Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
Lehrstuhl Automatentheorie

Grundlagen der Theoretischen Informatik

Skript zur Lehrveranstaltung von Prof. Dr.-Ing. F. Baader

Bearbeiter: Dr.-Ing. T. Hinze,
H. Dietze,
M. Fruth

Dresden im Januar 2005

Inhaltsverzeichnis

Organisation der Lehrveranstaltung

Verantwortlicher Hochschullehrer:

Prof. Dr.-Ing. Franz Baader

Technische Universität Dresden
Fakultät Informatik
Institut für Theoretische Informatik
01062 Dresden

`http://lat.inf.tu-dresden.de/~baader/`

Aktuelle Informationen zur Lehrveranstaltung:

`http://wwwtcs.inf.tu-dresden.de/GThI/`

Hinweis

Dieses Skript ist als Hilfestellung für Studenten gedacht. Trotz großer Mühe beim Erstellen kann keine Garantie für Fehlerfreiheit übernommen werden. Es wird nochmals darauf hingewiesen, daß der prüfungsrelevante Stoff durch die Vorlesung bestimmt wird und nicht mit dem Skriptinhalt vollständig übereinstimmen muß.

Einführung

Die Lehrveranstaltung „Grundlagen der Theoretischen Informatik“ gibt eine Einführung in zwei wichtige Bereiche der Theoretischen Informatik:

1. Automatentheorie und formale Sprachen (Teile I, II)
2. Berechenbarkeit und Komplexität (Teile III, IV)

zu 1.: Hier werden *Wörter* und Mengen von Wörtern (**formale Sprachen**) untersucht.

- Wie beschreibt man solche Mengen?
  $\longrightarrow$ **Automaten** oder **Grammatiken** ...
- Beispiel: Daten (d.h. Eingabe und Ausgabe von Programmen) sind Wörter. Die formale Sprache beschreibt die syntaktisch korrekten Daten.

zu 2.: Hier betrachtet man *Funktionen* (von Zahlen nach Zahlen oder von Wörtern nach Wörtern).

- **Berechenbarkeit:** Gibt es ein Programm, das die Funktion berechnet? Gibt es *nichtberechenbare* Funktionen?
  $\longrightarrow$ *abstrakte Maschinenmodelle* und *Programmiersprachen*
- **Komplexität:** Wieviel *Zeit* oder *Speicherplatz* benötigt eine Berechnung?

Automatentheorie und formale Sprachen

Dieser Teil der Theoretischen Informatik beschäftigt sich mit *Wörtern* und Mengen von Wörtern (*formalen Sprachen*).

Wichtige Fragestellungen:

1. Wie beschreibt man die (meist unendlichen) Mengen von Wörtern mit endlichem Aufwand?

   - *Automaten* oder *Maschinen*, die genau die Elemente der Menge akzeptieren.
   - *Grammatiken*, die genau die Elemente der Menge generieren (vgl. „Programmierung“: kontextfreie Grammatiken (EBNF) zur Beschreibung der Syntax von Programmiersprachen).
   - *Ausdrücke*, welche 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, welche 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 Problemstellungen sind für eine Sprachklasse entscheidbar und mit welchem Aufwand?

   - *Wortproblem:* geg. eine Beschreibung der Sprache L (Automat, Grammatik, ...) und ein Wort w. Gilt $w \in L$?
     Anwendungsbeispiele:
     – Programmiersprache, deren Syntax durch eine kontextfreie Grammatik beschrieben ist. Entscheide für ein geg. Programm P, ob dieses syntaktisch korrekt ist.

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

- *Leerheitsproblem:* geg. eine Beschreibung der Sprache L. Gilt $L \neq \emptyset$?
  Anwendungsbeispiel:
  Wenn ein Suchpattern die leere Sprache beschreibt, so muß man die Dateien nicht durchsuchen.
- *Äquivalenzproblem:* Beschreiben zwei verschiedene Beschreibungen dieselbe Sprache?
  Anwendungsbeispiel:
  Zwei Lehrbücher über eine Programmiersprache beschreiben die Syntax durch verschiedene Grammatiken. Sind diese wirklich äquivalent?

3. Welche Abschlußeigenschaften hat eine Sprachklasse?
   z.B. Abschluß unter Durchschnitt, Vereinigung und Komplement (L_1, L_2 enthalten, so 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: Statt „$L_1 = L_2$?“ entscheidet man, ob $(L_1 \cap \overline{L_2}) \cup (L_2 \cap \overline{L_1}) = \emptyset$.

Teil I

Endliche Automaten und reguläre Sprachen

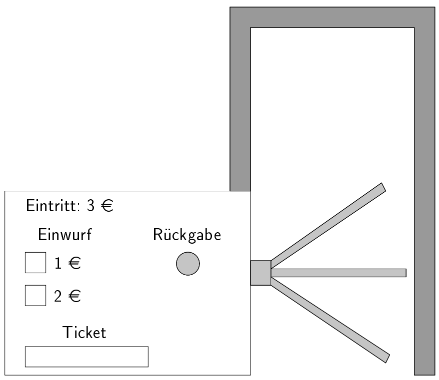
Einführung

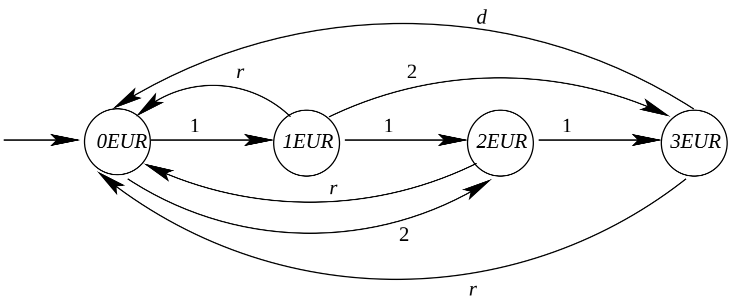
Endliche Automaten sind gekennzeichnet durch:

- endliche Menge von (internen) Zuständen
- Übergänge zwischen Zuständen; abhängig von der internen Struktur des Automaten und von der Eingabe

Im Prinzip sind Rechner ebenfalls endliche Automaten: Sie haben nur endlich viel Speicherplatz und daher nur eine endliche Menge möglicher Konfigurationen (Belegung der Speicherzellen). Die Konfigurationsübergänge werden bestimmt durch Verdrahtung und Eingaben (Tastatur, Peripheriegeräte).
<u>Aber:</u> Wegen der extrem großen Anzahl von Zuständen sind endliche Automaten nicht die geeignete Beschreibungsebene für das Verhalten von Rechenmaschinen. Man abstrahiert daher von der Speicherplatzbeschränkung und nimmt potentiell unendlich großen Speicher an (z.B. unendliches Band bei Turingmaschine).

Beispiel (Eintrittsautomat)
Beispiel für einen endlichen Automaten, bei dem dies die richtige Beschreibungsebene ist:
Eingabe: $1, 2, r, d$ *(d: Drehgitter dreht sich)*
Zustände: 0EUR, 1EUR, 2EUR, 3EUR





Teilweise interessiert man sich auch für Automaten mit Ausgabe (z.B. Rückgabe des Geldes, Freigeben der Sperre des Drehgitters). Wir werden diesen Aspekt hier nicht betrachten. Bevor wir den Begriff des endlichen Automaten formal einführen können, benötigen wir noch einige grundlegende Definitionen:

Alphabet: Eine endliche Menge von Buchstaben (z.B. $a, b, c, \ldots$) heißt Alphabet und wird meist mit Σ bezeichnet.

Wort: Eine endliche Folge von Buchstaben $w = a_1 \ldots a_n$ mit $a_i \in \Sigma$ heißt Wort über dem Alphabet Σ.

Länge des Wortes: $|w| = n$; z.B. $|aba| = 3$

Anzahl der Vorkommen von Buchstaben: $|w|_a$ zählt die Vorkommen von a in w; z.B. $|aba|_a = 2$, $|aba|_b = 1$, $|aba|_c = 0$

Leeres Wort: ε hat Länge 0

Menge aller Wörter über Σ: Σ^*

Menge aller nichtleeren Wörter über Σ: $\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$

Formale Sprache: Eine Menge L von Wörtern mit $L \subseteq \Sigma^*$ heißt formale Sprache über Σ. Beispiele:

- Menge aller Wörter über $\{a, \ldots, z\}$, die korrekte Wörter der deutschen Sprache sind
- Menge der Wörter, die das Stichwort „theorie“ enthalten

Operationen auf Sprachen oder Wörtern

Boolesche Operationen: $L_1 \cup L_2, \quad L_1 \cap L_2, \quad \overline{L_1} := \Sigma^* \setminus L_1, \quad L_1 \setminus L_2 := L_1 \cap \overline{L_2}$

Konkatenation: $u \cdot v := uv$; z.B. $abb \cdot ab = abbab$
$L_1 \cdot L_2 := \{u \cdot v \mid (u \in L_1) \wedge (v \in L_2)\}$
Beachte: Der Konkatenationspunkt wird häufig weggelassen (L_1L_2 statt $L_1 \cdot L_2$).

Kleene-Stern: iterierte Konkatenation

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

Präfix, Suffix, Infix:
u ist Präfix von v gdw. $v = uw$ für ein $w \in \Sigma^*$.
u ist Suffix von v gdw. $v = wu$ für ein $w \in \Sigma^*$.
u ist Infix von v gdw. $v = w_1uw_2$ für $w_1, w_2 \in \Sigma^*$.

1 Nichtdeterministische endliche Automaten

Wir betrachten zunächst den etwas allgemeineren Begriff des *Transitionssystems*.

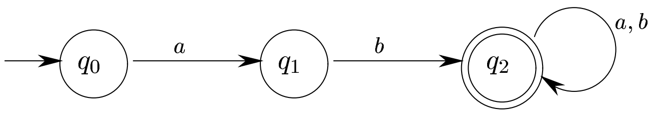
Definition 1.1 (Transitionssystem)
Ein Transitionssystem ist von der Form $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$, wobei

- *Q eine Menge von Zuständen ist (nicht notwendigerweise endlich!),*
- *Σ ein endliches Alphabet ist,*
- *$I \subseteq Q$ eine Menge von Anfangszuständen ist,*
- *$\Delta \subseteq Q \times \Sigma \times Q$ eine Übergangsrelation (Transitionsrelation) ist,*
- *$F \subseteq Q$ eine Menge von Endzuständen ist.*

Ein endlicher Automat hat endlich viele Zustände und nur einen Anfangszustand.

Definition 1.2 (nichtdeterministischer endlicher Automat)
Das Transitionssystem $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$ heißt nichtdeterministischer endlicher Automat (NEA), wenn $|Q| < \infty$ und $|I| = 1$. Anstelle von $(Q, \Sigma, \{q_0\}, \Delta, F)$ schreiben wir $(Q, \Sigma, q_0, \Delta, F)$.

Beispiel 1.3
$\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b\}$, $F = \{q_2\}$,
$\Delta = \{(q_0, a, q_1), (q_1, b, q_2), (q_2, a, q_2), (q_2, b, q_2)\}$ wird graphisch dargestellt als:



Definition 1.4 (Pfad)
Ein Pfad in einem Transitionssystem $\mathcal{A}$ ist eine Folge
$\pi = (p_0, a_1, p_1)(p_1, a_2, p_2) \dots (p_{n-1}, a_n, p_n)$ mit $(p_i, a_{i+1}, p_{i+1}) \in \Delta$ für $i = 0, \dots, n-1$.
π ist ein Pfad von p nach q, wenn $p = p_0$ und $q = p_n$.
Die Beschriftung des Pfades π ist das Wort $\beta(\pi) := a_1 \dots a_n$. Die Länge des Pfades π ist n.
Für $n = 0$ sprechen wir vom leeren Pfad, welcher die Beschriftung ε hat.

Die Existenz eines Pfades in $\mathcal{A}$ von p nach q mit Beschriftung w beschreiben wir durch $p \xrightarrow{w}_{\mathcal{A}} q$. Für Mengen $Q_1, Q_2 \subseteq Q$ heißt $Q_1 \xrightarrow{w}_{\mathcal{A}} Q_2$, daß es $p_1 \in Q_1$ und $p_2 \in Q_2$ gibt mit $p_1 \xrightarrow{w}_{\mathcal{A}} p_2$.
Für den NEA aus Beispiel 1.3 gilt für alle $w \in \{a, b\}^*$: $q_0 \xrightarrow{abw}_{\mathcal{A}} q_2$.

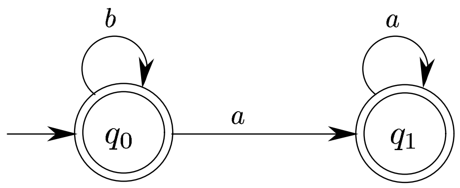
Definition 1.5 (akzeptierte Sprache)
Das Transitionssystem $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$ akzeptiert das Wort $w \in \Sigma^*$ gdw. $I \xrightarrow{w}_{\mathcal{A}} F$ gilt.
Die von $\mathcal{A}$ akzeptierte (erkannte) Sprache ist $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}$.

Für den NEA aus Beispiel 1.3 ist $L(\mathcal{A}) = \{abw \mid w \in \Sigma^*\}$, d.h. $\mathcal{A}$ akzeptiert genau die Wörter über $\Sigma = \{a, b\}$, welche mit ab beginnen.

Definition 1.6 (Erkennbarkeit einer Sprache)
Eine Sprache $L \subseteq \Sigma^*$ heißt erkennbar, wenn es einen NEA $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

Da wir uns bei einem Transitionssystem i.a. nur für die akzeptierten Sprachen interessieren, bezeichnen wir zwei Transitionssysteme als äquivalent, wenn sie dieselben Sprachen akzeptieren.

Beispiel 1.7
$L = \{w \in \{a,b\}^* \mid ab$ ist nicht Infix von $w\}$ ist erkennbar:



akzeptiert L.

Für manche Konstruktionen ist es günstig, auch endliche Automaten zu betrachten, bei denen Übergänge mit Wörtern beschriftet sind.

Definition 1.8 (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, wobei für alle $(q, w, q') \in \Delta$ gilt: $|w| \leq 1$ (d.h. $w \in \Sigma$ oder $w = \varepsilon$).

Pfade, Pfadbeschriftungen und akzeptierte Sprache werden entsprechend wie für NEAs definiert. Zum Beispiel hat der Pfad $(q_0, ab, q_1)(q_1, \varepsilon, q_2)(q_2, bb, q_3)$ die Beschriftung $ab \cdot \varepsilon \cdot bb = abbb$.

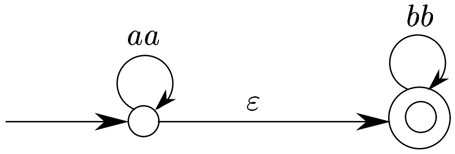
Satz 1.9
Zu jedem NEA mit Wortübergängen kann man effektiv einen äquivalenten NEA konstruieren.

Man zeigt dies mit Umweg über ε-NEAs.

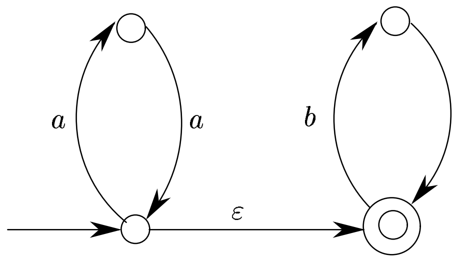
Lemma 1.10
Zu jedem NEA mit Wortübergängen kann man effektiv einen äquivalenten ε-NEA konstruieren.

Beweis:
Man ersetzt jeden Wortübergang $(q, a_1 \dots a_n, q')$ mit $n > 1$ durch Buchstabenübergänge $(q, a_1, p_1), (p_1, a_2, p_2), \dots, (p_{n-1}, a_n, q')$, wobei $p_1, \dots, p_{n-1}$ jeweils neue Hilfszustände sind (die nicht zur Endzustandsmenge dazugenommen werden). Man sieht leicht, daß dies einen äquivalenten ε-NEA liefert. $\square$

Beispiel 1.11



wird zu



Lemma 1.12
Zu jedem ε-NEA kann man effektiv einen äquivalenten NEA konstruieren.

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

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

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

1. Δ' und F' können effektiv bestimmt werden:
   - $p \xrightarrow{a}_{\mathcal{A}} q$ gilt genau dann, wenn es Zustände $p', q' \in Q$ gibt mit $p \xrightarrow{\varepsilon}_{\mathcal{A}} p'$, $\quad (p', a, q') \in \Delta$, $\quad q' \xrightarrow{\varepsilon}_{\mathcal{A}} q$. Δ ist eine endliche Menge und $p \xrightarrow{\varepsilon}_{\mathcal{A}} p'$ und $q' \xrightarrow{\varepsilon}_{\mathcal{A}} q$ ist ein Erreichbarkeitsproblem in einem endlichen Graphen (Gibt es einen Pfad von p nach q, dessen Übergänge nur mit ε beschriftet sind?).
   - „$q_0 \xrightarrow{\varepsilon}_{\mathcal{A}} F$?“ ist ebenfalls ein Erreichbarkeitsproblem in einem endlichen Graphen.

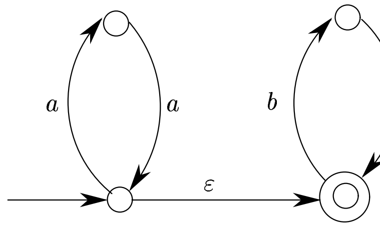
2. $L(\mathcal{A}) \subseteq L(\mathcal{A}')$:
   Es sei $w = a_1 \ldots a_n \in L(\mathcal{A})$ und $\pi = (p_0, a_1, p_1)(p_1, a_2, p_2) \ldots (p_{n-1}, a_n, p_n)$ ein Pfad in $\mathcal{A}$ mit $p_0 = q_0$, $p_n \in F$ und $a_i \in \Sigma \cup \{\varepsilon\}$. Wir betrachten die Indizes $i_1, \ldots, i_m$ mit $a_{i_j} \neq \varepsilon$. Offenbar ist $w = a_{i_1} \ldots a_{i_m}$.
   <u>1. Fall:</u> $(m = 0)$
   Es gibt keinen solchen Index, d.h. $a_i = \varepsilon$ für $i = 1, \ldots, n$ und damit $w = \varepsilon$. Es gilt also $q_0 = p_0 \xrightarrow{\varepsilon}_{\mathcal{A}} p_n \in F$, was $q_0 \in F'$ liefert. Es ist daher $w = \varepsilon \in L(\mathcal{A}')$.
   <u>2. Fall:</u> $(m > 0)$
   Nach Definition von Δ' ist nun aber $(p_0, a_{i_1}, p_{i_1})(p_{i_1}, a_{i_2}, p_{i_2}) \ldots (p_{i_{m-1}}, a_{i_m}, p_n)$ ein Pfad im NEA $\mathcal{A}'$. Aus $p_n \in F$ folgt $p_n \in F'$, was $w = a_{i_1} \ldots a_{i_m} \in L(\mathcal{A}')$ zeigt.

3. $L(\mathcal{A}') \subseteq L(\mathcal{A})$:
   Es sei $w = a_1 \ldots a_n \in L(\mathcal{A}')$ und $(p_0, a_1, p_1)(p_1, a_2, p_2) \ldots (p_{n-1}, a_n, p_n)$ mit $p_0 = q_0$, $p_n \in F'$ ein entsprechender Pfad. Nach Definition von Δ' gilt nun in $\mathcal{A}$: $p_i \xrightarrow{a_{i+1}}_{\mathcal{A}} p_{i+1}$ für $i = 0, \ldots, n-1$, also gibt es in $\mathcal{A}$ einen Pfad von p_0 nach p_n mit Beschriftung $w = a_1 \ldots a_n$.
   <u>1. Fall:</u> $(p_n \in F)$

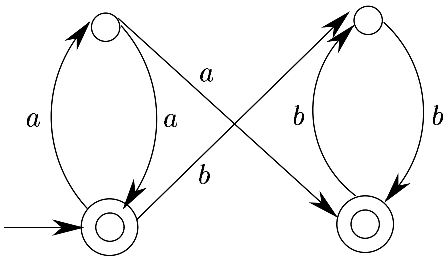
Dann folgt sofort $w \in L(\mathcal{A})$.
2. Fall: ($p_n \in F' \setminus F$, d.h. $p_n = q_0$)
Es gilt nun aber (nach Definition von F'): $p_n = q_0 \xrightarrow{\varepsilon}_{\mathcal{A}} \hat{p}$ für ein $\hat{p} \in F$. Daher gibt es in $\mathcal{A}$ einen Pfad von p_0 nach $\hat{p}$ mit Beschriftung w. $\square$

Lemma 1.12 kann man auch dazu verwenden zu zeigen, daß es zu jedem endlichen Transitionssystem $\mathcal{A} = (Q, \Sigma, I, \Delta, F)$ einen äquivalenten NEA $\mathcal{A}'$ gibt. Man definiert für einen neuen Zustand $q_0 \notin Q$ einen ε-NEA $\mathcal{A}'' = (Q \cup \{q_0\}, \Sigma, q_0, \Delta'', F)$ mit $\Delta'' = \Delta \cup \{(q_0, \varepsilon, q) \mid q \in I\}$. $\mathcal{A}''$ ist äquivalent zu $\mathcal{A}$. Man erhält nun $\mathcal{A}'$ aus $\mathcal{A}''$ wie im Beweis des Lemmas beschrieben.

Beispiel 1.13 *(zu Lemma 1.12)*
Der ε-NEA aus Beispiel 1.11



wird zu



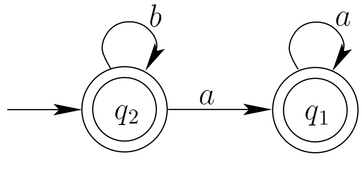
2 Deterministische endliche Automaten

Die bisher betrachteten NEAs heißen nichtdeterministisch, weil es in ihnen zu einem Paar $(q, a) \in Q \times \Sigma$ mehr als einen Übergang geben kann, d.h. $q', q'' \in Q$ mit $q' \neq q''$ und $(q, a, q') \in \Delta$, $(q, a, q'') \in \Delta$.

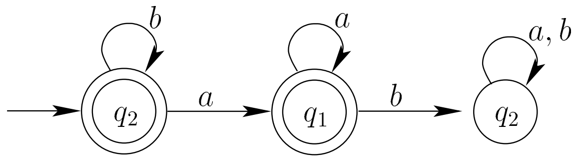
Definition 2.1 (deterministischer endlicher Automat)
Ein NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ heißt deterministisch (DEA), falls es für alle $q \in Q$, $a \in \Sigma$ genau ein $q' \in Q$ gibt mit $(q, a, q') \in \Delta$.

Anstelle der Übergangsrelation Δ verwenden wir dann die Übergangsfunktion $\delta : Q \times \Sigma \to Q$ mit $\delta(q, a) = q'$ gdw. $(q, a, q') \in \Delta$. DEAs werden in der Form $(Q, \Sigma, q_0, \delta, F)$ geschrieben.
Beachte:
Zusätzlich zur Eindeutigkeit eines Übergangs haben wir auch die Existenz eines Übergangs für jedes Paar (q, a) gefordert.
In der Literatur wird für deterministische Automaten manchmal nur die Eindeutigkeit des Übergangs (für (q, a) gibt es höchstens ein q' mit $(q, a, q') \in \Delta$) gefordert, während man bei Existenz aller Übergänge (für (q, a) gibt es mindestens ein q' mit $(q, a, q') \in \Delta$) von vollständigen Automaten spricht. Bezüglich der akzeptierten Sprachklasse macht dies keinen Unterschied.

Beispiel 2.2
Der NEA



erfüllt zwar die Eindeutigkeitsanforderung, er ist aber kein DEA in unserem Sinn, da für (q_1, b) kein Übergang existiert. Einen äquivalenten DEA erhält man durch Hinzunahme eines „Papierkorbzustandes“:



Definition 2.3 (kanonische Fortsetzung von δ)
Die kanonische Fortsetzung von $\delta : Q \times \Sigma \to Q$ auf eine Funktion $\delta^* : Q \times \Sigma^* \to Q$ wird induktiv (über die Wortlänge) definiert:

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

Der Einfachheit halber schreiben wir in Zukunft auch für Wörter w $\delta(q, w)$ statt $\delta^*(q, w)$.
Beachte:
Für einen DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ gilt:

a) $\delta(q, w) = q'$ gdw. q' ist der eindeutige Zustand mit $q \xrightarrow{w}_{\mathcal{A}} q'$.

b) $L(\mathcal{A}) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$

c) $\delta(q, uv) = \delta(\delta(q, u), v)$

Wir zeigen nun, daß die bei der Definition von DEAs gemachten Einschränkungen (Existenz und Eindeutigkeit von Übergängen) keine echten Einschränkungen sind:

Satz 2.4 (Rabin/Scott)
Zu jedem NEA kann man effektiv einen äquivalenten DEA konstruieren.

Es folgt, daß NEAs und DEAs dieselbe Klasse von Sprachen akzeptieren. Der Beweis verwendet die sogenannte Potenzmengenkonstruktion: die Zustandsmenge des DEA ist die Potenzmenge 2^Q der Zustandsmenge Q des NEA.
Idee:
Sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA. Um für $w \in \Sigma^*$ zu entscheiden, ob $w \in L(\mathcal{A})$ ist, betrachtet man $\{q \in Q \mid q_0 \xrightarrow{w}_{\mathcal{A}} q\} \in 2^Q$. Es ist $w \in L(\mathcal{A})$ gdw. diese Menge enthält mindestens einen Endzustand.
Wir definieren einen DEA mit Zustandsmenge 2^Q und Übergangsfunktion δ so, daß $\delta(\{q_0\}, w) = \{q \mid q_0 \xrightarrow{w}_{\mathcal{A}} q\}$.
Beweis von Satz 2.4
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:

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

Behauptung: $q' \in \delta(\{q\}, w)$ gdw. $q \xrightarrow{w}_{\mathcal{A}} q'$.
Daraus folgt $L(\mathcal{A}) = L(\mathcal{A}')$, da:

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

Beweis der Behauptung: Induktion über $|w|$.

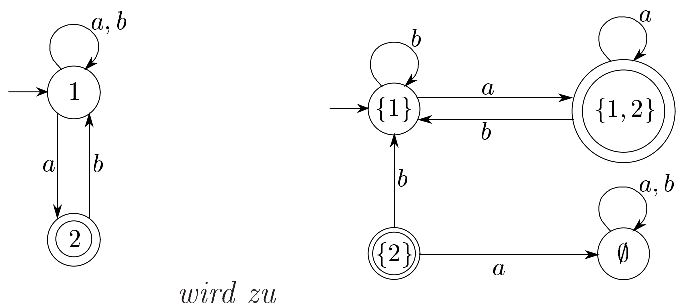
$|w| = 0$:

$\delta(\{q\}, \varepsilon) = \{q\}$. q ist der einzige Zustand, der mit ε von q aus erreichbar ist (keine ε-Übergänge!).

$|w| = n + 1$:

Für Wörter der Länge n gelte die Behauptung schon. Es ist $w = ua$ für $a \in \Sigma$ und für das Wort $u \in \Sigma^*$ der Länge n.
Nach Definition 2.3 ist $\delta(\{q\}, ua) = \delta(\delta(\{q\}, u), a) = \bigcup\limits_{q' \in \delta(\{q\}, u)} \{q'' \mid (q', a, q'') \in \Delta\}$.
Nach Induktionsvoraussetzung gilt $q' \in \delta(\{q\}, u)$ gdw. $q \xrightarrow{u}_{\mathcal{A}} q'$. Aus $q \xrightarrow{u}_{\mathcal{A}} q'$ und $(q', a, q'') \in \Delta$ ergibt sich $q \xrightarrow{ua}_{\mathcal{A}} q''$. Umgekehrt gilt $q \xrightarrow{ua}_{\mathcal{A}} q''$ gdw. es q' gibt mit $q \xrightarrow{u}_{\mathcal{A}} q'$ und $(q', a, q'') \in \Delta$. Es folgt daher $q'' \in \delta(\{q\}, ua)$ gdw. $q \xrightarrow{ua}_{\mathcal{A}} q''$. $\square$

Beispiel 2.5



wird zu

Nachteilig an dieser Konstruktion ist, daß die Zustandsmenge exponentiell vergrößert wird. Allgemein kann man dies nicht vermeiden, in manchen Fällen kommt man aber doch mit weniger Zuständen aus. Um aus einem gegebenen DEA einen äquivalenten DEA mit minimaler Zustandszahl zu konstruieren, benötigt man 2 Schritte:

1. Schritt: Eliminieren unerreichbarer Zustände

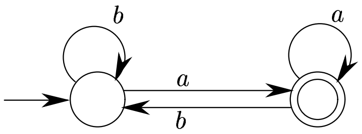
Definition 2.6 (Erreichbarkeit eines Zustandes)
Ein Zustand q des DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ heißt erreichbar, falls es ein $w \in \Sigma^*$ gibt mit $\delta(q_0, w) = q$. Sonst heißt q unerreichbar.

