \max\{\mathrm{g}(y) \mid \mathrm{g} : \mathbb{N} \to \mathbb{N} \text{ wird von einem LOOP-Programm der Länge } x \text{ berechnet}\}$$

Behauptung 1:

Die Funktion

$$\mathrm{d} : \mathbb{N} \to \mathbb{N} \text{ mit } z \mapsto \mathrm{f}(z, z)$$

ist nicht LOOP-berechenbar, denn:

Sei P ein LOOP-Programm, das d berechnet und sei $n = |P|$.

Wir betrachten $\mathrm{d}(n) = \mathrm{f}(n, n)$. Nach Definition ist $\mathrm{f}(n, n)$ verschieden von jedem 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 formal Beweisen, dass eine TM jedes LOOP-Programm simulieren kann). Da alle diese Aufrufe terminieren, kann man in endlicher Zeit den maximalen so erhaltenen Funktionswert berechnen (und dann um eins erhöhen).

□

Eine prominente *nicht* primitiv rekursive, aber berechenbare Funktion ist die sogenannte *Ackermannfunktion* A, die wie folgt definiert ist:

$$\begin{aligned} \mathrm{A}(0, y) &:= y + 1 \\ \mathrm{A}(x + 1, 0) &:= \mathrm{A}(x, 1) \\ \mathrm{A}(x + 1, y + 1) &:= \mathrm{A}(x, \mathrm{A}(x + 1, y)) \end{aligned}$$

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 formaler, induktiver Beweis der Wohldefiniertheit ist nicht schwierig.

Man kann leicht zeigen, dass A Turing-berechenbar ist. Schwieriger ist der Beweis, dass A nicht LOOP-berechenbar, da die Funktionswerte *schneller wachsen* als bei jeder LOOP-berechenbaren Funktion (siehe [Schö97]).

14. μ-rekursive Funktionen und While-Programme

Um die primitiv rekursiven Funktionen so zu erweitern, dass man alle (Turing-)berechenbaren Funktionen abdeckt, fügt mn eine weitere Operation, die *unbeschränkte Minimalisierung* hinzu. Auf Seiten der LOOP-Programme entspricht dies der Erweiterung um eine *While-Schleife*. Diese unterscheidet sich von der LOOP-Schleife dadurch, dass die Anzahl der Durchläufe nicht von Anfang an feststeht.

Definition 14.1 (unbeschränkte Minimalisierung)
Es sei $n \geq 0$. Die Funktion $\mathrm{f} : \mathbb{N}^n \to \mathbb{N}$ entsteht aus $\mathrm{g} : \mathbb{N}^{n+1} \to \mathbb{N}$ durch *unbeschränkte Minimalisierung* (Anwendung des *μ-Operators*), falls gilt:

$$\mathrm{f}(x_1, \ldots, x_n) = \begin{cases} y & \text{falls } \mathrm{g}(x_1, \ldots, x_n, y) = 0 \text{ und} \\ & \mathrm{g}(x_1, \ldots, x_n, z) \text{ ist definiert und } \neq 0 \\ & \text{für alle } 0 \leq z < y \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Wir schreiben dann auch $\mathrm{f} = \mu\mathrm{g}$.

Der μ-Operator sucht also nach dem kleinsten y, so dass $\mathrm{g}(\underline{x}, y) = 0$ ist. Dabei müssen aber alle vorherigen Werte $\mathrm{g}(\underline{x}, z)$ für $z < y$ definiert sein. Der Wert $\mu g(\underline{x})$ ist also in zwei Fällen undefiniert: (i) wenn kein y existiert für dass $g(\underline{xy}) = 0$ gilt und (ii) wenn für das kleinste y mit $g(\underline{xy}) = 0$ gilt: es gibt ein z mit $0 \leq z < y$ und $g(\underline{xy})$ undefiniert.

Wir betrachten zwei Beispiele für Funktionen, die sich mittels des μ-Operators definieren lassen. Beide Beispiele zeigen auch, dass eine μ-rekursive Funktion im Gegensatz zu den primitiv rekursiven Funktionen nicht unbedingt total sein muß.

Beispiel 14.2

- Die 1-stellige, überall undefinierte Funktion undef kann definiert werden als

$$\mathrm{undef}(x_1) = \mu(x_1 + s(x_2)).$$

- Um die Funktion

$$\mathrm{quo}(x, y) = \begin{cases} x/y & \text{falls } x/y \in \mathbb{N} \\ \text{undefiniert} & \text{sonst} \end{cases}$$

  zu definieren, beobachten wir zunächst, dass die Funktion

$$\mathrm{neq}(x, y) = \begin{cases} 1 & \text{falls } x \neq y \\ 0 & \text{sonst} \end{cases}$$

  primitiv rekursiv und damit auch μ-rekursiv ist: $\mathrm{neq}(x, y) = \mathrm{sign}((x \dot{-} y) + (y \dot{-} x))$. Wir definieren nun $\mathrm{quo}(x, y)$ wie folgt:

$$\mathrm{quo}(x, y) = \mu(x \neq z \cdot y).$$

Definition 14.3 (Klasse der μ-rekursiven Funktionen)
Die Klasse der *μ-rekursiven Funktionen* besteht aus den Funktionen, welche man aus den *Grundfunktionen* (Definition 13.1) durch Anwenden von

- Komposition,
- primitiver Rekursion und
- unbeschränkter Minimalisierung

erhält.

Definition 14.4 (Syntax von WHILE-Programmen)
Die Syntax von WHILE-Programmen enthält alle Konstrukte in der Syntax von LOOP-Programmen und zusätzlich

4) Falls P ein WHILE-Programm ist und $i \geq 0$, so ist auch
   `WHILE` $x_i \neq 0$ `DO` P `END`
   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:

`LOOP` x `DO` P `END`
kann simuliert werden durch:

y `:=` $x + 0$`;`
`WHILE` $y \neq 0$ `DO` y `:=` $y \dot{-} 1$`;`P `END`

wobei y eine neue Variable ist.

Definition 14.5 (WHILE-berechenbar)
Eine (partielle) Funktion $\mathrm{f} : \mathbb{N}^k \to \mathbb{N}$ heißt *WHILE-berechenbar*, falls es ein WHILE-Programm P gibt, das f in dem 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 $\mathrm{f}(n_1, \ldots, n_k)$ in der Variablen x_0, falls dieser Wert definiert ist.
- Sonst stoppt P nicht.

Beispiel:
Die Funktion

$$\mathrm{f} : \mathbb{N}^2 \to \mathbb{N} \text{ mit } (x, y) \mapsto \begin{cases} x - y & \text{falls} \quad x \geq y \\ \text{undefiniert} & \text{sonst} \end{cases}$$

ist WHILE-berechenbar durch das folgende Programm:

```
WHILE x2 ≠ 0 DO
  IF x1 ≠ 0 THEN
    x1 := x1 ∸ 1;
    x2 := x2 ∸ 1;
  END
END;
x0 := x1
```

Beachte, dass das IF-Konstrukt in der Syntax von WHILE-Programmen eigentlich nicht vorhanden ist. Es kann aber leicht wie folgt simuliert werden:

```
y := x1;
WHILE y ≠ 0 DO
  x1 := x1 ∸ 1;
  x2 := x2 ∸ 1;
  y := 0
END
```

Satz 14.6

Die Klasse der μ-rekursiven Funktionen stimmt mit der Klasse der WHILE-berechenbaren Funktionen überein.

Beweis. Wir müssen hierzu den Beweis von Satz 13.18 um die Behandlung des zusätzlichen μ-Operators/WHILE-Konstrukts ergänzen.

1) Alle μ-rekursiven Funktionen sind WHILE-berechenbar:

   Es sei $\mathrm{f} = \mu\mathrm{g}$ und P (nach Induktionsvoraussetzung) ein WHILE-Programm für g. Dann berechnet das folgende Programm die Funktion f:

   ```
   x0 := 0;
   y := g(x1, ..., xn, x0);      (realisierbar mittels P)
   WHILE y ≠ 0 DO
     x0 := x0 + 1;
     y := g(x1, ..., xn, x0);    (realisierbar mittels P)
   END
   ```

   Dieses Programm terminiert nicht, wenn eine der Berechnung der Funktion g mittels des Programmes 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.

2) Alle WHILE-berechenbaren Funktionen sind μ-rekursiv:

   Betrachte das Programm `WHILE` $x_i \neq 0$ `DO` Q `END`.

   Wie bei der Behandlung von LOOP im Beweis von Satz 13.18 erhalten wir eine μ-rekursive Funktion

   $$\mathrm{h} : \mathbb{N}^2 \to \mathbb{N},$$

für die

- $\mathrm{h}(z, n)$ die Kodierung der Werte der Variablen $x_0, \ldots, x_k$ ist nachdem,
- beginnend mit $x_0 = \mathrm{c}_0^{(k+1)}(z), \ldots, x_k = \mathrm{c}_k^{(k+1)}(z)$
- das Programm Q n-mal ausgeführt wurde.

Der μ-Operator, angewandt auf die Komposition $\mathrm{c}_i^{(k+1)}(\mathrm{h})$, sucht nach der kleinsten Iterationszahl, so dass die Variable $x_i = 0$ wird. Daher ist

$$\mathrm{g}_P(z) = \mathrm{h}(z, (\mu \mathrm{c}_i^{(k+1)}(\mathrm{h}(z)))).$$

Man beachte, dass sowohl die primitive Rekursion (bei der Definition von h) als auch die unbeschränkte Minimalisierung verwendet wurde, um das WHILE-Konstrukt zu behandelt.

□

Es bleibt noch zu zeigen, dass die Klasse der μ-rekursiven/WHILE-berechenbaren Funktionen mit der Klasse der Turing-berechenbaren Funktionen übereinstimmt.

Satz 14.7

Jede μ-rekursive Funktion ist Turing-berechenbar.

Beweis. Es ist leicht zu zeigen, dass die Grundfunktionen Turing-berechenbar sind. (Übung)

Komposition: Seien $\mathcal{A}_\mathrm{g}, \mathcal{A}_{\mathrm{h}_1}, \ldots, \mathcal{A}_{\mathrm{h}_m}$ DTM, die $\mathrm{g}, \mathrm{h}_1, \ldots, \mathrm{h}_m$ berechnen. Konstruiere wie folgt eine DTM, die die Komposition f von g und $\mathrm{h}_1, \ldots, \mathrm{h}_m$ berechnet:

Verwende $m + 1$ Bänder:

- Kopiere die Eingabe $\underline{x}$ auf Band 2, ..., Band $(m + 1)$
- $\mathcal{A}_{\mathrm{h}_i}$ berechnet $\mathrm{h}_i(\underline{x})$ auf Band $(i + 1)$
- Überschreibe Band 1 mit $a^{\mathrm{h}_1(\underline{x})} \not b \ldots \not b\, a^{\mathrm{h}_m(\underline{x})}$
- Berechne g-Wert davon mit $\mathcal{A}_\mathrm{g}$ auf Band 1

Wenn eine der Maschinen $\mathcal{A}_\mathrm{g}, \mathcal{A}_{\mathrm{h}_1}, \ldots, \mathcal{A}_{\mathrm{h}_m}$ nicht terminiert, so terminiert auch die hier konstruierte Maschine $\mathcal{A}_\mathrm{f}$ nicht. Das ist korrekt, da in diesem Fall der Funktionswert der Komposition f underfiniert ist.

Primitive Rekursion: Seien $\mathcal{A}_\mathrm{g}, \mathcal{A}_\mathrm{h}$ DTM für g, h. Konstruiere wie folgt eine DTM, die die Funktion f berechnet, welche sich aus g und $\mathrm{h}_1, \ldots, \mathrm{h}_m$ mittels primitiver Rekursion ergibt:

Verwende 4 Bänder:

- Band 1: Speichert $\underline{xy}$
- Band 2: Zählt von 0 hoch bis y (aktueller Wert: z)

- Band 3: Enthält $f(\underline{x}, z)$ (für $z = 0$ mittels $\mathcal{A}_g$ berechnet)
- Band 4: Berechnung von $\mathcal{A}_h$

Man überzeugt sich leicht davon, dass die konstruierte Maschine genau dann nicht terminiert, wenn der Funktionswert von f undefiniert ist.

μ-Operator: Sei $\mathcal{A}_g$ eine DTM für g. Konstruiere wie folgt eine DTM, die die Funktion f berechnet, welche sich aus g durch unbeschränkte Minimalisierung ergibt:

Verwende 3 Bänder:

- Band 1: Speichert Eingabe $\underline{x}$
- Band 2: Zählt von 0 hoch (aktueller Wert: z)
- Band 3: Berechne $g(\underline{x}, z)$ mittels $\mathcal{A}_g$ und teste, ob Wert $= 0$ ist

Auch hier terminiert die konstruierte Maschine genau in den "richtigen Fällen".

□

Satz 14.8

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 $Q = \{q_1, \ldots, q_k\}$ mit q_1 Startzustand, d.h. wir können Alphabetssymbole und Zustände über ihren Index als natürliche Zahl beschreiben.

Die Konfiguration

$$a_{i_1} \ldots a_{i_l} q_m a_{j_1} \ldots a_{j_r}$$

wird dargestellt als

$$x_1 = (i_1, \ldots, i_l)_b \qquad := \sum_{\nu=1}^{l} i_\nu \cdot b^{l-\nu}$$

$$x_2 = m$$

$$x_3 = (j_r, \ldots, j_1)_b \qquad := \sum_{\rho=1}^{r} j_\rho \cdot b^{\rho-1},$$

wobei $b = |\Gamma| + 1$ ist, d.h.

- $a_{i_1} \dots a_{i_l}$ repräsentiert x_1 in b-närer Zahlendarstellung und
- $a_{j_r} \dots a_{j_1}$ (Reihenfolge!) repräsentiert x_3 in b-närer Zahlendarstellung.

Ist beispielsweise $b = |\Gamma| = 10$, also $\Gamma = \{a_1, \dots, a_9\}$, so wird die Konfiguration

$$a_4 a_2 a_7 a_1 \, q_3 \, a_1 a_8 a_3 \quad \text{dargestellt durch} \quad x_1 = 4271,\ x_2 = 3,\ x_3 = 381.$$

Wie brauchen $b = |\Gamma| + 1$ statt $b = |\Gamma|$, 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 $x_3 = (j_r, \dots, j_1)_b$, so ist $j_1 = x_3 \bmod b$.