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

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

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

Beispiel:
Im Automaten $\mathcal{A}'$ von Beispiel 2.5 sind die Zustände $\{2\}$ und $\emptyset$ nicht erreichbar. Durch Weglassen dieser Zustände erhält man $\mathcal{A}'_0$:

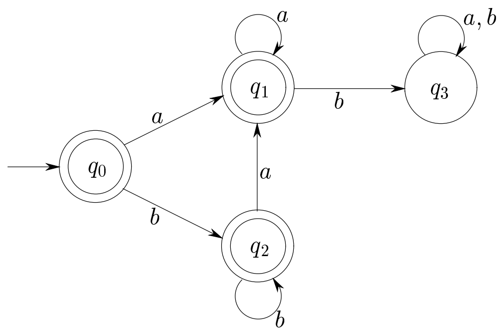


Bei der Potenzmengenkonstruktion kann man die Betrachtung unerreichbarer Elemente von 2^Q vermeiden, indem man mit $\{q_0\}$ beginnt und nur die davon erreichbaren Mengen sukzessive konstruiert.

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 akzeptierte Sprache.

Beispiel 2.7



erkennt dieselbe Sprache wie der DEA aus Beispiel 2.2, hat aber einen Zustand mehr. Dies kommt daher, daß q_0 und q_2 äquivalent sind.

Definition 2.8 (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. $L(\mathcal{A}_q) = L(\mathcal{A}_{q'})$.

Lemma 2.9

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

2) *$\sim_\mathcal{A}$ ist verträglich mit der Übergangsfunktion, d.h. $q \sim_\mathcal{A} q' \Rightarrow \forall a \in \Sigma : \delta(q,a) \sim_\mathcal{A} \delta(q',a)$.*

3) *$\sim_\mathcal{A}$ kann effektiv bestimmt werden.*

Beweis:

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

2) läßt sich wie folgt herleiten:

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

3) Wir definieren Approximationen $\sim_k$ von $\sim_{\mathcal{A}}$:

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

Es gilt $Q \times Q \supseteq \sim_0 \supseteq \sim_1 \supseteq \sim_2 \supseteq \ldots \supseteq \sim_{\mathcal{A}}$. Da Q endlich ist, gibt es ein k mit $\sim_k = \sim_{k+1}$. Man zeigt leicht, daß dann $\sim_k = \sim_{\mathcal{A}}$ ist. $\square$

Die $\sim_{\mathcal{A}}$-Äquivalenzklasse eines Zustands $q \in Q$ schreiben wir $\tilde{q} := \{q' \in Q \mid q \sim_{\mathcal{A}} q'\}$.
Beispiel:
Für den Automaten aus Beispiel 2.7 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 $\delta(q_1, b) \notin F$.
- $\sim_2 = \sim_1 = \sim_{\mathcal{A}}$.

Definition 2.10 (Quotientenautomat)
Der Quotientenautomat $\tilde{\mathcal{A}} = (\tilde{Q}, \Sigma, \tilde{q}_0, \tilde{\delta}, \tilde{F})$ *zu* $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ *ist definiert durch:*

- $\tilde{Q} := \{\tilde{q} \mid q \in Q\}$
- $\tilde{\delta}(\tilde{q}, a) := \widetilde{\delta(q,a)}$ *(repräsentantenunabhängig wegen Lemma 2.9)*
- $\tilde{F} := \{\tilde{q} \mid q \in F\}$

Lemma 2.11
$\tilde{\mathcal{A}}$ *ist äquivalent zu* $\mathcal{A}$.

Beweis:
$w \in L(\mathcal{A})$ gdw. $\delta(q_0, w) \in F$ gdw. $\widetilde{\delta(q_0, w)} \in \tilde{F}$ gdw. $\tilde{\delta}(\tilde{q}_0, w) \in \tilde{F}$ gdw. $w \in L(\tilde{\mathcal{A}})$. $\square$

Definition 2.12 (reduzierter Automat zu einem DEA)
Für einen DEA $\mathcal{A}$ *bezeichnet* $\mathcal{A}_{red} := \tilde{\mathcal{A}}_0$ *den reduzierten Automaten, den man aus* $\mathcal{A}$ *durch Eliminieren unerreichbarer Zustände und Zusammenfassen äquivalenter Zustände erhält.*

Wir werden im folgenden zeigen:

- $\mathcal{A}_{red}$ kann nicht weiter vereinfacht werden, d.h. er ist der kleinste DEA, mit dem man $L(\mathcal{A})$ akzeptieren kann.
- $\mathcal{A}_{red}$ hängt nur von $L(\mathcal{A})$ und nicht von $\mathcal{A}$ ab, d.h. gilt $L(\mathcal{A}) = L(\mathcal{B})$, so auch $\mathcal{A}_{red} \simeq \mathcal{B}_{red}$ (gleich bis auf Zustandsumbenennung, isomorph).

Dazu konstruieren wir zu einer erkennbaren Sprache L einen „kanonischen" Automaten $\mathcal{A}_L$ und zeigen, daß jeder reduzierte Automat, der L akzeptiert, isomorph zu $\mathcal{A}_L$ ist.

Definition 2.13 (Nerode-Rechtskongruenz)
Es sei $L \subseteq \Sigma^*$ eine beliebige Sprache. Für $u, v \in \Sigma^*$ definieren wir:
$u \simeq_L v$ gdw. $\forall w \in \Sigma^* : uw \in L \Leftrightarrow vw \in L$.

Beispiel 2.14 *(vgl. Beispiele 1.7, 2.2, 2.7)*
$L = \{w \in \{a, b\}^* \mid ab$ ist nicht Infix von $w\}$

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

Lemma 2.15 (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: $L = \bigcup_{u \in L} [u]$, wobei $[u] := \{v \mid u \simeq_L v\}$.

4) Ist $L = L(\mathcal{A})$ für einen DEA $\mathcal{A}$, so ist die Anzahl der $\simeq_L$-Klassen $\leq$ der Zustandszahl von $\mathcal{A}$. (Die Anzahl der $\simeq_L$-Klassen heißt Index von $\simeq_L$.)

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:
   $(\star)$ $uww' \in L \Leftrightarrow vww' \in L$
   Nun ist aber $ww' \in \Sigma^*$ und daher folgt $(\star)$ aus $u \simeq_L v$.

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

   „ $\subseteq$ " : klar, da $u \in [u]$.

   „ $\supseteq$ " : Sei $u \in L$ und $v \in [u]$. 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 $L = L(\mathcal{A})$.
   Behauptung:
   Aus $\delta(q_0, u) = \delta(q_0, v)$ folgt $u \simeq_L v$, denn:
   $\forall w : uw \in L$ gdw. $\delta(q_0, uw) \in F$
   gdw. $\delta(\delta(q_0, u), w) \in F$
   gdw. $\delta(\delta(q_0, v), w) \in F$
   gdw. $\delta(q_0, vw) \in F$
   gdw. $vw \in L$
   Also gibt es maximal so viele verschiedene Klassen, wie es verschiedene Zustände gibt (Schubfachprinzip). $\square$

Beispiel 2.14 *(Fortsetzung)*
$\simeq_L$ hat drei Klassen: $[\varepsilon] = \{b\}^*$, $[a] = \{b\}^* \cdot \{a\}^+$, $[ab] = \Sigma^* \cdot \{ab\} \cdot \Sigma^*$
Man kann die $\simeq_L$-Klassen nun als Zustände eines Automaten für L verwenden.

Definition 2.16 (Transitionssystem $\mathcal{A}_L$ zu einer Sprache L)
Zu $L \subseteq \Sigma^*$ ist das Transitionssystem $\mathcal{A}_L := (Q', \Sigma, q_0', \delta', F')$ definiert durch:

- $Q' := \{[u] \mid u \in \Sigma^*\}$
- $q_0' := [\varepsilon]$
- $\delta'([u], a) := [ua]$ *(repräsentantenunabhängig wegen Lemma 2.15, 2))*
- $F' := \{[u] \mid u \in L\}$

Lemma 2.17
Hat $\simeq_L$ endlichen Index, so ist $\mathcal{A}_L$ ein DEA mit $L = L(\mathcal{A}_L)$.

Beweis:
Hat $\simeq_L$ endlich viele Klassen, so hat $\mathcal{A}_L$ endlich viele Zustände. Außerdem gilt:

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

Satz 2.18 (Satz von Nerode)
Eine Sprache L ist erkennbar gdw. $\simeq_L$ hat endlichen Index (d.h. endlich viele Klassen).

Beweis:

„$\Rightarrow$“: Ergibt sich unmittelbar aus Lemma 2.15, 4).

„$\Leftarrow$“: Ergibt sich unmittelbar aus Lemma 2.17, da $\mathcal{A}_L$ DEA ist, der L akzeptiert. $\square$

Beispiel 2.19 (nichterkennbare 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 werden im nächsten Abschnitt noch eine weitere Möglichkeit sehen, wie man für eine Sprache zeigen kann, daß sie nicht regulär ist.
Zunächst untersuchen wir aber den Zusammenhang zwischen reduzierten Automaten und der Nerode-Rechtskongruenz.

Definition 2.20 (isomorph)
Zwei DEAs $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ und $\mathcal{A}' = (Q', \Sigma, q_0', \delta', F')$ sind isomorph ($\mathcal{A} \simeq \mathcal{A}'$) gdw. es eine Bijektion $\mathrm{f} : Q \to Q'$ gibt mit:

- $\mathrm{f}(q_0) = q_0'$

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

Lemma 2.21
$\mathcal{A} \simeq \mathcal{A}' \;\Rightarrow\; L(\mathcal{A}) = L(\mathcal{A}')$

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

$$\begin{array}{lll} w \in L(\mathcal{A}) & \text{gdw.} & \delta(q_0,w) \in F \\ & \text{gdw.} & f(\delta(q_0,w)) \in F' \qquad (\text{wegen } F' = f(F)) \\ & \text{gdw.} & \delta'(f(q_0),w) \in F' \\ & \text{gdw.} & \delta'(q_0',w) \in F' \qquad (\text{wegen } q_0' = f(q_0)) \\ & \text{gdw.} & w \in L(\mathcal{A}') \qquad \Box \end{array}$$

Wir können nun Minimalität und Eindeutigkeit des reduzierten Automaten zeigen.

Satz 2.22
Es sei L eine erkennbare Sprache. Dann gilt:

1) *$\mathcal{A}_L$ hat minimale Zustandszahl unter allen DEAs, welche L erkennen.*

2) *Ist $\mathcal{A}$ ein DEA mit $L(\mathcal{A}) = L$, so ist der reduzierte Automat $\mathcal{A}_{red} := \tilde{\mathcal{A}}_0$ isomorph zu $\mathcal{A}_L$.*

Beweis:

1) Mit Satz 2.18 hat $\simeq_L$ endlichen Index. Daher ist mit Lemma 2.17 $\mathcal{A}_L$ ein DEA für L. Mit Lemma 2.15, 4) hat jeder DEA, der L akzeptiert, mindestens soviele Zustände, wie $\simeq_L$ Klassen (d.h. wie $\mathcal{A}_L$ Zustände) hat.

2) 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 $f : Q \to Q'$ und zeigen, daß sie ein Isomorphismus ist. Für $q \in Q$ existiert (mindestens) ein Wort $w_q \in \Sigma^*$ 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 $f(q) := [w_q]$.

I) f ist injektiv:
Wir müssen zeigen, daß aus $p \neq q$ auch $[w_p] \neq [w_q]$ 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, $\delta(q_0, w_p w) \in F \Leftrightarrow \delta(q_0, w_q w) \in F$ gilt nicht, d.h. wiederum $w_p w \in L \Leftrightarrow w_q w \in L$ gilt nicht. Also ist $w_p \not\simeq_L w_q$, d.h. $[w_p] \neq [w_q]$.

II) f ist surjektiv:
Folgt aus Injektivität und $|Q| \geq |Q'|$ (Aussage 1) des Lemmas).

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

IV) $f(F) = F'$:
$q \in F$ gdw. $\delta(q_0, w_q) = q \in F$ gdw. $w_q \in L$ gdw. $[w_q] \in F'$

V) $f(\delta(q,a)) = \delta'(f(q),a)$:
Es sei $\delta(q,a) =: p$. Dann ist $f(\delta(q,a)) = [w_p]$.

Außerdem ist $\delta'(\mathrm{f}(q), a) = \delta'([w_q], a) = [w_q a]$. Es bleibt also $[w_p] = [w_q a]$ zu zeigen, d.h. $\forall w : w_p w \in L$ gdw. $w_q a w \in L$.
Dies ist offenbar dann der Fall, wenn $\delta(q_0, w_p) = \delta(q_0, w_q a)$ ist:
$\delta(q_0, w_q a) = \delta(\delta(q_0, w_q), a) = \delta(q, a) = p = \delta(q_0, w_p)$ □

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

Korollar 2.23
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). Hat man NEAs gegeben, so kann man diese zuerst deterministisch machen und dann das Korollar anwenden.

3 Nachweis der Nichterkennbarkeit

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: Es genügt ja nicht zu sagen, daß man keinen Automaten dafür gefunden hat (unendlich viele Automaten).
Wir haben bereits gesehen, daß man den Satz von Nerode (Satz 2.18) dazu verwenden kann. Ein anderes Hilfsmittel hierfür ist das Pumping-Lemma:

Lemma 3.1 (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:
Es sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA mit $L(\mathcal{A}) = L$. Wir wählen $n_0 = |Q|$. Für jedes Wort $w = a_1 \ldots a_m \in L$ 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 $p_m \in F$.
Ist $m \geq n_0$, so können die $m + 1$ Zustände $p_0, \ldots, p_m$ nicht alle verschieden sein, da $|Q| = n_0 < m + 1$. Es gibt daher ein $i < j$ mit $p_i = p_j$. Für $x := a_1 \ldots a_i$, $y := a_{i+1} \ldots a_j$, $z := a_{j+1} \ldots a_m$ gilt daher $y \neq \varepsilon$ (da $i < j$) und $q_0 = p_0 \xrightarrow{x}_{\mathcal{A}} p_i \xrightarrow{y}_{\mathcal{A}} p_i = p_j \xrightarrow{z}_{\mathcal{A}} p_m \in F$.
Folglich gilt für alle $k \geq 0$ auch $p_i \xrightarrow{y^k}_{\mathcal{A}} p_i$, was $xy^k z \in L$ zeigt. □

Wir zeigen mit Hilfe dieses Lemmas (nochmals), daß $\{a^n b^n \mid n \geq 0\}$ nicht erkennbar ist.

Beispiel 3.2
$L = \{a^n b^n \mid n \geq 0\}$ *ist nicht erkennbar.*

Beweis:
Angenommen, L ist erkennbar. Dann gibt es eine Zahl n_0 mit den in Lemma 3.1 beschriebenen Eigenschaften. Es sei nun n so, daß $2n \geq n_0$ ist. Da $a^n b^n \in L$ ist, hat es eine Zerlegung

$a^n b^n = xyz$ mit $|y| \geq 1$ und $xy^k z \in L$ für alle $k \geq 0$.

1. Fall: y liegt ganz in a^n.
D.h. $x = a^{n_1}$, $y = a^{n_2}$, $z = a^{n_3} b^n$ mit $n_2 > 0$ und $n = n_1 + n_2 + n_3$. Damit ist aber $xz = xy^0 z = a^{n_1+n_3} b^n \notin L$, da $n_1 + n_3 < n$. Widerspruch.

2. Fall: y liegt ganz in b^n.
Führt entsprechend zu einem Widerspruch.

3. Fall: y enthält as und bs.
D.h. $x = a^{n_1}$, $y = a^{n_2} b^{n'_1}$, $z = b^{n'_2}$ mit $n_2 \neq 0 \neq n'_1$. Dann ist aber $xyyz = a^{n_1} a^{n_2} b^{n'_1} a^{n_2} b^{n'_1} b^{n'_2} \notin L$ (da nach bs nochmal as kommen). Widerspruch.

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

Als eine weitere Konsequenz von Lemma 3.1 erhält man, daß das Leerheitsproblem für erkennbare Sprachen entscheidbar ist.

Satz 3.3
Es sei L eine erkennbare Sprache (gegeben durch NEA oder DEA). Dann kann man effektiv entscheiden, ob $L = \emptyset$ oder nicht.

Beweis:

Es sei L erkennbar und n_0 die zugehörige Zahl aus Lemma 3.1. Dann gilt:
($\star$) $\quad L \neq \emptyset$ gdw. $\exists w \in L$ mit $|w| < n_0$
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 4.4).
Beweis von ($\star$):

„$\Leftarrow$“: 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 $xy^0 z = xz \in L$. Dies ist ein Widerspruch zur Minimalität von w. □

Mit Hilfe der einfachen Variante des Pumping-Lemmas gelingt es nicht immer, Nichterkennbarkeit nachzuweisen.

Beispiel 3.4
Ist $L = \{a^n b^m \mid n \neq m\}$ erkennbar? Versucht man, Nichterkennbarkeit mit Lemma 3.1 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 $n + m \geq 3$.

1. Fall: $n > m$
D.h. $n = m + i$ *für* $i \geq 1$.
1.1.: $i > 1$, *d.h.* $n - 1 > m$.
Für $x = \varepsilon$, $y = a$, $z = a^{n-1} b^m$ *gilt für alle* $k \geq 0$: $xy^k z = a^k a^{n-1} b^m \in L$, *da* $k + n - 1 \geq n - 1 > m$ *(wegen* $n - m = i > 1$*).*
1.2.: $i = 1$, *d.h.* $n = m + 1$.
Wegen $n + m \geq 3$ *folgt* $n = m + 1 \geq 2$.
Für $x = \varepsilon$, $y = a^2$, $z = a^{n-2} b^m$ *gilt*
a) $\quad xy^0 z \in L$, *da* $n - 2 = m - 1 \neq m$

b) $xy^kz \in L$ *für* $k \geq 1$*, da* $(n-2) + k \cdot 2 \geq n > m$

2. Fall: $n < m$

kann entsprechend behandelt werden. □

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 3.5 (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 *mit* $uvw \in L$ *und* $|v| \geq n_0$ *gibt es eine Zerlegung* $v = xyz$ *mit*

- $y \neq \varepsilon$
- $uxy^kzw \in L$ *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 $q_0 \xrightarrow{u}_{\mathcal{A}} p \xrightarrow{v}_{\mathcal{A}} q \xrightarrow{w}_{\mathcal{A}} f \in F$.
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 3.1 weitermachen. □

Was ist der Vorteil dieses Lemmas beim Nachweis der Nichterkennbarkeit? Man weiß, daß man bereits beliebig lange Teilstücke zerlegen kann. Dadurch kann man das y geeignet positionieren (im Beispiel 3.4 im kürzeren Block).

Beispiel 3.4 *(Fortsetzung)*
$L = \{a^n b^m \mid n \neq m\}$ *ist nicht erkennbar.*

Beweis:

Angenommen, doch; dann gibt es $n_0 \geq 1$, das die in Lemma 3.5 geforderten Eigenschaften hat. Wir betrachten nun die Wörter $u := \varepsilon$, $v := a^{n_0}$, $w := b^{n_0!+n_0}$. Offenbar ist $uvw = a^{n_0}b^{n_0!+n_0} \in L$. Mit Lemma 3.5 gibt es eine Zerlegung $v = xyz$ mit $y \neq \varepsilon$ und $uxy^kzw \in L$ für alle $k \geq 0$. Es sei $x = a^{n_1}$, $y = a^{n_2}$, $z = a^{n_3}$, $n_1 + n_2 + n_3 = n_0$, $n_2 > 0$. Offenbar existiert ein l mit $n_2 \cdot l = n_0!$ (da $0 < n_2 \leq n_0$). Es ist nun aber $n_1 + (l+1)n_2 + n_3 = n_0 + n_0!$, was $uxy^{l+1}zw \notin L$ liefert (Widerspruch). □

4 Abschlußeigenschaften und Entscheidungsprobleme

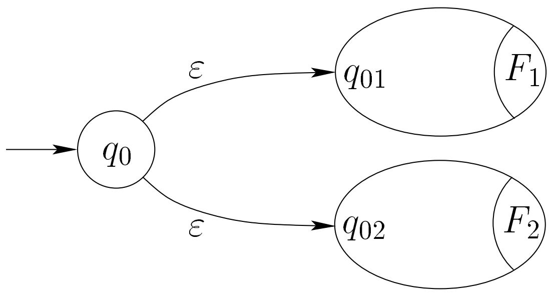
Wir zeigen zunächst, daß die Klasse der erkennbaren Sprachen unter den Booleschen Operationen sowie Konkatenation und Kleene-Stern abgeschlossen ist.

Satz 4.1
Sind L_1*,* L_2 *erkennbar, so auch* $L_1 \cup L_2$, $\overline{L_1}$, $L_1 \cap L_2$, $L_1 \setminus L_2$, $L_1 \cdot L_2$ *und* L_1^*.

Beweis:

Es sei $\mathcal{A}_i = (Q_i, \Sigma, q_{0i}, \Delta_i, F_i)$ ein NEA für L_i $(i = 1, 2)$. O.B.d.A. sei $Q_1 \cap Q_2 = \emptyset$.

1) Der folgende ε-NEA akzeptiert $L_1 \cup L_2$:
   $\mathcal{A} := (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, q_0, \Delta, F_1 \cup F_2)$, wobei $q_0 \notin Q_1 \cup Q_2$ und $\Delta := \Delta_1 \cup \Delta_2 \cup \{(q_0, \varepsilon, q_{01}), (q_0, \varepsilon, q_{02})\}$.



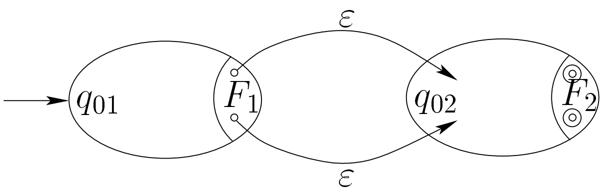
Mit Lemma 1.12 gibt es zu $\mathcal{A}$ einen äquivalenten NEA.

2) 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. Der DEA für $\overline{L_1}$ ist nun $\overline{\mathcal{A}} := (Q, \Sigma, q_0, \delta, Q \setminus F)$. Es gilt nämlich:
$$\begin{array}{lll} w \in \overline{L_1} & \text{gdw.} & w \notin L(\mathcal{A}_1) \\ & \text{gdw.} & w \notin L(\mathcal{A}) \\ & \text{gdw.} & \delta(q_0, w) \notin F \\ & \text{gdw.} & \delta(q_0, w) \in Q \setminus F \\ & \text{gdw.} & w \in L(\overline{\mathcal{A}}) \end{array}$$
<u>Beachte:</u> Man darf dies nicht direkt mit dem NEA machen!

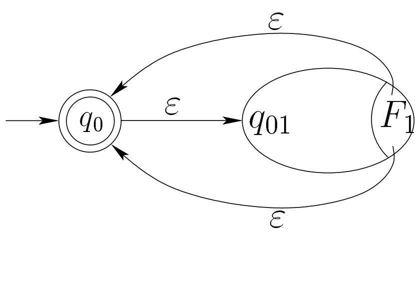
3) $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ zeigt wegen 1) und 2) auch 3). Da die Potenzmengenkonstruktion aber sehr aufwendig sein kann, ist es günstiger, direkt einen NEA für $L_1 \cap L_2$ zu konstruieren, den sogenannten Produktautomaten $\mathcal{A} := (Q_1 \times Q_2, \Sigma, (q_{01}, q_{02}), \Delta, F_1 \times F_2)$ 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. Daraus ergibt sich leicht $L(\mathcal{A}) = L_1 \cap L_2$.

4) $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$.

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



6) Der folgende ε-NEA akzeptiert L_1^*:
$\mathcal{A} := (Q_1 \cup \{q_0\}, \Sigma, q_0, \Delta, \{q_0\})$, wobei $q_0 \notin Q_1$ und $\Delta := \Delta_1 \cup \{(f, \varepsilon, q_0) \mid f \in F_1\} \cup \{(q_0, \varepsilon, q_{01})\}$.



□

Beachte: Alle angegebenen Konstruktionen sind effektiv. Die Automaten für $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 \cdot L_2$ und L_1^* sind polynomiell in der Größe der Automaten für L_1, L_2. Beim Komplement kann die Konstruktion exponentiell sein, wenn man von einem NEA ausgeht.

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

Beispiel 4.2
$L := \{a^n b^m \mid n \neq m\}$ ist nicht erkennbar (vgl. Beispiel 3.4). Anstatt dies direkt mit Lemma 3.5 zu zeigen, kann man auch verwenden, daß bereits bekannt ist, daß $L' := \{a^n b^n \mid n \geq 0\}$ nicht erkennbar ist. Wäre nämlich L erkennbar, so auch $L' = \overline{L} \cap \{a\}^* \cdot \{b\}^*$. Da wir schon wissen, daß L' nicht erkennbar ist, kann auch L nicht erkennbar sein.

Wir betrachten nun das Leerheitsproblem, das Wortproblem und das Äquivalenzproblem. Dabei gehen wir davon aus, daß die erkennbare Sprache durch einen DEA oder NEA gegeben ist. Bzgl. Entscheidbarkeit macht es keinen Unterschied, ob man einen DEA oder NEA gegeben hat, da man aus einem NEA effektiv einen äquivalenten DEA konstruieren kann (Satz 2.4). Bzgl. Komplexität kann der Unterschied aber relevant sein, 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 3.3), 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 4.3
Es sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA. Dann kann man „$L(\mathcal{A}) \neq \emptyset$?“ in der Zeit $O(|Q| + |\Delta|)$ entscheiden.

Beweis:
Man kann $\mathcal{A}$ als gerichteten Graphen $G = (Q, E)$ auffassen mit $E := \{(q_1, q_2) \mid (q_1, a, q_2) \in \Delta$ 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 (vgl. Vorlesung „Algorithmen und Datenstrukturen“). Insbesondere ist also das Leerheitsproblem für erkennbare Sprachen mit polynomiellem Aufwand lösbar. $\square$

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

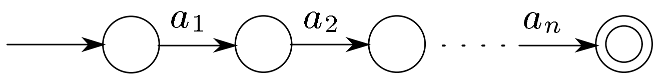
Ist $L = L(\mathcal{A})$ für einen DEA $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$, so kann man einfach, beginnend mit q_0, durch Anwendung von δ berechnen, zu welchem Zustand in $\mathcal{A}$ man mit w kommt und prüfen, ob dies ein Endzustand ist. Nimmt man Σ als konstant an, so benötigt eine Anwendung von δ nur konstante Zeit. Dies liefert:

Satz 4.4
Es sei $\mathcal{A} = (Q, \Sigma, q_0, \delta, F)$ ein DEA und $w \in \Sigma^*$. Dann kann man „$w \in L(\mathcal{A})$?“ in der Zeit $O(|w|)$ entscheiden.

Für einen NEA ist dies nicht so einfach, da man ja verschiedene mit w beschriftete Pfade haben kann und man diese (im schlimmsten Fall) alle betrachten muß, um festzustellen, ob einer davon mit einem Endzustand aufhört.

Satz 4.5
Es sei $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$ ein NEA und $w \in \Sigma^*$. Dann kann man „$w \in L(\mathcal{A})$?“ in der Zeit $O(|w| \cdot (|Q| + |\Delta|))$ entscheiden.