Ändern dieses Symbols zu a_j
Der neue Wert von x_3 ist $(j_r, \dots, j_2, j)_b = \mathrm{div}((j_r, \dots, j_2, j_1)_b, b) \cdot b + j$

Verschieben des Schreib-Lesekopfes
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 $q_0 aaa \not b aaaaa$ erzeugt werden, repräsentiert durch $x_1 = 2$, $x_2 = 1$, $x_3 = 111211111$ 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 $x_3 \bmod b$
   - wird mit den oben dargestellten arithmetischer Operationen das aktuelle Symbol verändert und
   - der Schreib-Lesekopf 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. □

Insgesamt haben wir also gezeigt:

Theorem 14.9

Die folgenden Klassen von Funktionen stimmen überein:

1) Turing-berechenbare Funktionen

2) WHILE-berechenbare Funktionen

3) μ-rekursive Funktionen

Diese Äquivalenz ist ein wichtiges Argument für die Korrektheit der Church-Turing These.

15. (Partielle) Entscheidbarkeit und Aufzählbarkeit

In diesem Kapitel führen wir den Begriff eines Entscheidungsproblems und der Entscheidbarkeit formal ein, außerdem die eng verwandten Begriffe partielle Entscheidbarkeit und rekursive Aufzählbarkeit. Wir werden dann einige grundlegende Zusammenhänge zwischen diesen Begriffen kennenlernen. Wir verwenden hier direkt Turing-Maschinen als zugrundeliegendes Berechnungsmodell, begründet durch die Church-Turing-These und die in den vorhergehenden Abschnitten dargestellte Äquivalenz zu anderen konkreten Berechnungsmodellen.

In der Informatik erfordern viele Probleme nur eine ja/nein Antwort anstelle eines "echten" Funktionswertes, z.B.:

- Das Leerheitsproblem für NEAs: gegeben ein NEA $\mathcal{A}$, ist $L(\mathcal{A}) = \emptyset$?
- Das Wortproblem für kontextfreie Grammatiken: gegeben eine kontextfreie Grammatik G und ein Wort, ist $w \in L(G)$?
- Das Äquivalenzproblem für kontextsensitive Grammatiken: gegeben kontextsensitive Grammatiken G_1 und G_2, gilt $L(G_1) = L(G_2)$?
- etc

Derartige Probleme nennen wir Entscheidungsprobleme. Wir formalisieren sie nicht als Funktionen, sondern als Mengen $P \subseteq \Sigma^*$ über einem geeigneten Alphabet Σ, also als formale Sprache. Das assoziierte ja/nein-Problem ist dann einfach: ist ein gegebenes Wort $w \in \Sigma^*$ enthalten in P?

Als Beispiel für die Formalisierung eines Entscheidungsproblems als formale Sprache betrachten wir das *Äquivalenzproblem* für kontextsensitive Sprachen. Jede Grammatik $G = (N, \Sigma, P, S)$ kann als Wort $\text{code}(G) \in \Gamma^*$ über einer festen (d.h. nicht von G abhängigen) Grammatik Γ aufgefasst werden. Das Äquivalenzproblen für Typ 1-Sprachen ist dann die Sprache

$$\{(\text{code}(G_1)\#\text{code}(G_2)) \mid G_1, G_2 \text{ kontextsensitiv}, L(G_1) = L(G_2)\} \subseteq \Gamma^*,$$

wobei # einfach ein Trennsymbol ist, dass es erlaubt, die beiden Eingaben zu unterscheiden.

Definition 15.1 (entscheidbar, partiell entscheidbar, rekursiv aufzählbar)
Eine Sprache $L \subseteq \Sigma^*$ heißt

1) *entscheidbar*, falls es eine DTM gibt, die bei Eingabe $w \in \Sigma^*$
   - 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 $q_0 w$

2) *partiell entscheidbar*, falls es eine DTM gibt, die bei Eingabe $w \in \Sigma^*$
   - terminiert, falls $w \in L$ ist
   - nicht terminiert sonst.

3) *rekursiv aufzählbar*, falls L von einer Aufzähl-Turingmaschine aufgezählt wird. Diese ist wie folgt definiert:

   Eine *Aufzähl-Turingmaschine* $\mathcal{A}$ 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 hat w als *Ausgabe*.

   Die durch $\mathcal{A}$ *aufgezählte Relation* ist

   $$R = \{w \in \Sigma^* \mid w \text{ ist Ausgabe einer Ausgabekonfiguration, die von } \mathcal{A} \text{ ausgehend von Startkonfiguration } q_0 \not b \text{ erreicht wird}\}.$$

Entscheidbarkeit entspricht der intuitiven Existenz eines Algorithmus, der das Entscheidungsproblem (in endlicher Zeit) löst. In der Tat haben wir bereits zahlreiche Entscheidbarkeitsbeweise geführt (z.B. für das Wortproblem für NEAs), wobei wir jedoch einen intuitiven Berechenbarkeitsbegriff anstatt TMs verwendet haben. Eine DTM, die eine Sprache L entscheidet, nennen wir ein *Entscheidungsverfahren* für L. Eine DTM, die L partiell entscheidet, nennen wir ein *partielles Entscheidungsverfahren* für L. Beachte, dass ein Entscheidungsverfahren immer terminiert, ein partielles Entscheidungsverfahren jedoch nicht. Bei partiellen Entscheidungsverfahren ist Terminierung ja sogar die verwendete Akzeptanzbedingung. Beachte auch, dass eine Aufzähl-TM dasselbe Wort mehrfach aufzählen kann.

Bemerkung 15.2.

1) Entscheidbarkeit kann als Spezialfall der Berechenbarkeit *totaler* Funktionen gesehen werden: eine Sprache L ist entscheidbar g.d.w. ihre *charakteristische Funktion*, also die totale Funktion χ_L, definiert durch

   $$\chi_L(w) := \begin{cases} a & \text{wenn } w \in L \\ \varepsilon & \text{sonst} \end{cases}$$

   berechenbar ist.

2) Man sieht leicht: eine Sprache L ist partiell entscheidbar gdw. L der Definitionsbereich einer einstelligen berechenbaren *partiellen* Funktion ist.

3) Eine Sprache L ist partiell entscheidbar gdw. sie Turing-erkennbar ist:

   - Turing-erkennbar $\Rightarrow$ partiell entscheidbar

     Wir können mit Satz 11.6 o.B.d.A. annehmen, dass es eine DTM $\mathcal{A}$ gibt, die L erkennt. Bei $w \notin L$ kann $\mathcal{A}$ in nicht-Endzustand halten, wohingegen ein partielles Entscheidungsverfahren nicht terminieren darf.

     Modifikation: wenn $\mathcal{A}$ bei $w \notin L$ in nicht-Endzustand anhält, dann gehe in Endlosschleife.

   - partiell entscheidbar $\Rightarrow$ Turing-erkennbarheidbar

     Partielles Entscheidungsverfahren $\mathcal{A}$ für L kann bei $w \in L$ in beliebigem Zustand anhalten, wohingegen es bei Turing-Erkennbarkeit ein Endzustand sein muß.

     Modifikation: wenn $\mathcal{A}$ bei $w \in L$ in nicht-Endzustand anhält, dann wechsle in einem Schritt in Endzustand (der keine Folgezustände hat).

Es bestehen sehr enge Zusammenhänge zwischen den in Definition 15.1 eingeführten Begriffen. So stellen sich rekursive Aufzählbarkeit und partielle Entscheidbarkeit als äquivalent heraus und eine Sprache L ist entscheidbar gdw. L und das Komplement von L partiell entscheidbar sind.

Satz 15.3
Es sei $L \subseteq \Sigma^*$.

1) L ist rekursiv aufzählbar gdw. L ist partiell entscheidbar.

2) Ist L entscheidbar, so auch partiell entscheidbar.

3) Ist L entscheidbar, so ist auch das Komplement $\overline{L} = \Sigma^* \setminus L$ entscheidbar.

4) L ist entscheidbar gdw. L und $\overline{L}$ partiell entscheidbar sind.

Beweis.

1) „⇒“: Es sei L rekursiv aufzählbar und $\mathcal{A}$ eine Aufzähl-DTM für L. Die Maschine $\mathcal{A}'$, die ein partielles Entscheidungsverfahren für L ist, arbeitet wie folgt:

   - Sie speichert die Eingabe w auf zusätzliches Band
   - Sie beginnt mit der Aufzählung von L.
   - Bei jeder Ausgabekonfiguration überprüft sie, ob die entsprechende Ausgabe mit w übereinstimmt. Wenn ja, so terminiert $\mathcal{A}'$. Sonst sucht sie die nächste Ausgabekonfiguration von $\mathcal{A}$.
   - Terminiert $\mathcal{A}$, ohne dass w ausgegeben wurde, so gehe in Endlosschleife.

   $\mathcal{A}'$ terminiert daher genau dann nicht, wenn w nicht in der Aufzählung vorkommt.

„⇐“: Es sei $\mathcal{A}$ ein partielles Entscheidungsverfahren für L und

$$\Sigma^* = \{w_1, w_2, w_3, \ldots\}$$

Die Maschine $\mathcal{A}'$ arbeitet wie folgt:

1) Führe einen Schritt der Berechnung von $\mathcal{A}$ auf Eingabe w_1 aus
2) Führe zwei Schritt der Berechnung von $\mathcal{A}$ auf Eingaben w_1 und w_2 aus
   $\vdots$
n) Führe n Schritte der Berechnung von $\mathcal{A}$ auf Eingaben $w_1, \ldots, w_n$ aus
   $\vdots$

Terminiert $\mathcal{A}$ für eine dieser Eingaben, so gebe diese Eingabe aus und mache weiter.

Beachte:
Man kann nicht $\mathcal{A}$ zunächst auf Eingabe w_1 zu Ende laufen lassen, da $\mathcal{A}$ auf w_1 nicht terminieren muss.

2) Eine DTM $\mathcal{A}$, die L entscheidet, wird wie folgt modifiziert:
   - hält $\mathcal{A}$ in nicht-akzeptierender Stoppkonfiguration, so gehe in Endlosschleife.
   - keine Änderung, wenn $\mathcal{A}$ in akzeptierender Stoppkonfiguration stoppt.
3) Eine DTM $\mathcal{A}$, die L berechnet, wird wie folgt zu einer DTM für $\overline{L}$ modifiziert:
   - setze $F = Q \setminus F$ (tausche Endzustände und nicht-Endzustände)

   Diese Konstruktion liefert das gewünschte Resultat, weil $\mathcal{A}$ deterministisch ist und auf jeder Eingabe terminiert.
4) „⇒“: Ergibt sich aus 2) und 3).

   „⇐“: Sind L und $\overline{L}$ partiell entscheidbar, so mit 1) auch rekursiv aufzählbar.

   Für Eingabe w lässt man die Aufzähl-DTMs $\mathcal{A}$ und $\mathcal{A}'$ für L und $\overline{L}$ parallel laufen (d.h. jeweils abwechselnd ein Schritt von $\mathcal{A}$ auf einem Band gefolgt von einem Schritt von $\mathcal{A}'$ auf dem anderen).

   Die Eingabe w kommt in einer der beiden Aufzählungen vor:

$$w \in \Sigma^* = L \cup \overline{L}$$

   Kommt w bei $\mathcal{A}$ vor, so erzeuge Ausgabe a, sonst Ausgabe ε.

□

16. Universelle Maschinen und unentscheidbare Probleme

Wir beweisen das fundamentale Resultat, dass es Probleme gibt, die nicht entscheidbar sind. Ein wichtiges Hilfsmittel dabei ist eine *universelle Turingmaschine*, die wie ein Interpreter für Programmiersprachen funktioniert und damit *jede* Turingmaschinen simulieren kann. Die universelle Maschine erhält als Eingabe

- eine Beschreibung der zu simulierenden Turingmaschine $\mathcal{A}$;
- das Eingabewort w, auf dem $\mathcal{A}$ simuliert werden soll.

Sie akzeptiert ihre Eingabe gdw. w von $\mathcal{A}$ akzeptiert wird. Wie jede andere Turingmaschine bekommt auch die universelle TM ein Wort als Eingabe. Um Turingmaschinen als Eingabe verwenden zu können, ist es also wichtig, diese als Wörter zu kodieren.

Konventionen:

- *Arbeitsalphabete* der betrachteten Turingmaschinen sind endliche Teilmengen von $\{a_0, a_1, a_2, \ldots\}$, wobei wir der Lesbarkeit halber an Stelle von a_0 meist a schreiben, an Stelle von a_1 schreiben wir b, und an Stelle von a_2 schreiben wir $\not b$.
- *Zustandsmengen* sind endliche Teilmengen von $\{q_0, q_1, q_2, \ldots\}$, wobei q_0 stets der *Anfangszustand* ist.

Definition 16.1 (Kodierung einer Turingmaschine)
Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_1, \Delta, F)$ eine Turingmaschine, die o.B.d.A. die obigen Konventionen erfüllt.

1) Eine *Transition*

$$t = (q_i, a_j, a_{j'}, \begin{matrix} l \\ r \\ n \end{matrix}, q_{i'})$$

wird *kodiert* durch

$$\mathrm{code}(t) = a^i b a^j b a^{j'} b \begin{matrix} a \\ aa \\ aaa \end{matrix} b a^{i'} bb.$$

2) Besteht Δ aus den Transitionen $t_1, \ldots, t_k$ und ist $F = \{q_{i_1}, \ldots, q_{i_r}\}$, so wird $\mathcal{A}$ *kodiert* durch

$$\mathrm{code}(\mathcal{A}) = \mathrm{code}(t_1) \ldots \mathrm{code}(t_k) b a^{i_1} b \ldots b a^{i_r} bbb.$$

Beachte, dass die einzelnen Transitionen durch *bb* getrennt sind, die Kodierung der Übergangsrelation durch *bbb* abgeschlossen wird und die Gesamtkodierung der TM durch den zweiten Block *bbb* gekennzeichnet ist.