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



Dieser Automat hat $|w| + 1$ Zustände. Offenbar ist $w \in L(\mathcal{A})$ gdw. $L(\mathcal{A}) \cap L(\mathcal{A}_w) \neq \emptyset$. Wie groß ist nun der Produktautomat zu $\mathcal{A}$ und $\mathcal{A}_w$?
Zustände: $|Q| \cdot (|w| + 1)$
Übergänge: Für jeden Übergang $\bigcirc \xrightarrow{a_i} \bigcirc$ gibt es maximal $|\Delta|$ mögliche Übergänge in $\mathcal{A}$.
Also maximal $|w| \cdot |\Delta|$ viele Übergänge im Produktautomaten. Nach Satz 4.3 ist daher der Aufwand zum Testen von $L(\mathcal{A}) \cap L(\mathcal{A}_w) \neq \emptyset$:
$O(|Q| \cdot (|w| + 1) + |w| \cdot |\Delta|) = O(|w| \cdot (|Q| + |\Delta|))$ $\square$

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

Wie bereits erwähnt, kann man das Äquivalenzproblem auf das Leerheitsproblem reduzieren:
$L_1 = L_2$ gdw. $(L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2) = \emptyset$

Satz 4.6
Das Äquivalenzproblem ist für erkennbare Sprachen entscheidbar. Sind die Sprachen durch DEAs gegeben, so ist dies in polynomieller Zeit möglich.

Beweis:
Wir haben gesehen, daß die Automatenkonstruktionen, welche Abschluß unter Vereinigung, Durchschnitt und Komplement zeigen, effektiv sind. Daraus ergibt sich Entscheidbarkeit (auch bei NEAs). Die Konstruktionen für Vereinigung und Durchschnitt sind polynomiell. Beim Komplement ist dies nur dann der Fall, wenn bereits DEAs vorliegen. $\square$

Wir werden später sehen, daß sich der exponentielle Zeitaufwand für die Potenzmengenkonstruktion (wahrscheinlich) bei NEAs nicht vermeiden läßt: das Äquivalenzproblem für NEAs ist $PSPACE$-vollständig (schlimmer als NP).

5 Reguläre Ausdrücke und Sprachen

Wir hatten bisher verschiedene äquivalente Charakterisierungen der Klasse der erkennbaren Sprachen mit Hilfe von Transitionssystemen und Rechtskongruenzen 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 $\mathcal{A}$.

(4) $L = L(\mathcal{A})$ für ein endliches Transitionssystem $\mathcal{A}$.

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

(6) Die Nerode-Rechtskongruenz $\simeq_L$ hat endlichen Index.

Im folgenden betrachten wir eine weitere Charakterisierung mit Hilfe regulärer Ausdrücke.

Definition 5.1 (Syntax regulärer Ausdrücke)
Es 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 5.2
$((a \cdot b^*) + \emptyset^*)^* \in Reg_\Sigma$ *für* $\Sigma = \{a, b\}$

Notation:
Um Klammern zu sparen, lassen wir Außenklammern weg und vereinbaren, daß * stärker bindet als $\cdot$ und daß $\cdot$ stärker bindet als $+$. Außerdem lassen wir $\cdot$ meist ganz wegfallen. Der Ausdruck aus Beispiel 5.2 kann also geschrieben werden als $(ab^* + \emptyset^*)^*$.

Jedem regulären Ausdruck r über Σ wird eine formale Sprache $L(r)$ zugeordnet.

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

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

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

Beispiel 5.4

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

Bemerkung:
Statt $L(r)$ schreiben wir im folgenden häufig einfach r.
Dies ermöglicht es zu schreiben:

- $abb \in ab^* + b$ (eigentlich $abb \in L(ab^* + b)$)
- $(ab)^*a = a(ba)^*$ (eigentlich $L((ab)^*a) = L(a(ba)^*)$)

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

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

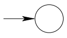
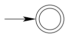
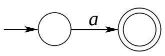
1) *L ist regulär.*

2) *L ist erkennbar.*

Beweis:

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

Verankerung:

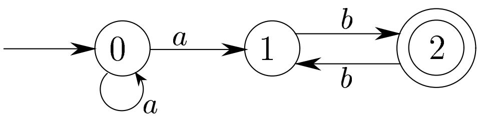
- $L(\emptyset) = \emptyset$ erkennbar:  ist NEA für $\emptyset$ (kein Endzustand).
- $L(\varepsilon) = \{\varepsilon\}$ erkennbar:  ist NEA für $\{\varepsilon\}$.
- $L(a) = \{a\}$ erkennbar:  ist NEA für $\{a\}$.

Schritt:

Weiß man bereits, daß $L(r)$ und $L(s)$ erkennbar sind, so folgt mit Satz 4.1 (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 $L = L(\mathcal{A})$. Wie in Definition 2.8 sei für $q \in Q$ der NEA $\mathcal{A}_q := (Q, \Sigma, q, \Delta, F)$ und $L_q = L(\mathcal{A}_q)$ definiert, d.h. insbesondere $L = L_{q_0}$.
Der Zusammenhang zwischen den L_qs kann nun als Gleichungssystem beschrieben werden, dessen Lösungen eindeutig bestimmte reguläre Sprachen sind.

Beispiel 5.6



$$
\begin{aligned}
L_0 &= \{a\} \cdot L_0 \cup \{a\} \cdot L_1 \\
L_1 &= \{b\} \cdot L_2 \\
L_2 &= \{b\} \cdot L_1 \cup \{\varepsilon\}
\end{aligned}
$$

Allgemein:

Für $p, q \in Q$ sei $A_{p,q} = \{a \in \Sigma \mid (p, a, q,) \in \Delta\}$ und für $p \in Q$ sei $B_p = \begin{cases} \{\varepsilon\} & \text{falls } p \in F \\ \emptyset & \text{falls } p \notin F \end{cases}$
Damit erfüllen die Sprachen L_p die folgenden Gleichungen: Für alle $p \in Q$ gilt:
$L_p = (\bigcup\limits_{q \in Q} A_{p,q} \cdot L_q) \cup B_p$

Behauptung:

Das Gleichungssystem

$(\star)\qquad X_p = (\bigcup_{q\in Q} A_{p,q}\cdot X_q)\cup B_p \qquad (p\in Q)$

hat genau eine Lösung und diese besteht aus regulären Sprachen.
Aus der Behauptung folgt, daß die Sprachen L_p regulär sind, da sie ja das System $(\star)$ lösen.
Wir zeigen zunächst, wie man eine einzige Gleichung der Form

$(\star\star)\qquad X = A\cdot X\cup B$

lösen kann. Daraus ergibt sich dann die Lösung von Gleichungssystemen der Form $(\star)$ durch Induktion über die Anzahl der Variablen.

Lemma 5.7 (Arden)
Es seien $A, B \subseteq \Sigma^*$ und $\varepsilon \notin A$. Die Gleichung $(\star\star)\quad X = A\cdot X\cup B$ hat als eindeutige Lösung $X = A^*\cdot B$.

Beachte:

Sind A, B regulär, so auch A^*B.

Aus dem Lemma folgt dann für die Gleichung $(\star)$, die wir als Gleichung der Form $(\star\star)$ für ein fixiertes $p\in Q$ auffassen:

$$X_p = A_{p,p}\cdot X_p \cup \left(\left(\bigcup_{p\neq q} A_{p,q}\cdot X_q\right)\cup B_p\right)$$

hat als eindeutige Lösung

$$X_p = A_{p,p}^*\cdot\left(\left(\bigcup_{q\neq p} A_{p,q}\cdot X_q\right)\cup B_p\right).$$

Setzt man diese Lösung in $(\star)$ ein, so erhält man ein System mit einer Variablen weniger. Nach Induktion hat dieses eine eindeutige Lösung, die aus regulären Sprachen besteht. Da in obiger Lösung für X_p nur reguläre Operationen $(\cup, \cdot, ^*)$ verwendet werden, ist auch diese Lösung regulär (und eindeutig nach Arden).

Beispiel 5.6 *(Fortsetzung)*

$$\begin{aligned} X_0 &= \{a\}\cdot X_0 \cup \{a\}\cdot X_1\\ X_1 &= \{b\}\cdot X_2\\ X_2 &= \{b\}\cdot X_1\cup\{\varepsilon\}\end{aligned}$$

Auflösen nach X_0:

$X_0 = \{a\}^*\cdot\{a\}\cdot X_1$

Einsetzen ändert die anderen Gleichungen nicht.

Auflösen nach X_1:

$X_1 = \emptyset^*\cdot\{b\}\cdot X_2 = \{b\}\cdot X_2$

Einsetzen liefert:

$X_2 = \{b\}\cdot\{b\}\cdot X_2\cup\{\varepsilon\}$

Auflösen nach X_2:

$X_2 = (\{b\}\cdot\{b\})^*$

Damit ist $X_1 = \{b\}\cdot(\{b\}\cdot\{b\})^*$ und $X_0 = \{a\}^*\cdot\{a\}\cdot\{b\}\cdot(\{b\}\cdot\{b\})^* = L(\mathcal{A})$. Als regulärer Ausdruck: $a^*ab(bb)^*$.

Beweis von Lemma 5.7:

1) A^*B ist Lösung:
$A \cdot A^* \cdot B \cup B = (A \cdot A^* \cup \{\varepsilon\}) \cdot B = A^* \cdot B$

2) Eindeutigkeit:
Es sei L eine Lösung, d.h. $L = AL \cup B$.

2.1) $A^*B \subseteq L$.
Wir zeigen durch Induktion über n: $A^nB \subseteq L$.

- $A^0B = B \subseteq AL \cup B = L$
- Gelte $A^nB \subseteq L$. Es folgt: $A^{n+1}B = AA^nB \subseteq AL \subseteq AL \cup B = L$

2.2) $L \subseteq A^*B$.
Angenommen, dies gilt nicht. Es sei $w \in L$ ein Wort minimaler Länge mit $w \notin A^*B$. $w \in L = AL \cup B$, d.h. $w \in AL$ oder $w \in B$. Für $w \in B$ folgt $w \in A^*B$ (Widerspruch). Aus $w \in AL$ folgt, daß es $u \in A$ und $v \in L$ gibt mit $w = uv$. Wegen $\varepsilon \notin A$ ist $|v| < |w|$. Wegen Minimalität von w folgt $v \in A^*B$ (Widerspruch, da dann $w = uv \in A^*B$). $\square$

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

- endliche Automaten mit Ausgabe:

  Übergänge $p \xrightarrow{a/v}_{\mathcal{A}} q$, wo $v \in \Gamma^*$ Wort über einem Ausgabealphabet ist. Solche Automaten beschreiben spezielle Funktionen $\Sigma^* \to \Gamma^*$.

- algebraische Theorie formaler Sprachen:

  Jeder Sprache L wird ein Monoid M_L (syntaktisches Monoid) zugeordnet. Klassen von Sprachen entsprechen Klassen von Monoiden, z.B. L regulär gdw. M_L endlich.

- Automaten auf unendlichen Wörtern:

  Büchi-Automaten sind endliche Automaten, bei denen man unendliche Wörter (unendliche Folgen von Buchstaben) eingibt.

- Baumautomaten:

  Bäume statt Wörter als Eingaben.

Teil II

Grammatiken, kontextfreie Sprachen und Kellerautomaten

Einführung

Wir werden hier Klassen formaler Sprachen untersuchen, die allgemeiner sind als die der regulären Sprachen. Insbesondere werden wir die Klasse der kontextfreien Sprachen genauer betrachten, da durch sie z.B. die Syntax von Programmiersprachen (zumindest in großen Teilen) beschreibbar ist.
Zunächst führen wir allgemein den Begriff der Grammatik ein. Klassen formaler Sprachen erhält man durch einschränkende Bedingungen an die Form der Grammatik.

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 erzeugt werden können durch wiederholtes Ersetzen.

Beispiel 6.1

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

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

Das Symbol S ist hier ein Hilfssymbol (nichtterminales Symbol) und man ist nur an den erzeugten Wörtern interessiert, die das Hilfssymbol nicht enthalten (Terminalwörter). Man sieht leicht, daß dies hier alle 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 sind (N: Nichtterminalsymbole, Σ: Terminalsymbole),*
- *$S \in N$ das Startsymbol ist,*

- $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ *eine endliche Menge von Ersetzungsregeln (Produktionen) ist.*

Produktionen $(u, v) \in P$ *schreibt man gewöhnlich als* $u \longrightarrow v$.

Beispiel 6.3
$G = (N, \Sigma, P, S)$ *mit* $N = \{S, B\}$, $\Sigma = \{a, b, c\}$,
$P = \{S \longrightarrow aSBc, S \longrightarrow abc, cB \longrightarrow Bc, bB \longrightarrow bb\}$

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)
Es sei $G = (N, \Sigma, P, S)$ *eine Grammatik und* x, y *seien aus* $(N \cup \Sigma)^*$.

1) *y aus x direkt ableitbar:*
   $x \vdash_G y$ *gdw.* $\exists x_1, x_2 \in (N \cup \Sigma)^* : \exists u \longrightarrow v \in P : x = x_1 u x_2 \wedge y = x_1 v x_2$

2) *y aus x in n Schritten ableitbar:*
   $x \vdash_G^n y$ *gdw.* $\exists x_0, x_1, \ldots, x_n \in (N \cup \Sigma)^* : x_0 = x \wedge x_n = y \wedge x_i \vdash_G x_{i+1}$ *für* $0 \leq i < n$

3) *y aus x ableitbar:*
   $x \vdash_G^* y$ *gdw.* $\exists n \geq 0 : x \vdash_G^n y$

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)*
$S \vdash_G abc$, *d.h.* $abc \in L(G)$
$S \vdash_G aSBc \vdash_G aaSBcBc \vdash_G aaabcBcBc \vdash_G aaabBccBc \vdash_G^2 aaabBBccc \vdash_G^2 aaabbbccc$
Es gilt: $L(G) = \{a^n b^n c^n \mid n \geq 1\}$

Beweis:

„$\supseteq$": Für $n = 1$ ist $abc \in L(G)$ klar. Für $n > 1$ sieht man leicht:
$S \vdash_G^{n-1} a^{n-1}S(Bc)^{n-1} \vdash_G a^n bc(Bc)^{n-1} \vdash_G^* a^n bB^{n-1}c^n \vdash_G^* a^n b^n c^n$

„$\subseteq$": Gelte $S \vdash_G^* w$ mit $w \in \Sigma^*$. Wird sofort die Produktion $S \longrightarrow abc$ verwendet, so ist $w = abc \in \{a^n b^n c^n \mid n \geq 1\}$. Sonst betrachten wir die Stelle in der Ableitung, an der S das letzte mal auftritt: $S \vdash_G^* a^{n-1}Su$ mit $u \in \{c, B\}^*$ und $|u|_B = |u|_c = n - 1$ (nur $S \longrightarrow aSBc$, $cB \longrightarrow Bc$ angewendet, da noch kein b vorhanden). $a^{n-1}Su \vdash_G a^n bcu \vdash_G^* w$. Um von $a^n bcu$ aus nun alle nichtterminalen Symbole B zu entfernen, muß man $bB \longrightarrow bb$ anwenden. Damit B aber auf b stößt, muß es links von allen cs stehen. □

Beispiel 6.5
$G = (N, \Sigma, P, S)$ *mit* $N = \{S, B\}$, $\Sigma = \{a, b\}$,
$P = \{S \longrightarrow aS, S \longrightarrow bS, S \longrightarrow abB, B \longrightarrow aB, B \longrightarrow bB, B \longrightarrow \varepsilon\}$
$L(G) = \Sigma^* \cdot \{a\} \cdot \{b\} \cdot \Sigma^*$

Die Grammatiken aus Beispiel 6.5, 6.3 und 6.1 gehören zu unterschiedlichen Stufen der Chomsky-Hierarchie:

Definition 6.6 (Typen von Chomsky-Grammatiken)
Es sei $G = (N, \Sigma, P, S)$ eine Grammatik.

- *Jede Grammatik G heißt Grammatik vom Typ 0.*
- *G heißt Grammatik vom Typ 1 (kontextsensitiv), falls jede Produktion von G die Form*
  - *$u_1Au_2 \longrightarrow u_1wu_2$ mit $A \in N$, $u_1, u_2, w \in (\Sigma \cup N)^*$ und $|w| \geq 1$ oder*
  - *$S \longrightarrow \varepsilon$ hat.*

  Ist $S \longrightarrow \varepsilon \in P$, so kommt S nicht auf der rechten Seite einer Produktion vor.
- *G heißt Grammatik vom Typ 2 (kontextfrei), falls jede Regel von G die Form $A \longrightarrow w$ hat mit $A \in N$, $w \in (\Sigma \cup N)^*$.*
- *G heißt Grammatik vom Typ 3 (rechtslinear), falls jede Regel von G die Form $A \longrightarrow uB$ oder $A \longrightarrow u$ hat mit $A, B \in N$, $u \in \Sigma^*$.*

kontextfrei: linke Seite nur Nichtterminalsymbol A, das daher stets unabhängig vom Kontext im Wort ersetzt werden kann.

kontextsensitiv: $u_1Au_2 \longrightarrow u_1wu_2$. Hier ist die Ersetzung von A durch w abhängig davon, daß der richtige Kontext (u_1 links und u_2 rechts) vorhanden ist. $|w| \geq 1$ sorgt dafür, daß die Produktionen die Wörter verlängern (Ausnahme $S \longrightarrow \varepsilon$, die aber nur zur Erzeugung von ε dient).

Definition 6.7 (Klasse der Typ-i-Sprachen)
Für $i = 0, 1, 2, 3$ ist die Klasse der Typ-i-Sprachen definiert als
$\mathcal{L}_i := \{L(G) \mid G$ *ist Grammatik vom Typ* $i\}$.

Wir werden sehen: $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$.
Nach Definition der Grammatiktypen gilt offenbar $\mathcal{L}_3 \subseteq \mathcal{L}_2$ und $\mathcal{L}_1 \subseteq \mathcal{L}_0$.

7 Rechtslineare Grammatiken und reguläre Sprachen

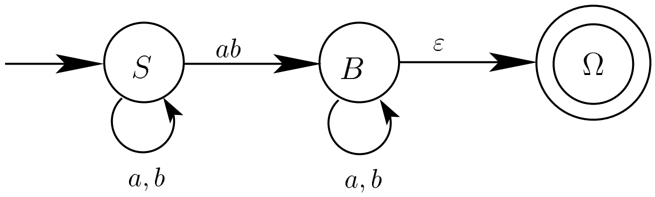
Satz 7.1
Die Typ-3-Sprachen sind genau die regulären/erkennbaren Sprachen, d.h.
$\mathcal{L}_3 = \{L \mid L$ *ist regulär*$\}$.

Beweis:

1) Es sei $L \in \mathcal{L}_3$, d.h. $L = L(G)$ für eine Typ-3-Grammatik $G = (N, \Sigma, P, S)$. Es gilt $w \in L(G)$ gdw.
   ($\star$) Es gibt eine Ableitung $S = B_0 \vdash_G w_1B_1 \vdash_G w_1w_2B_2 \vdash_G \ldots \vdash_G w_1 \ldots w_{n-1}B_{n-1} \vdash_G w_1 \ldots w_{n-1}w_n$ für Produktionen $B_{i-1} \longrightarrow w_iB_i \in P$ ($i = 1, \ldots, n$) und $B_{n-1} \longrightarrow w_n \in P$.
   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 $\Omega \notin N$ neuer 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)*
$P = \{S \longrightarrow aS, S \longrightarrow bS, S \longrightarrow abB, B \longrightarrow aB, B \longrightarrow bB, B \longrightarrow \varepsilon\}$ *liefert*

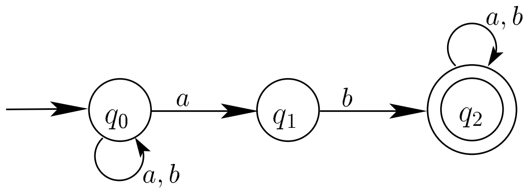


2) Es sei $L = L(\mathcal{A})$ für einen NEA $\mathcal{A} = (Q, \Sigma, q_0, \Delta, F)$. Wir definieren daraus eine Typ-3-Grammatik $G = (N, \Sigma, P, S)$ wie folgt:

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

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

Beispiel 7.2



liefert die Grammatik mit den Produktionen
$P = \{q_0 \longrightarrow aq_0, q_0 \longrightarrow bq_0, q_0 \longrightarrow aq_1, q_1 \longrightarrow bq_2, q_2 \longrightarrow aq_2, q_2 \longrightarrow bq_2, q_2 \longrightarrow \varepsilon\}$.

Korollar 7.3
$\mathcal{L}_3 \subset \mathcal{L}_2$.

<u>Beweis:</u>
Wir wissen bereits $\mathcal{L}_3 \subseteq \mathcal{L}_2$. Mit Beispiel 6.1 ist $L := \{a^n b^n \mid n \geq 0\} \in \mathcal{L}_2$. Im Teil I haben wir gezeigt, daß L nicht erkennbar/regulär ist, d.h. mit Satz 7.1 folgt $L \notin \mathcal{L}_3$. $\quad\square$

Beispiel 7.4
Als ein weiteres Beispiel für eine kontextfreie Sprache, die nicht regulär ist, betrachten wir $\{a^n b^m \mid n \neq m\}$. *Man kann diese Sprache mit folgender kontextfreier Grammatik erzeugen:* $G = (N, \Sigma, P, S)$ *mit* $N = \{S, S_\geq, S_\leq\}$, $\Sigma = \{a, b\}$, $P = \{S \longrightarrow aS_\geq, S \longrightarrow S_\leq b, S_\geq \longrightarrow aS_\geq b, S_\geq \longrightarrow aS_\geq, S_\geq \longrightarrow \varepsilon, S_\leq \longrightarrow aS_\leq b, S_\leq \longrightarrow S_\leq b, S_\leq \longrightarrow \varepsilon\}$.
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 kontextfreier Grammatiken

Wir werden zeigen, daß man die Klasse aller kontextfreien Sprachen bereits mit kontextfreien Grammatiken einer eingeschränkten Form erzeugen kann.
Zwei Grammatiken heißen äquivalent, falls sie dieselbe Sprache erzeugen.
Zunächst zeigen wir, daß man „überflüssige" Symbole aus kontextfreien Grammatiken eliminieren kann.

Definition 8.1 (terminierendes und erreichbares Symbol, 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.*

Lemma 8.2
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:

„$\subseteq$": Offenbar enthalten alle E_i nur erreichbare Symbole

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

Beispiel 8.3
$P = \{S \longrightarrow aS, S \longrightarrow SB, S \longrightarrow SS, S \longrightarrow \varepsilon, A \longrightarrow ASB, A \longrightarrow C, B \longrightarrow Cb\}$
$E_0 = \{S\} \subset E_1 = \{S, B\} \subset E_2 = \{S, B, C\} = E_3$
Es ist also A *das einzige nichterreichbare Symbol.*

Lemma 8.4
Zu einer kontextfreien Grammatik $G = (N, \Sigma, P, S)$ kann man effektiv die Menge der terminierenden Symbole bestimmen.

Beweis:
Wir definieren dazu

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

Es ist $T_1 \subseteq T_2 \subseteq \ldots \subseteq N$, und es gibt daher ein k mit $T_k = T_{k+1} = \bigcup_{i \geq 1} T_i$.

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

„$\subseteq$“: Offenbar sind alle Elemente von T_i $(i \geq 1)$ terminierend:
$i = 1$: $A \vdash_G w \in \Sigma^*$.
$i \to i+1$: $A \vdash_G u_1 B_1 u_2 B_2 \ldots u_n B_n u_{n+1}$, $B_i \vdash_G^* w_i \in \Sigma^*$

„$\supseteq$“: Zeige durch Induktion über i:
$A \vdash_G^{\leq i} w \in \Sigma^* \Rightarrow A \in T_i$.
$i = 1$: $A \vdash_G^{\leq 1} w \in \Sigma^* \Rightarrow A \longrightarrow w \in P \Rightarrow A \in T_1$
$i \to i+1$: $A \vdash_G u_1 B_1 \ldots u_n B_n u_{n+1} \vdash_G^{\leq i} w = u_1 w_1 \ldots u_n w_n u_{n+1}$, $u_j w_j \in \Sigma^*$, $B_j \in N$
$\Rightarrow$ Für alle j gilt: $B_j \vdash_G^{\leq i} w_j \in \Sigma^*$
$\Rightarrow B_j \in T_i$ (Induktion) $\Rightarrow A \in T_{i+1}$ $\square$

Aus Lemma 8.4 folgt unmittelbar:

Satz 8.5
Das Leerheitsproblem ist für kontextfreie Sprachen entscheidbar.

Beweis:
Offenbar gilt $L(G) \neq \emptyset$ gdw. $\exists w \in \Sigma^* : S \vdash_G^* w$ gdw. S ist terminierend. $\square$

Lemma 8.2 und 8.4 zusammen zeigen, wie man unerreichbare und nichtterminierende Symbole eliminieren kann.

Satz 8.6
Zu jeder kontextfreien Grammatik G mit $L(G) \neq \emptyset$ kann man effektiv eine äquivalente reduzierte kontextfreie Grammatik erzeugen.

Beweis:
Zu $G = (N, \Sigma, P, S)$ definieren wir $G' := (N', \Sigma, P', S)$, 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: Aus $L(G) \neq \emptyset$ folgt $S \in N'$!

In einem zweiten Schritt eliminieren wir unerreichbare Symbole, d.h. $G'' := (N'', \Sigma, P'', S)$, wobei $N'' := \{A \in N' \mid A \text{ ist erreichbar in } G'\}$ und $P'' := \{A \longrightarrow w \in P' \mid A \in N''\}$.

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. $\square$

Vorsicht: Symbole, die in G noch erreichbar waren, müssen es in G' nicht sein.

Als nächstes eliminieren wir Regeln der Form $A \longrightarrow \varepsilon$.

Lemma 8.7
Es sei G eine kontextfreie Grammatik. Dann läßt sich eine Grammatik G' ohne Regeln der Form $A \longrightarrow \varepsilon$ konstruieren mit $L(G') = L(G) \setminus \{\varepsilon\}$.

Beweis:

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

$$\begin{aligned} N_1 &:= \{A \in N \mid A \longrightarrow \varepsilon \in P\} \\ N_{i+1} &:= N_i \cup \{A \in N \mid A \longrightarrow B_1 \dots B_n \in P \text{ für } B_j \in N_i\} \end{aligned}$$

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 $A \longrightarrow u_1 B_1 \dots u_n B_n u_{n+1}$ mit $B_i \in N_k$ und $u_i \in (\Sigma \cup N \setminus N_k)^*$ die Regeln $A \longrightarrow u_1 \beta_1 u_2 \dots u_n \beta_n u_{n+1}$ hinzu mit $\beta_i \in \{B_i, \varepsilon\}$ und $u_1 \beta_1 u_2 \dots u_n \beta_n u_{n+1} \neq \varepsilon$. $\square$

Beispiel:
$P = \{S \longrightarrow aS, S \longrightarrow SS, S \longrightarrow bA, A \longrightarrow BB, B \longrightarrow CC, B \longrightarrow aAbC, C \longrightarrow \varepsilon\}$
$N_0 = \{C\}, N_1 = \{C, B\}, N_2 = \{C, B, A\} = N_3$
$P' = \{S \longrightarrow aS, S \longrightarrow SS, S \longrightarrow bA, S \longrightarrow b,$
$A \longrightarrow BB, A \longrightarrow B,$
$B \longrightarrow CC, B \longrightarrow C,$
$B \longrightarrow aAbC, B \longrightarrow abC, B \longrightarrow aAb, B \longrightarrow ab\}$
Die Ableitung $S \vdash bA \vdash bBB \vdash bCCB \vdash bCCCC \vdash^* b$ kann in G' direkt durch $S \vdash b$ erreicht werden.

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.*

Satz 8.9
Zu jeder kontextfreien Grammatik G kann effektiv eine äquivalente ε-freie Grammatik konstruiert werden.

Beweis:

Konstruiere G' wie im Beweis von Lemma 8.7 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 $S_0 \longrightarrow \varepsilon$. $\square$

Korollar 8.10
$\mathcal{L}_2 \subseteq \mathcal{L}_1$.

Beweis:

Offenbar ist jede ε-freie kontextfreie Grammatik eine Typ-1-Grammatik. (Produktionen: $u_1 A u_2 \longrightarrow u_1 w u_2$ mit $|w| \geq 1$, kontextfrei: $u_i = \varepsilon$; $S \longrightarrow \varepsilon$ und S nicht auf rechter Seite). $\square$

Wir zeigen nun, daß man auch auf Regeln der Form $A \longrightarrow B$ verzichten kann.

Satz 8.11
Zu jeder kontextfreien Grammatik kann man effektiv eine äquivalente kontextfreie Grammatik konstruieren, die keine Regeln der Form $A \longrightarrow B$ $(A, B \in N)$ enthält.