Bemerkung 16.2.

1) Wir werden als Eingabe für die universelle TM Wörter der Form $x = \text{code}(\mathcal{A})w$ mit $w \in \Sigma^*$ verwenden; die Eingabe w für $\mathcal{A}$ findet man leicht nach dem zweiten Block *bbb*.
2) Es gibt eine DTM $\mathcal{A}_{CODE}$, welche die Sprache

$$CODE = \{\text{code}(\mathcal{A}) \mid \mathcal{A} \text{ ist DTM über } \Sigma\}$$

entscheidet, denn:

- Überprüfe bei Eingabe w zunächst, ob w eine Turingmaschine kodiert (d.h. eine Folge von Transitionskodierungen gefolgt von einer Endzustandsmengenkodierung ist).
- Überprüfe dann, ob die kodierte Turingmaschine deterministisch ist (bei jeder Transition wird nachgeschaut, ob es eine andere mit demselben Anfangsteil gibt).

Die pure Existenz einer Kodierung von Turingmaschinen als Wörter ist bereits eine interessante Tatsache. Man erhält beispielsweise unmittelbar, dass es nur abzählbar viele Turingmaschinen gibt. Analog zu Cantors Beweis, dass es überabzählbar viele reelle Zahlen gibt, kann man beweisen, dass es überabzählbar viele verschiedene Sprachen gibt. Hieraus folgt natürlich sofort die Existenz unentscheidbarer Sprachen. Wir wollen im folgenden aber einen konstruktiveren Beweis führen, der sich zudem auf eine Sprache bezieht, die zwar unentscheidbar, aber trotzdem noch partiell entscheidbar ist. Wir konstruieren dazu zunächst wie angekündigt eine universelle Turingmaschine.

Satz 16.3 (Turing)
Es gibt eine universelle DTM $\mathcal{U}$ über Σ, d.h. eine DTM mit der folgenden Eigenschaft: Für alle DTM $\mathcal{A}$ und alle $w \in \Sigma^*$ gilt:

$$\mathcal{U} \textit{ akzeptiert } \text{code}(\mathcal{A})w \quad \underline{gdw}. \quad \mathcal{A} \textit{ akzeptiert } w.$$

Es ist also $\mathcal{U}$ ein „Turing-Interpreter für Turingmaschinen“.

Beweis. $\mathcal{U}$ führt bei Eingabe $\text{code}(\mathcal{A})w$ die $\mathcal{A}$-Berechnung

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \dots$$

in kodierter Form aus, d.h. $\mathcal{U}$ erzeugt sukzessive Bandbeschriftungen

$$\text{code}(\mathcal{A})\text{code}(k_0),\ \text{code}(\mathcal{A})\text{code}(k_1),\ \text{code}(\mathcal{A})\text{code}(k_2),\ \dots$$

Wir brauchen also noch eine *Kodierung von Konfigurationen als Wörter*:

$$\text{code}(a_{i_1} \dots a_{i_l} q_j a_{i_{l+1}} \dots a_{i_r}) = a^{i_1} b \dots a^{i_l} b a^j b b a^{i_{l+1}} b \dots a^{i_r} b$$

Beachte, dass die Kopfposition durch *bb* gekennzeichnet ist.

Arbeitsweise von $\mathcal{U}$ bei Eingabe $\text{code}(\mathcal{A})w$:

- Erzeuge aus code($\mathcal{A}$)w die Kodierung der Anfangskonfiguration

$$\text{code}(\mathcal{A})\text{code}(\underbrace{q_1 w}_{k_0}).$$

- Simuliere die Schritte von $\mathcal{A}$, ausgehend vom jeweiligen Konfigurationskode

$$\ldots ba^j bba^i b \ldots \text{ (Zustand } q_j\text{, gelesenes Symbol } a_i\text{)}.$$

  - Suche eine Transitionskodierung code(t), die mit $a^j ba^i$ beginnt.
  - Falls es so eine gibt, Änderungen der Konfigurationskodierung entsprechend t.
  - Sonst geht $\mathcal{U}$ in Stoppzustand. Dieser ist akzeptierend gdw. a^j in der Kodierung der Endzustandsmenge vorkommt.

  Es ist nicht schwer, das im Detail auszuarbeiten. Beachte allerdings, dass jede dieser Aufgaben viele Einzelschritte erfordert. Um z.B. ein code(t) zu finden, das mit $a^j ba^i$ 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 code(t) (Vergleich Symbol für Symbol).

□

Wir führen nun den angekündigten Unentscheidbarkeitsbeweis.

Satz 16.4
Die Sprache

$$UNIV = \{\text{code}(\mathcal{A})w \in \Sigma^* \mid \mathcal{A} \textit{ ist DTM über } \Sigma,\ \textit{die } w \textit{ akzeptiert}\}$$

ist partiell entscheidbar, aber nicht entscheidbar.

Beweis.

1) Bei Eingabe x geht die DTM, welche ein partielles Entscheidungsverfahren für $UNIV$ ist, wie folgt vor:

   - Teste, ob x von der Form $x = x_1 w$ ist mit

$$x_1 \in CODE \text{ und } w \in \Sigma^*.$$

   - Wenn nein, gehe in Endlosschleife.
   - Andernfalls wende $\mathcal{U}$ auf x an. Wenn $\mathcal{U}$ in akzeptierender Stoppkonfiguration anhält, stoppe. Wenn $\mathcal{U}$ in nicht akzeptierender Stoppkonfiguration anhält, gehe in Endlosschleife.

2) Angenommen, $UNIV$ ist rekursiv. Dann ist auch $\overline{UNIV} = \Sigma^* \setminus UNIV$ rekursiv und es gibt eine DTM $\mathcal{A}_0$, die $\overline{UNIV}$ entscheidet.

   Wir betrachten nun die Sprache

   $$D = \{\text{code}(\mathcal{A}) \mid \text{code}(\mathcal{A})\text{code}(\mathcal{A}) \notin UNIV\}$$

   d.h. die Maschine $\mathcal{A}$ akzeptiert ihre eigene Kodierung nicht).

   Diese Sprache kann leicht mit Hilfe von $\mathcal{A}_0$ entschieden werden:

   - Bei Eingabe x dupliziert man x und
   - startet dann $\mathcal{A}_0$ mit Eingabe xx.

   Es sei $\mathcal{A}_D$ die DTM, die D entscheidet. Es gilt nun (für $\mathcal{A} = \mathcal{A}_D$):

   $$\begin{array}{lll} \mathcal{A}_D \text{ akzeptiert code}(\mathcal{A}_D) & \text{gdw. } \text{code}(\mathcal{A}_D) \in D & (\text{denn } L(\mathcal{A}_D) = D) \\ & \text{gdw. } \text{code}(\mathcal{A}_D)\text{code}(\mathcal{A}_D) \notin UNIV & (\text{nach Def. } D) \\ & \text{gdw. } \mathcal{A}_D \text{ akzeptiert code}(\mathcal{A}_D) \text{ nicht}. & (\text{nach Def. } UNIV) \end{array}$$

   Widerspruch. □

Die in Teil 2 des Beweises verwendete Vorgehensweise nennt man *Diagonalisierung*. Es gibt zwei Dimensionen

- Turingmaschinen
- Eingabe der TM,

und die Sprache D entspricht der Diagonalen: TM wird als Eingabe ihre eigene Kodierung gegeben.

Ein anderes bekanntes Beispiel für ein Diagonalisierungsargument ist Cantors Beweis für die Überabzählbarkeit von $\mathbb{R}$, siehe Mathe 1. Auch im Beweis von Satz 13.19 haben wir ein Diagonalisierungsargument benutzt.

Wir wenden im folgenden Einschub Diagonalisierung an, um zu zeigen, dass es nicht-kontextsensitive Typ-0-Sprachen gibt.

Satz 16.5
$\mathcal{L}_1 \subset \mathcal{L}_0$.

Beweis. Es sei

- $G_0, G_1, \ldots$ eine effektive (d.h. mit TM machbare) Aufzählung aller kontextsensitiven 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

$$L = \{w_i \mid i \geq 0 \wedge w_i \notin L(G_i)\} \quad \text{(Diagonalisierung!)}.$$

- L ist Turing-erkennbar und damit aus $\mathcal{L}_0$. 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 12.7)
- L ist nicht kontextsensitiv. Anderenfalls gäbe es einen Index k mit $L = L(G_k)$. Nun ist aber

$$\begin{aligned} w_k \in L(G_k) \quad &\underline{\text{gdw}}. \quad w_k \in L && \text{(denn } L(G_k) = L\text{)} \\ &\underline{\text{gdw}}. \quad w_k \notin L(G_k) && \text{(nach Def. } L\text{)} \end{aligned}$$

  Widerspruch.

□

Aus der Unentscheidbarkeit von $UNIV$ kann man weitere wichtige Unentscheidbarkeitsresultate herleiten. $UNIV$ ist offensichtlich nichts anderes als das Wortproblem für DTMs, kodiert als Turingmaschine. Es folgt daher sofort das folgende Theorem.

Satz 16.6

Das Wortproblem für DTM ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das zu jeder gegebenen DTM $\mathcal{A}$ und jedem Eingabewort w entscheidet, ob $\mathcal{A}$ das Wort w akzeptiert.

Eine Variante ist das Wortproblem für Typ 0-Grammatiken, d.h. die Frage, ob für eine gegebene Typ 0-Grammatik G und ein gegebenes Wort w gilt, dass $w \in L(G)$. Wäre diese Variante entscheidbar, so wäre auch das Wortproblem für DTMs entscheidbar, was ja nach Satz 16.6 nicht der Fall ist: gegeben eine DTM $\mathcal{A}$ und ein Eingabewort w könnte man zunächst $\mathcal{A}$ in eine äquivalente Typ 0-Grammatik G wandeln (wie im Beweis von Satz 12.1) und dann entscheiden, ob $w \in L(G)$. Beachte, dass die Übersetzung im Beweis von Satz 12.1 effektiv ist, also mit TMs implementierbar. Auch das Wortproblem für Typ 0-Grammatiken ist also unentscheidbar.

Satz 16.7

Das Halteproblem für DTM ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das zu jeder gegebenen DTM $\mathcal{A}$ entscheidet, ob $\mathcal{A}$ beginnend mit leerem Eingabeband terminiert.

Beweis. Wir zeigen: wäre das Halteproblem entscheidbar, so auch das Wortproblem. Das ist aber nicht der Fall.

Um von einer gegebenen DTM $\mathcal{A}$ und einem gegebenem Wort w zu entscheiden, ob $w \in L(\mathcal{A})$, könnte man dann nämlich wie folgt vorgehen:

Modifiziere $\mathcal{A}$ zu einer TM $\hat{\mathcal{A}}$:

- $\hat{\mathcal{A}}$ schreibt zunächst w auf das Band.
- Danach verhält sich $\hat{\mathcal{A}}$ wie $\mathcal{A}$.
- Stoppt $\mathcal{A}$ mit *akzeptierender* Stoppkonfiguration, so stoppt auch $\hat{\mathcal{A}}$.
  Stoppt $\mathcal{A}$ mit *nichtakzeptierender* Stoppkonfiguration, so geht $\hat{\mathcal{A}}$ in Endlosschleife.

Damit gilt:

$$\mathcal{A} \text{ akzeptiert } w \quad \underline{\text{gdw}}. \quad \hat{\mathcal{A}} \text{ hält mit leerem Eingabeband.}$$

Mit dem Entscheidungsverfahren für das Halteproblem, angewandt auf $\hat{\mathcal{A}}$, könnte man also das Wortproblem „ist w in $L(\mathcal{A})$“ entscheiden. □

Satz 16.8

Das Leerheitsproblem für DTM ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das bei gegebener DTM $\mathcal{A}$ entscheidet, ob $L(\mathcal{A}) = \emptyset$.

Beweis. Da das Halteproblem unentscheidbar ist, ist mit Satz 15.3 auch dessen Komplement unentscheidbar. Wäre aber das Leerheitsproblem entscheidbar, so auch das Komplement des Halteproblems.

Um bei gegebener DTM $\mathcal{A}$ zu entscheiden, ob $\mathcal{A}$ auf leerer Eingabe *nicht* hält, konstruiert man die DTM $\hat{\mathcal{A}}$ wie folgt:

- $\hat{\mathcal{A}}$ löscht seine Eingabe.
- Danach verhält sich $\hat{\mathcal{A}}$ wie $\mathcal{A}$.
- Stoppt die Berechnung, so geht $\hat{\mathcal{A}}$ in eine akzeptierende Stoppkonfiguration.

Offenbar hält $\mathcal{A}$ auf *nicht* dem leeren Eingabeband $\underline{\text{gdw}}$. $L(\hat{\mathcal{A}}) = \emptyset$. □

Man kann aus Satz 16.8 leicht folgern, dass auch das Leerheitsproblem für Typ 0-Grammatiken unentscheidbar ist, siehe die Bemerkung nach Satz 16.6.

Satz 16.9

Das Äquivalenzproblem für DTM ist unentscheidbar.

Beweis. Offenbar kann man leicht eine DTM $\hat{\mathcal{A}}$ konstruieren mit $L(\hat{\mathcal{A}}) = \emptyset$.

Wäre das Äquivalenzproblem

$$L(\mathcal{A}_1) = L(\mathcal{A}_2)?$$

entscheidbar, so könnte man durch den Test

$$L(\mathcal{A}) = L(\hat{\mathcal{A}})?$$

das Leerheitsproblem für $\mathcal{A}$ entscheiden. □

Auch hier folgt wieder, dass das korrespondierende Äquivalenzproblem für Typ 0-Grammatiken unentscheidbar ist. An dieser Stelle, wo wir die Existenz unentscheidbarer aber dennoch partiell entscheidbarer Probleme bewiesen haben, kommen wir kurz auf die Abschlußeigenschaften von Typ 0-Sprachen zurück. Es stellt sich heraus, dass diese nicht unter Komplement abgeschlossen sind. Dies ist eine fundamentale Beobachtung aus aus der Sicht der Theorie der Berechenbarkeit.