Beweisskizze:

1) Bestimme zu jedem $A \in N$ die Menge $N(A) := \{B \in N \mid A \vdash_G^* B\}$ (effektiv machbar).

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

Beispiel:
$P = \{S \longrightarrow A, A \longrightarrow B, B \longrightarrow aA, B \longrightarrow b\}$
$N(S) = \{S, A, B\}, \quad N(A) = \{A, B\}, \quad N(B) = \{B\}$
$P' = \{B \longrightarrow aA, A \longrightarrow aA, S \longrightarrow aA, B \longrightarrow b, A \longrightarrow b, S \longrightarrow b\}$

Wir zeigen nun, daß man sich auf Regeln sehr einfacher Form beschränken kann, nämlich $A \longrightarrow a$ und $A \longrightarrow BC$.

Satz 8.12 (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 Regeln der Form $A \longrightarrow B$ (Reihenfolge!).

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 $A \longrightarrow B_1 \ldots B_n$ für $n > 2$ werden ersetzt durch $A \longrightarrow B_1 C_1$, $C_1 \longrightarrow B_2 C_2$, $\ldots$, $C_{n-2} \longrightarrow B_{n-1} B_n$, wobei die C_i jeweils neue Symbole sind. $\square$

Beachte:

Für einen NEA $\mathcal{A}$ (eine rechtslineare Grammatik G) kann man $w \in L(\mathcal{A})$ ($w \in L(G)$) z.B. deshalb entscheiden, da ein Pfad in $\mathcal{A}$, der w erkennt (eine Ableitung in G, die w erzeugt) genau die Länge $|w|$ haben muß. Es gibt aber nur endlich viele Pfade (Ableitungen) dieser festen Länge.
Bei allgemeinen kontextfreien Grammatiken kann man keine Schranke für die Länge einer Ableitung von w aus S angeben, wohl aber bei Grammatiken in Chomsky-Normalform:

- 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: $w \in L(G) \setminus \{\varepsilon\}$ wird durch eine Ableitung der Länge $2|w|-1$ erzeugt.
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 8.13
Es sei $G = (N, \Sigma, P, S)$ eine kontextfreie Grammatik in Chomsky-Normalform und $w = a_1 \dots a_n \in \Sigma^*$. Wir definieren:

- $w_{ij} := a_i \dots a_j$ *(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 das folgende iterative Verfahren zur Bestimmung von N_{1k}:

Algorithmus 8.14 (CYK-Algorithmus von Cocke, Younger, Kasami)
`for` i `:=` 1 `to` n `do`
  N_{ii} `:=` $\{A \mid A \longrightarrow a_i \in P\}$
`for` d `:=` 1 `to` $n-1$ `do` *(wachsende Differenz $d = j - i$)*
  `for` i `:=` 1 `to` $n-d$ `do`
    j `:=` $i+d$
    N_{ij} `:=` $\emptyset$
    `for` k `:=` i `to` $j-1$ `do`
      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 die Differenzen $k-i$ und $j-k+1$ kleiner als das aktuelle d, also sind N_{ik} und $N_{(k+1)j}$ bereits berechnet.

Satz 8.15
Für eine gegebene Grammatik in Chomsky-Normalform entscheidet Algorithmus 8.14 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 ist die Suche nach den Produktionen $A \longrightarrow BC$ und $A \longrightarrow a_i$ auch konstant.

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

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

Eine weitere interessante Normalform für kontextfreie Grammatiken ist die Greibach-Normalform, bei der jede Produktion mindestens ein Terminalsymbol erzeugt.

Satz 8.16

Jede kontextfreie Grammatik läßt sich effektiv umformen in eine äquivalente Grammatik, die nur Regeln der Form $A \longrightarrow aw$ ($A \in N, a \in \Sigma, w \in N^*$) (und eventuell $S \longrightarrow \varepsilon$, wobei S nicht rechts vorkommt) enthält (ohne Beweis!). Eine derartige Grammatik heißt Grammatik in Greibach-Normalform.

9 Abschlußeigenschaften kontextfreier Sprachen

Satz 9.1

Die Klasse $\mathcal{L}_2$ der kontextfreien Sprachen ist unter Vereinigung, Konkatenation und Kleene-Stern abgeschlossen.

Beweis:

Es sei $L_1 = L(G_1)$ und $L_2 = L(G_2)$ für kontextfreie Grammatiken $G_i = (N_i, \Sigma, P_i, S_i)$ ($i = 1, 2$). 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) $G' := (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \longrightarrow S_1 S_2\}, S)$ 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, sogenannte Ableitungsbäume dar.

Beispiel 9.2
$P = \{S \longrightarrow SbS, S \longrightarrow a\}$
Drei Ableitungen von ababa:

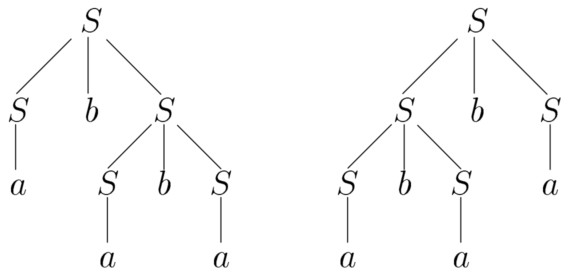
1) $S \vdash SbS \vdash abS \vdash abSbS \vdash ababS \vdash ababa$

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.

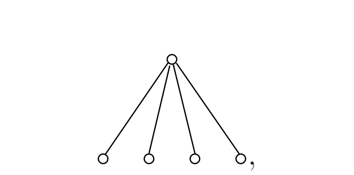
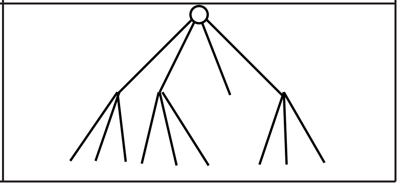
Allgemein:
Die Knoten des Ableitungsbaumes sind mit Elementen aus $\Sigma \cup N$ beschriftet. Ein mit A beschrifteter Knoten kann mit $\alpha_1, \ldots, \alpha_n$ beschriftete Nachfolgerknoten haben, wenn $A \longrightarrow \alpha_1 \ldots \alpha_n \in P$ ist.
Ein Ableitungsbaum, dessen Wurzel mit A 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 9.3 (Pumping-Lemma für kontextfreie Sprachen)
Es sei G eine ε-freie kontextfreie Grammatik ohne Regeln der Form $A \longrightarrow B$, die m nichtterminale Symbole enthält und für die die Länge der rechten Regelseiten $\leq k$ ist. Es sei $n = k^{m+1}$. Dann gibt es für jedes $z \in L(G)$ mit $|z| > n$ eine Zerlegung $z = uvwxy$ mit:

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

Beweis:

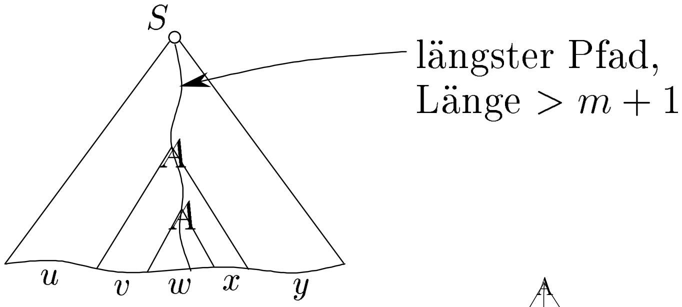
1) Ein Baum der Tiefe $\leq t$ und der Verzweigungszahl $\leq k$ hat maximal k^t viele Blätter:

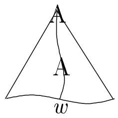
eine Ebene: $\leq k$ Blätter	zwei Ebenen: $\leq k^2$ Blätter
	

etc.

Da der Ableitungsbaum für z als Blattanzahl $|z| > k^{m+1}$ hat, ist die maximale Tiefe (längster Pfad von Wurzel bis Blatt) $> m + 1$.

2) Da es nur m verschiedene Elemente in N gibt, kommt auf diesem längsten Pfad ein nichtterminales Symbol A zweimal vor.



Wir wählen hier o.B.d.A. A so, daß es in dem Baum  keine andere Variablenwiederholung gibt. Daher hat dieser Baum eine Tiefe $\leq m + 1$, was $|vwx| \leq k^{m+1} = n$ 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. $\square$

Satz 9.4
$\mathcal{L}_2 \subset \mathcal{L}_1$, da $L = \{a^n b^n c^n \mid n \geq 1\} \in \mathcal{L}_1 \setminus \mathcal{L}_2$.

Beweis:

1) Wir zeigen zunächst $L \notin \mathcal{L}_2$. 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 $n_0 = k^{m+1}$ die zugehörige Zahl aus Lemma 9.3. Wir betrachten $z = a^{n_0}b^{n_0}c^{n_0} \in L = L(G)$. Mit Satz 9.3 gibt es eine Zerlegung $z = uvwxy$, $vx \neq \varepsilon$ und $uv^iwx^iy \in L$ für alle $i \geq 0$.

1. Fall: v enthält verschiedene Buchstaben. Man sieht leicht, daß dann $uv^2wx^2y \notin a^*b^*c^* \supseteq L$.

2. Fall: x enthält verschiedene Buchstaben. Führt zu entsprechendem Widerspruch.

3. Fall: v enthält lauter gleiche Buchstaben und x enthält lauter gleiche Buchstaben. Dann gibt es einen Buchstaben aus $\{a, b, c\}$, der in xv nicht vorkommt. Daher kommt dieser in $uv^0wx^0y = uwy$ weiterhin n_0-mal vor. Aber $|uwy| < 3n_0$, was $uwy \notin L$ zeigt.

2) In Beispiel 6.3 hatten wir eine Grammatik G mit $L(G) = \{a^n b^n c^n \mid n \geq 1\}$ angegeben. Leider enthält G eine Produktion, die nicht die Bedingungen für kontextsensitive Produktionen ($uAv \longrightarrow uwv$, $|w| \geq 1$) erfüllt: $cB \longrightarrow Bc$. Wir modifizieren G wie folgt:

- Ersetze in allen Produktionen c durch ein neues nichtterminales Symbol C und nimm die Produktion $C \longrightarrow c$ hinzu: $\{S \longrightarrow aSBC, S \longrightarrow abC, CB \longrightarrow BC, bB \longrightarrow bb, C \longrightarrow c\}$.
- Ersetze nun $CB \longrightarrow BC$ durch die kontextsensitiven Produktionen $CB \longrightarrow A_1B, A_1B \longrightarrow A_1A_2, A_1A_2 \longrightarrow BA_2, BA_2 \longrightarrow BC$. Diese Produktionen können nur dazu verwendet werden, $CB \longrightarrow BC$ zu simulieren. $\square$

Beachte:

Auf diese Art kann man leicht zeigen, daß jede nichtkürzende Produktion $u \longrightarrow v$ (mit $|u| \leq |v|$) durch kontextsensitive Produktionen ersetzt werden kann, ohne die Sprache zu ändern.

Korollar 9.5

Die Klasse $\mathcal{L}_2$ der kontextfreien Sprachen ist nicht unter Durchschnitt 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 $\mathcal{L}_2$:

$$\{a^n b^n c^m \mid n \geq 1, m \geq 1\} = \underbrace{\underbrace{\{a^n b^n \mid n \geq 1\}}_{\in \mathcal{L}_2} \cdot \underbrace{\{c^m \mid m \geq 1\}}_{= c^+ \in \mathcal{L}_3 \subseteq \mathcal{L}_2}}_{\in \mathcal{L}_2 \text{ (Konkatenation)}}$$

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.

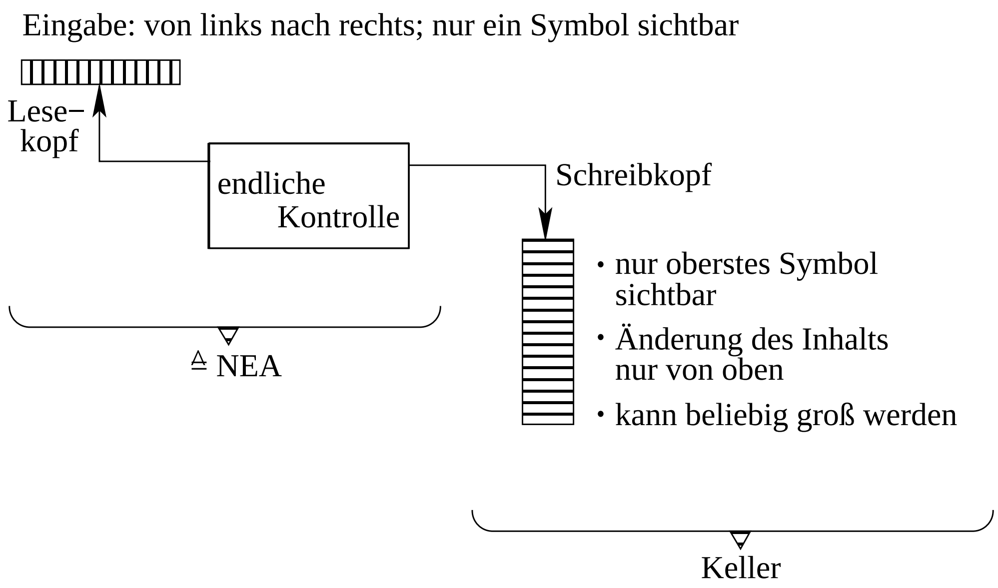
3) $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$.
Wäre $\mathcal{L}_2$ unter Komplement abgeschlossen, so auch unter $\cap$, da $\mathcal{L}_2$ unter $\cup$ abgeschlossen ist. $\square$

Beachte:

Man kann daher das Äquivalenzproblem für kontextfreie Sprachen nicht einfach auf das Leerheitsproblem reduzieren. Wir werden später sehen, daß das Äquivalenzproblem für kontextfreie Sprachen sogar 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 (potentiell unendliche) Speicherkomponente, einen sogenannten Keller, erweitern.



Kellerautomat

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:

(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).

Den aktuellen Zustand einer Kellerautomatenberechnung kann man beschreiben 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:

- $(q, aw, Z\gamma) \vdash (q', w, \beta\gamma)$ *falls* $(q, a, Z, \beta, q') \in \Delta$
- $(q, w, Z\gamma) \vdash (q', w, \beta\gamma)$ *falls* $(q, \varepsilon, Z, \beta, q') \in \Delta$
- $\mathcal{K} \vdash^* \mathcal{K}'$ *gdw.* $\exists n \geq 0\ \exists \mathcal{K}_0, \ldots, \mathcal{K}_n$ *mit* $\mathcal{K}_0 = \mathcal{K}$, $\mathcal{K}_n = \mathcal{K}'$ *und* $\mathcal{K}_i \vdash \mathcal{K}_{i+1}$ *für* $0 \leq i < n$.

Der Automat $\mathcal{A}$ akzeptiert das Wort $w \in \Sigma^*$ gdw. $(q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$ (Eingabewort ganz gelesen und Keller leer). Die von $\mathcal{A}$ akzeptierte Sprache ist $L(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \text{ akzeptiert } w\}$.

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* Δ:

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	ε	q_1	*(erstes b, entnimm Z)*
q_1	b	Z	ε	q_1	*(weitere b's, entnimm Z)*
q_1	ε	Z_0	ε	f	*(lösche das Kellerstartsymbol)*

Konfigurationsübergänge:

1) $(q_0, aabb, Z_0) \vdash (q_0, abb, ZZ_0) \vdash (q_0, bb, ZZZ_0) \vdash (q_1, b, ZZ_0) \vdash (q_1, \varepsilon, Z_0) \vdash (f, \varepsilon, \varepsilon)$ – *akzeptiert*

2) $(q_0, aab, Z_0) \vdash^* (q_0, b, ZZZ_0) \vdash (q_1, \varepsilon, ZZ_0)$ – *kein Übergang mehr möglich*

3) $(q_0, abb, Z_0) \vdash (q_0, bb, ZZ_0) \vdash (q_1, b, Z_0) \vdash (f, b, \varepsilon)$ – *nicht akzeptiert*

Der Kellerautomat aus Beispiel 10.3 ist deterministisch, da es zu jeder Konfiguration höchstens eine Folgekonfiguration gibt.

Definition 10.4
Der PDA $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ heißt deterministisch, falls die folgenden beiden Bedingungen erfüllt sind:

1) $\forall q \in Q\ \forall a \in \Sigma \cup \{\varepsilon\}\ \forall Z \in \Gamma$ *existiert höchstens ein Übergang der Form* $(q, a, Z, \ldots, \ldots) \in \Delta$.

2) *Existiert ein Tupel* $(q, \varepsilon, Z, \ldots, \ldots) \in \Delta$, *so existiert kein Tupel der Form* $(q, a, Z, \ldots, \ldots) \in \Delta$ *mit* $a \in \Sigma$.

Beispiel 10.5
Die Sprache $L = \{w \overleftarrow{w} \mid w \in \{a, b\}^*\}$ (wobei für $w = a_1 \ldots a_n$ gilt $\overleftarrow{w} = a_n \ldots a_1$) wird von einem nichtdeterministischen PDA akzeptiert.
Idee: Der PDA legt die erste Worthälfte im Keller ab und greift darauf in umgekehrter Reihenfolge beim Lesen der zweiten Worthälfte zurück. So kann Nichtübereinstimmung festgestellt werden (kein Übergang bei Nichtübereinstimmung).

Nichtdeterminismus ist intuitiv nötig, da man „raten" muß, wann die Wortmitte erreicht ist (Man kann beweisen, daß kein deterministischer PDA diese Sprache akzeptiert).
$Q = \{q_0, q_1, q_2, f\}$, $\Gamma = \{a, b, Z_0\}$, $\Sigma = \{a, b\}$, Δ :

q_0	ε	Z_0	ε	f	*akzeptiert* ε
q_0	a/b	Z_0	a/bZ_0	q_1	*Lesen*
q_1	a	b	ab	q_1	*und*
q_1	b	a	ba	q_1	*Speichern*
q_1	a	a	aa	q_1	*der ersten*
q_1	b	b	bb	q_1	*Worthälfte*
q_1	b	b	ε	q_2	*Lesen der*
q_1	a	a	ε	q_2	*zweiten Hälfte*
q_2	b	b	ε	q_2	*und Vergleichen*
q_2	a	a	ε	q_2	*mit erster*
q_2	ε	Z_0	ε	f	*Entfernen Kellerstartsymb.*

Es gibt auch einen anderen Akzeptanzbegriff für PDAs, bei denen man nicht fordert, daß der Keller leer ist, sondern daß ein Endzustand (aus einer Menge $F \subseteq Q$) erreicht ist. Dieser Akzeptanzbegriff liefert aber dieselbe Sprachklasse.

Wir werden nun zeigen, daß man mit Kellerautomaten genau die kontextfreien Sprachen akzeptieren kann.

Satz 10.6
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. □

Beachte:
Jedes Wort in $L(G)$ kann durch eine Linksableitung erzeugt werden (da G kontextfrei ist).

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

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

Beispiel:
$P = \{S \longrightarrow \varepsilon, S \longrightarrow aSa, S \longrightarrow bSb\}$ liefert die Übergänge:

q	ε	S	ε	q	$(S \longrightarrow \varepsilon)$
q	ε	S	aSa	q	$(S \longrightarrow aSa)$
q	ε	S	bSb	q	$(S \longrightarrow bSb)$
q	a	a	ε	q	(a entfernen)
q	b	b	ε	q	(b entfernen)

Die Ableitung $S \vdash aSa \vdash abSba \vdash abba$ entspricht der Konfigurationsfolge $(q, abba, S) \vdash (q, abba, aSa) \vdash (q, bba, Sa) \vdash (q, bba, bSba) \vdash (q, ba, Sba) \vdash (q, ba, ba) \vdash (q, a, a) \vdash (q, \varepsilon, \varepsilon)$.

Behauptung:

Für $u, v \in \Sigma^*$ und $\alpha \in \{\varepsilon\} \cup N \cdot (\Sigma \cup N)^*$ gilt: $S \vdash_G^* u\alpha$ mit Linksableitung gdw. $(q, uv, S) \vdash^* (q, v, \alpha)$.

<u>Beachte:</u>

Für $\alpha = \varepsilon = v$ folgt: $S \vdash_G^* u$ gdw. $(q, u, S) \vdash^* (q, \varepsilon, \varepsilon)$, d.h. $L(G) = L(\mathcal{A})$.

„$\Leftarrow$“: Induktion über die Anzahl k der Übergänge mit Transitionen der Form (I): $(q, \varepsilon, A, \gamma, q)$.

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

$k \to k+1$**:** $(q, \underbrace{u_1 u_2}_{u} v, S) \vdash^* (q, u_2 v, A\alpha') \overset{\text{letzte Trans. (I)}}{\vdash} (q, u_2 v, \underbrace{\gamma\alpha'}_{u_2\alpha}) \overset{\text{Trans. (II)}}{\vdash^*} (q, v, \alpha)$

Induktion liefert: $S \vdash^* u_1 A\alpha'$ und außerdem wegen $A \longrightarrow \gamma \in P$: $u_1 A\alpha' \vdash u_1\gamma\alpha' = u_1 u_2 \alpha = u\alpha$.

„$\Rightarrow$“: Induktion über die Länge der Linksableitung

$k = 0$**:** Dann ist $u = \varepsilon$, $\alpha = S$ und $(q, v, S) \vdash^0 (q, v, S)$

$k \to k+1$**:** $S \vdash_G^{k+1} u\alpha$, d.h. $S \vdash_G^k u'A\beta \vdash \underbrace{u'\gamma\beta}_{=u\alpha}$ mit $u' \in \Sigma^*$ (da Linksableitung) und $A \longrightarrow \gamma \in P$. Es sei u'' das längste Anfangsstück von $\gamma\beta$, das in Σ^* liegt. Dann ist $u'u'' = u$ und α ist der Rest von $\gamma\beta$, d.h. $\gamma\beta = u''\alpha$.

Induktion liefert: $(q, u' \underbrace{u''v}_{v'}, S) \vdash^* (q, u''v, A\beta) \vdash (q, u''v, \gamma\beta) = (q, u''v, u''\alpha)$. Außerdem gibt es Übergänge $(q, u''v, u''\alpha) \vdash^* (q, v, \alpha)$. $\square$

„$2 \longrightarrow 1$“: Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, Z_0, \Delta)$ ein PDA. Die Nichtterminalsymbole der zugehörigen Grammatik G sind alle Tripel $[p, Z, q] \in Q \times \Gamma \times Q$.

<u>Idee:</u>

Es soll gelten: $[p, Z, q] \vdash_G^* u \in \Sigma^*$ gdw. $\mathcal{A}$ kommt vom Zustand p in den Zustand q durch Lesen von u auf dem Eingabeband und Löschen von Z aus dem Keller (ohne dabei die Symbole unter Z anzutasten).

Wie realisiert man dies durch geeignete Produktionen? Betrachte den PDA-Übergang $(p, a, Z, X_1 \ldots X_n, p') \in \Delta$. Hier wird a auf dem Eingabeband verbraucht (falls nicht $a = \varepsilon$). Z wird durch $X_1 \ldots X_n$ ersetzt. Um den Kellerinhalt zu erreichen, den man durch einfaches Wegstreichen von Z erhalten würde, muß man also nun noch $X_1 \ldots X_n$ löschen. Löschen von X_i kann durch das Nichtterminalsymbol $[p_{i-1}, X_i, p_i]$ (für geeignete Zwischenzustände p_{i-1}, p_i) beschrieben werden.

<u>Formale Definition:</u>

$$
\begin{aligned}
G &:= (N, \Sigma, P, S) \text{ mit} \\
N &:= \{S\} \cup \{[p, Z, q] \mid p, q \in Q, Z \in \Gamma\} \\
P &:= \{S \longrightarrow [q_0, Z_0, q] \mid q \in Q\} \cup \\
&\quad\ \{[p, Z, q] \longrightarrow a[p_0, X_1, p_1][p_1, X_2, p_2] \ldots [p_{n-1}, X_n, q] \mid p, q, p_0, p_1, \ldots, p_{n-1} \in Q, \\
&\quad\ a \in \Sigma \cup \{\varepsilon\} \text{ und } (p, a, Z, X_1 \ldots X_n, p_0) \in \Delta \text{ und } q = p_0 \text{ falls } n = 0\}
\end{aligned}
$$

Beachte:

Für $n = 0$ hat man die Produktion $[p, Z, q] \longrightarrow a$, welche dem Übergang $(p, a, Z, \varepsilon, q) \in \Delta$ entspricht.

Behauptung:

Für alle $p, q \in Q, u \in \Sigma^*, Z \in \Gamma$ gilt
$(\star)$ $\qquad (p, u, Z) \vdash^*_{\mathcal{A}} (q, \varepsilon, \varepsilon)$ gdw. $[p, Z, q] \vdash^*_G u$
Für $p = p_0$ und $Z = Z_0$ folgt daraus: $(q_0, u, Z_0) \vdash^* (q, \varepsilon, \varepsilon)$ gdw. $S \vdash_G [q_0, Z_0, q] \vdash^*_G u$, d.h. $u \in L(\mathcal{A})$ gdw. $u \in L(G)$.

Beweis der Behauptung durch Induktion über die Länge der Konfigurationsfolge („$\Rightarrow$") bzw. die Länge der Ableitung („$\Leftarrow$"). $\quad\square$

Beispiel: (vgl. Beispiel 10.3)

PDA für $\{a^n b^n \mid n \geq 1\}$.

$(q_0, ab, Z_0) \overset{(q_0,a,Z_0,ZZ_0,q_0)}{\vdash} (q_0, b, ZZ_0) \overset{(q_0,b,Z,\varepsilon,q_1)}{\vdash} (q_1, \varepsilon, Z_0) \overset{(q_1,\varepsilon,Z_0,\varepsilon,f)}{\vdash} (f, \varepsilon, \varepsilon)$
entspricht der Ableitung
$S \vdash_G [q_0, Z_0, f] \vdash_G a[q_0, Z, q_1][q_1, Z_0, f] \vdash_G ab[q_1, Z_0, f] \vdash_G ab.$

Teil III

Berechenbarkeit

Einführung

Aus der Sicht der Theorie der formalen Sprachen geht es in diesem Teil darum, Automatenmodelle für die Typ-0- und die Typ-1-Sprachen zu finden. Allgemeiner geht es aber darum, die intuitiven Begriffe „berechenbare Funktion“ und „entscheidbares Problem“ zu formalisieren.
Um für eine (eventuell partielle) Funktion $f : \mathbb{N}^k \to \mathbb{N}$ (bzw. $(\Sigma^*)^k \to \Sigma^*$) intuitiv klarzumachen, daß sie berechenbar ist, genügt es, ein Berechnungsverfahren (einen Algorithmus) für die Funktion anzugeben (z.B. in Form eines Modula-, Pascal- oder C-Programms oder einer abstrakten Beschreibung der Vorgehensweise bei der Berechnung). Entsprechend hatten wir in den Teilen I, II die Entscheidbarkeit von Problemen (Wortproblem, Leerheitsproblem, Äquivalenzproblem, ...) dadurch begründet, daß wir intuitiv beschrieben haben, wie man die Probleme mit Hilfe eines Rechenverfahrens (d.h. effektiv) entscheiden kann. Aus dieser Beschreibung hätte man jeweils ein Modula-, Pascal-, etc. -Programm zur Entscheidung des Problems gewinnen können.
Beim Nachweis der Nichtentscheidbarkeit bzw. Nichtberechenbarkeit ist eine solche intuitive Vorgehensweise nicht mehr möglich, da man hier formal nachweisen muß, daß es kein Berechnungsverfahren geben kann. Damit ein solcher Beweis durchführbar ist, benötigt man eine formale Definition dessen, was man unter einem Berechnungsverfahren versteht. Man will also ein Berechnungsmodell, das

1) einfach ist, damit formale Beweise erleichtert werden (z.B. nicht Programmiersprache ADA),
2) berechnungsuniversell ist, d.h. alle intuitiv berechenbaren Funktionen damit berechnet werden können (z.B. nicht nur endliche Automaten).

Wir werden in diesem Teil drei derartige Modelle betrachten:

- Turingmaschinen
- μ-rekursive Funktionen
- WHILE-Programme

Es gibt noch eine Vielzahl anderer Modelle:

- Minskymaschinen
- GOTO-Programme

- URMs (unbeschränkte Registermaschinen)
- Pascal-Programme
- ...