Satz 16.10

$\mathcal{L}_0$ ist nicht unter Komplement abgeschlossen.

Beweis. Wir wissen von der in Satz 16.4 eingeführten Sprache $UNIV$:

- $UNIV$ ist partiell entscheidbar, d.h. gehört zu $\mathcal{L}_0$.
- $UNIV$ ist *nicht* entscheidbar.

Das Komplement $\overline{UNIV}$ gehört aber nicht zu $\mathcal{L}_0$: andernfalls wäre $\overline{UNIV}$ ja auch partiell entscheidbar und mit Satz 15.3 (Teil 4) würde folgen, dass $UNIV$ *entscheidbar* ist. □

Das in den Beweisen der Sätze 16.6 bis 16.9 gewählte Vorgehen nennt man *Reduktion*:

- Das Lösen eines Problems P_1 (z.B. Halteproblem) wird auf das Lösen eines Problem P_2 (z.B. Äquivalenzproblem) reduziert.
- Wäre daher P_2 entscheidbar, so auch P_1.
- Weiss man bereits, dass P_1 unentscheidbar ist, so folgt daher, dass auch P_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, dass man dann reduzieren kann). Formal lassen sich Reduktionen wie folgt definieren.

Definition 16.11 (Reduktion)

1) Eine *Reduktion von $L_1 \subseteq \Sigma^*$ auf $L_2 \subseteq \Sigma^*$* ist eine berechenbare Funktion

$$\mathrm{f} : \Sigma^* \to \Sigma^*,$$

für die gilt:

$$w \in L_1 \quad \underline{\text{gdw}}. \quad \mathrm{f}(w) \in L_2.$$

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.

Lemma 16.12

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). □

Wir werden im folgenden noch einige Beispiele für Reduktionen sehen. Zunächst zeigen wir mit Hilfe einer Reduktion des Halteproblems folgendes sehr starke Resultat:

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 Funktionen, die die Eigenschaft erfüllen, aber nicht alle Turing-erkennbaren Funktionen erfüllen sie.

Zum Beispiel ist das Anhalten auf dem leeren Wort eine semantische und nichttriviale Eigenschaft. Also ist die Unentscheidbarkeit des Halteproblems eine konkrete Instanz des hier bewiesenen, sehr allgemeinen Resultates. Da nach der Church-Turing-These DTMs äquivalent zu jedem anderen Berechnungsmodell sind, kann man diese Aussage intuitiv so verstehen, dass *alle interessanten Eigenschaften*, die das *Verhalten von Programmen* betreffen, unentscheidbar sind. Man kann beispielsweise kein Programm schreiben, dass ein Programm als Eingabe erhält und entscheidet, ob dieses terminiert oder sich in einer Endlosschleife verfängt. 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

$$E \subseteq \{L \subseteq \Sigma^* \mid L \text{ is Turing-erkennbar}\}.$$

Es folgt das angekündigte Resultat.

Satz 16.13 (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

$$L(E) := \{\text{code}(\mathcal{A}) \mid \mathcal{A} \textit{ DTM mit } L(\mathcal{A}) \in E\}$$

unentscheidbar.

Beweis. Sei E wie in Satz 16.13. Angenommen, $L(E)$ ist entscheidbar.

Wir zeigen, dass man dann auch ein Entscheidungsverfahren für das Halteproblem erhält, im Widerspruch zu Satz 16.7.

Das Ziel ist es, zu einer gegebenen DTM $\mathcal{A}$ eine DTM $\hat{\mathcal{A}}$ zu konstruieren, so dass

$$\mathcal{A} \text{ hält auf der leeren Eingabe} \quad \underline{\text{gdw}}. \quad L(\widehat{A}) \in E,$$

denn dann könnte man ein Entscheidungsverfahren für $L(E)$ verwenden, um zu entscheiden, ob $\mathcal{A}$ auf der leeren Eingabe hält.

Zur Konstruktion von $\hat{\mathcal{A}}$ verwenden wir

(a) eine Sprache $L_E \in E$ und eine dazugehörige DTM $\mathcal{A}_E$ mit $L(\mathcal{A}_E) = L_E$ (so eine Sprache und DTM existieren, da E nichttrivial)

(b) die Annahme, dass die leere Sprache E nicht erfüllt (dazu später mehr).

Konstruiere nun zu gegebener DTM $\mathcal{A}$ die DTM $\hat{\mathcal{A}}$ wie folgt:

- $\hat{A}$ speichert zunächst die Eingabe w
- danach löscht $\hat{A}$ das Eingabeband und verhält sich genau wie $\mathcal{A}$ (auf der leeren Eingabe)
- wenn $\mathcal{A}$ terminiert, so wird die ursprüngliche Eingabe w auf dem Eingabeband wiederhergestellt; dann verhält sich $\hat{A}$ wie die Maschine $\mathcal{A}_E$.

Man sieht leicht, dass $\hat{\mathcal{A}}$ sich wie gewünscht verhält:

1. wenn $\mathcal{A}$ auf ε anhält, dann erkennt $\hat{\mathcal{A}}$ die Sprache L_E, also $L(\hat{\mathcal{A}}) \in E$;
2. wenn $\mathcal{A}$ auf ε nicht anhält, dann erkennt $\hat{\mathcal{A}}$ die leere Sprache, also $L(\hat{\mathcal{A}}) \notin E$.

Wir gehen nun noch kurz auf die obige Annahme (b) ein. Wenn die leere Sprache E erfüllt, 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 oben zeigen, dass $L(\overline{E})$ unentscheidbar. Unentscheidbarkeit von $L(E)$ folgt dann mit Teil 3 von Satz 15.3 und der Beobachtung, dass

$$L(\overline{E}) = \overline{L(E)} \cap CODE.$$

□

17. 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 die 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 Abschnitt betrachten wir einige ausgewählte Probleme dieser Art.

Das folgende von Emil Post definierte Problem ist sehr nützlich, um mittels Reduktion Unentscheidbarkeit zu zeigen.

Definition 17.1 (Postsches Korrespondenzproblem)
Eine Instanz des *Postschen Korrespondenzproblems (PKP)* ist gegeben durch eine endliche Folge

$$P = (x_1, y_1), \ldots, (x_k, y_k)$$

von Wortpaaren mit $x_i, y_i \in \Sigma^+$, für ein endliches Alphabet Σ.

Eine *Lösung* des Problems ist eine *Indexfolge* $i_1, \ldots, i_m$ mit

- $m > 0$ und
- $i_j \in \{1, \ldots, k\}$,

so dass gilt: $x_{i_1} \cdots x_{i_m} = y_{i_1} \cdots y_{i_m}$.

Beispiel:

1) $P_1 = (a, aaa), (abaa, ab), (aab, b)$

   hat z.B. die Folgen 2, 1 und 1, 3 als Lösungen

   $abaa|a$ $\quad$ $a|aab$
   $ab|aaa$ $\quad$ $aaa|b$

   und damit auch 2, 1, 1, 3 sowie 2, 1, 2, 1, 2, 1, ...

2) $P_2 = (ab, aba), (baa, aa), (aba, baa)$

   hat keine Lösung: jede Lösung müsste mit dem Index 1 beginnen, da in allen anderen Paaren das erste Symbol von x_i and y_i verschieden ist. Danach kann man nur mit dem Index 3 weitermachen (beliebig oft). Dabei bleibt die Konkatenation $y_{i_1} \cdots y_{i_m}$ stets länger als die Konkatenation $x_{i_1} \cdots x_{i_m}$.

Um die Unentscheidbarkeit des PKP zu zeigen, 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 man beginnen muss, ist festgelegt.

Lemma 17.2
Das MPKP kann auf das PKP reduziert werden.

Beweis. Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$ eine Instanz des MPKP über dem Alphabet Σ. Es seien $\#, \$$ Symbole, die nicht in Σ vorkommen.

Wir definieren die Instanz $\mathrm{f}(P)$ des PKP über $\hat{\Sigma} = \Sigma \cup \{\#, \$\}$ wie folgt:

$$\mathrm{f}(P) := (x'_0, y'_0), (x'_1, y'_1), \ldots, (x'_k, y'_k), (x'_{k+1}, y'_{k+1}),$$

wobei gilt:

- Für $1 \leq i \leq k$ entsteht x'_i aus x_i, indem man *hinter jedem* Symbol ein $\#$ einfügt. Ist z.B. $x_i = abb$, so ist $x'_i = a\#b\#b\#$.
- Für $1 \leq i \leq k$ entsteht y'_i aus y_i, indem man *vor jedem* Symbol ein $\#$ einfügt.
- $x'_0 := \#x'_1$ und $y'_0 := y'_1$
- $x'_{k+1} := \$$ und $y'_{k+1} := \#\$$

Offenbar ist f berechenbar, und man kann leicht zeigen, dass gilt:

„Das MPKP P hat eine Lösung." gdw. „Das PKP $\mathrm{f}(P)$ hat eine Lösung." □

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 PKP nicht zulässig. Die Konstruktion liefert:

$$\begin{array}{rcccc} & (x'_0, y'_0) & (x'_1, y'_1) & (x'_2, y'_2) & (x'_3, y'_3) \\ \mathrm{f}(P) = & (\#a\#, \#a\#a\#a), & (a\#, \#a\#a\#a), & (a\#a\#b\#, \#b), & (\$, \#\$) \end{array}$$

Die Lösung 1, 2 von P liefert die Lösung 0, 2, 3 von $\mathrm{f}(P)$:

$\#a\#|\, ab\#a\#b\#|\$$
$\#a\#\, b\#a|\#b|\#\$$

Die Sequenz 2,1 liefert keine Lösung von $\mathrm{f}(P)$: wegen der Verwendung des $\#$-Symbols muss jede Lösung von $\mathrm{f}(P)$ 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 17.3
Das Halteproblem kann auf das MPKP reduziert werden.

Beweis. Gegeben sei eine DTM $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ und ein Eingabewort $w \in \Sigma^*$.

Wir müssen zeigen, wie man $\mathcal{A}, w$ effektiv in eine Instanz $\mathrm{f}(\mathcal{A}, w)$ des MPKP überführen kann, so dass gilt:

$$\mathcal{A} \text{ hält auf Eingabe } w \text{ gdw. } \mathrm{f}(\mathcal{A}, w) \text{ hat eine Lösung.}$$

Wir verwenden für das MPKP $\mathrm{f}(\mathcal{A}, w)$ das Alphabet $\Gamma \cup Q \cup \{\#\}$ mit $\# \notin \Gamma \cup Q$. Das MPKP $\mathrm{f}(\mathcal{A}, w)$ besteht aus den folgenden Wortpaaren:

1) Anfangsregel:

$$(\#,\ \#q_0w\#)$$

2) Kopierregeln:

$$(a,\ a) \text{ für alle } a \in \Gamma \cup \{\#\}$$

3) Übergangsregeln:

$$\begin{aligned}
(qa,\ q'a') &\quad \text{falls } (q,a,a',n,q') \in \Delta \\
(qa,\ a'q') &\quad \text{falls } (q,a,a',r,q') \in \Delta \\
(bqa,\ q'ba') &\quad \text{falls } (q,a,a',l,q') \in \Delta \text{ und } b \in \Gamma \\
(\#qa,\ \#q'\not{b}\,a') &\quad \text{falls } (q,a,a',l,q') \in \Delta \\
(q\#,\ q'a'\#) &\quad \text{falls } (q,\not{b},a',n,q') \in \Delta \\
(q\#,\ a'q'\#) &\quad \text{falls } (q,\not{b},a',r,q') \in \Delta \\
(bq\#,\ q'ba'\#) &\quad \text{falls } (q,\not{b},a',l,q') \in \Delta \\
(\#q\#,\ \#q'\not{b}\,a'\#) &\quad \text{falls } (q,\not{b},a',l,q') \in \Delta
\end{aligned}$$

4) Löschregeln:

   (aq,q) und (qa,q) für alle $a \in \Gamma$ und $q \in Q$ Stoppzustand

   (O.B.d.A. hänge in $\mathcal{A}$ 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:

$$(q\#\#,\#) \text{ für alle } q \in Q \text{ mit } q \text{ Stoppzustand}$$

Falls $\mathcal{A}$ bei Eingabe w hält, so gibt es eine Berechnung

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_t$$

mit $k_0 = q_0w$ 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. $(q, a, a', r, q') \in \Delta$ und $k_0 = \not{b}\, qab \not{b}$

$\#|a|q\, a|b|b|\#$
$\#\ a\ q\, a\ b\ b\ \#\,|\,a\,|\,a', q'|\,b\,|\,b\,|\,\#$

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:

- Man muss mit k_0 beginnen, da wir das MPKP betrachten.
- Durch Kopier- und Übergangsregeln kann man die erzeugten Wörter offensichtlich nicht gleich lang machen.
- Daher muss ein Stoppzustand erreicht werden, damit Lösch- und Abschlussregeln eingesetzt werden können.

□

Da das Halteproblem unentscheidbar ist, folgt die Unentscheidbarkeit des MPKP und damit (wegen Lemma 17.2) die Unentscheidbarkeit des PKP.

Satz 17.4
Das PKP ist unentscheidbar.

Wir verwenden dieses Resultat, um Unentscheidbarkeit von Problemen für kontextfreie und kontextsensitive Sprachen nachzuweisen. Wir zeigen zunächst:

Lemma 17.5
Das Schnitt-Leerheitsproblem für kontextfreie Grammatiken ist unentscheidbar, d.h. gegeben zwei kontextfreie Grammatiken G_1, G_2 ist nicht entscheidbar, ob gilt:

$$L(G_1) \cap L(G_2) = \emptyset.$$

Beweis. Wir reduzieren das PKP auf das Komplement des Schnitt-Leerheitsproblems, d.h. wir zeigen:

Zu jeder Instanz P des PKP kann man effektiv kontextfreie Grammatiken $G_P^{(l)}, G_P^{(r)}$ konstruieren, so dass gilt:

$$P \text{ hat Lösung} \quad \underline{\text{gdw}}. \quad L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset.$$

Da das PKP unentscheidbar ist und ein Problem entscheidbar ist gdw. sein Komplement entscheidbar ist, folgt die Aussage des Lemmas.

Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$. Wir definieren $G_P^{(l)} = (N_l, \Sigma_l, P_l, S_l)$ mit

- $N_l = \{S_l\}$,
- $\Sigma_l = \Sigma \cup \{1, \ldots, k\}$ und
- $P_l = \{S_l \longrightarrow w_i S_l i, S_l \longrightarrow w_i i \mid 1 \leq i \leq k\}$.

$G_P^{(r)}$ wird entsprechend definiert. Es gilt:

$$L(G_P^{(l)}) = \{x_{i_1} \ldots x_{i_m} i_m \ldots i_1 \mid m \geq 1, i_j \in \{1, \ldots, k\}\}$$

$$L(G_P^{(r)}) = \{y_{i_1} \ldots y_{i_m} i_m \ldots i_1 \mid m \geq 1, i_j \in \{1, \ldots, k\}\}$$

Daraus folgt nun unmittelbar:

$$\begin{aligned} & L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset \\ \underline{\text{gdw}}. \quad & \exists m \geq 1 \; \exists i_1, \ldots, i_m \in \{1, \ldots, k\} : x_{i_1} \ldots x_{i_m} i_m \ldots i_1 = y_{i_1} \ldots y_{i_m} i_m \ldots i_1 \\ \underline{\text{gdw}}. \quad & P \text{ hat Lösung.} \end{aligned}$$

□

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.

Wir wissen jedoch bereits, dass jede kontextfreie Sprache auch kontextsensitiv ist und dass die kontextsensitiven Sprachen unter Schnitt abgeschlossen sind (Satz 12.6). Daraus folgt der folgende Satz.

Satz 17.6

Für kontextsensitive Grammatiken sind das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.

Beweis. **Beachte:**

Es existiert eine einfache Reduktion des Schnitt-Leerheitsproblems kontextfreier Sprachen auf das Leerheitsproblem kontextsensitiver Sprachen: gegeben kontextfreie Grammatiken G_1 und G_2, konstruiere kontextsensitive 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

$$L(G) = \emptyset \text{ gdw. } L(G) = L(G_\emptyset) \quad (G_\emptyset : \text{ kontextsensitive Grammatik mit } L(G_\emptyset) = \emptyset).$$

□

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 Grammatiken *effektiv* in das entsprechende Automatenmodell übersetzen kann (und umgekehrt).

Satz 17.7

Für kontextfreie Grammatiken ist das Äquivalenzproblem unentscheidbar.

Beweis.

1. Man kann sich leicht überlegen, dass die Sprachen $L(G_P^{(l)})$ und $L(G_P^{(r)})$ aus dem Beweis von Lemma 17.5 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 (effektiv berechenbaren) deterministischen Kellerautomaten und damit eine kontextfreie Grammatik für

   $$\overline{L(G_P^{(l)})}$$

   (siehe z.B. [Wege93], Satz 8.1.3).

3. Es sei $\overline{G}$ die kontextfreie Grammatik mit $L(\overline{G}) = \overline{L(G_P^{(l)})}$. Nun gilt:

   $$\begin{aligned} L(G_P^{(l)}) \cap L(G_P^{(r)}) = \emptyset \quad & \text{gdw. } L(G_P^{(r)}) \subseteq L(\overline{G}) \\ & \text{gdw. } L(G_P^{(r)}) \cup L(\overline{G}) = L(\overline{G}) \\ & \text{gdw. } L(G_\cup) = L(\overline{G}), \end{aligned}$$

   wobei $G_\cup$ die effektiv konstruierbare kontextfreie Grammatik für $L(G_P^{(r)}) \cup L(\overline{G})$ ist.

4. Wäre also das Äquivalenzproblem für kontextfreie Grammatiken entscheidbar, so auch

   $$L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset$$

   und damit das PKP.

□

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 einerseits die Idee zugrunde, dass mit größeren Eingaben üblicherweise auch der Ressourcenbedarf zum Verarbeiten der Eingabe wächst. Andererseits wird aber von konkreten Eingaben abstrahiert und nur deren Größe betrachtet: es ist durchaus vorstellbar, dass der Ressourcenverbrauch auf verschiedenen Eingaben derselben Größe stark variiert. Man interessiert sich dann für den *maximalen* Ressourcenbedarf unter allen Eingaben derselben Größe.

Es ist wichtig, sauber zwischen folgenden Fragestellungen zu entscheiden:

Komplexität eines konkreten Algorithmus/Programmes. Wieviel Ressourcen verbaucht dieser Algorithmus? Derartige Fragestellungen gehören in die praktische Informatik und zum 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. Dies ist in manchen praktischen Anwendungen realistisch, für andere aber deutlich zu pessimistisch, da dort der schlimmste Fall selten oder niemals vorkommt.

18. Komplexitätsklassen

Ene 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 18.1 (f(n)-zeitbeschränkt, f(n)-platzbeschränkt)
Es sei $\mathrm{f} : \mathbb{N} \to \mathbb{N}$ eine Funktion und $\mathcal{A}$ eine DTM über Σ.

1) $\mathcal{A}$ heißt f(n)-*zeitbeschränkt*, falls für alle $w \in \Sigma^*$ die Maschine $\mathcal{A}$ bei Eingabe w nach $\leq \mathrm{f}(|w|)$ Schritten anhält.
2) $\mathcal{A}$ heißt f(n)-*platzbeschränkt*, falls für alle $w \in \Sigma^*$ die Maschine $\mathcal{A}$ bei Eingabe w $\leq \mathrm{f}(|w|)$ 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)-zeitbeschränkte TM muß das nicht unbedingt der Fall sein.

Definition 18.2 (Komplexitätsklassen)

$$\begin{aligned}
\mathsf{DTIME}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine } \mathrm{f}(n)\text{-}\mathbf{zeit}\text{beschränkte } \mathbf{D}\text{TM } \mathcal{A} \text{ mit } L(\mathcal{A}) = L\} \\
\mathsf{NTIME}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine } \mathrm{f}(n)\text{-}\mathbf{zeit}\text{beschränkte } \mathbf{N}\text{TM } \mathcal{A} \text{ mit } L(\mathcal{A}) = L\} \\
\mathsf{DSPACE}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine } \mathrm{f}(n)\text{-}\mathbf{platz}\text{beschränkte } \mathbf{D}\text{TM } \mathcal{A} \text{ mit } L(\mathcal{A}) = L\} \\
\mathsf{NSPACE}(\mathrm{f}(n)) &:= \{L \mid \text{es gibt eine } \mathrm{f}(n)\text{-}\mathbf{platz}\text{beschränkte } \mathbf{N}\text{TM } \mathcal{A} \text{ mit } L(\mathcal{A}) = L\}
\end{aligned}$$

Im folgenden Beobachten wir einige elementare Zusammenhänge zwischen Komplexitätsklassen.

Satz 18.3

1) $\mathsf{DTIME}(\mathrm{f}(n)) \subseteq \mathsf{DSPACE}(\mathrm{f}(n)) \subseteq \mathsf{NSPACE}(\mathrm{f}(n))$
2) $\mathsf{DTIME}(\mathrm{f}(n)) \subseteq \mathsf{NTIME}(\mathrm{f}(n))$
3) $\mathsf{NSPACE}(\mathrm{f}(n)) \subseteq \mathsf{DTIME}(2^{\mathcal{O}(\mathrm{f}(n))})$

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 $L \in \mathsf{NSPACE}(\mathrm{f}(n))$. Dann gibt es eine $\mathrm{f}(n)$-platzbeschränkte NTM $\mathcal{A}$ mit $L(\mathcal{A}) = L$. Mittels der Konstruktion aus dem Beweis von Satz 11.6 könnten wir $\mathcal{A}$ in eine DTM $\mathcal{A}'$ wandeln, so dass $L(\mathcal{A}) = L(\mathcal{A}')$. Allerdings terminiert $\mathcal{A}'$ nicht auf jeder Eingabe und ist daher nicht $2^{\mathcal{O}(\mathrm{f}(n))}$-zeitbeschränkt. Wir brauchen also eine bessere Konstruktion.

Wir stellen dazu die Berechnungen von $\mathcal{A}$ auf Eingaben w der Länge n als gerichteten Graphen dar, den sogenannten *Konfigurationsgraph* $G_{\mathcal{A},n} = (V_{\mathcal{A},n}, E_{\mathcal{A},n})$, wobei

- $V_{\mathcal{A},n}$ die Menge der Konfigurationen von $\mathcal{A}$ mit Länge höchstens $\mathrm{f}(n)$ ist und
- $E_{\mathcal{A},n}$ diejenigen Kanten (k, k') enthält, so dass k' von k in einem Berechnungsschritt erreicht werden kann.

Es ist nun offensichtlich, dass w von $\mathcal{A}$ akzeptiert wird gdw. in $G_{\mathcal{A},n}$ vom Knoten $q_0 w$ aus eine akzeptierende Stoppkonfiguration erreichbar ist (wenn also ein Pfad von $\mathcal{A}$ zu einer solchen Konfiguration existiert).

Eine terminierende DTM $\mathcal{A}'$ mit $L(\mathcal{A}') = L(\mathcal{A})$ kann also wie folgt vorgehen:

- Bei Eingabe w der Länge n, konstruiere Konfigurationsgraph $G_{\mathcal{A},n}$
- überprüfe, ob von $q_0 w$ aus eine akzeptierende Stoppkonfiguration erreichbar ist.

Wir analysieren nun den Zeitbedarf von $\mathcal{A}'$. Zunächst schätzen wir die Größe von $G_{\mathcal{A},n}$. Die Anzahl der Konfigurationen von $\mathcal{A}$ der Länge $\leq \mathrm{f}(n)$ ist beschränkt durch

$$\mathrm{akonf}_{\mathcal{A}}(n) := |Q| \cdot \mathrm{f}(n) \cdot |\Gamma|^{\mathrm{f}(n)}$$

wobei $|Q|$ die Anzahl der möglichen Zustände beschreibt, $\mathrm{f}(n)$ die Anzahl der möglichen Kopfpositionen und $|\Gamma|^{\mathrm{f}(n)}$ die Anzahl der möglichen Beschriftungen von Bändern der Länge $\mathrm{f}(n)$.

Man sieht nun leicht, dass $|V_{\mathcal{A},n}| = \mathrm{akonf}_{\mathcal{A}}(n) \in 2^{\mathcal{O}(\mathrm{f}(n))}$ (beachte, dass $|Q|$ und $|\Gamma|$ Konstanten sind, da sie nicht von der Eingabe abhängen) und damit auch $|E_{\mathcal{A},n}| \in 2^{\mathcal{O}(\mathrm{f}(n))}$. Zudem ist es für eine DTM einfach, den Graph $G_{\mathcal{A},n}$ in Zeit $2^{\mathcal{O}(\mathrm{f}(n))}$ zu konstruieren. Es müssen danach noch $\leq 2^{\mathcal{O}(\mathrm{f}(n))}$ Erreichbarkeitstests gemacht werden (einer für jede akzeptierende Stoppkonfiguration in $G_{\mathcal{A},n}$), von denen jeder lineare Zeit (in der Größe des Graphen $G_{\mathcal{A},n}$) benötigt. Insgesamt ergibt sich ein Zeitbedarf von $2^{\mathcal{O}(\mathrm{f}(n))}$. □

Wir betrachten die folgenden *fundamentalen* Komplexitätsklassen:

Definition 18.4 ($\mathsf{P}, \mathsf{NP}, \mathsf{PSpace}, \mathsf{NPSpace}, \mathsf{ExpTime}$)

$$\mathsf{P} := \bigcup_{\mathrm{p\ Polynom\ in\ } n} \mathsf{DTIME}(\mathrm{p}(n))$$

$$\mathsf{NP} := \bigcup_{\mathrm{p\ Polynom\ in\ } n} \mathsf{NTIME}(\mathrm{p}(n))$$

$$\mathsf{PSpace} := \bigcup_{\mathrm{p\ Polynom\ in\ } n} \mathsf{DSPACE}(\mathrm{p}(n))$$

$$\mathsf{NPSpace} := \bigcup_{\text{p Polynom in } n} \mathsf{NSPACE}(\mathrm{p}(n))$$

$$\mathsf{ExpTime} := \bigcup_{\text{p Polynom in } n} \mathsf{DTIME}(2^{\mathrm{p}(n)})$$

Aus Satz 18.3 ergibt sich sofort der folgende Zusammenhang zwischen diesen Klassen:

Korollar 18.5
$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{NPSpace} \subseteq \mathsf{ExpTime}$.

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 B
- Im Gegensatz dazu nimmt man an, dass NP viele Probleme enthält, die *nicht effizient lösbar* sind (engl. *intractable*)
- Die in Polynomialzeit lösbaren Probleme (also die in P) stimmen in allen bekannten deterministischen Berechnungsmodellen überein. Dies nennt man die *erweiterte Church-Turing These*.

Die Bedeutung der Komplexitätsklasse P ist dadurch hinreichend motiviert. Im Gegensatz dazu ist die Bedeutung von NP etwas schwieriger einzusehen, da nicht-deterministische Maschinen in der Praxis ja nicht existieren. Auch diese Klasse ist aber von großer Bedeutung, da

1. sehr viele natürlich 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 zwei typische Beispiele für Probleme in NP, die wir später noch genauer analysieren werden.

Definition 18.6 (SAT)
SAT ist die Menge der erfüllbaren („satisfiable“) aussagenlogischen Formeln über den Operatoren $\neg, \wedge, \vee$.

Beispiel:

- Die Formel $(x_1 \wedge x_2) \vee \neg x_1$ ist erfüllbar, da sie von der Belegung $x_1 \mapsto 1$, $x_2 \mapsto 1$ wahr gemacht wird.
- $x_1 \wedge \neg x_1$ ist *nicht* erfüllbar.

Streng genommen geht es natürlich um eine geeignete Kodierung dieses Problems als formale Sprache, so muß man beispielsweise die potentiell unendlich vielen Aussagenvariablen $x_1, x_2, \ldots$ mittels eines endlichen Alphabetes darstellen. Von solchen Details, die leicht auszuarbeiten sind, werden wir im folgenden aber abstrahieren.