Es hat sich herausgestellt, daß all diese Modelle äquivalent sind, d.h. die gleiche Klasse von Funktionen berechnen. Außerdem ist es bisher nicht gelungen, ein formales Berechnungsmodell zu finden, so daß

- die dadurch berechneten Funktionen noch intuitiv berechenbar erscheinen,
- dadurch Funktionen berechnet werden, 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 Churchsche These.

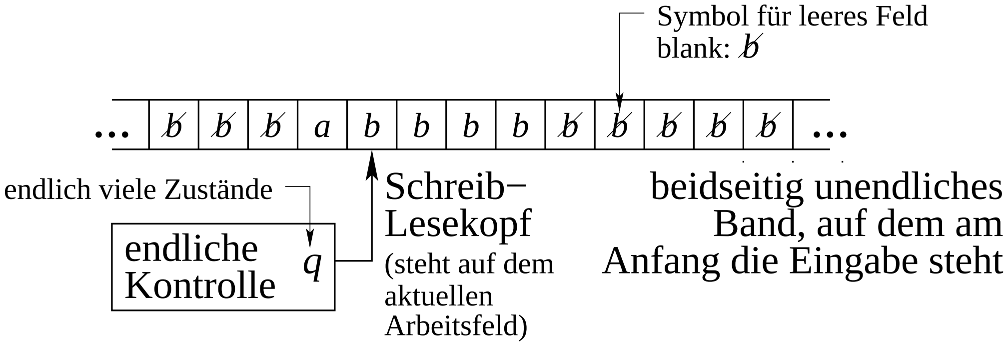
Churchsche These:

Die (intuitiv) berechenbaren Funktionen sind genau die mit Turingmaschinen (und damit mit GOTO-, WHILE-, Pascal-Programmen, URMs, ...) berechenbaren Funktionen.

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 sehr viele Indizien, die für die Richtigkeit der These sprechen (Vielzahl äquivalenter Berechnungsmodelle).
Im folgenden betrachten wir:

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

11 Turingmaschinen



Zu jedem Zeitpunkt sind nur endlich viele Symbole auf dem Band verschieden von $\not b$.

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* $\Sigma \subseteq \Gamma$, $\not{b} \in \Gamma \setminus \Sigma$,
- $q_0 \in Q$ *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 $\left(q, a, a', \begin{matrix} r \\ l \\ n \end{matrix}, q'\right)$:
Im Zustand q mit a auf dem Arbeitsfeld kann $\mathcal{A}$ 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 <u>deterministisch</u>, falls es für jedes Tupel $(q, a) \in Q \times \Gamma$ <u>höchstens ein</u> Tupel der Form $(q, a, \ldots, \ldots) \in \Delta$ gibt.
<u>NTM</u> steht im folgenden für (möglicherweise nichtdeterministische) Turingmaschinen und <u>DTM</u> für deterministische.
Einen Berechnungszustand (<u>Konfiguration</u>) einer Turingmaschine kann man beschreiben durch ein Wort $\alpha q \beta$ mit $\alpha, \beta \in \Gamma^+$, $q \in Q$:

- q ist der momentane Zustand
- α ist die Beschriftung des Bandes links vom Arbeitsfeld
- β ist die Beschriftung des Bandes beginnend beim Arbeitsfeld nach rechts

Dabei werden (um endliche Wörter α, β zu erhalten) unendlich viele Blanks weggelassen, d.h. α, β umfassen mindestens den Bandabschnitt, auf dem Symbole $\neq \not{b}$ stehen.

<u>Beispiel:</u>
Der Zustand der Maschine zu Beginn des Abschnitts wird durch $aqbbbb$ (aber auch durch $\not{b}\not{b}aqbbbb\not{b}$) beschrieben.

Die Übergangsrelation Δ ermöglicht die folgenden <u>Konfigurationsübergänge</u>:

- $\left.\begin{array}{rcl} \alpha q a \beta & \vdash_{\mathcal{A}} & \alpha a' q' \beta \\ \alpha q a & \vdash_{\mathcal{A}} & \alpha a' q' \not{b} \end{array}\right\}$ falls $(q, a, a', r, q') \in \Delta$
- $\left.\begin{array}{rcl} \alpha b q a \beta & \vdash_{\mathcal{A}} & \alpha q' b a' \beta \\ b q a \beta & \vdash_{\mathcal{A}} & \not{b} q' b a' \beta \end{array}\right\}$ falls $(q, a, a', l, q') \in \Delta$
- $\alpha q a \beta \vdash_{\mathcal{A}} \alpha q' a' \beta$ falls $(q, a, a', n, q') \in \Delta$

Gilt $k \vdash_{\mathcal{A}} k'$, so heißt k' <u>Folgekonfiguration</u> von k. Die Konfiguration $\alpha q \beta$ heißt <u>akzeptierend</u>, falls $q \in F$, und sie heißt <u>Stoppkonfiguration</u>, falls sie keine Folgekonfiguration hat. Die von $\mathcal{A}$ <u>akzeptierte Sprache</u> ist
$L(\mathcal{A}) = \{w \in \Sigma^* \mid \not{b} q_0 w \not{b} \vdash_{\mathcal{A}}^* k,\ \text{wobei } k \text{ akzeptierende Stoppkonfiguration ist}\}$.

Definition 11.2 (Turing-akzeptierbar, Turing-berechenbar)

1) Die Sprache $L \subseteq \Sigma^*$ heißt Turing-akzeptierbar, falls es eine NTM $\mathcal{A}$ gibt mit $L = L(\mathcal{A})$.

2) Die (partielle) Funktion $\mathrm{f} : (\Sigma^*)^n \to \Sigma^*$ heißt Turing-berechenbar, falls es eine DTM $\mathcal{A}$ gibt mit $\forall x_1, \ldots, x_n \in \Sigma^* \ : \ \not{b}\, q_0 x_1 \not{b} \ldots \not{b}\, x_n \not{b} \vdash_{\mathcal{A}}^* k$ Stoppkonfiguration gdw. $(x_1, \ldots, x_n) \in \mathrm{dom(f)}$ (Definitionsbereich von f).
Im Fall $(x_1, \ldots, x_n) \in \mathrm{dom(f)}$ muß die Beschriftung des Bandes in der Stoppkonfiguration rechts vom Schreib-Lesekopf bis zum ersten Symbol $\notin \Sigma$ der Wert $y = \mathrm{f}(x_1, \ldots, x_n)$ der Funktion sein, d.h. k muß die Form uqyv haben mit $v \in (\Gamma \setminus \Sigma) \cdot \Gamma^* \cup \{\varepsilon\}$.

Beachte:

1) Undefiniertheit des Funktionswertes von f entspricht der Tatsache, daß die Maschine bei dieser Eingabe nicht terminiert.

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).

Beispiel 11.3
Die Funktion $\mathrm{f} : \mathbb{N} \to \mathbb{N}, \quad n \mapsto 2n$ 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	*stop*	$2 \cdot 0 = 0$
q_0	a	b	r	q_1	*ersetze a durch b* $(\star)$
q_1	a	a	r	q_1	*laufe nach rechts über a's*
q_1	c	c	r	q_1	*und bereits geschriebene c's*
q_1	$\not{b}$	c	n	q_2	*schreibe weiteres c*
q_2	c	c	l	q_2	*laufe zurück über c's und*
q_2	a	a	l	q_2	*a's*
q_2	b	b	r	q_0	*bei erstem b eins nach rechts und weiter wie* $(\star)$ *oder*
q_0	c	c	r	q_3	*alle a's bereits ersetzt*
q_3	c	c	r	q_3	*laufe nach rechts bis Ende der c's*
q_3	$\not{b}$	$\not{b}$	l	q_4	*letztes c erreicht*
q_4	c	a	l	q_4	*ersetze c's und b's*
q_4	b	a	l	q_4	*durch a's*
q_4	$\not{b}$	$\not{b}$	r	*stop*	*bleibe am Anfang der erzeugten 2n a's stehen*

Man sieht hier, daß das Programmieren von Turingmaschinen sehr 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 11.4
Die Sprache $L = \{a^n b^n c^n \mid n \geq 0\}$ *ist Turing-akzeptierbar. Die Turingmaschine, welche* L *akzeptiert, geht wie folgt vor:*
Sie ersetzt das erste a *durch* a'*, das erste* b *durch* b' *und das erste* c *durch* c'*; läuft zurück zum zweiten* a*, ersetzt es durch* a'*, das zweite* b *durch* b' *und das zweite* c *durch* c' *etc.*
Dies wird solange gemacht, bis nach erzeugtem c' *ein* $\not b$ *steht. Danach wird von rechts nach links geprüft, ob – in dieser Reihenfolge – nur noch ein* c'*-Block, dann ein* b'*-Block und dann ein* a'*-Block (abgeschlossen durch* $\not b$*) vorhanden ist.*
Eine entsprechende Turingmaschine $\mathcal{A}$ *kann z.B. wie folgt definiert werden:*

$$\begin{aligned}\mathcal{A} = (&\{q_0, q_{akz}, finde\_b, finde\_c, zu\_Ende\_?, zu\_Ende\_!, zurück\},\\ &\{a, b, c\},\\ &\{a, a', b, b', c, c', \not b\},\\ &q_0, \Delta, \{q_{akz}\}) \text{ mit } \Delta :\end{aligned}$$

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}

<u>Varianten von Turingmaschinen:</u>
In der Literatur werden verschiedene Versionen der Definition der Turingmaschine angegeben, die aber alle äquivalent zueinander sind, d.h. dieselben Sprachen akzeptieren 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

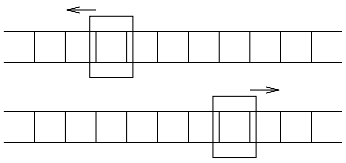
Wir betrachten das zweite Beispiel genauer und zeigen Äquivalenz zur in Definition 11.1 eingeführten 1-Band-TM.

Definition 11.5 (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* $\Delta \subseteq Q \times \Gamma^k \times \Gamma^k \times \{r, l, n\}^k \times Q$.

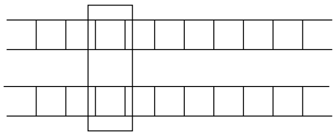
Dabei bedeutet $(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$:
Vom Zustand q aus mit $a_1, \ldots, a_k$ auf den Arbeitsfeldern der k Bänder kann $\mathcal{A}$ das Symbol a_i auf dem i-ten Band durch b_i ersetzen, in den Zustand q' gehen und die Schreib-Leseköpfe der 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 auch verschieden bewegen können:



Wären die Köpfe gekoppelt, so hätte man im Prinzip nicht mehrere Bänder, sondern ein Band mit mehreren Spuren:

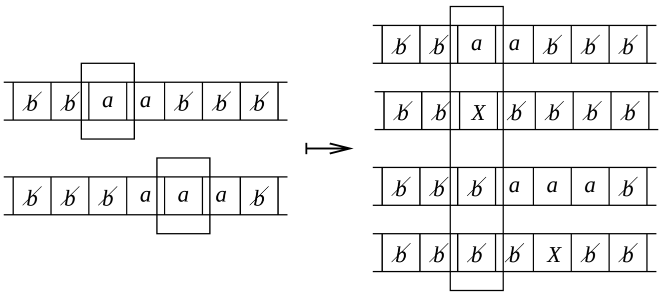


k Spuren erhält man einfach, indem man eine normale NTM (nach Definition 11.1) verwendet, die als Bandalphabet Γ^k statt Γ hat.
Offenbar kann man jede 1-Band-NTM (nach Definition 11.1) durch eine k-Band-NTM ($k > 1$) simulieren, indem man nur das erste Band wirklich verwendet. Der nächste Satz zeigt, daß auch die Umkehrung gilt:

Satz 11.6
Wird die Sprache L durch eine k-Band-NTM akzeptiert, so auch durch eine 1-Band-NTM.

Beweis:

Es sei $\mathcal{A}$ eine k-Band-NTM. Gesucht ist eine 1-Band-NTM $\mathcal{A}'$ und eine Kodierung $k \mapsto k'$ der Konfigurationen k von $\mathcal{A}$ durch Konfigurationen k' von $\mathcal{A}'$, so daß gilt: $k_1 \vdash_{\mathcal{A}} k_2$ gdw. $k_1' \vdash_{\mathcal{A}'}^* k_2'$ (mehr Schritte nötig zur Simulation eines Schrittes von $\mathcal{A}$).
Arbeitsalphabet von $\mathcal{A}'$: $\Gamma^{2k} \cup \Sigma \cup \{\not b\}$;
$\Sigma \cup \{\not b\}$ wird für Eingabe benötigt; Γ^{2k} sorgt dafür, daß sich $\mathcal{A}'$ wie eine 1-Band-NTM mit $2k$ Spuren verhält.
Kodierung: 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.



Hierbei wird angenommen, daß das Arbeitsfeld von $\mathcal{A}'$ stets bei dem am weitesten links stehenden X liegt.
Initialisierung: Zunächst wird die Anfangskonfiguration $\not{b}\, q_0 a_1 \ldots a_m \not{b}$ von $\mathcal{A}'$ in die Kodierung der entsprechenden Anfangskonfiguration von $\mathcal{A}$ umgewandelt (durch entsprechende Übergänge):

$\not{b}$	$\not{b}$	a_1	a_2	$\ldots$	a_m	$\not{b}$	Spur 1 und 2 kodieren
$\not{b}$	$\not{b}$	X	$\not{b}$	$\ldots$	$\not{b}$	$\not{b}$	Band 1
$\not{b}$	$\not{b}$	$\not{b}$	$\not{b}$	$\ldots$	$\not{b}$	$\not{b}$	Spur 3 und 4 kodieren
$\not{b}$	$\not{b}$	X	$\not{b}$	$\ldots$	$\not{b}$	$\not{b}$	Band 2

$\vdots$

Simulation der Übergänge: Betrachte $(q, (a_1, \ldots, a_k), (b_1, \ldots, b_k), (d_1, \ldots, d_k), q') \in \Delta$.

- Von links nach rechts suche die mit X markierten Felder. Dabei merke man sich (in dem Zustand der TM) die Symbole, welche jeweils über dem X stehen. Außerdem zählt man (durch Zustand der TM) mit, wieviele X man schon gelesen hat, um festzustellen, wann das k-te erreicht ist. Man bestimmt so, ob das aktuelle Tupel tatsächlich $(a_1, \ldots, a_k)$ ist.
- Von rechts nach links gehend überdrucke man die a_i bei den X-Marken jeweils durch das entsprechende b_i und verschiebt die X-Marken gemäß d_i.
- Bleibe bei der am weitesten links stehenden X-Markierung, lasse den Kopf dort stehen und gehe in Zustand q'.

Bemerkung 11.7

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.*

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-akzeptierbaren Sprachen machen kann, ohne an Ausdrucksstärke zu verlieren.

Satz 11.8
Zu jeder NTM gibt es eine DTM, die dieselbe Sprache akzeptiert.

Beweis:

Wegen Satz 11.6 und Bemerkung 11.7 genügt es, eine deterministische 3-Band-Turingmaschine zu konstruieren. Es sei $\mathcal{A} = (Q, \Sigma, \Gamma, q_0, \Delta, F)$ eine NTM. Die Maschine $\mathcal{A}'$ soll für wachsendes n auf dem dritten Band jeweils alle Konfigurationsfolgen $k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} k_2 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_n$ beginnend mit der Startkonfiguration $k_0 = \not{b}\, q_0 w \not{b}$ erzeugen. 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 = maximale Anzahl von Transitionen in Δ pro festem Paar $(q, a) \in Q \times \Gamma$ (entspricht dem maximalen Verzweigungsgrad der nichtdeterministischen Berechnung). Eine Indexfolge $i_1, \ldots, i_n$ mit $i_j \in \{1, \ldots, r\}$ bestimmt dann von k_0 aus für n Schritte die Auswahl

der jeweiligen Transition (und somit von k_0 aus eine feste Konfigurationsfolge $k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_n$). Zählt man daher alle endlichen Wörter über $\{1, \ldots, r\}$ auf und zu jedem Wort $i_1 \ldots i_n$ die zugehörige Konfigurationsfolge, so erhält man eine Aufzählung aller endlichen Konfigurationsfolgen. $\mathcal{A}'$ realisiert dies auf den drei Bändern wie folgt:

- Auf Band 1 bleibt die Eingabe gespeichert.
- Auf dem zweiten Band werden sukzessive alle Wörter $i_1 \ldots i_n \in \{1, \ldots, r\}^*$ erzeugt.
- Für jedes dieser Wörter wird auf dem dritten Band die zugehörige Konfigurationsfolge realisiert. Erreicht man hierbei eine akzeptierende Konfiguration von $\mathcal{A}$, so geht auch $\mathcal{A}'$ in eine akzeptierende Konfiguration. $\square$

12 Zusammenhang zwischen Turingmaschinen und Grammatiken

Wir zeigen zunächst den Zusammenhang zwischen Typ-0-Sprachen und Turing-akzeptierbaren Sprachen. Dieser beruht darauf, daß es eine Entsprechung von Ableitungen einer Typ-0-Grammatik einerseits und Konfigurationsfolgen von Turingmaschinen andererseits gibt. Beim Übergang von der Turingmaschine zur Grammatik dreht sich allerdings die Richtung um:

- eine akzeptierende Konfigurationsfolge beginnt mit dem zu akzeptierenden Wort
- eine Ableitung endet mit dem erzeugten Wort

Satz 12.1
Eine Sprache L gehört zu $\mathcal{L}_0$ gdw. sie Turing-akzeptierbar ist.

Beweis:

„$\Rightarrow$“: Es sei $L = L(G)$ für eine Typ-0-Grammatik $G = (N, \Sigma, P, S)$ mit $|P| = k$ Regeln. Wir geben eine 2-Band-NTM an, die $L(G)$ akzeptiert (die nach Satz 11.6 äquivalent zu einer 1-Band-NTM ist).

1. Band: speichert Eingabe w

2. Band: ausgehend von S werden gemäß den Regeln von G ableitbare Wörter gebildet

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 sucht die Turingmaschine weiter. 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) Wechsle nun (nichtdeterministisch) in einen der Zustände Regel$_1$, …, Regel$_k$ (entspricht der Entscheidung, daß man diese Regel anwenden will).
4) Es sei Regel$_i$ der gewählte Zustand und $\alpha \longrightarrow \beta$ die entsprechende Produktion. 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 $\neq \not b$ 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 Inschrift 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": 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 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 man sich in der ersten Komponente w merkt und in der zweiten Komponente die Berechnung simuliert.

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, $\Sigma \cup \{\not b\}$ zum Speichern von w und Γ zur $\mathcal{A}$-Berechnung dienen.

Regeln:

1. Phase: $S \longrightarrow Bq_0A$, $A \longrightarrow [a, a]A$ für alle $a \in \Sigma$, $A \longrightarrow B$, $B \longrightarrow [\not b, \not b]B$, $B \longrightarrow \varepsilon$
Man erhält also somit für alle $a_1 \ldots a_n \in \Sigma^*, k, l, \geq 0$:
$S \vdash_G^* [\not b, \not b]^k q_0[a_1, a_1] \ldots [a_n, a_n][\not b, \not b]^l$

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 $a_1 \ldots a_n$ erzeugen können, muß in der zweiten Phase das „Nachschieben" von Blanks am Rand nicht mehr behandelt werden.

3. Phase: Aufräumen und erzeugen von $a_1 \ldots a_n$, wenn die TM akzeptiert hat

- $\left.\begin{array}{lll} q[a, b] \longrightarrow EaE & \text{für} & a \in \Sigma, b \in \Gamma \\ q[\not b, b] \longrightarrow E & \text{für} & b \in \Gamma \end{array}\right\}$ falls $q \in F$ und es keine Transition der Form $(q, b, \ldots, \ldots) \in \Delta$ 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:
$w \in L(G)$ gdw. $\mathcal{A}$ akzeptiert w. $\square$

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 $\begin{array}{rcll} u & \longrightarrow & v & \text{mit } u \in N_i^+ \text{ und } v \in N_i^* \\ X_a & \longrightarrow & a & \text{mit } X_a \in N_i \text{ und } a \in \Sigma \end{array}$
Für den Durchschnitt 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-akzeptierbare Sprachen nicht unter Komplement abgeschlossen sind (Satz 16.10). $\square$

Wir werden später außerdem zeigen, daß für Turing-akzeptierbare Sprachen (und damit für $\mathcal{L}_0$) alle bisher betrachteten Entscheidungsprobleme unentscheidbar sind (Sätze 16.6, 16.8, 16.9):

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. Man kann mit der Beweisidee von Satz 12.1 auch eine Charakterisierung kontextsensitiver Sprachen durch spezielle Turingmaschinen erhalten. Diese Maschinen dürfen nur auf dem Bandabschnitt arbeiten, auf dem anfangs die Eingabe stand. Um ein Überschreiten der dadurch gegebenen Bandgrenzen zu verhindern, verwendet man Randmarker $\not c$, \$.

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

- $\$, \not{c} \in \Gamma \setminus \Sigma$
- *Übergänge $(q, \not{c}, \ldots)$ sind nur in der Form $(q, \not{c}, \not{c}, r, q')$ erlaubt (linker Rand darf nicht überschritten werden).*
- *Übergänge $(q, \$, \ldots)$ sind nur in der Form $(q, \$, \$, l, q')$ erlaubt.*
- *$\not{c}$ und $\$$ dürfen nicht geschrieben werden.*

Ein gegebener LBA $\mathcal{A}$ akzeptiert die Sprache
$L(\mathcal{A}) := \{w \in \Sigma^* \mid \not{c} q_0 w \$ \vdash^* k,$ wobei k akzeptierende Stoppkonfiguration ist$\}$.

Korollar 12.5
Eine Sprache L gehört zu $\mathcal{L}_1$ gdw. sie von einem LBA akzeptiert wird.

Beweis:

„⇒“: 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 (längere können nie mehr zu w abgeleitet werden). Daher kommt man mit $|w|$ vielen Feldern aus. Beachte: Zwei Bänder liefern nicht doppelt so langes Band, sondern größeres Alphabet.

„⇐“: Durch Modifikation der Beweisidee von Satz 12.1 gelingt es, zu einem LBA eine Grammatik zu konstruieren, die keine kürzenden Regeln ($\alpha \longrightarrow \beta$ mit $|\alpha| > |\beta|$) hat. Wie im Beweis von Satz 9.4 illustriert, kann man diese in eine kontextsensitive Grammatik transformieren.
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 kürzenden Regeln $E[\not{b}, \not{b}] \longrightarrow E$, $[\not{b}, \not{b}]E \longrightarrow E$ weg. Es gibt allerdings noch einige technische Probleme:

- Man muß die Randmarker $\not{c}$ und $\$$ einführen und am Schluß löschen.
- Man muß das Hilfssymbol E und den Zustand q löschen.

Lösung: Führe diese nicht als zusätzliche Symbole ein, sondern kodiere sie in die anderen Symbole hinein (z.B. statt $[a, b]q[a', b'][a'', b'']$ verwende $[a, b][q, a', b'][a'', b'']$). □

Satz 12.6
$\mathcal{L}_1$ ist abgeschlossen unter $\cup, \cdot, ^*, \cap$ und Komplement.

Beweis:

Für $\cup, \cdot, ^*$ und $\cap$ geht dies wie bei $\mathcal{L}_0$. Komplement: schwierig, war lange offen und wurde dann in den 1980ern unabhängig von zwei Forschern gezeigt (Immerman und Szelepcsenyi). □

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

Satz 12.7
Für $\mathcal{L}_1$ ist das Wortproblem entscheidbar.

Beweis:

Da kontextsensitive Produktionen (bis auf Spezialfall $S \longrightarrow \varepsilon$) nichtkürzend sind, muß man zur Entscheidung „$w \in L(G)$?“ nur alle aus S ableitbaren Wörter aus $(N \cup \Sigma)^*$ der Länge $\leq |w|$ erzeugen. Dies sind endlich viele. □

13 Berechenbarkeit, Entscheidbarkeit, Aufzählbarkeit

Wir werden hier – gemäß der These von Church – die Begriffe Turing-berechenbar und (intuitiv) berechenbar als synonym verwenden. Wegen der anfangs erwähnten Äquivalenz von Berechnungsmodellen treffen alle Definitionen und Resultate auch auf die anderen Modelle zu. Wir werden meist intuitiv die Existenz eines Berechnungsverfahrens (einer DTM) begründen. Diese intuitiven Argumente können aber leicht (wenn auch im Detail technisch und zeitaufwendig) in TM-Konstruktionen übersetzt werden.

Definition 13.1 (Berechenbarkeit)
Wir betrachten partielle oder totale Funktionen $\mathrm{f} : (\Sigma^*)^n \to \Sigma^*$. f *heißt berechenbar (partiell rekursiv), falls sie Turing-berechenbar ist. Ist* f *total (d.h.* $\mathrm{dom(f)} = (\Sigma^*)^n$*), so heißt* f *rekursiv.*

Bei partiellen Funktionen entspricht Undefiniertheit des Funktionswertes der Tatsache, daß das Berechnungsverfahren (die DTM) bei dieser Eingabe nicht terminiert.

Definition 13.2 (entscheidbar, partiell entscheidbar, rekursiv aufzählbar)
Wir betrachten n-stellige Relationen $R \subseteq (\Sigma^*)^n$.

1) *R heißt entscheidbar (rekursiv), falls ihre charakteristische Funktion* $\chi_R : (\Sigma^*)^n \to \Sigma^*$ *mit* $(x_1, \dots, x_n) \mapsto \begin{cases} a & \text{falls } (x_1, \dots, x_n) \in R \\ \varepsilon & \text{sonst} \end{cases}$ *berechenbar ist. Dabei ist a ein Element von* Σ *(beliebig aber fest gewählt).*

2) *R heißt partiell entscheidbar (partiell rekursiv), falls R Definitionsbereich einer berechenbaren Funktion* f *ist, d.h. es gibt eine DTM, die bei Eingabe* $(x_1, \dots, x_n) \in (\Sigma^*)^n$ *terminiert, falls* $(x_1, \dots, x_n) \in R$ *ist, und nicht terminiert sonst.*

3) *R heißt rekursiv aufzählbar, falls R 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 für eine n-stellige Relation ist von der Form*
$uq_{Ausgabe}\, x_1 \not{b}\, x_2 \not{b} \dots x_n \not{b} v$ *mit* $u, v \in \Gamma^*$ *und* $x_1, \dots, x_n \in \Sigma^*$.
Diese Konfiguration hat $(x_1, \dots, x_n)$ *als Ausgabe. Die durch* $\mathcal{A}$ *aufgezählte Relation ist*

$$R = \{(x_1, \dots, x_n) \in (\Sigma^*)^n \mid (x_1, \dots, x_n) \textit{ ist Ausgabe einer Ausgabekonfiguration, die von } \mathcal{A} \textit{ ausgehend von } \not{b}\, q_0 \not{b} \textit{ erreicht wird}\}.$$

Bemerkung 13.3

1) *Probleme wie z.B. das Wortproblem für kontextfreie Sprachen können als Relationen aufgefaßt werden: Es sei* $G = (N, \Sigma, P, S)$ *eine kontextfreie Grammatik und* $w \in \Sigma^*$. *Die Grammatik G kann als Wort* $\mathrm{code}(G) \in \Gamma^*$ *über einem erweiterten Alphabet* Γ *aufgefaßt werden. Das Wortproblem für G entspricht der Relation* $R = \{(\mathrm{code}(G), w) \mid w \in L(G)\} \subseteq (\Gamma^*)^2$.

2) *Für* $n = 1$ *stimmen die Begriffe „partiell entscheidbar" und „Turing-akzeptierbar" überein,* <u>*denn:*</u> *Bei Turing-akzeptierbar war neben dem Anhalten der (o.B.d.A. deterministischen) TM zusätzlich gefordert, daß man einen akzeptierenden Zustand erreicht hat. Man kann aber einfach von allen nichtakzeptierenden Stoppkonfigurationen aus in eine Endlosschleife gehen.*

3) Äquivalente Charakterisierungen für rekursiv aufzählbar im Fall $n = 1$ sind:

- *R ist Wertebereich einer partiell rekursiven Funktion.*
- *$R = \emptyset$ oder R ist Wertebereich einer rekursiven Funktion.*

Satz 13.4
Es sei $R \subseteq (\Sigma^*)^n$ eine Relation.

1) R ist rekursiv aufzählbar gdw. R ist partiell entscheidbar.

2) Ist R entscheidbar, so auch partiell entscheidbar.

3) Ist R entscheidbar, so auch $\overline{R} = (\Sigma^*)^n \setminus R$.