Lemma 18.7
SAT $\in$ NP.

Beweis. Die NTM, die SAT in polynomieller Zeit erkennt, arbeitet wie folgt:

- Teste, ob die Eingabe eine aussagenlogische Formel ist.

  Dies ist eine Instanz des Wortproblems für kontextfreie Grammatiken, das in Polynomialzeit lösbar ist (z.B. mit dem CYK-Algorithmus)

- Die Variablen in der Eingabeformel φ seien $x_1, \ldots, x_n$. Schreibe nicht-deterministisch ein beliebiges Wort $u \in \{0,1\}^*$ der Länge n hinter die Eingabe auf das Band

- Betrachte u als Belegung mit $x_i \mapsto 1$ gdw. das i-te Symbol von u eine 1 ist.

  Überprüfe nun deterministisch und in Polynomialzeit, ob diese Belegung die Formel φ erfüllt (man überlegt sich leicht, dass dies tatsächlich möglich ist). Akzeptiere, wenn das der Fall ist und verwerfe sonst.

Die NTM akzeptiert ihre Eingabe gdw. es eine erfolgreiche Berechnung gibt gdw. die Eingabe eine erfüllbar aussagenlogische Formel ist. □

Wie betrachten ein weiteres Problem in NP.

Definition 18.8 (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 18.9
CLIQUE $\in$ NP.

Beweis. Eine NTM, die CLIQUE in polynomieller Zeit erkennt, konstruiert man analog zum Beweis von Lemma 18.7: gegeben (G, k),

- Schreibe nicht-deterministisch eine Knotenmenge C der Grösse k auf das Band
- Überprüfe deterministisch und in Polynomialzeit, ob C eine Clique in G ist.

□

Algorithmen, wie sie in den Beweisen von Lemma 18.7 und 18.7 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* Wahrheitsbelegung B für $x_1, \ldots, x_n$
- Überprüfe determinstisch und in Polyzeit, ob B 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.

Wir werden später den Zusammenhang der (wahrscheinlich) sehr unterschiedlichen Komplexitätsklassen P und NP genauer untersuchen. Interessanterweise stellen sich die *Platz*-komplexitätsklassen PSpace und $\mathsf{NPSpace}$, die ja komplett analog zu P und NP definiert sind, als identisch heraus. Das folgende Resultat ist als *Savitch's Theorem* bekannt und wurde um 1970 bewiesen.

Satz 18.10
$\mathsf{PSpace} = \mathsf{NPSpace}$.

Der Beweis beruht auf einer cleveren Simulation von NTMs mittels DTMs; für Details wird auf Spezialvorlesungen zur Komplexitätstheorie verwiesen. Savitch's Theorem hat unter anderem folgende Konsequenzen:

- Die Komplexitätsklasse $\mathsf{NPSpace}$ wird im allgemeinen nicht explizit betrachtet;
- Wenn man nachweisen will, dass ein Problem in PSpace ist, dann kann man o.B.d.A. eine nicht-deterministische Turingmaschine verwenden, was oft praktisch ist (man kann also *raten*).

Betrachten wir an dieser Stelle noch einmal die Inklusionen aus Korollar 18.5:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSpace} \subseteq \mathsf{ExpTime}. \qquad (*)$$

Mit einem verfeinerten Diagonalisierungsargument kann man nachweisen, dass

$$\mathsf{P} \neq \mathsf{ExpTime}$$

man kann mit einer DTM in exponentieller Zeit also mehr Probleme lösen als in polynomieller Zeit. Das bedeutet natürlich, dass auch mindestens eine der drei Inklusionen in $(*)$ echt sein muss. Leider weiß man bis heute nicht, welche Inklusion das ist. Insbesondere wird die Frage, ob

$$\mathsf{P} = \mathsf{NP} \qquad \text{oder} \qquad \mathsf{P} \neq \mathsf{NP}$$

als wichtigstes offenes Problem der Informatik angesehen. Es glaubt allerdings kaum jemand, dass $\mathsf{P} = \mathsf{NP}$. Ein Beweis von $\mathsf{P} \neq \mathsf{NP}$ würde tiefe neue Einsichten in die Komplexitätstheorie gewähren und hätte wahrscheinlich sehr nützliche Anwendungen in der Kryptographie (wo man an schwer zu lösenden Problemen interessiert ist). Ein Beweis von $\mathsf{P} = \mathsf{NP}$ hätte (wahrscheinlich) revolutionäre Konsequenzen weil dann auf einen Schlag sehr viele Probleme effizient lösbar wären, die heute als nicht effizienz lösbar angenommen werden.

19. NP-vollständige Probleme

Wegen $\mathsf{P} \subseteq \mathsf{NP}$ enthält NP auch einfach lösbare Probleme, es sind also nicht *alle* Probleme in NP schwierig zu lösen. Wie kann man die schwierigen Probleme in NP identifizieren und von den einfachen abgrenzen, wenn nicht einmal bekannt ist, ob $\mathsf{P} = \mathsf{NP}$ gilt? Man behilft 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 gilt $\mathsf{P} = \mathsf{NP}$ (was als extrem unwahrscheinlich angesehen wird).

Um *polynomiellen Mehraufwand* zu formalisieren, verfeinern wir in geeigneter Weise den Begriff der Reduktion.

Definition 19.1 (Polynomialzeitreduktion, $\leq_p$)

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 $L \leq_p L'$.

Die meisten wichtigen Eigenschaften von Reduktionen gelten auch für Polynomialzeitreduktionen, insbesondere:

Lemma 19.2
Wenn $L_1 \leq_p L_2$ und $L_2 \leq_p L_3$, dann $L_1 \leq_p L_3$.

Der Beweis dieses Lemmas ähnelt dem Beweis von Lemma 19.4 und wird als Übung gelassen. Wir definieren nun die zentralen Begriffe dieses Kapitels.

Definition 19.3 (NP-hart, NP-vollständig)

1) Eine Sprache L heißt NP-*hart* wenn für *alle* $L' \in \mathsf{NP}$ gilt: $L' \leq_p L$.
2) L heißt *NP-vollständig* wenn $L \in \mathsf{NP}$ und L ist NP-hart.

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 19.4
Für jede NP-vollständige Sprache L gilt: wenn $L \in \mathsf{P}$, dann $\mathsf{P} = \mathsf{NP}$.

Beweis. Es sei L NP-vollständig und $L \in P$, Dann gibt es ein Polynom $p(n)$ und eine $p(n)$-zeitbeschränkte DTM $\mathcal{A}$ mit $L(\mathcal{A}) = L$.

Wir müssen zeigen, dass daraus $\mathsf{NP} \subseteq \mathsf{P}$ folgt.

Sei also $L' \in \mathsf{NP}$. Da L NP-vollständig, gilt $L' \leq_p L$, d.h. es gibt eine Reduktion f von L' auf L, die in Zeit $q(n)$ berechenbar ist, wobei $q(n)$ ein Polynom ist.

Die Polynomialzeit-DTM, die L entscheidet, geht wie folgt vor:

- Bei Eingabe w berechnet sie f(w). Sie benötigt $\leq$ q($|w|$) viel Zeit. Daher ist auch die Länge der erzeugten Ausgabe $\leq |w| +$ q($|w|$).
- Wende Entscheidungsverfahren für L auf f(w) an.

Insgesamt benötigt man somit

$$\leq \mathrm{q}(|w|) + \mathrm{p}(|w| + \mathrm{q}(|w|))$$

viele Schritte, was ein Polynom in $|w|$ ist. □

Es ist zunächst nicht unmittelbar klar, warum NP-vollständige Probleme überhaupt existieren sollten. Überraschenderweise stellt sich aber heraus, dass es sehr viele solche Probleme gibt. Ein besonders wichtiges ist das Erfüllbarkeitsproblem der Aussagenlogik. Das folgende sehr bekannte Resultat wurde unabhängig voneinander von Cook und Levin bewiesen.

Satz 19.5

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 Probleme gemeinsam ist, dass sie (per Definition von NP) als das Wortproblem einer polynomiell zeitbeschränkten NTM aufgefasst werden können.

Sei also $\mathcal{A}$ eine $p(n)$-zeitbeschränkte NTM, mit $p(n)$ Polynom. Unser Ziel ist, für jede Eingabe w eine AL-Formel φ_w zu finden, so dass

1. w wird von $\mathcal{A}$ akzeptiert gdw. φ_w ist erfüllbar und
2. φ_w kann in polynomieller Zeit (in $|w|$) konstruiert werden.

Die Konstruktion von φ_w beruht auf den folgenden Ideen. Jede Berechnung von $\mathcal{A}$ auf Eingabe $w = a_0 \cdots a_{n-1}$ kann man wie folgt als Matrix darstellen:

$\not b$	$\cdots$	$\not b$	a_0, q_0	a_1	$\cdots$	a_{n-1}	$\not b$	$\cdots$	$\not b$
$\not b$	$\cdots$	$\not b$	b	a_1, q	$\cdots$	a_{n-1}	$\not b$	$\cdots$	$\not b$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Die Anzahl der Zeilen ist beschränkt durch $p(n)$, die maximale Anzahl Schritte von $\mathcal{A}$ auf der Eingabe w. Um eine Nummerierung der Spalten zu erhalten, weisen wir der Spalte, in der in der ersten Zeile (Startkonfiguration!) der Kopf steht, die Nummer 0 zu. Da $\mathcal{A}$ aufgrund der Zeitbeschränkung maximal $p(n)$ Schritte nach links und nach rechts machen kann, sind die Spalten mit den Nummern $-p(n), \ldots, 0, p(n)$ ausreichend.

Diese Matrix lässt sich wiederrum mittels polynomiell vieler Aussagenvariablen darstellen, nämlich:

- $B_{a,i,t}$: 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

wobei $0 \leq t \leq p(n)$, $-p(n) \leq i \leq p(n)$, $a \in \Gamma$ und $q \in Q$. Wenn beispielsweise in der Startkonfiguration die Zelle 3 mit dem Symbol a beschriftet ist, so wird dies dadurch repräsentiert, dass $B_{a,3,0} \mapsto 1$ und $B_{b,3,0} \mapsto 0$ für alle $b \in \Gamma \setminus \{a\}$.

Die für die Eingabe w konstruierte Formel φ_w verwendet die Variablen $B_{a,i,t}$, $K_{i,t}$ und $Z_{q,t}$. Die generelle Idee ist, φ_w so zu konstruieren, dass erfüllende Belegungen von φ_w genau den akzeptierenden Berechnungen von $\mathcal{A}$ auf w entsprechen. Genauer ist φ_w eine Konjunktion, die aus den folgenden Konjunkten besteht:

- Die Berechnung beginnt mit der Startkonfiguration für $w = a_0 \cdots a_{n-1}$:

$$\psi_{\text{ini}} := Z_{q_0,0} \wedge K_{0,0} \wedge \bigwedge_{i<n} B_{a_i,i,0} \wedge \bigwedge_{n \leq i \leq p(n)} B_{\not b,i,0} \wedge \bigwedge_{-p(n) \leq i < 0} B_{\not b,i,0}.$$

- Die Übergangsrelation wird respektiert:

$$\psi_{\text{move}} := \bigwedge_{t \leq p(n)} \bigwedge_{-p(n) \leq i \leq p(n)} \bigwedge_{a \in \Gamma, q \in Q \text{ kein Stoppzustand}} \Big(\big(B_{a,i,t} \wedge K_{i,t} \wedge Z_{p,t}\big) \rightarrow \bigvee_{(q,a,a',M,q') \in \Delta \text{ so dass } -p(n) \leq M(i) \leq p(n)} \big(B_{a',M(i),t+1} \wedge K_{M(i),t+1} \wedge Z_{q',t+1}\big) \Big)$$

  wobei $M \in \{r, \ell, n\}$, $r(i) = i + 1$, $\ell(i) = i - 1$ 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).

- Zellen, die nicht unter dem Kopf sind, ändern sich nicht:

$$\psi_{\text{keep}} := \bigwedge_{t < p(n)} \bigwedge_{-p(n) \leq i \leq p(n)} \bigwedge_{a \in \Gamma} \Big(\big(\neg K_{i,t} \wedge B_{a,i,t}\big) \rightarrow B_{a,i,t+1} \Big)$$

- Die Eingabe wird akzeptiert:

$$\psi_{\text{acc}} := \bigwedge_{q \in Q \setminus F \text{ Stoppzustand}} \bigwedge_{t \leq p(n)} \neg Z_{q_f,t}$$

Es würde hier nicht funktionieren, das Vorkommen eines akzeptierenden Stoppzustandes zu fordern, denn dann wären unerwünschte Belegungen der folgenden Form möglich: eine verwerfende Berechnung von $\mathcal{A}$, die weniger als die maximal möglichen $p(n)$ Schritte macht, gefolgt von einer Konfiguration mit akzeptierendem Stoppzustand, obwohl $\mathcal{A}$ gar keine weiteren Schritte macht.

- Bandbeschriftung, Kopfposition, Zustand sind eindeutig und definiert:

$$\psi_{\text{aux}} := \bigwedge_{t,q,q',q\neq q'} \neg(Z_{q,t} \wedge Z_{q',t}) \wedge \bigwedge_{t,i,a,a',a\neq a'} \neg(B_{a,i,t} \wedge B_{a',i,t}) \wedge \bigwedge_{t,i,j,i\neq j} \neg(K_{i,t} \wedge K_{j,t})$$

$$\bigwedge_{t\leq p(n)} \bigvee Q \wedge \bigwedge_{t\leq p(n)} \bigvee_{-p(n)\leq i\leq p(n)} K_{i,t} \wedge \bigwedge_{t\leq p(n)} \bigwedge_{-p(n)\leq i\leq p(n)} \bigvee \Gamma$$

Setze nun

$$\varphi_w = \varphi_{\text{ini}} \wedge \varphi_{\text{move}} \wedge \varphi_{\text{keep}} \wedge \varphi_{\text{acc}} \wedge \varphi_{\text{aux}}.$$

Man überprüft leicht, dass $\mathcal{A}$ die Eingabe w akzeptiert gdw. φ_w erfüllbar. Idee:

"$\Leftarrow$" Wenn w von $\mathcal{A}$ akzeptiert wird, dann gibt es akzeptierende Berechnung

$$k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \cdots \vdash_{\mathcal{A}} k_m$$

von $\mathcal{A}$ auf w. Erzeuge daraus in der offensichtlichen Weise eine Belegung für die Variablen $B_{a,i,t}$, $K_{i,t}$ und $Z_{q,t}$, zum Beispiel:

$B_{a,i,t} \mapsto 1$ 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 die Folge $k_0, k_1, \ldots, k_m$ zuvor zu $k_0, k_1, \ldots, k_{p(n)}$ durch $p(n) - m$-maliges Wiederholen der Konfiguration k_m.