4) R ist entscheidbar gdw. R und $\overline{R}$ partiell entscheidbar sind.

Beweis:

1) **„$\Rightarrow$“:** Es sei $\mathcal{A}$ eine Aufzähl-DTM für R. Die Maschine $\mathcal{A}'$ arbeitet wie folgt:

- Sie speichert die Eingabekonfiguration $\not{b}\, q_0 \not{b}\, x_1 \not{b}\, x_2 \ldots \not{b}\, x_n \not{b}$ ab (z.B. auf zusätzlichem Band).
- Sie beginnt (auf anderem Band) mit der Aufzählung von R.
- Bei jeder Ausgabekonfiguration überprüft sie, ob die entsprechende Ausgabe mit der gespeicherten Eingabe übereinstimmt. Wenn ja, so terminiert $\mathcal{A}'$. Sonst sucht sie die nächste Ausgabekonfiguration von $\mathcal{A}$.
- Terminiert $\mathcal{A}$, ohne daß $(x_1, \ldots, x_n)$ ausgegeben wurde, so gehe in Endlosschleife. $\mathcal{A}'$ terminiert daher genau dann nicht, wenn $(x_1, \ldots, x_n)$ nicht in der Aufzählung vorkommt.

„$\Leftarrow$“: Es sei $\mathcal{A}$ ein partielles Entscheidungsverfahren für R, d.h. $\mathcal{A}$ berechnet f mit dom(f) $= R$. Es sei $\vec{x}^{(1)}, \vec{x}^{(2)}, \vec{x}^{(3)}, \ldots$ eine Auflistung von $(\Sigma^*)^n$. Die Maschine $\mathcal{A}'$ arbeitet wie folgt:

1) Simuliere die Berechnung von $\mathcal{A}$ mit Eingabe $\vec{x}^{(1)}$ <u>einen</u> Schritt.
2) Simuliere die Berechnung von $\mathcal{A}$ mit Eingabe $\vec{x}^{(1)}$ <u>zwei</u> Schritte und mit Eingabe $\vec{x}^{(2)}$ <u>zwei</u> Schritte.

   ⋮

n) Simuliere die Berechnung von $\mathcal{A}$ mit Eingaben $\vec{x}^{(1)}, \ldots, \vec{x}^{(n)}$ jeweils <u>n Schritte</u>.

   ⋮

Terminiert $\mathcal{A}$ für eine dieser Eingaben, so gebe diese Eingabe aus und mache weiter.
<u>Beachte:</u> Man kann nicht $\mathcal{A}$ zunächst ganz mit Eingabe $\vec{x}^{(1)}$ laufen lassen (ohne Schrittbeschränkung), da $\mathcal{A}$ darauf nicht terminieren muß. Das gewählte Vorgehen nennt man <u>dove-tailing</u> (Ineinander-Verzahnen mehrerer Berechnungen).

2) Eine DTM $\mathcal{A}$, die χ_R berechnet, wird wie folgt modifiziert: Ist die Ausgabe ε (d.h. in der Stoppkonfiguration von $\mathcal{A}$ steht der Kopf auf $\not{b}$), so gehe in Endlosschleife. Sonst gehe in Stoppkonfiguration.

3) Eine DTM $\mathcal{A}$, die χ_R berechnet, wird wie folgt zu einer DTM für $\chi_{\overline{R}}$ modifiziert: Ist die Ausgabe von $\mathcal{A}$ ε, so erzeuge Ausgabe a. Sonst erzeuge Ausgabe ε.

4) „$\Rightarrow$“ ergibt sich aus 2) und 3).

„⇐“: Sind R und $\overline{R}$ partiell entscheidbar, so mit 1) auch rekursiv aufzählbar. Für Eingabe $\vec{x}$ läßt man die Aufzähl-DTMs $\mathcal{A}$ und $\mathcal{B}$ für R und $\overline{R}$ parallel laufen (d.h. jeweils abwechselnd ein Schritt von $\mathcal{A}$ auf einem Band gefolgt von einem Schritt von $\mathcal{B}$ auf dem anderen). Die Eingabe $\vec{x}$ kommt in einer der beiden Aufzählungen vor ($\vec{x} \in (\Sigma^*)^n = R \cup \overline{R}$). Kommt $\vec{x}$ bei $\mathcal{A}$ vor, so erzeuge Ausgabe a, sonst Ausgabe ε. □

Für $\mathrm{f} : (\Sigma^*)^n \to \Sigma^*$ definieren wir:
$\mathrm{Graph(f)} := \{(x_1, \ldots, x_n, x_{n+1}) \mid (x_1, \ldots, x_n) \in \mathrm{dom(f)} \text{ und } \mathrm{f}(x_1, \ldots, x_n) = x_{n+1}\}$

Satz 13.5
f *ist berechenbar gdw.* Graph(f) *ist rekursiv aufzählbar.*

Beweis:

„⇐“: Sei Graph(f) rekursiv aufzählbar. Um den Funktionswert $\mathrm{f}(x_1, \ldots, x_n)$ zu berechnen, startet man das Aufzählungsverfahren und wartet, bis ein Tupel der Form $(x_1, \ldots, x_n, y)$ auftaucht. Wenn ja, so ist y der Funktionswert. Sonst ist $(x_1, \ldots, x_n) \notin \mathrm{dom(f)}$, und die Maschine terminiert nicht (zählt unendlich auf oder geht in Endlosschleife, wenn Aufzählung abgebrochen wird, ohne daß Tupel der Form $(x_1, \ldots, x_n, \cdot)$ gefunden wurde).

„⇒“: Mit Satz 13.4 genügt es zu zeigen, daß Graph(f) partiell entscheidbar ist. Das partielle Entscheidungsverfahren für Graph(f) erhält man wie folgt: Berechne bei Eingabe $(x_1, \ldots, x_n, x_{n+1})$ den Funktionswert von f an der Stelle $(x_1, \ldots, x_n)$.

1. Fall: $\mathrm{f}(x_1, \ldots, x_n)$ ist definiert und $= x_{n+1}$. Dann terminiere.

2. Fall: $\mathrm{f}(x_1, \ldots, x_n)$ ist definiert und $\neq x_{n+1}$. Gehe in Endlosschleife.

3. Fall: $\mathrm{f}(x_1, \ldots, x_n)$ ist undefiniert. Dann terminiert die Berechnung des Wertes bereits nicht. □

14 Primitiv rekursive Funktionen und Loop-Programme

In den nächsten beiden Abschnitten betrachten wir Berechnungsmodelle, die eng verwandt sind mit modernen imperativen und funktionalen Programmiersprachen. Die in diesem Abschnitt eingeführten Modelle sind nicht stark genug, um alle (Turing-)berechenbaren Funktionen zu erfassen. Wir werden im Abschnitt 15 zeigen, wie man diese Modelle erweitern muß, um berechnungsuniverselle 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. $\pi : \Sigma^* \to \mathbb{N}$ bijektiv mit π und π^{-1} berechenbar.

Definition 14.1 (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)*

Aus diesen einfachen Funktionen kann man mit Hilfe von Operatoren komplexere Funktionen aufbauen.

Definition 14.2 (Komposition)
Die Funktion $\mathrm{f} : \mathbb{N}^n \to \mathbb{N}$ *entsteht aus* $\mathrm{g} : \mathbb{N}^m \to \mathbb{N}$ *und* $\mathrm{h}_1, \ldots, \mathrm{h}_m : \mathbb{N}^n \to \mathbb{N}$ *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 $\mathrm{h}_i(x_1, \ldots, x_n)$ undefiniert ist oder alle h_i-Werte definiert sind, aber g von diesen Werten undefiniert ist.

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 (entspricht call-by-value-Auswertung).

Definition 14.3 (primitive Rekursion)
Es sei $n \geq 0$. *Die Funktion* $\mathrm{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ *entsteht aus* $\mathrm{g} : \mathbb{N}^n \to \mathbb{N}$ *und* $\mathrm{h} : \mathbb{N}^{n+2} \to \mathbb{N}$ *durch primitive Rekursion, falls gilt:*
$\mathrm{f}(x_1, \ldots, x_n, 0) = \mathrm{g}(x_1, \ldots, x_n)$
$\mathrm{f}(x_1, \ldots, x_n, y + 1) = \mathrm{h}(x_1, \ldots, x_n, \mathrm{f}(x_1, \ldots, x_n, y), y)$

Auch hier setzen sich wie bei der Komposition wieder undefinierte Werte fort. Durch Induktion über die letzte Komponente zeigt man leicht, daß dieses Schema die Funktion f eindeutig definiert.

Beispiel 14.4 (Addition)
Die Addition natürlicher Zahlen kann durch primitive Rekursion wie folgt definiert werden:

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

Das heißt also: add *entsteht durch primitive Rekursion aus den Funktionen*
$\mathrm{g} : \mathbb{N} \to \mathbb{N}$ *mit* $x \mapsto x$, *d.h.* $\mathrm{g} = \pi_1^{(1)}$ *ist Grundfunktion,*
$\mathrm{h} : \mathbb{N}^3 \to \mathbb{N}$ *mit* $(x, z, y) \mapsto z + 1$, *d.h.* $\mathrm{h}(x, z, y) = \mathrm{s}(\pi_2^{(3)}(x, z, y))$.
Also entsteht h *durch Komposition aus Grundfunktionen.*

Definition 14.5 (Klasse der primitiv rekursiven Funktionen)
Die Klasse der primitiv rekursiven Funktionen besteht aus allen Funktionen, die man aus den Grundfunktionen durch endlich oftes 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 14.6
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 14.4 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:
$\mathrm{mult}(x, 0) = 0$,

$\mathrm{mult}(x, y+1) = \mathrm{add}(x, \mathrm{mult}(x, y))$

Exponentiation erhält man durch primitive Rekursion aus der Multiplikation:
$\exp(x, 0) = 1 = \mathrm{s}(\mathrm{null}^{(1)}(x))$,
$\exp(x, y+1) = \mathrm{mult}(x, \exp(x, y))$

Die Funktion min1 : $\mathbb{N} \to \mathbb{N}$ mit $x \mapsto x \dot{-} 1 := \begin{cases} x-1 & x > 0 \\ 0 & x = 0 \end{cases}$ ist ebenfalls primitiv rekursiv:
$\mathrm{min1}(0) = 0 = \mathrm{null}^{(0)}()$,
$\mathrm{min1}(y+1) = y = \pi_2^{(2)}(\mathrm{min1}(y), y)$

Übung: Zeige, daß min : $\mathbb{N}^2 \to \mathbb{N}$ mit $(x, y) \mapsto x \dot{-} y := \begin{cases} x-y & x \geq y \\ 0 & \text{sonst} \end{cases}$ primitiv rekursiv ist.

Beispiel 14.7
Aus den bisher betrachteten Funktionen erhält man damit die Funktion c : $\mathbb{N}^2 \to \mathbb{N}$ *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.*

Lemma 14.8
Die Funktion c *aus Beispiel 14.7 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 $\mathrm{c}_1(\mathrm{c}(x, y)) = y$
- $\mathrm{c}(\mathrm{c}_0(z), \mathrm{c}_1(z)) = z$

Diese ergeben sich im Prinzip aus dem Beweis von Lemma 14.8, d.h.

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

Um zu zeigen, daß c_0, c_1 ebenfalls primitiv rekursiv sind, benötigen wir noch etwas Vorarbeit.

Die Funktionen sign : $\mathbb{N} \to \mathbb{N}$ und $\overline{\mathrm{sign}} : \mathbb{N} \to \mathbb{N}$ mit
$\mathrm{sign}(x) = \begin{cases} 0 & x = 0 \\ 1 & x > 0 \end{cases}$ $\qquad \overline{\mathrm{sign}}(x) = \begin{cases} 1 & x = 0 \\ 0 & x > 0 \end{cases}$
sind primitiv rekursiv:
$\mathrm{sign}(0) = 0, \qquad \mathrm{sign}(y+1) = 1$
$\overline{\mathrm{sign}}(0) = 1, \qquad \overline{\mathrm{sign}}(1) = 0$
sind primitiv rekursive Definitionsschemata dafür.

Definition 14.10 (Fallunterscheidung)
Es seien $\mathrm{g}_1, \mathrm{g}_2, \mathrm{h} : \mathbb{N}^n \to \mathbb{N}$ *gegeben. Die Funktion* $\mathrm{f} : \mathbb{N}^n \to \mathbb{N}$ *entsteht daraus durch Fallunterscheidung, falls für alle* $\underline{x} \in \mathbb{N}^n$ *gilt:*

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

Lemma 14.11
Sind $\mathrm{g}_1, \mathrm{g}_2, \mathrm{h}$ *primitiv rekursiv, so auch* f.

Beweis:
$\mathrm{f}(\underline{x}) = \mathrm{g}_1(\underline{x}) \cdot \overline{\mathrm{sign}}(\mathrm{h}(\underline{x})) + \mathrm{g}_2(\underline{x}) \cdot \mathrm{sign}(\mathrm{h}(\underline{x}))$. □

Definition 14.12 (beschränkte Minimalisierung)
Es sei $n \geq 0$*. Die Funktion* $\mathrm{f} : \mathbb{N}^{n+1} \to \mathbb{N}$ *entsteht aus* $\mathrm{g} : \mathbb{N}^{n+1} \to \mathbb{N}$ *durch beschränkte Minimalisierung, falls gilt:*

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

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

Lemma 14.13
Ist g *primitiv rekursiv, so auch* $\overline{\mu}\mathrm{g}$.

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

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

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

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

Es handelt sich hier also um eine Fallunterscheidung. Es bleibt zu zeigen, daß die Funktion $\mathrm{h}(\underline{x}, z, y) = \begin{cases} 0 & \text{falls} \quad z \leq y \text{ oder } \mathrm{g}(\underline{x}, y+1) = 0 \\ 1 & \text{sonst} \end{cases}$ primitiv rekursiv ist.
Offenbar ist aber $\mathrm{h}(\underline{x}, z, y) = \mathrm{sign}((z \dot{-} y) \cdot \mathrm{g}(\underline{x}, y+1))$ und damit primitiv rekursiv. □

Wir kommen nun zurück zu den Umkehrfunktionen der Bijektion aus Beispiel 14.7. Betrachten wir zunächst die Teilbarkeitsrelation in der Definition von c_0.

Lemma 14.14
Die Funktion $\mathrm{teilt} : \mathbb{N}^2 \to \mathbb{N}$ *mit* $(x, y) \mapsto \begin{cases} 0 & x|y \\ 1 & \textit{sonst} \end{cases}$ *ist primitiv rekursiv.*

Beweis:
Die Funktion $\mathrm{h}(x, z, y) = \begin{cases} 0 & x \cdot z = y \\ 1 & \text{sonst} \end{cases}$ 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 es gilt:

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

Damit ist aber $\mathrm{teilt}(x, y) = \overline{\mu}\mathrm{h}(x, y, y) \dot{-} y$. □

Lemma 14.15
Die Umkehrfunktion c_0 *der Funktion* c *aus Beispiel 14.7 ist primitiv rekursiv.*

Beweis:
$\mathrm{c}_0(z) = \max\{x \in \mathbb{N} \mid 2^x | (z+1)\}$. Es genügt dafür, nach dem kleinsten $i \leq z+1$ zu suchen mit $2^{(z+1) \dot{-} i} | (z+1)$, d.h. nach dem kleinsten $i \leq z+1$ mit $\mathrm{teilt}(2^{(z+1) \dot{-} i}, z+1) = 0$. Dies gelingt durch beschränkte Minimalisierung. □

Um zu zeigen, daß auch c_1 primitiv rekursiv ist, betrachten wir die ganzzahlige Division $\mathrm{div} : \mathbb{N}^2 \to \mathbb{N}$ 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 14.16
Die Umkehrfunktion c_1 *der Funktion* c *aus Beispiel 14.7 ist primitiv rekursiv.*

Beweis:
Es genügt zu zeigen, daß div primitiv rekursiv ist. Wir betrachten dazu die Funktion $\mathrm{f}(x, y, z) = \begin{cases} 0 & \text{falls} \quad z \cdot y > x \\ 1 & \text{falls} \quad z \cdot y \leq x \end{cases}$. 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 (unter Verwendung geeigneter Projektionen) auch $\mathrm{g}(x, y) := \overline{\mu}\mathrm{f}(x, y, x)$. Es gilt nun aber

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

Ist $y > 1$, so existiert so ein i stets 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 $\mathrm{div}(x, y) = x$. Also gilt: $\mathrm{div}(x, y) = \mathrm{g}(x, y) \dot{-} 1$. □

Wir betrachten nun eine einfache imperative Programmiersprache, die genau die primitiv rekursiven Funktionen berechnen kann.
LOOP-Programme sind aus den folgenden Komponenten aufgebaut:

- Variablen: $x_0, x_1, x_2, \ldots$
- Konstanten: 0, 1, 2, ... (also die Elemente von $\mathbb{N}$)
- Trennsymbole: `;` und `:=`
- Operationssymbole: $+$ und $\dot{-}$
- Schlüsselwörter: `LOOP`, `DO`, `END`

Definition 14.17 (Syntax LOOP)
Die Syntax von LOOP-Programmen ist induktiv definiert:

1) *Jede Wertzuweisung* x_i `:=` $x_j + c$ *und* x_i `:=` $x_j \dot{-} c$ *für* $i, j \geq 0$ *und* $c \in \mathbb{N}$ *ist ein LOOP-Programm.*

2) *Falls* P_1 *und* P_2 *LOOP-Programme sind, so ist auch* P_1`;`P_2 *(Hintereinanderausführung) ein LOOP-Programm.*

3) *Falls* P *ein LOOP-Programm ist und* $i \geq 0$, *so ist auch* `LOOP` x_i `DO` P `END` *ein LOOP-Programm.*

Die Semantik dieser einfachen Sprache ist wie folgt definiert:
Bei einem LOOP-Programm, das eine Funktion $\mathrm{f} : \mathbb{N}^k \to \mathbb{N}$ 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.

Definition 14.18 (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.

Bemerkung:
Offenbar 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.

Beispiel 14.19
Die Additionsfunktion ist LOOP-berechenbar:

```
x0 := x1 + 0;
LOOP x2 DO x0 := x0 + 1 END
```

Mit den Programmkonstrukten von Definition 14.17 kann man auch andere in Programmiersprachen vorhandene Konstrukte simulieren:

```
IF x = 0 THEN P END
```

kann z.B. 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 und $\neq x_0$ ist.

Satz 14.20
Die Klasse der primitiv rekursiven Funktionen stimmt mit der Klasse der LOOP-berechenbaren Funktionen überein.

Beweis:

(I) Alle primitiv rekursiven Funktionen sind LOOP-berechenbar:

- 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.
- 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

```
z1 := g(x1,...,xn);   (⋆)
z2 := 0;
z3 := x(n+1) ∸ 1;
LOOP z3 DO
  z1 := h(x1,...,xn,z1,z2);   (⋆)
  z2 := z2 + 1
END
```

  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, z_3 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 $\mathrm{c}^{(n)} : \mathbb{N}^n \to \mathbb{N}$ $(n \geq 2)$ sowie die zugehörigen Umkehrfunktionen $\mathrm{c}_0^{(n)},\ldots,\mathrm{c}_{n-1}^{(n)}$. Wir definieren dazu

$$\begin{aligned}
\mathrm{c}^{(n)}(x_1,\ldots,x_n) &:= \mathrm{c}(x_1,\mathrm{c}(x_2,\ldots,\mathrm{c}(x_{n-1},x_n)\ldots)) \text{ und} \\
\mathrm{c}_0^{(n)}(z) &:= \mathrm{c}_0(z) \\
\mathrm{c}_1^{(n)}(z) &:= \mathrm{c}_1(\mathrm{c}_2(z)) \\
&\vdots \\
\mathrm{c}_{n-2}^{(n)}(z) &:= \mathrm{c}_1(\mathrm{c}_2^{(n-2)}(z)) \\
\mathrm{c}_{n-1}^{(n)}(z) &:= \mathrm{c}_2^{n-1}(z)
\end{aligned}$$

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

Wir zeigen durch Induktion über den Aufbau von LOOP-Programmen, 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.

1) Hat P die Form x_i `:=` $x_j + c$, so ist $\mathrm{g}_P(z) = \mathrm{c}^{(k+1)}(\mathrm{c}_0^{(k+1)}(z), \ldots, \mathrm{c}_{i-1}^{(k+1)}(z), \mathrm{c}_j^{(k+1)}(z) + c, \mathrm{c}_{i+1}^{(k+1)}(z), \ldots)$. 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:
   $\mathrm{h}(x, 0) = x$,
   $\mathrm{h}(x, y + 1) = \mathrm{g}_Q(\mathrm{h}(x, y))$
   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 LOOP-Programm P primitiv rekursiv ist.
   Ist $\mathrm{f} : \mathbb{N}^r \to \mathbb{N}$ die von P berechnete Funktion, so gilt:
   $\mathrm{f}(z_1, \ldots, z_r) = \mathrm{c}_0^{(k+1)}(\mathrm{g}_P(\mathrm{c}^{(k+1)}(0, z_1, \ldots, z_r, \underbrace{0, \ldots, 0}_{k-r})))$. □

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

Satz 14.21
Es gibt totale berechenbare Funktionen, die nicht LOOP-berechenbar sind.

<u>Beweis:</u>
Wir definieren die Länge von LOOP-Programmen induktiv über deren Aufbau:

1) $|x_i$ `:=` $x_j + c| := i + j + c + 1$
   $|x_i$ `:=` $x_j \dot{-} c| := i + j + c + 1$
2) $|P\,;Q| := |P| + |Q| + 1$
3) $|$`LOOP` x_i `DO` P `END`$| := |P| + i + 1$

Für eine gegebene Länge n gibt es nur endlich viele LOOP-Programme dieser Länge. Deshalb macht die folgende Definition Sinn:

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

<u>Behauptung1:</u>
Die Funktion $\mathrm{d} : \mathbb{N} \to \mathbb{N}$ 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 $\mathrm{f}(n, n)$ <u>größer</u> als der maximale Funktionswert, den ein LOOP-Programm der Länge n bei Eingabe n berechnen kann. Dies widerspricht der Tatsache, daß d von einem LOOP-Programm der Länge n berechnet wird. □

<u>Behauptung2:</u>
Die Funktion d ist (intuitiv) berechenbar, denn:
Bei Eingabe z zählt man die endlich vielen LOOP-Programme der Länge z auf und wendet

sie jeweils auf die Eingabe z an. Da alle diese Aufrufe terminieren, kann man in endlicher Zeit den maximalen so erhaltenen Funktionswert berechnen.

Mit der Churchschen These ist diese Funktion auch Turing-berechenbar.
Eine prominente nicht primitiv rekursive, aber berechenbare Funktion ist die sogenannte Ackermannfunktion A, die wie folgt definiert ist:

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

Durch (geschachtelte) Induktion zeigt man leicht, daß A dadurch eindeutig definiert ist. Man kann zeigen, daß A zwar Turing-berechenbar ist, aber nicht LOOP-berechenbar, da sie schneller wächst als jede LOOP-berechenbare Funktion (siehe [Schö_97]).

15 μ-rekursive Funktionen und While-Programme

Um alle berechenbaren Funktionen zu erhalten, muß man die primitiv rekursiven Funktionen um eine weitere Operation, die unbeschränkte Minimalisierung, erweitern. Die LOOP-Programme muß man um eine While-Schleife (d.h. eine Schleife ohne vorher festgelegte Anzahl der Durchläufe) erweitern.

Definition 15.1 (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:

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

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

Beachte: Der μ-Operator sucht 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.

Beispiel 15.2

1) $\mathrm{g}_1(x,y) = \begin{cases} x-y & \textit{falls } x \geq y \\ \textit{undefiniert} & \textit{falls } x<y \end{cases}$

$\mu\mathrm{g}_1(x) = x$, d.h. man erhält durch Anwenden des μ-Operators auf die partielle Funktion g_1 die totale Identitätsfunktion $\mathrm{f}_1 : \mathbb{N} \to \mathbb{N}$ mit $x \mapsto x$.

2) $\mathrm{g}_2(x,y) = \begin{cases} y-x & \textit{falls } x \leq y \\ \textit{undefiniert} & \textit{falls } x>y \end{cases}$

$\mu\mathrm{g}_2(x) = \begin{cases} 0 & \textit{falls } x=0 \\ \textit{undefiniert} & \textit{falls } x>0 \quad \textit{(vor Wert 0 hat man undefiniert)} \end{cases}$

3) $\mathrm{g}_3(x,y) = x+y$

$\mu\mathrm{g}_3(x) = \begin{cases} 0 & \textit{falls } x=0 \\ \textit{undefiniert} & \textit{falls } x>0 \quad \textit{(kein Wert 0)} \end{cases}$

Durch Anwenden von μ kann man also partielle Funktionen aus totalen erhalten.

Definition 15.3 (Klasse der μ-rekursiven Funktionen)
Die Klasse der μ-rekursiven Funktionen besteht aus den Funktionen, welche man aus den Grundfunktionen (Definition 14.1) durch endlich oftes Anwenden von Komposition, primitiver Rekursion und unbeschränkter Minimalisierung erhält.

Definition 15.4 (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.

Beachte: 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 15.5 (WHILE-berechenbar)
Die (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 15.6
Die Funktion $\mathrm{f} : \mathbb{N}^2 \to \mathbb{N}$ 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
  y := x1;
  WHILE y ≠ 0 DO
    x1 := x1 -̇ 1;
    x2 := x2 -̇ 1;
    y := 0
  END
END;
x0 := x1
```

Satz 15.7
Die Klasse der μ-rekursiven Funktionen stimmt genau mit der Klasse der WHILE-berechenbaren Funktionen überein.

Beweis:

Wir müssen hierzu den Beweis von Satz 14.20 um die Behandlung des zusätzlichen Operators/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:

x_0 := 0;
y := $\mathrm{g}(x_1, \ldots, x_n, x_0)$; (Realisierbar mittels P)
WHILE $y \neq 0$ DO
  x_0 := $x_0 + 1$;
  y := $\mathrm{g}(x_1, \ldots, x_n, x_0)$
END

Beachte: Dieses Programm ist nur deshalb korrekt, weil wir in der Definition von μ gefordert haben, daß alle vorherigen Werte nicht nur $\neq 0$, sondern auch definiert sein müssen.

2) Alle WHILE-berechenbaren Funktionen sind μ-rekursiv:
Betrachte das Programm WHILE $x_i \neq 0$ DO Q END. Wie im Beweis von Satz 14.20 bei der Behandlung von LOOP beschaffen wir uns 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 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.
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 $\mathrm{g}_P(z) = \mathrm{h}(z, (\mu \mathrm{c}_i^{(k+1)}(\mathrm{h}(z))))$. □

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

Satz 15.8
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.
Verwende $m + 1$ Bänder:

- Kopiere die Eingabe $\underline{x}$ auf Band2, ..., Band$(m + 1)$
- $\mathcal{A}_{\mathrm{h}_i}$ berechnet $\mathrm{h}_i(\underline{x})$ auf Band $(i + 1)$
- Überschreibe Band1 mit $\not b\, a^{\mathrm{h}_1(\underline{x})}\ \not b \ldots \not b\, a^{\mathrm{h}_m(\underline{x})}\ \not b$
- Berechne g-Wert davon mit $\mathcal{A}_\mathrm{g}$

Primitive Rekursion: Seien $\mathcal{A}_\mathrm{g}, \mathcal{A}_\mathrm{h}$ DTM für g, h.
Verwende 4 Bänder:

Band1: Speichert $\underline{xy}$
Band2: Zählt von 0 ab hoch bis y (aktueller Wert: z)
Band3: Enthält $\mathrm{f}(\underline{x}, z)$ (für $z = 0$ mittels $\mathcal{A}_\mathrm{g}$ berechnet)
Band4: Berechnung von $\mathcal{A}_\mathrm{h}$

μ-Operator: Sei $\mathcal{A}_g$ eine DTM für g.
Verwende 3 Bänder:
Band1: Speichert Eingabe $\underline{x}$
Band2: Zählt von 0 ab hoch (aktueller Wert: z)
Band3: Berechne $g(\underline{x}, z)$ mittels $\mathcal{A}_g$ und teste, ob Wert $= 0$ ist
□

Satz 15.9
Jede Turing-berechenbare Funktion ist WHILE-berechenbar.

Beweisskizze:
Um diesen Satz zu beweisen, müssen wir Konfigurationen von Turingmaschinen der Form $\alpha q \beta$ für $\alpha, \beta \in \Gamma^+$ und $q \in Q$ in drei natürlichen Zahlen kodieren. Diese werden dann in den drei Programmvariablen x_1, x_2, x_3 des WHILE-Programms gespeichert. Es sei o.B.d.A. $\Gamma = \{a_1, \ldots, a_n\}$ und $Q = \{q_1, \ldots, q_k\}$. Die Konfiguration $a_{i_1} \ldots a_{i_l} q_m a_{j_1} \ldots a_{j_r}$ wird dargestellt durch

$$x_1 = (i_1, \ldots, i_l)_b := \sum_{\nu=1}^{l} i_\nu \cdot b^{l-\nu}$$
$$x_2 = m$$
$$x_3 = (j_r, \ldots, j_1)_b := \sum_{\rho=1}^{r} j_\rho \cdot b^\rho,$$

wobei $b > |\Gamma|$ ist, d.h. $a_{i_1} \ldots a_{i_l}$ repräsentiert x_1 in b-närer Zahlendarstellung und $a_{j_r} \ldots a_{j_1}$ (Reihenfolge!) repräsentiert x_3 in b-närer Zahlendarstellung.

Das Herauslesen des aktuellen Symbols a_{j_1} kann daher wie folgt erreicht werden:
Ist $x_3 = (j_r, \ldots, j_1)_b$, so ist $j_1 = x_3 \bmod b$.

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

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.

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 $x_3 \bmod b$.
   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 möglich.

3) Aus dem Wert von x_3 nach Termination der WHILE-Schleife wird der Ausgabewert herausgelesen und in die Variable x_0 geschrieben. □

Insgesamt haben wir also gezeigt:

Theorem 15.10
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 Churchschen These.

16 Universelle Maschinen und unentscheidbare Probleme

Wir werden hier zeigen, daß es Relationen gibt, die nicht Turing-entscheidbar sind (d.h. die charakteristische Funktion ist nicht Turing-berechenbar). Mit der Churchschen These sind diese Relationen dann nicht entscheidbar im intuitiven Sinn.
Dazu konstruieren wir eine universelle Turingmaschine, die alle Turingmaschinen simulieren kann. Die universelle Maschine erhält als zusätzliche Eingabe eine Kodierung der zu simulierenden Turingmaschine.

Konventionen:

Arbeitsalphabete der betrachteten Turingmaschinen sind Teilmengen von $\{a_1, a_2, a_3, \ldots\}$, wobei $a_1 = a, a_2 = b, a_3 = \not b$.

Zustandsmengen sind Teilmengen von $\{q_1, q_2, q_3, \ldots\}$, wobei q_1 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* $t = (q_i, a_j, a_{j'}, \begin{array}{c} l \\ r \\ n \end{array}, q_{i'})$ *wird kodiert durch* $\mathrm{code}(t) = a^i b a^j b a^{j'} b \begin{array}{c} a \\ aa \\ aaa \end{array} b a^{i'} bb$.

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* $\mathrm{code}(\mathcal{A}) = \mathrm{code}(t_1) \ldots \mathrm{code}(t_k) b a^{i_1} b \ldots b a^{i_r} bbb$.

Bemerkung 16.2

1) *Für $x = \mathrm{code}(\mathcal{A}) w$ mit $w \in \Sigma^*$ folgt w genau auf den zweiten Block bbb, kann also aus x eindeutig wieder herausgelesen werden.*

2) *Es gibt eine DTM $\mathcal{A}_{CODE}$, welche die Relation $CODE = \{\mathrm{code}(\mathcal{A}) \mid \mathcal{A}$ 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).

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: $\mathcal{U}$ akzeptiert $\mathrm{code}(\mathcal{A})w$ gdw. $\mathcal{A}$ akzeptiert w.

Es ist also $\mathcal{U}$ ein „Turing-Interpreter für Turingmaschinen“.

Beweis:

$\mathcal{U}$ führt bei Eingabe code$(\mathcal{A})w$ die $\mathcal{A}$-Berechnung $k_0 \vdash_\mathcal{A} k_1 \vdash_\mathcal{A} \ldots$ in kodierter Form aus, d.h. $\mathcal{U}$ erzeugt sukzessive Bandbeschriftungen
code$(\mathcal{A})$code(k_0), code$(\mathcal{A})$code(k_1), code$(\mathcal{A})$code$(k_2), \ldots$

Konfigurationskodierung:

$\text{code}(a_{i_1} \ldots a_{i_l} q_j a_{i_{l+1}} \ldots a_{i_r}) = a^{i_1} b \ldots a^{i_l} b a^j b b a^{i_{l+1}} b \ldots a^{i_r} b$

Arbeitsweise von $\mathcal{U}$: (bei Eingabe code$(\mathcal{A})w$)

- Erzeuge aus code$(\mathcal{A})w$ die Anfangsbeschriftung code$(\mathcal{A})$code$(\underbrace{\not b\, q_1 w \not b}_{k_0})$.
- Simuliere die $\mathcal{A}$-Schritte, ausgehend vom jeweiligen Konfigurationskode $\ldots ba^j bba^i b \ldots$ (Zustand q_j, gelesenes Symbol a_i).
  - Aufsuchen einer 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. $\square$

Satz 16.4
Die Relation $UNIV = \{\text{code}(\mathcal{A})w \in \Sigma^* \mid \mathcal{A} \textit{ ist DTM über } \Sigma, \textit{ die } w \textit{ akzeptiert}\}$ *ist partiell entscheidbar, aber nicht entscheidbar.*

Beweis:

1) Wir zeigen, daß $UNIV$ Turing-akzeptierbar (und damit partiell entscheidbar) ist: Bei Eingabe x geht die TM, welche $UNIV$ akzeptiert, wie folgt vor:
   - Teste, ob x von der Form $x = x_1 w$ mit $x_1 \in CODE$ und $w \in \Sigma^*$ ist.
   - Wenn ja, so wende $\mathcal{U}$ auf x an.

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 Menge 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 $\mathcal{A} = \mathcal{A}_D$):
   $$\begin{array}{lll} \mathcal{A}_D \text{ akzeptiert code}(\mathcal{A}_D) & \text{gdw.} & \text{code}(\mathcal{A}_D) \in D \\ & \text{gdw.} & \text{code}(\mathcal{A}_D)\text{code}(\mathcal{A}_D) \notin UNIV \\ & \text{gdw.} & \mathcal{A}_D \text{ akzeptiert code}(\mathcal{A}_D) \underline{\text{nicht}}. \end{array}$$
   Widerspruch. $\square$

Die in Teil 2 des Beweises verwendete Vorgehensweise nennt man Diagonalisierung. Wir hatten hier ja zwei Dimensionen (Kodierung der TM, Eingabe der TM), und die Menge D identifiziert die beiden (betrachtet TM, die als Eingabe ihre eigene Kodierung haben), d.h. D bewegt sich in der Diagonalen.
Ein anderes Beispiel für ein Diagonalisierungsargument ist Cantors Beweis für die Überabzählbarkeit von $\mathbb{R}$. Wir wenden im folgenden Einschub Diagonalisierung an, um zu zeigen, daß es nichtkontextsensitive 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 $L \subseteq \{a, b\}^*$ durch $w_i \in L :\Leftrightarrow w_i \notin L(G_i)$ (Diagonalisierung!). Es ist L Turing-akzeptierbar, da man für eine kontextsensitive Grammatik G_i und ein Wort w_i entscheiden kann, ob $w_i \in L(G_i)$ ist (Beachte: nichtkürzende Produktionen $\Rightarrow$ Aufzählen aller ableitbaren Wörter bis zur Länge $|w_i|$).
Wäre L kontextsensitiv, so gäbe es einen Index k mit $L = L(G_k)$. Nun ist aber $w_k \in L(G_k)$ gdw. $w_k \in L$ gdw. $w_k \notin L(G_k)$. Widerspruch. $\square$

Aus der Unentscheidbarkeit von $UNIV$ kann man weitere wichtige Unentscheidbarkeitsresultate herleiten.

Satz 16.6
Das Wortproblem für DTM (und damit auch für $\mathcal{L}_0$) 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.

Beweis:

Wenn das Wortproblem für DTM entscheidbar wäre, so wäre auch $UNIV$ entscheidbar (ist es aber nicht).
Um zu entscheiden, ob code($\mathcal{A}$)$w \in UNIV$ ist, müßte man ja nur das Entscheidungsverfahren für das Wortproblem feststellen lassen, ob $\mathcal{A}$ das Wort w akzeptiert. $\square$

Satz 16.7
Das Halteproblem für DTM ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das zu jeder gegebenen DTM $\hat{\mathcal{A}}$ entscheidet, ob $\hat{\mathcal{A}}$ beginnend mit leerem Eingabeband terminiert.

Beweis:

Wäre das Halteproblem entscheidbar, so auch das Wortproblem. Um zu gegebener TM $\mathcal{A}$ und gegebenem Wort w zu entscheiden, ob $\mathcal{A}$ das Wort w akzeptiert, könnte man dann nämlich wie folgt vorgehen: Modifiziere $\mathcal{A}$ zu einer TM $\hat{\mathcal{A}}$:

- Bei leerem Band schreibt $\hat{\mathcal{A}}$ zunächst w auf das Band.
- Danach verhält sich $\hat{\mathcal{A}}$ wie $\mathcal{A}$.
- Stoppt $\mathcal{A}$ mit akzeptierender Stoppkonfiguration, so stoppt auch $\hat{\mathcal{A}}$. Hält $\mathcal{A}$ mit nichtakzeptierender Stoppkonfiguration, so geht $\hat{\mathcal{A}}$ in eine Endlosschleife.

Damit gilt: $\hat{\mathcal{A}}$ hält mit leerem Eingabeband gdw. $\mathcal{A}$ akzeptiert w. Mit dem Entscheidungsverfahren für das Halteproblem (angewandt auf $\hat{\mathcal{A}}$) könnte man also das Wortproblem („Ist w in $L(\mathcal{A})$?“) entscheiden. $\square$

Satz 16.8
Das Leerheitsproblem für DTM (und damit auch für $\mathcal{L}_0$) ist unentscheidbar, d.h. es gibt kein Berechnungsverfahren, das bei gegebener DTM $\hat{\mathcal{A}}$ entscheidet, ob es eine Eingabe w gibt, auf der $\hat{\mathcal{A}}$ terminiert.

Beweis:

Wäre das Leerheitsproblem entscheidbar, so auch das Halteproblem. Um bei gegebener DTM $\mathcal{A}$ zu entscheiden, ob $\mathcal{A}$ auf leerer Eingabe 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 gibt es eine Eingabe, für die $\hat{\mathcal{A}}$ hält gdw. $\mathcal{A}$ auf leerem Eingabeband hält. $\square$

Satz 16.9
Das Äquivalenzproblem für DTM (und damit auch für $\mathcal{L}_0$) 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. $\square$

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.

Wäre $\overline{UNIV} \in \mathcal{L}_0$, d.h. partiell entscheidbar, so würde aber mit Satz 13.4 (Teil 4) folgen, daß $UNIV$ entscheidbar ist. $\square$

Das in den Beweisen der Sätze 16.6 bis 16.9 gewählte Vorgehen nennt man Reduktion:
Ein Problem P_1 (z.B. Halteproblem) wird auf ein 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. Formaler betrachten wir (o.B.d.A.) einstellige Relationen $R \subseteq \Sigma^*$.

Definition 16.11 (Reduktion)

1) *Eine Reduktion von $R_1 \subseteq \Sigma^*$ auf $R_2 \subseteq \Sigma^*$ ist eine berechenbare Funktion $\mathrm{f} : \Sigma^* \to \Sigma^*$, für die gilt: $w \in R_1$ gdw. $\mathrm{f}(w) \in R_2$.*

2) *Wir schreiben $R_1 \leq_m R_2$ (R_1 wird auf R_2 reduziert), falls es eine Reduktion von R_1 nach R_2 gibt.*

Lemma 16.12

1) *$R_1 \leq_m R_2$ und R_2 entscheidbar $\Rightarrow$ R_1 entscheidbar.*

2) *$R_1 \leq_m R_2$ und R_1 unentscheidbar $\Rightarrow$ R_2 unentscheidbar.*

Beweis:

1) Um „$w \in R_1$“ zu entscheiden, berechnet man $\mathrm{f}(w)$ und entscheidet „$\mathrm{f}(w) \in R_2$“.
2) Folgt unmittelbar aus 1). □

Mit Hilfe einer Reduktion des Halteproblems kann man zeigen: Jede nichttriviale semantische Eigenschaft von Programmen (DTM) ist unentscheidbar.

Semantisch heißt hier: Die Eigenschaft hängt nicht von der syntaktischen Form des Programms, sondern nur von der berechneten Funktion ab.

Nichttrivial: Es gibt berechenbare Funktionen, die sie erfüllen, aber nicht alle berechenbaren Funktionen erfüllen sie.

Satz 16.13 (Satz von Rice)
Es sei E eine Eigenschaft partiell berechenbarer Funktionen $\mathrm{f} : \Sigma^* \to \Sigma^*$, *so daß gilt:*
$\emptyset \subset \{\mathrm{f} : \Sigma^* \to \Sigma^* \mid \mathrm{f}\ \textit{erfüllt}\ E\} \subset \{\mathrm{f} : \Sigma^* \to \Sigma^* \mid \mathrm{f}\ \textit{ist partiell berechenbar}\}$
Dann ist $L(E) := \{\mathrm{code}(\mathcal{A}) \mid \textit{die von } \mathcal{A} \textit{ berechnete Funktion } \mathrm{f} : \Sigma^* \to \Sigma^* \textit{ erfüllt } E\}$ *unentscheidbar.*

Beweis:

Angenommen, $L(E)$ ist entscheidbar. Wir zeigen, daß man dann auch ein Entscheidungsverfahren für das Halteproblem erhält (Widerspruch zu Satz 16.7). Mit f_u bezeichnen wir die überall undefinierte Funktion, welche offenbar partiell berechenbar ist. O.B.d.A. erfülle f_u die Eigenschaft E. Sonst könnte man statt E die Eigenschaft $\overline{E}$: „f erfüllt E nicht“ betrachten. Mit $L(E)$ ist auch $L(\overline{E}) = \overline{L(E)}$ entscheidbar.
Da nicht alle partiell berechenbaren Funktionen E erfüllen, gibt es eine partiell berechenbare Funktion g, welche E nicht erfüllt. Es sei nun $\mathcal{A}_{\mathrm{g}}$ eine DTM für g und $\mathcal{A}_u$ eine für f_u. Wir konstruieren nun eine DTM $\mathcal{A}'$, welche eine Funktion $\mathrm{f} : \Sigma^* \to \Sigma^*$ berechnet mit

$$w = \mathrm{code}(\mathcal{A}) \text{ für eine DTM} \mathcal{A}, \text{ die auf leerer Eingabe terminiert gdw.} \mathrm{f}(w) \notin L(E) \quad (\star).$$

Wäre daher $L(E)$ entscheidbar, so auch das Halteproblem.

Konstruktion von $\mathcal{A}'$:

Die DTM $\mathcal{A}'$ testet bei Eingabe w zunächst, ob w Kodierung einer DTM ist.
Falls nein, so gibt $\mathcal{A}'$ $\mathrm{code}(\mathcal{A}_u) \in L(E)$ aus.
Falls ja, so ist $w = \mathrm{code}(\mathcal{A})$ für eine DTM $\mathcal{A}$. Die DTM $\mathcal{A}'$ gibt dann $\mathrm{code}(\mathcal{A}'')$ aus, wobei $\mathcal{A}''$ eine DTM ist, die sich wie folgt verhält:

- $\mathcal{A}''$ ignoriert zunächst die Eingabe x und simuliert das Verhalten von $\mathcal{A}$ auf dem leeren Eingabeband.
- Im Anschluß (d.h. falls $\mathcal{A}$ auf leerem Eingabeband terminiert) verhält sich $\mathcal{A}''$ wie $\mathcal{A}_{\mathrm{g}}$ bei Eingabe x.

Damit gilt für $\mathcal{A}''$ offenbar:

- Terminiert $\mathcal{A}$ auf leerer Eingabe nicht, so berechnet $\mathcal{A}''$ die Funktion f_u, d.h. $\mathrm{code}(\mathcal{A}'') \in L(E)$, da f_u E erfüllt.
- Terminiert $\mathcal{A}$ auf leerer Eingabe, so berechnet $\mathcal{A}''$ die Funktion g, d.h. $\mathrm{code}(\mathcal{A}'') \notin L(E)$, da g E nicht erfüllt.

Insgesamt haben wir also gezeigt, daß die von $\mathcal{A}'$ berechnete Funktion f die Bedingung $(\star)$ erfüllt, das Halteproblem also auf das Problem, $L(E)$ zu entscheiden, reduziert. Da das Halteproblem unentscheidbar ist, folgt die Unentscheidbarkeit von $L(E)$. □

17 Weitere unentscheidbare Probleme

Die bisher als unentscheidbar nachgewiesenen Probleme betreffen alle nur (semantische) Eigenschaften von Programmen. Aber auch Probleme, die (direkt) nichts mit Programmen zu tun haben, können unentscheidbar sein.
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 $x_i, y_i \in \Sigma^+$ 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: $x_{i_1} \ldots x_{i_m} = y_{i_1} \ldots y_{i_m}$.

Beispiel 17.2

1) $P_1 = (a, aaa), (abaa, ab), (aab, b)$
hat z.B. die Folgen 2, 1 und 1, 3 als Lösungen

$abaa\|a$	$a\|aab$
$ab\|aaa$	$aaa\|b$

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:

$ab|aba|aba| \ldots$
$aba|baa|baa| \ldots$

Um die Unentscheidbarkeit des PKP zu zeigen, führen wir zunächst ein Zwischenproblem ein, das <u>modifizierte PKP</u> (MPKP): Hier muß für die Lösung zusätzlich $i_1 = 1$ gelten, d.h. das Wortpaar, mit dem man beginnen muß, ist festgelegt.

Lemma 17.3
Das MPKP kann auf das PKP reduziert werden.

<u>Beweis:</u>
Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$ eine Instanz der MPKP über dem Alphabet Σ. Es seien $\#, \$$ Buchstaben, die nicht in Σ vorkommen. Wir definieren die Instanz 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 <u>hinter</u> jedem Buchstaben 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 <u>vor</u> jedem Buchstaben 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 $f(P)$ hat eine Lösung." $\square$

Beispiel:

$P = (a, aaa), (aab, b)$ ist als MPKP lösbar mit Lösung 1, 2.

$$f(P) = \overset{(x'_0, y'_0)}{(\#a\#, \#a\#a\#a)}, \overset{(x'_1, y'_1)}{(a\#, \#a\#a\#a)}, \overset{(x'_2, y'_2)}{(a\#a\#b\#, \#b)}, \overset{(x'_3, y'_3)}{(\$, \#\$)}$$

Die Lösung 1, 2 von P liefert die Lösung 0, 2, 3 von $f(P)$:

$\#a\#|a\#a\#b\#|\$$

$\#a\#a\#a|\#b|\#\$$

Offenbar muß jede Lösung von $f(P)$ mit 0 beginnen.

Wäre daher das PKP entscheidbar, so auch das MPKP. Es genügt also zu zeigen, daß das MPKP unentscheidbar ist.

Lemma 17.4

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 $f(\mathcal{A}, w)$ des MPKP überführen kann, so daß gilt: $\mathcal{A}$ hält auf Eingabe w gdw. $f(\mathcal{A}, w)$ hat eine Lösung.
Wir verwenden für das MPKP $f(\mathcal{A}, w)$ das Alphabet $\Gamma \cup Q \cup \{\#\}$ mit $\# \notin \Gamma \cup Q$. Das MPKP $f(\mathcal{A}, w)$ besteht aus den folgenden Wortpaaren:

1) Anfangsregel:
   $(\#, \#\not b\, q_0 w \not b\#)$

2) Kopierregeln:
   (a, a) für alle $a \in \Gamma \cup \{\#\}$

3) Übergangsregeln:
   $(qa, q'a')$ falls $(q, a, a', n, q') \in \Delta$
   $(qa, a'q')$ falls $(q, a, a', r, q') \in \Delta$
   $(bqa, q'ba')$ falls $(q, a, a', l, q') \in \Delta$ und $b \in \Gamma$
   $(\#qa, \#q'\not b\, a')$ falls $(q, a, a', l, q') \in \Delta$
   $(q\#, q'a'\#)$ falls $(q, \not b, a', n, q') \in \Delta$
   $(q\#, a'q'\#)$ falls $(q, \not b, a', r, q') \in \Delta$
   $(bq\#, q'ba'\#)$ falls $(q, \not b, a', l, q') \in \Delta$
   $(\#q\#, \#q'\not b\, a'\#)$ falls $(q, \not b, a', l, q') \in \Delta$

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 ab.)

5) Abschlußregel:
   $(q\#\#, \#)$ für alle $q \in Q$ mit q Stoppzustand

Falls $\mathcal{A}$ bei Eingabe w hält, so gibt es eine Folge von Konfigurationsübergängen $k_0 \vdash_{\mathcal{A}} k_1 \vdash_{\mathcal{A}} \ldots \vdash_{\mathcal{A}} k_t$ mit $k_0 = \not b\, q_0 w \not b$ und $k_t = u\hat{q}v$ mit $\hat{q}$ Stoppzustand. Daraus kann man nun eine Lösung des MPKP bauen. Zunächst erzeugt man

$\#k_0\#k_1\#k_2\#\ldots\#$
$\#k_0\#k_1\#k_2\#\ldots\#k_t\#$

- 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 $k_0 = \not b\, qab \not b$

  $\#|\not b|qa|b|\not b|\#|$
  $\#\ \not b\ \ qa\ \ b\not b\ \ \#|\not b|a'q'|b|\not b|\#|$

  Man erhält so: $\begin{array}{l}\#k_0\# \\ \#k_0\#k_1\#\end{array}$

- 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. $\begin{array}{l}\ldots\#a\hat{q}|b|\#|\hat{q}\ b|\# \\ \ldots\#a\hat{q}\ b\ \#|\hat{q}|b|\#|\hat{q}|\#\end{array}$

- 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 nicht gleich lang machen.
- Daher muß ein Stoppzustand erreicht werden, damit Lösch- und Abschlußregeln eingesetzt werden können. $\square$

Da das Halteproblem unentscheidbar ist, folgt die Unentscheidbarkeit des MPKP und damit (wegen Lemma 17.3) die Unentscheidbarkeit des PKP.

Satz 17.5
Das PKP ist unentscheidbar.

Wir verwenden dieses Resultat, um Unentscheidbarkeit von Problemen für kontextfreie und kontextsensitive Sprachen nachzuweisen. Wir zeigen zunächst:

Lemma 17.6
Es ist nicht entscheidbar, ob für kontextfreie Grammatiken G_1, G_2 gilt: $L(G_1) \cap L(G_2) \neq \emptyset$.

Beweis:

Wir reduzieren das PKP auf dieses Problem, 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 hat Lösung gdw. $L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset$.
Da das PKP unentscheidbar ist, folgt dann die Aussage des Lemmas.

Es sei $P = (x_1, y_1), \dots, (x_k, y_k)$. Wir definieren $G_P^{(l)} = (N_l, \Sigma_l, P_l, S_l)$ mit $N_l = \{S_l\}$, $\Sigma_l = \Sigma \cup \{1, \dots, k\}$ und $P_l = \{S_l \longrightarrow w_i S_l i, S_l \longrightarrow w_i i \mid 1 \le i \le k\}$. $G_P^{(r)}$ wird entsprechend definiert. Es gilt:
$L(G_P^{(l)}) = \{x_{i_1} \dots x_{i_m} i_m \dots i_1 \mid m \ge 1, i_j \in \{1, \dots, k\}\}$
$L(G_P^{(r)}) = \{y_{i_1} \dots y_{i_m} i_m \dots i_1 \mid m \ge 1, i_j \in \{1, \dots, k\}\}$
Daraus folgt nun unmittelbar:
$L(G_P^{(l)}) \cap L(G_P^{(r)}) \neq \emptyset$ gdw.
$\exists m \ge 1 \; \exists i_1, \dots, i_m \in \{1, \dots, k\} : x_{i_1} \dots x_{i_m} i_m \dots i_1 = y_{i_1} \dots y_{i_m} i_m \dots i_1$ gdw.
P hat Lösung. $\square$

Da jede kontextfreie Sprache auch kontextsensitiv ist und da die kontextsensitiven Sprachen unter Durchschnitt abgeschlossen sind (Satz 12.6), folgt daraus unmittelbar:

Satz 17.7
Für $\mathcal{L}_1$ sind das Leerheitsproblem und das Äquivalenzproblem unentscheidbar.

Beachte: Leerheitsproblem ist Spezialfall des Äquivalenzproblems, da $L(G) = \emptyset$ gdw. $L(G) = L(G_\emptyset)$ ($G_\emptyset$: kontextsensitive Grammatik für $\emptyset$).

Für kontextfreie Sprachen wissen wir bereits, daß das Leerheitsproblem entscheidbar ist. Dies ist kein Widerspruch zu Lemma 17.6, da $\mathcal{L}_2$ nicht unter Durchschnitt abgeschlossen ist.