Indem man alle Konjunkte von φ_w durchgeht, überprüft man leicht, dass die erhaltene Belegung φ_w erfüllt. Also ist φ_w erfüllbar.

"$\Rightarrow$" Aus einer Belegung der Variablen $B_{a,i,t}$, $K_{i,t}$, $Z_{q,t}$, die φ_w erfüllt, liest man eine Konfigurationsfolge $k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \cdots \vdash_{\mathcal{A}} k_m$ ab, zum Beispiel:

Die i-te Zelle in k_t wird mit a beschriftet wenn $B_{a,i,t} \mapsto 1$.

Die Konfigurationsfolge endet, sobald ein Stoppzustand abgelesen wurde. Unter Verwendung der Tatsache, dass die Belegung alle Konjunkte von φ_w erfüllt, zeigt man nun leicht, dass es sich bei der abgelesenen Konfigurationsfolge um eine akzeptierende Berechnung von $\mathcal{A}$ auf w handelt.

□

Analog zu unserem Vorgehen bei der Unentscheidbarkeit erhalten wir weitere NP-vollständige Probleme durch polynomielle Reduktion von bereits als NP-vollständig nachgewiesenen Problemen wie SAT. Dies beruht auf Punkt 2) des folgenden Satzes, der die wichtigsten Zusammenhänge von NP und Polynomialzeitreduktionen zusammenfasst.

Satz 19.6

1) Ist $L_2 \in \mathsf{NP}$ und gilt $L_1 \leq_p L_2$, so ist auch L_1 in NP.

2) Ist L_1 NP-hart und gilt $L_1 \leq_p L_2$, so ist auch L_2 NP-hart.

Beweis.

1) Wegen $L_2 \in \mathsf{NP}$ gibt es eine polynomialzeitbeschränkte NTM $\mathcal{A}$, die L_2 akzeptiert. Wegen $L_1 \leq_p L_2$ gibt es eine Polynomialzeitreduktion f von L_1 auf L_2, also

$$w \in L_1 \ \ \underline{\text{gdw}}. \ \ \text{f}(w) \in L_2.$$

Die polynomialzeitbeschränkte NTM für L_1 arbeitet wie folgt:

- Bei Eingabe w berechnet sie zunächst f(w).
- Dann wendet sie $\mathcal{A}$ auf f(w) an.

2) Sei L_1 NP-hart und $L_1 \leq_p L_2$. Wir müssen zeigen, dass für alle $L \in \mathsf{NP}$ gilt:

$$L \leq_p L_2.$$

Sei also $L \in \mathsf{NP}$. Gesucht: Polynomialzeitreduktion f von L auf L_2, also

$$w \in L \ \ \underline{\text{gdw}}. \ \ \text{f}(w) \in L_2$$

Diese erhält man wie folgt:

- Da L_1 NP-hart ist, gibt es eine Polynomialzeitreduktion g von L auf L_1, also

$$w \in L \ \ \underline{\text{gdw}}. \ \ \text{g}(w) \in L_1.$$

- Wegen $L_1 \leq_p L_2$ gibt es eine Polynomialzeitreduktion h von L_1 auf L_2, also

$$u \in L_1 \ \ \underline{\text{gdw}}. \ \ \text{h}(u) \in L_2.$$

Dann ist f(w) := h(g(w)) wie gewünscht:

$$w \in L \ \ \underline{\text{gdw}}. \ \ \text{g}(w) \in L_1 \ \ \underline{\text{gdw}}. \ \ \text{h}(\text{g}(w)) \in L_2. \qquad \square$$

Mit Punkt 2) von Satz 19.6 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 Beispiele 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 $\mathsf{3SAT}$ spielt eine wichtige Rolle, da es oft einfacher ist, eine Reduktion von $\mathsf{3SAT}$ auf ein gegebenes Problem L zu finden als eine Reduktion von SAT.

Definition 19.7 (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 19.8

3SAT *ist* NP*-vollständig.*

Beweis.

1) 3SAT $\in$ NP folgt unmittelbar aus SAT $\in$ NP, da jedes 3SAT-Problem 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:

$$\varphi \text{ erfüllbar} \quad \underline{\text{gdw}}. \quad \varphi' \text{ erfüllbar.}$$

Beachte:

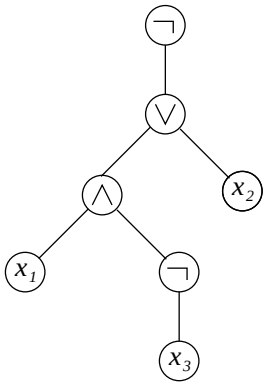
Es ist nicht gefordert, dass φ und φ' im logischen Sinne äquivalent sind, also von denselben Belegungen erfüllt werden.

Die Umformung erfolgt in mehreren Schritten, die wir am Beispiel der Formel

$$\neg((x_1 \wedge \neg x_3) \vee x_2)$$

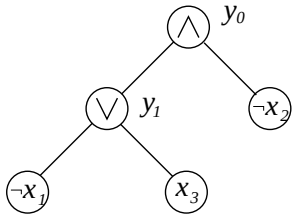
veranschaulichen.

Wir stellen diese Formel als Baum dar:



1. Schritt: Wende wiederholt die *de Morgan'schen Regeln* an und eliminiere doppelte Negationen, um die Negationszeichen zu den Blättern des Baumes zu schieben.

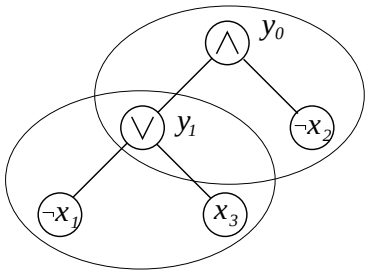
Dies ergibt den folgenden Baum (die Beschriftung y_0, y_1) wird im nächsten Schritt erklärt:



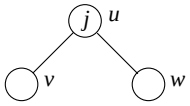
2. Schritt: Ordne jedem inneren Knoten eine neue Variable aus $\{y_0, y_1, \ldots\}$ zu, wobei die Wurzel y_0 erhält.

Intuitiv repräsentiert jede Variable y_i die Teilformel von φ, an dessen 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 $j \in \{\wedge, \vee\}$ ordnen wir eine Formel der folgenden Form zu:

$$(u \Leftrightarrow (v\ j\ w))$$

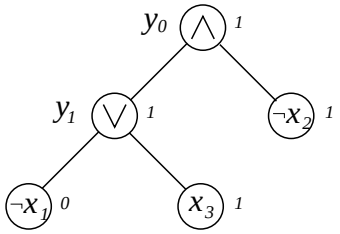
Diese Formeln werden nun konjunktiv mit y_0 verknüpft, was die Formel φ_1 liefert.

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.:



$$\begin{aligned} x_1 &= 1 \\ x_2 &= 0 \\ x_3 &= 1 \\ y_1 &= 1 \\ y_0 &= 1 \end{aligned}$$

Umgekehrt ist jede erfüllende Belegung für φ_1 auch eine für φ. Genauer gesagt kann man von den Blättern zu den Wurzeln 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 Belegungm 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 in 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.

Satz 19.9

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:

- $V := \{\langle i, j\rangle \mid 1 \leq i \leq m \text{ und } 1 \leq j \leq 3\}$
- $E := \{\{\langle i, p\rangle, \langle i', p'\rangle\}) \mid i \neq i' \text{ und } \ell_{ij} \neq \overline{\ell}_{pq}\}$, wobei

$$\overline{\ell} = \begin{cases} \neg x & \text{falls } \ell = x \\ x & \text{falls } \ell = \neg x \end{cases} .$$

- $k = m$

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 Graph G wie im Beweis von Satz 19.9, 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 19.10 (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 19.11
Mengenüberdeckung ist NP*-vollständig.*

Beweis.

1) Mengenüberdeckung ist in NP:
   - Wähle nichtdeterministisch Indices $i_1, \ldots, i_n$ und
   - 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 $M := \{1, \ldots, m, m+1, \ldots, m+n\}$.
   Für jedes $i \in \{1, \ldots, n\}$ sei

   $$T_i := \{j \mid x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{m+i\}$$
   $$T_i' := \{j \mid \neg x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{m+i\}$$

   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.

   "$\Rightarrow$" Für eine gegebene Belegung der Variablen, welche φ erfüllt, wählen wir
   - T_i, falls $x_i \mapsto 1$
   - T_i', falls $x_i \mapsto 0$

   Dies liefert n Mengen, in denen jedes Element von M vorkommt:
   - j mit $1 \leq j \leq m$, da jedes K_j zu 1 evaluiert wird.
   - $m+i$ mit $1 \leq i \leq n$, da für jedes i entweder T_i oder T_i' gewählt wird.

   "$\Leftarrow$" Sei umgekehrt

   $$\{U_1, \ldots, U_n\} \subseteq \{T_1, \ldots, T_n, T_1', \ldots, T_n'\} \text{ mit } U_1 \cup \ldots \cup U_n = M$$

   gegeben.

   Da für $1 \leq i \leq n$ das Element $m+i$ in $U_1 \cup \ldots \cup U_n$ vorkommt, kommt T_i oder T_i' in $\{U_1, \ldots, U_n\}$ vor. Da wir nur n verschiedene Mengen haben, kommt jeweils genau eines von beiden vor.

Wir definieren nun die Belegung:

$$x_i = \begin{cases} 1 & \text{falls} \quad T_i \in \{U_1, \ldots, U_n\} \\ 0 & \text{falls} \quad U_i \in \{U_1, \ldots, U_n\} \end{cases}$$

Diese erfüllt jedes K_j, da 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 19.8, gib ein Menge $U_1, \ldots, U_4 \subseteq \{T_1, \ldots, T_4, T_1', \ldots, T_4'\}$ an, so dass $U_1 \cup \cdots \cup U_4 = M$ und finde die dazugehörige Belegung, die φ erfüllt.

Es gibt eine große 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“
> von M. Garey und D. Johnson, 1979.

Das Buch, welches zu den meistzitierten in der Informatik überhaupt gehört, 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.

Bemerkung:
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 19.3.

Definition 19.12 (PSpace-hart, PSpace-vollständig)

1) Eine Sprache L heißt PSpace-*hart* wenn für *alle* $L' \in$ PSpace gilt: $L' \leq_p L$.
2) L heißt PSpace-*vollständig* wenn $L \in$ PSpace und L ist PSpace-hart.

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 Probleme, 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 kontextsensitive Grammatiken.

Es gibt also auch für diese Probleme wahrscheinlich keine Polynomialzeit-Algorithmen.

V. Appendix

Der Appendix führt in knapper Weise Material ein, dass für das Verständnis der Vorlesung benötigt wird, jedoch nicht im engeren Sinne Gegenstand der Vorlesung ist.

A. Endliche Automaten als Graphen

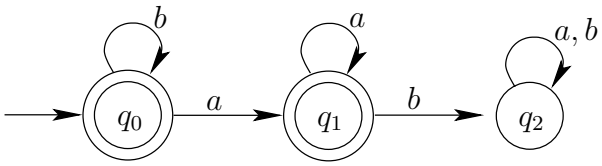
Manchmal ist es bequem, einen NEA als gerichteten Graph aufzufassen.

Definition A.1 (Gerichteter Graph)
Ein *gerichteter Graph* $G = (V, E)$ besteht aus einer Menge V von *Knoten* und einer Menge $E \subseteq V \times V$ von *Kanten*.

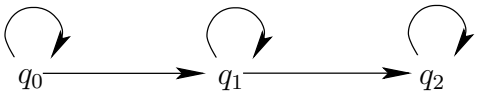
Graphen sind eine fundamentale Struktur in der Informatik und der Mathematik. Sie können z.B. zur Repräsentation von Kommunikationsnetzen und Datenstrukturen verwendet werden. Ein NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ kann als gerichteter Graph (V, E) dargestellt werden mit

- $V = Q$ und
- $E = \{(q, q') \mid \exists a \in \Sigma : (q, a, q') \in \Delta\}$.

Man fasst also die Zustände als Knoten und die Übergänge als Kanten auf, wobei die Alphabetssymbole sowie Start- und Endzustände wegabstrahiert werden. Zum Beispiel wird damit aus dem NEA



der folgende Graph:



Es ist natürlich auch möglich, nur solche Übergänge als Kanten in den Graph aufzunehmen, die bestimmte, ausgewählte Symbole betreffen.

Die Darstellung von Automaten als Graphen ist insbesondere für die algorithmische Behandlung von Automaten nützlich. Eine zentrale Rolle spielt dabei das Erreichbarkeitsproblem.

Definition A.2 (Erreichbar)
Sei $G = (V, E)$ ein gerichteter Graph. Ein Knoten $v \in V$ ist *erreichbar* von einem Knoten $u \in V$ wenn es eine Knotenfolge $v_1, \ldots, v_n$ gibt, mit $n \geq 1$, so dass $v_1 = u$, $v_n = v$, und $(v_i, v_{i+1}) \in E$ für $1 \leq i < n$.

Das *Erreichbarkeitsproblem* ist nun das folgende Problem: gegeben ein endlicher gerichteter Graph $G = (V, E)$ und zwei Knoten $u_0, v_0 \in V$, entscheide ob v_0 von u_0 erreichbar ist. Ein einfacher Algorithmus, um dieses Problem zu lösen, ist der folgende: gegeben $G = (V, E)$ und u_0, v_0, berechne eine Folge von Knotenmengen $V_0 \subseteq V_1 \subseteq V_2 \cdots$ so dass

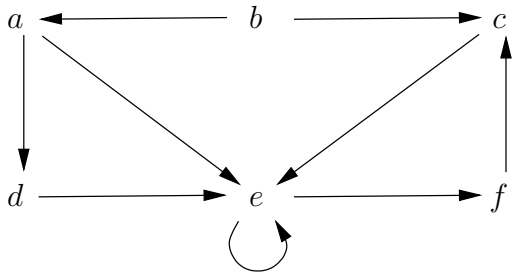
- $V_0 = \{u_0\}$ und
- $V_i \subseteq V$ für alle $i \geq 0$.

Konkret berechnet man die V_i für $i > 0$ wie folgt:

$$V_{i+1} := V_i \cup \{v' \mid \exists v \in V_i : (v, v') \in E\}.$$

Die Folge wird Schritt für Schritt berechnet. Sobald $V_i = V_{i+1}$ gilt, stoppt der Algorithmus, gibt „ja“ zurück wenn $v_0 \in V_i$ und „nein“ sonst. Wir werden in der Übung zeigen, dass der Algorithmus tatsächlich Erreichbarkeit entscheidet.

Beispiel A.3
Wir verwenden den Algorithmus, um zu entscheiden, ob in folgendem Graphen der Knoten f vom Knoten a aus erreichbar ist:



Die berechnete Knotenfolge ist

$$\{a\}, \{a, d, e\}, \{a, d, e, f\}, \{a, d, e, f, c\}, \{a, d, e, f, c\},$$

Da f in der zuletzt berechneten Menge enthalten ist, antwortet der Algorithmus mit „ja“.

Satz A.4

Das Erreichbarkeitsproblem für gerichtete Graphen ist effektiv entscheidbar.

Wir führen eine kurze Laufzeitanalyse durch. Sei $|V| = n$. Der oben angegebene Algorithmus stoppt spätestens bei der Menge V_n: wenn $V_i \neq V_{i+1}$ für alle $i < n$, denn gilt offenbar $|V_{i+1}| > |V_i|$ für alle $i < n$. Es folgt $|V_n| \geq n$, also $V_j = V$. Damit kann kein weiterer Knoten zu V_n hinzugefügt werden.

Wird der Algorithmus vorsichtig implementiert, so läuft er in Linearzeit: bei Eingabe (V, E) macht er $c \cdot (|V| + |E|)$ elementare Schritte, wobei c eine Konstante ist. Die Vorstellung der Details ist in dieser Vorlesung nicht möglich, sie finden sich z.B. im Buch „Introduction to Algorithms“ von Cormen, Leiserson, Rivest und Stein. Man beachte, dass bei naiver Implementierung bereits ein einzelner Test $V_i \neq V_{i+1}$ quadratisch viele (genauer gesagt $|V_i| \cdot |V_{i+1}|$) Vergleichsoperationen benötigt!

B. Laufzeitanalyse von Algorithmen und O-Notation

Laufzeitanalyse

Ein gegebenes Berechnungsproblem lässt sich meist durch viele verschiedene Algorithmen lösen. Um den „besten“ dieser vielen möglichen Algorithmen zu bestimmen, analysiert man deren Ressourcenverbrauch. In diesem Zusammenhang ist die wichtigste Ressource der *Zeitverbrauch*, also die Anzahl Rechenschritte, die der Algorithmus ausführt. Andere Ressourcen, die eine Rolle spielen können, sind beispielsweise der Speicherverbrauch oder der Kommunikationsbedarf, wenn der Algorithmus auf mehreren Prozessoren oder Rechnern verteilt ausgeführt wird. Ein Algorithmus mit geringem Ressourcenbedarf ist dann einem Algorithmus mit höherem Ressourcenbedarf vorzuziehen.

Die *Laufzeit* eines Algorithmus A auf einer Eingabe x ist die Anzahl *elementarer Rechenschritte*, die A gestartet auf x ausführt, bevor er anhält. Was ein elementarer Rechenschritt ist, wird in der VL „Theoretische Informatik II“ genauer untersucht. Bis dahin betrachten wir die üblichen Operationen eines Prozessors als elementare Rechenschritte, also z.B. Addition und Multiplikation. Die zentrale Eigenschaft eines elementaren Rechenschrittes ist, dass er in konstanter Zeit ausgeführt werden kann—insbesondere nehmen wir an, dass die benötigte Zeit unabhängig von den konkreten Argumenten ist, auf die der Rechenschritt angewendet wird.

Man beschreibt die Laufzeit eines Algorithmus immer in Abhängigkeit von der *Größe seiner Eingabe*. Dem liegt die Intuition zugrunde, dass das Verarbeiten einer grösseren Eingabe im allgemeinen länger dauert als das einer kleinen. So kann man z.B. offensichtlich schneller entscheiden, ob ein gegebener NEA ein Eingabewort der Länge 1 akzeptiert als eines der Länge 128. Der Zeitverbrauch eines Algorithmus kann also beschrieben werden durch eine Funktion

$$f : \mathbb{N} \to \mathbb{N}$$

die jeder Eingabelänge $n \in \mathbb{N}$ eine Laufzeit $f(n)$ zuordnet. Man beachte, dass diese Darstellung von der *konkreten Eingabe* abstrahiert, d.h., die Laufzeit des Algorithmus auf verschiedenen Eingaben derselben Länge wird nicht unterschieden. Diese kann durchaus sehr unterschiedlich sein: für einen gebenen NEA $\mathcal{A}$, in dem der Startzustand keine a-Übergang erlaubt, ist es trivial, zu entscheiden, dass das Wort

$$abbbababababbbbbbbbababbabbabbbbbbbbaaabbbbaaaaaaab$$

nicht akzeptiert wird, wohingegen dies für das gleichlange, aber mit b beginnende Wort

$$bbbababababbbbbbbbababbabbabbbbbbbbaaabbbbaaaaaaba$$

viel mehr Schritte erfordern kann. Die durch die Funktion $f : \mathbb{N} \to \mathbb{N}$ beschrieben Laufzeit ist als *Worst Case Komplexität* zu verstehen: $f(n)$ beschreibt eine obere Schranke, also eine maximale Schrittzahl, die für keine Eingabe der Länge n überschritten wird (die aber für „einfache“ Eingaben unter Umständen stark *unter*schritten werden kann).

Welche Laufzeit kann als *effizient* angesehen werden und welche nicht? Die wichtigste Grenze verläuft zwischen polynomiellem und exponentiellem Laufzeitverhalten, wobei ersteres im allgemeinen mit „machbar“ gleichgesetzt wird und letzteres mit steigender Eingabegröße so schnell wächst, dass größere Eingaben nicht verarbeitet werden können. Bei *polynomieller Laufzeit* ist die Funktion f also ein Polynom wie zum Beispiel

$$n, \quad n^2, \quad 5n, \quad 7n^2, \quad 8n^3, \quad 3n^3 + 2n^2 + 5n + 7.$$

Bei *exponentieller Laufzeit* ist sie eine Exponentialfunktion wie zum Beispiel

$$2^n, \quad 5^n, \quad 2^{n^2}, \quad n^n, \quad 17(5^{23n}).$$

Es ist wichtig, sich vor Augen zu führen, dass exponentielle Laufzeit so dramatisch ist, dass sie von schnellerer Hardware nicht kompensiert werden kann. Man betrachte z.B. die Laufzeit 2^n auf Eingaben der (sehr moderaten!) Größe $n = 128$. Die sich ergebende Anzahl Schritte ist so gewaltig, dass ein moderner 3Ghz-Prozessor länger rechnen müsste als vom Anfang des Universums bis heute. Aber auch Polynome höheren Grades sind recht schnell wachsende Funktionen. Interessanterweise findet man aber nur äußerst selten Algorithmen, deren Laufzeit zwar polynomiell ist, aber nicht durch ein Polynom kleinen Grades (meist ≤ 5, oft ≤ 3) beschrieben werden kann. Dies rechtfertigt die übliche Gleichsetzung von „polynomiell“ und „machbar“. Für sehr zeitkritische Probleme kann aber auch eine Laufzeit von n^2 zu lang sein. Als besonders effizient gilt *Linearzeit*, also Laufzeiten der Form $c \cdot n$, wobei $c \in \mathbb{N}$ eine Konstante ist.

O-Notation

Bei der Laufzeitanalyse von Algorithmen abstrahiert man so gut wie immer von konkreten Konstanten. Man würde also beispielsweise nicht zwischen einem Algorithmus mit

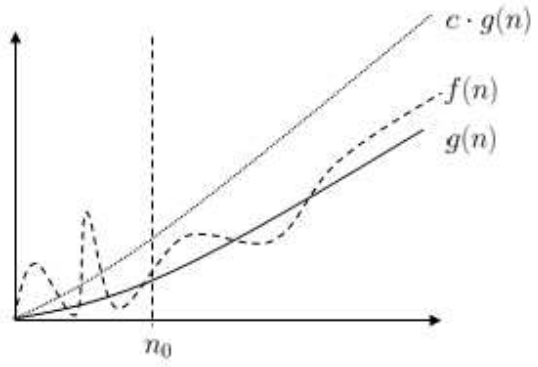
Laufzeit $2n$ und einem mit Laufzeit $4n$ unterscheiden (außer natürlich in einer konkreten Implementierung, wo derartige Unterschiede sehr wichtig sein können!). Dies hat mehrere Gründe. Erstens ist es schwierig und fehleranfällig, alle involvierten Konstanten zu identifizieren und während der für die Laufzeitabschätzung benötigten Rechnungen „mitzuschleppen". Und zweitens sind die konkreten Konstanten oft abhängig von Implementierungsdetails wie den konkret zur Verfügung stehenden elementaren Rechenschritten und den zur Repräsentation der Eingabe verwendeten Datenstrukturen. Die sogenannte „O-Notation" stellt eine einfach Methode zur Verfügung, um konkrete Konstanten auszublenden.

Definition B.1 (O-Notation)
Seien f und g Funktionen von $\mathbb{N}$ nach $\mathbb{N}$. Wir schreiben

$$f \in O(g)$$

wenn es eine Konstante $c > 0$ und Schranke $n_0 \geq 0$ gibt, so dass $f(n) \leq c \cdot g(n)$ für alle $n > n_0$.

Mit anderen Worten bedeutet $f \in O(g)$, dass f „schließlich nicht wesentlich schneller wächst" als g. Die folgende Graphik illustriert dies anhand zweier Funktionen $f(n)$ und $g(n)$ mit $f \in O(g)$:



Wie in Definition B.1 gefordert, liegt f schliesslich (d.h. ab der Schranke n_0) unterhalb der mit der Konstanten c skalierten Funktion $c \cdot g(n)$. Auch wenn der absolute Wert von $f(n)$ an vielen Stellen größer ist als der von $g(n)$, wächst f also nicht wesentlich schneller als g.

Wir verwenden die Laufzeitbeschreibung $O(f(n))$, wenn wir ausdrücken wollen, dass die Laufzeit $f(n)$ ist, bis auf Konstanten (repräsentiert durch c) und endlich viele Ausnahmen (repräsentiert durch n_0). Insbesondere beschreibt

- $O(n)$ Linearzeit;
- $O(n^2)$ quadratische Zeit und
- $\bigcup_{i \geq 1} n^i$ polynomielle Zeit.

Es existieren verscheidene Rechenregeln für die O-Notation wie zum Beispiel

- $O(O(f) = O(f)$;
- $O(f) + O(g) = O(f + g)$ und
- $O(f) \cdot O(g) = O(f \cdot g)$.

Mehr Informationen zur O-Notation finden sich beispielsweise im Buch „Concrete Mathematics“ von Graham, Knuth und Patashnik.

Entscheidbarkeit in Theoretische Informatik I

In „Theoretische Informatik I“ verwenden wir zwei Begriffe, die erst in „Theoretische Informatik II“ formal definiert und im Detail untersucht werden können: Effektivität und Entscheidbarkeit. Diese werden im folgenden kurz und informell erklärt. Es macht aus verschiedenen Gründen Sinn, zwischen zwei Arten von Problemen zu unterscheiden:

- *Entscheidungsprobleme*, bei denen die möglichen Antworten nur „ja“ und „nein“ sind;
- *Berechnungsprobleme*, bei denen eine echte Ausgabe (wie etwa eine Zahl, ein Wort, oder einen Automaten) berechnet with.

Wie wir in „Theoretische Informatik II“ sehen werden, gibt es Probleme, die zwar wohldefiniert sind, aber so schwierig, dass es *gar keinen* Algorithmus gibt, um sie zu lösen. Ein Beispiel für ein solches Problem ist das Leerheitsproblem für kontextsensitive Grammatiken. Die Existenz derartiger Probleme führt zu folgenden Begriffen: wir nennen

- ein Entscheidungsproblem *entscheidbar*, wenn es einen Algorithmus gibt, der dieses Problem löst und *unentscheidbar* sonst.
- ein Berechnungsproblem *effektiv berechenbar*, wenn es einen Algorithmus gibt, der dieses Problem löst.

Abkürzungsverzeichnis

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
geg. ..... gegeben
i.a. ..... im allgemeinen
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 Allgemeingültigkeit
PDA ..... pushdown automaton (Kellerautomat)
PKP ..... Postsches Korrespondenzproblem
PL1 ..... Prädikatenlogik erster Stufe
SAT ..... satisfiability problem (Erfüllbarkeitstest der Aussagenlogik)
TM ..... Turingmaschine (allgemein)
u.a. ..... unter anderem
URM ..... unbeschränkte Registermaschine
vgl. ..... vergleiche
z.B. ..... zum Beispiel
□ ..... was zu beweisen war (q.e.d)

Literaturverzeichnis

[Koz06] Dexter Kozen. *Automata and Computability.* Springer Verlag, 2007

[Hop06] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullmann. *Introduction to Automata Theory, Languages, and Computation.* Dritte Ausgabe. Addison Wesley, 2006

[Schö97] Uwe Schöning. *Theoretische Informatik – kurzgefaßt.* Spektrum Akademischer Verlag, 1997

[Wege93] Ingo Wegener. *Theoretische Informatik.* Teubner-Verlag, 1993