Satz 17.8
Für $\mathcal{L}_2$ ist das Äquivalenzproblem unentscheidbar.

Beweis:

Man kann sich leicht überlegen, daß die Sprachen $L(G_P^{(l)})$ und $L(G_P^{(r)})$ aus dem Beweis von Lemma 17.6 durch deterministische Kellerautomaten akzeptiert werden können. Die von deterministischen Kellerautomaten akzeptierten kontextfreien Sprachen sind nun aber 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. [Wege_93], Satz 8.1.3). Es sei $\overline{G}$ die kontextfreie Grammatik mit $L(\overline{G}) = \overline{L(G_P^{(l)})}$. Nun gilt:

$$\begin{aligned} & L(G_P^{(l)}) \cap L(G_P^{(r)}) = \emptyset \\ \text{gdw.} \quad & L(G_P^{(r)}) \subseteq L(\overline{G}) \\ \text{gdw.} \quad & L(G_P^{(r)}) \cup L(\overline{G}) = L(\overline{G}) \\ \text{gdw.} \quad & L(G_\cup) = L(\overline{G}), \end{aligned}$$

wobei $G_\cup$ die effektiv konstruierbare kontextfreie Grammatik für $L(G_P^{(r)}) \cup L(\overline{G})$ ist.
Wäre also das Äquivalenzproblem für kontextfreie Grammatiken entscheidbar, so auch $L(G_P^{(l)}) \cap L(G_P^{(r)}) = \emptyset$ und damit das PKP. $\square$

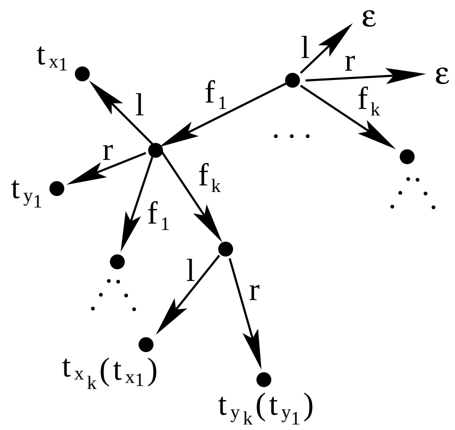
Als nächstes verwenden wir das PKP, um zu zeigen, daß die Gültigkeit von Formeln der Prädikatenlogik erster Stufe (PL1) unentscheidbar ist.

Satz 17.9
Das PKP kann auf die Gültigkeit von PL1-Formeln reduziert werden.

Beweis:
Es sei $P = (x_1, y_1), \ldots, (x_k, y_k)$ ein PKP über dem Alphabet $\Sigma = \{a_1, \ldots, a_n\}$.
Wir verwenden

- einstellige Funktionssymbole $a_1, \ldots, a_n$,
- einstellige Funktionssymbole $f_1, \ldots, f_k$,
- einstellige Funktionssymbole ℓ, r,
- das Gleichheitssymbol $=$ (welches als Gleichheit interpretiert werden muß),
- ein Konstantensymbol ε.

Ein Wort $w = a_{i_1} \ldots a_{i_m} \in \Sigma^*$ wird dargestellt als Term $t_w = a_{i_m}(a_{i_{m-1}}(\ldots(a_{i_1}(\varepsilon))\ldots))$. Die Konkatenation entspricht daher dem Einsetzen von Termen, d.h. $t_{uv} = t_v(t_u)$.
Idee: Der Suchbaum für eine Lösung des PKP kann wie folgt dargestellt werden:



Dies wird durch die folgende Formel beschrieben:

$$\begin{array}{lll} B_P: & \exists x. & (\ell(x) = \varepsilon \quad \wedge \quad r(x) = \varepsilon) \wedge \\ & \forall x. & (\ell(f_1(x)) = t_{x_1}(\ell(x)) \neq \varepsilon \quad \wedge \quad r(f_1(x)) = t_{y_1}(r(x)) \neq \varepsilon \wedge \\ & & \vdots \\ & & \ell(f_k(x)) = t_{x_k}(\ell(x)) \neq \varepsilon \quad \wedge \quad r(f_k(x)) = t_{y_k}(r(x)) \neq \varepsilon) \end{array}$$

Die Lösbarkeit des PKP wird dann ausgedrückt durch die Formel:

$$F_P: \quad B_P \quad \Rightarrow \quad \exists z.(\ell(z) = r(z) \neq \varepsilon)$$

Behauptung: F_P ist gültig gdw P eine Lösung hat.
denn: Jede Interpretation, die B_P erfüllt, „enthält“ den Suchbaum.
„⇒“: Angenommen, F_P ist gültig. Wir benutzen den Suchbaum, um eine Interpretation $\mathcal{I}$ zu definieren, die B_P erfüllt:

$$\begin{array}{lll} dom(\mathcal{I}) & := & \Sigma^* \cup \{1, \ldots, k\}^* \\ \varepsilon^{\mathcal{I}} & := & \varepsilon \\ a_i & : & dom(\mathcal{I}) \to dom(\mathcal{I}) \end{array}$$

$$a_i^{\mathcal{I}}(u) := \begin{cases} ua_i & \text{falls } u \in \Sigma^* \\ \varepsilon & \text{sonst} \end{cases}$$

$$\begin{array}{rcl} f_j & : & dom(\mathcal{I}) \to dom(\mathcal{I}) \\ f_j^{\mathcal{I}}(u) & := & \begin{cases} uj & \text{falls } u \in \{1, \ldots, k\}^* \\ j & \text{sonst} \end{cases} \end{array}$$

$$\begin{array}{rcl} \ell & : & dom(\mathcal{I}) \to dom(\mathcal{I}) \\ \ell^{\mathcal{I}}(u) & := & \begin{cases} x_{i_1} \ldots x_{i_m} & \text{falls } u = i_1 \ldots i_m \in \{1, \ldots, k\}^* \\ \varepsilon & \text{sonst} \end{cases} \end{array}$$

$$\begin{array}{rcl} r & : & dom(\mathcal{I}) \to dom(\mathcal{I}) \\ r^{\mathcal{I}}(u) & := & \begin{cases} y_{i_1} \ldots y_{i_m} & \text{falls } u = i_1 \ldots i_m \in \{1, \ldots, k\}^* \\ \varepsilon & \text{sonst} \end{cases} \end{array}$$

Man sieht leicht, daß $\mathcal{I}$ die Formel B_P wahr macht:

- $\exists x.\ell(x) = \varepsilon \wedge r(x) = \varepsilon$ ist erfüllt, da $\ell^{\mathcal{I}}(\varepsilon^{\mathcal{I}}) = \varepsilon = \varepsilon^{\mathcal{I}}$ und $r^{\mathcal{I}}(\varepsilon^{\mathcal{I}}) = \varepsilon = \varepsilon^{\mathcal{I}}$.
- Sei $u \in dom(\mathcal{I})$:
  - $u = i_1 \ldots i_m \in \{1, \ldots, k\}^*$:
    $$\begin{array}{rcccl} \ell^{\mathcal{I}}(f_j^{\mathcal{I}}(u)) & = & \ell^{\mathcal{I}}(i_1 \ldots i_m j) & = & x_{i_1} \ldots x_{i_m} x_j \\ & & & & \shortparallel \\ t_{x_j}^{\mathcal{I}}(\ell^{\mathcal{I}}(u)) & = & t_{x_j}^{\mathcal{I}}(x_{i_1} \ldots x_{i_m}) & = & x_{i_1} \ldots x_{i_m} x_j \end{array}$$
  - $u \notin \{1, \ldots, k\}^*$:
    $$\begin{array}{rcccl} \ell^{\mathcal{I}}(f_j^{\mathcal{I}}(u)) & = & \ell^{\mathcal{I}}(j) & = & x_j \\ & & & & \shortparallel \\ t_{x_j}^{\mathcal{I}}(\ell^{\mathcal{I}}(u)) & = & t_{x_j}^{\mathcal{I}}(\varepsilon) & = & x_j \end{array}$$
- Für r geht das entsprechend.

Also erfüllt $\mathcal{I}$ die Formel B_P. Damit muß $\mathcal{I}$ auch $\exists z.(\ell(z) = r(z) \neq \varepsilon)$ erfüllen, d.h. es gibt ein $u \in dom(\mathcal{I})$ mit $\ell^{\mathcal{I}}(u) = r^{\mathcal{I}}(u) \neq \varepsilon^{\mathcal{I}} = \varepsilon$.

1. Fall: $u = i_1 \ldots i_m \in \{1, \ldots, k\}^+$. Dann ist

$$\begin{array}{rcl} \ell^{\mathcal{I}}(u) & = & x_{i_1} \ldots x_{i_m} \\ \shortparallel & & \\ r^{\mathcal{I}}(u) & = & y_{i_1} \ldots y_{i_m} \end{array}$$

d.h. $i_1 \ldots i_m$ ist eine Lösung des PKP.

2. Fall: $u \in \Sigma^*$.

- $u = \varepsilon : \ell^{\mathcal{I}}(u) = \varepsilon = r^{\mathcal{I}}(u)$ kann nicht sein
- $u \in \Sigma^+$: Dann ist ebenfalls $\ell^{\mathcal{I}}(u) = \varepsilon = r^{\mathcal{I}}(u)$, d.h. auch dieser Fall kann nicht eintreten.

„$\Leftarrow$“: Angenommen, P hat eine Lösung. Dann sei $\mathcal{I}$ eine Interpretation, die B_P wahr macht. Zu zeigen: $\mathcal{I}$ erfüllt auch $\exists z.(\ell(z) = r(z) \neq \varepsilon)$. Es sei $i_1 \dots i_m$ für $m > 0$ eine Lösung von P, d.h. $x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}$.

Da $\mathcal{I}$ die Formel B_P erfüllt, gilt:
$$\begin{array}{lcll} \ell^{\mathcal{I}}(f^{\mathcal{I}}_{i_m}(\cdots f^{\mathcal{I}}_{i_1}(\varepsilon^{\mathcal{I}}))) & = & t^{\mathcal{I}}_{x_{i_1}\dots x_{i_m}} & \neq \varepsilon^{\mathcal{I}} \\ r^{\mathcal{I}}(f^{\mathcal{I}}_{i_m}(\cdots f^{\mathcal{I}}_{i_1}(\varepsilon^{\mathcal{I}}))) & = & t^{\mathcal{I}}_{y_{i_1}\dots y_{i_m}} & \neq \varepsilon^{\mathcal{I}} \end{array}$$

Aus $x_{i_1} \dots x_{i_m} = y_{i_1} \dots y_{i_m}$ folgt $t^{\mathcal{I}}_{x_{i_1}\dots x_{i_m}} = t^{\mathcal{I}}_{y_{i_1}\dots y_{i_m}}$, d.h. $f^{\mathcal{I}}_{i_m}(\cdots f^{\mathcal{I}}_{i_1}(\varepsilon^{\mathcal{I}}))$ ist das gesuchte z. $\square$

Teil 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/Komplexität:

Rechenzeit (bei TM: Anzahl der Übergänge bis zum Halten)

Speicherplatz (bei TM: Anzahl der benutzten Felder)

soll abgeschätzt werden als Funktion in der Größe der Eingabe.

Man kann die Komplexität eines speziellen Algorithmus/Programms betrachten (wie in der Vorlesung „Algorithmen und Datenstrukturen") oder die Komplexität eines Entscheidungsproblems: Wieviel Aufwand benötigt der „beste" Algorithmus im „schlimmsten" Fall? Wir betrachten hier den zweiten Fall.

18 Komplexitätsklassen

Wir führen im folgenden den Begriff der Komplexitätsklasse allgemein ein und definieren dann einige fundamentale Zeit- und Platzkomplexitätsklassen. Zunächst führen wir formal ein, was es heißt, daß der Aufwand durch eine Funktion der Größe der Eingabe beschränkt ist.

Definition 18.1 ($\mathrm{f}(n)$-zeitbeschränkt, $\mathrm{f}(n)$-platzbeschränkt)
Es sei $\mathrm{f} : \mathbb{N} \to \mathbb{N}$ *eine Funktion und* $\mathcal{A}$ *eine DTM über* Σ.

1) $\mathcal{A}$ *heißt* $\mathrm{f}(n)$*-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* $\mathrm{f}(n)$*-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.
Wir betrachten im folgenden nicht (wie in „Algorithmen und Datenstrukturen") die Komplexität einzelner Algorithmen (DTMs), sondern die Komplexität von Entscheidungsproblemen, wobei wir uns o.B.d.A. auf einstellige Relationen (d.h. $L \subseteq \Sigma^*$) beschränken.

Definition 18.2 (Komplexitätsklassen)

$$\begin{array}{rcl} DTIME(\mathrm{f}(n)) & := & \{L \mid \textit{es gibt eine } \mathrm{f}(n)\textit{-}\underline{\textit{zeit}}\textit{beschränkte } \underline{\textit{DTM}}\textit{, die } L \; \underline{\textit{entscheidet}}\} \\ NTIME(\mathrm{f}(n)) & := & \{L \mid \textit{es gibt eine } \mathrm{f}(n)\textit{-}\underline{\textit{zeit}}\textit{beschränkte } \underline{\textit{NTM}}\textit{, die } L \; \underline{\textit{akzeptiert}}\} \\ DSPACE(\mathrm{f}(n)) & := & \{L \mid \textit{es gibt eine } \mathrm{f}(n)\textit{-}\underline{\textit{platz}}\textit{beschränkte } \underline{\textit{DTM}}\textit{, die } L \; \underline{\textit{entscheidet}}\} \\ NSPACE(\mathrm{f}(n)) & := & \{L \mid \textit{es gibt eine } \mathrm{f}(n)\textit{-}\underline{\textit{platz}}\textit{beschränkte } \underline{\textit{NTM}}\textit{, die } L \; \underline{\textit{akzeptiert}}\} \end{array}$$

Satz 18.3

1) $DTIME(\mathrm{f}(n)) \subseteq DSPACE(\mathrm{f}(n)) \subseteq NSPACE(\mathrm{f}(n))$

2) $DTIME(\mathrm{f}(n)) \subseteq NTIME(\mathrm{f}(n))$

Beweis:
Offenbar kann man in k Schritten höchstens k Felder benutzen. Außerdem ist eine DTM ja auch eine NTM. □

Wir betrachten die folgenden fundamentalen Komplexitätsklassen:

Definition 18.4 ($P, NP, PSPACE$)

$$\begin{array}{rcl} P & := & \bigcup\limits_{\mathrm{p}\ \textit{Polynom in}\ n} DTIME(\mathrm{p}(n)) \\ NP & := & \bigcup\limits_{\mathrm{p}\ \textit{Polynom in}\ n} NTIME(\mathrm{p}(n)) \\ PSPACE & := & \bigcup\limits_{\mathrm{p}\ \textit{Polynom in}\ n} NSPACE(\mathrm{p}(n)) \end{array}$$

Bei der Definition von $PSPACE$ hätte man genausogut $DSPACE$ statt $NSPACE$ verwenden können, d.h. $PSPACE = \bigcup\limits_{\text{p Polynom in } n} DSPACE(\mathrm{p}(n))$, denn:
Wir hatten gesehen, daß man jede NTM durch eine DTM simulieren kann. Nach einem Resultat von Savitch ist der Platzbedarf der DTM quadratisch im Platzbedarf der NTM (ohne Beweis).

Bemerkung 18.5
Die Bedeutung der in Definition 18.4 eingeführten Komplexitätsklassen ergibt sich u.a. aus den folgenden Überlegungen:

1) Alle üblichen berechnungsuniversellen Maschinenmodelle lassen sich auf TM in Polynomialzeit simulieren, d.h.: Benötigt das Maschinenmodell zur Berechnung n Schritte, so benötigt die simulierende TM $\mathrm{q}(n)$ *Schritte (*q *Polynom). Es folgt, daß die in Definition 18.4 eingeführten Komplexitätsklassen unabhängig vom speziellen Maschinenmodell sind.*

2) Probleme, die in P *sind, werden i.a. als effizient lösbar („tractable", d.h. „machbar") bezeichnet. Dies liegt daran, daß Polynome im Vergleich zu Exponentialfunktionen* ($n \mapsto 2^n$) *ein recht moderates Wachstum haben.*
Aber: In Anwendungen, bei denen man für sehr große Eingaben sehr schnell (in Echtzeit) Antworten haben will, ist ein Polynom mit hohem Grad häufig auch nicht mehr tolerierbar.

3) *NP enthält viele Probleme, für die derzeit nur deterministische Entscheidungsverfahren mit exponentiellem Zeitaufwand bekannt sind. Man bezeichnet diese Probleme daher häufig als nicht effizient lösbar („intractable“).*
   Aber: Die eingeführten Komplexitätsklassen (wie NP) betrachten stets den schwierigsten Fall („worst-case“-Komplexität). Es kann durchaus sein, daß es wegen gewisser pathologischer Situationen keine polynomiale Schranke für die Rechenzeit gibt, aber für typische Fälle oder im Mittel doch polynomiales Verhalten erzielt werden kann.

Wie ist das Verhältnis zwischen den Komplexitätsklassen? Aus Satz 18.3 ergibt sich:

$$P \subseteq NP \subseteq PSPACE$$

Ob diese Inklusionen echt sind, ist bisher nicht bekannt. Die Frage, ob P gleich NP ist (das $P = NP$-Problem), ist eines der fundamentalen offenen Probleme der Informatik, dessen Beantwortung weitreichende Konsequenzen hätte.

19 NP-vollständige Probleme

Es geht hier um Probleme in NP, die „mindestens so schwer“ wie jedes Problem in NP sind. Ein solches Problem ist SAT:

Definition 19.1 (SAT)
SAT ist die Menge der erfüllbaren („satisfiable“) Booleschen Ausdrücke mit Variablen x_i ($i \geq 1$, kodiert als Dualzahl bei Eingabe für TM) und Operatoren $\neg, \wedge, \vee$.

Beispiel:

- $(x_1 \wedge x_2) \vee \neg x_1$ 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.

Lemma 19.2
$SAT \in NP$

Beweis:
Eine NTM akzeptiert die Elemente von SAT wie folgt:

1) Teste, ob die Eingabe ein Boolescher Ausdruck ist (entspricht dem Wortproblem für kontextfreie Grammatiken, polynomial entscheidbar).

2) Durchlaufe den Ausdruck und ersetze nichtdeterministisch jede Variable x_i durch 0 oder 1 (polynomialer Aufwand).

3) Werte den Ausdruck aus und akzeptiere, falls das Ergebnis 1 ist (geht polynomial). □

Inwiefern ist nun SAT „das schwierigste“ NP-Problem?

Definition 19.3 (polynomialzeitberechenbar, -reduzierbar, NP-vollständig)

1) *Die Funktion* $\mathrm{f} : \Sigma^* \to \Sigma^*$ *heißt polynomialzeitberechenbar, falls es ein Polynom* p *und eine* $\mathrm{p}(n)$*-zeitbeschränkte DTM gibt, die* f *berechnet.*

2) $L \subseteq \Sigma^*$ ist polynomialzeitreduzierbar auf $L' \subseteq \Sigma^*$ (geschrieben als $L \leq_p L'$), falls es eine polynomialzeitberechenbare Funktion f *gibt mit*

$$w \in L \text{ gdw. } \mathrm{f}(w) \in L'$$

für alle $w \in \Sigma^*$.

3) L_0 heißt NP-vollständig, falls gilt:

- *$L_0 \in NP$ (L_0 ist in NP)*
- *Für alle $L \in NP$ gilt: $L \leq_p L_0$ (L_0 ist NP-hart, d.h. mindestens so hart zu lösen wie jedes andere NP-Problem.)*

Satz 19.4

1) Ist L_0 NP-vollständig, so gilt:

$$L_0 \in P \Rightarrow P = NP$$

(Es ist bisher kein NP-vollständiges L_0 bekannt, das in P ist!)

2) SAT ist NP-vollständig.

Beweis:

1) Es sei L_0 NP-vollständig und $L_0 \in P$, d.h. L_0 wird von einer p(n)-zeitbeschränkten DTM (für ein Polynom p) entschieden.
Wir müssen zeigen, daß daraus $NP \subseteq P$ folgt. Sei also $L \in NP$. Es ist daher $L \leq_p L_0$, d.h. es gibt ein f, das mit Zeitaufwand $\leq \mathrm{q}(n)$ berechenbar ist (für Polynom q), so daß

$$w \in L \text{ gdw. } \mathrm{f}(w) \in L_0.$$

Die DTM, welche L entscheidet, geht wie folgt vor:

- Bei Eingabe w berechnet sie f(w). Sie benötigt $\leq \mathrm{q}(|w|)$ viel Zeit. Daher ist auch die erzeugte Ausgabe $\leq |w| + \mathrm{q}(|w|)$.
- Wende Entscheidungsverfahren für L_0 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.

2) Wir beschreiben nur kurz die Idee:
Zu zeigen ist: $\forall L : L \in NP \Rightarrow L \leq_p SAT$. Sei $\mathcal{A}$ eine polynomialzeitbeschränkte NTM, welche $L \subseteq \Sigma^*$ akzeptiert. Die gesuchte Reduktionsfunktion f weist jedem Wort $w \in \Sigma^*$ einen Booleschen Ausdruck β_w zu, so daß

- Der Übergang von w zu β_w ist in Polynomialzeit berechenbar.
- $\mathcal{A}$ akzeptiert w gdw. β_w ist erfüllbar.

Die genaue Definition von β_w ist sehr aufwendig. Es muß ja die Arbeitsweise von $\mathcal{A}$ auf Eingabe w durch einen Booleschen Ausdruck beschrieben werden.
Idee:

- Aussagenvariablen $Band_{t,i,a}$ für „Zum Zeitpunkt t steht auf Feld i das Symbol a“. Da $\mathcal{A}$ polynomialzeitbeschränkt ist, muß man nur polynomial viele Bandzellen und Zeitpunkte betrachten!
- Entsprechend Aussagenvariablen $Kopf_{t,i}$ („Kopf zum Zeitpunkt t auf Feld i“), $Zustand_{t,q}$ („Zustand zum Zeitpunkt t ist q“).
- Übergänge: (p, a, a', r, q) liefert z.B.:
  $Band_{t,i,a} \wedge Kopf_{t,i} \wedge Zustand_{t,p} \longrightarrow Band_{t+1,i,a'} \wedge Kopf_{t+1,i+r} \wedge Zustand_{t+1,q}$.
- Kodierung der Eingabe w zum Zeitpunkt 0.
- Kodierungsbedingungen
  z.B. $\neg(Band_{t,i,a} \wedge Band_{t,i,b})$ für $a \neq b$ etc. □

Weitere NP-vollständige Probleme erhält man durch polynomielle Reduktion.

Satz 19.5

1) Ist $L_2 \in NP$ 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 $L_2 \in NP$ gibt es eine polynomialzeitbeschränkte NTM $\mathcal{A}$, welche L_2 akzeptiert und wegen $L_1 \leq_p L_2$ gibt es eine polynomialzeitberechenbare Funktion f mit $w \in L_1$ gdw. $\mathrm{f}(w) \in L_2$. Die polynomialzeitbeschränkte NTM für L_2 arbeitet wie folgt:

- Bei Eingabe w berechnet sie zunächst $\mathrm{f}(w)$.
- Dann wendet sie $\mathcal{A}$ auf $\mathrm{f}(w)$ an.

2) Wir müssen zeigen, daß für alle $L \in NP$ gilt: $L \leq_p L_2$. Die polynomialzeitberechenbare Reduktionsfunktion f mit $w \in L$ gdw. $\mathrm{f}(w) \in L_2$ erhält man wie folgt:

- Da L_1 NP-hart ist, gibt es eine polynomialzeitberechenbare Funktion g mit $w \in L$ gdw. $\mathrm{g}(w) \in L_1$.
- Wegen $L_1 \leq_p L_2$ gibt es eine polynomialzeitberechenbare Funktion h mit $u \in L_1$ gdw. $\mathrm{h}(u) \in L_2$.

Wir definieren $\mathrm{f}(w) := \mathrm{h}(\mathrm{g}(w))$. Dann gilt: $w \in L$ gdw. $\mathrm{g}(w) \in L_1$ gdw. $\mathrm{h}(\mathrm{g}(w)) \in L_2$. □

Zunächst zeigen wir, daß auch ein Teilproblem von SAT, bei dem man nur Boolesche Ausdrücke einer ganz speziellen Form zuläßt, bereits NP-hart ist.

Definition 19.6 ($3SAT$)
Eine 3-Klausel ist von der Form $l_1 \vee l_2 \vee l_3$, wobei l_i eine Variable oder eine negierte Variable ist. Ein $3SAT$-Problem ist eine endliche Konjunktion von 3-Klauseln. $3SAT$ ist die Menge der erfüllbaren $3SAT$-Probleme.

Satz 19.7
$3SAT$ ist NP-vollständig.

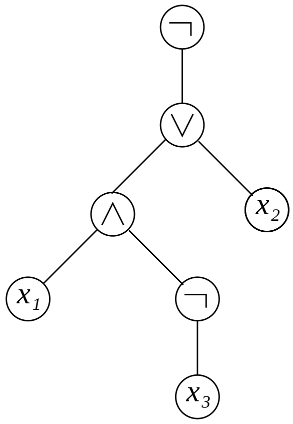
Beweis:

1) $3SAT \in NP$ folgt unmittelbar aus $SAT \in NP$, da jedes $3SAT$-Problem ein Boolescher Ausdruck ist.

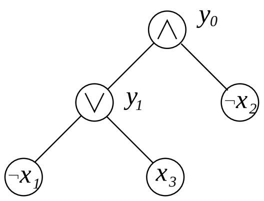
2) Es sei F ein beliebiger Boolescher Ausdruck. Wir geben ein polynomielles Verfahren an, das F in ein $3SAT$-Problem F' umwandelt, so daß gilt: F erfüllbar gdw. F' erfüllbar.
Beachte: Es ist nicht nötig, daß F und F' (als Boolesche Ausdrücke) äquivalent sind.
Die Umformung erfolgt in mehreren Schritten, die wir am Beispiel des Ausdrucks

$$\neg((x_1 \wedge \neg x_3) \vee x_2)$$

veranschaulichen. Wir stellen diesen Ausdruck als Baum dar:



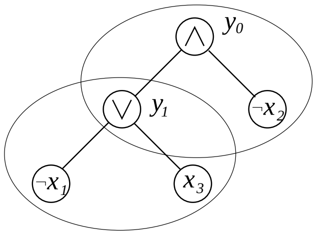
1. Schritt: Wende de Morgan an, um die Negationszeichen zu den Blättern zu schieben.

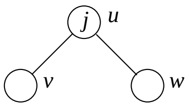


Dies erfordert nur einen Durchlauf durch den Baum, ist also in linearer Zeit realisierbar.

2. Schritt: Ordne jedem inneren Knoten eine neue Variable aus $\{y_0, y_1, \ldots\}$ zu, wobei die Wurzel y_0 erhält.

3. Schritt: Fasse jede Verzweigung (gedanklich) zu einer Dreiergruppe zusammen:

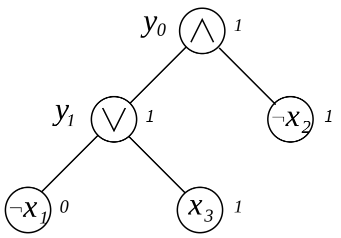


Jeder Verzweigung der Form 

mit $j \in \{\wedge, \vee\}$ ordnen wir eine Formel der Form $(u \Leftrightarrow (v\ j\ w))$ zu. Diese Formeln werden nun konjunktiv mit y_0 verknüpft, was die Formel F_1 liefert.

Im Beispiel: $y_0 \wedge (y_0 \Leftrightarrow (y_1 \wedge \neg x_2)) \wedge (y_1 \Leftrightarrow (\neg x_1 \vee x_3))$

Die Ausdrücke F und F_1 sind erfüllbarkeitsäquivalent, denn:
Eine erfüllende Belegung für F kann zu einer für F_1 erweitert werden, indem man die Werte für die Variablen y_i durch die Auswertung der entsprechenden Teilformel bestimmt, z.B.:



$x_1 = 1$
$x_2 = 0$
$x_3 = 1$
$y_1 = 1$
$y_0 = 1$

Umgekehrt ist jede erfüllende Belegung für F_1 auch eine für F.

4. Schritt: Jede Konjunktion in F_1 wird separat in eine konjunktive Normalform umgeformt:

$$\begin{aligned}
a \Leftrightarrow (b \vee c) &\leadsto (\neg a \vee (b \vee c)) \wedge (\neg(b \vee c) \vee a) \\
&\leadsto (\neg a \vee b \vee c) \wedge (\neg b \vee a) \wedge (\neg c \vee a) \\
&\leadsto (\neg a \vee b \vee c) \wedge (\neg b \vee a \vee a) \wedge (\neg c \vee a \vee a)
\end{aligned}$$

$$\begin{aligned}
a \Leftrightarrow (b \wedge c) &\leadsto (\neg a \vee (b \wedge c)) \wedge (\neg(b \wedge c) \vee a) \\
&\leadsto (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee a) \\
&\leadsto (\neg a \vee b \vee b) \wedge (\neg a \vee c \vee c) \wedge (\neg b \vee \neg c \vee a)
\end{aligned}$$

Dadurch erhält man Formeln der gewünschten $3SAT$-Form.
Beachte: Pro Teilformel ist der Aufwand hierfür konstant. □

Ein weiteres interessantes NP-vollständiges Problem ist das Mengenüberdeckungsproblem.

Definition 19.8 (Mengenüberdeckung)
Gegeben ist ein Mengensystem über einer endlichen Grundmenge M, d.h. $T_1, \ldots, T_k \subseteq M$ sowie eine Zahl $n \leq k$.
Frage: Gibt es eine Auswahl von n Mengen, die ganz M überdecken, d.h. $\{i_1, \ldots, i_n\} \subseteq \{1, \ldots, k\}$ mit $T_{i_1} \cup \ldots \cup T_{i_k} = M$.

Satz 19.9 *Mengenüberdeckung ist NP-vollständig.*

Beweis:

1) Mengenüberdeckung ist in *NP*: Wähle nichtdeterministisch Indices $i_1, \ldots, i_n$ und überprüfe, ob $T_{i_1} \cup \ldots \cup T_{i_k} = M$ gilt.

2) Um NP-Härte zu zeigen, reduzieren wir $3SAT$ auf Mengenüberdeckung. Sei also $F = K_1 \wedge \ldots \wedge K_m$ eine Instanz von $3SAT$, welche 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

$$\begin{aligned}
T_i &:= \{j \mid x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{m+i\} \\
T_i' &:= \{j \mid \neg x_i \text{ kommt in } K_j \text{ als Disjunkt vor}\} \cup \{m+i\}
\end{aligned}$$

Wir betrachten das Mengensystem:

$$T_1, \ldots, T_n, T_1', \ldots, T_n'$$

a) Für eine gegebene Belegung der Variablen, welche F erfüllt, wählen wir

- T_i, falls $x_i = 1$
- T'_i, falls $x_i = 0$

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.

b) Sei umgekehrt $\{U_1, \ldots, U_n\} \subseteq \{T_1, \ldots, T_n, T'_1, \ldots, T'_n\}$ 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. O.B.d.A. sei $U_i \in \{T_i, T'_i\}$ für $i = 1, \ldots, n$.
Wir definieren nun die Belegung:

$$x_i = \begin{cases} 1 & \text{falls} \quad U_i = T_i \\ 0 & \text{falls} \quad U_i \neq T_i \end{cases}$$

Diese erfüllt jedes K_j, da j in $U_1 \cup \ldots \cup U_n$ vorkommt. $\square$

Viele Probleme für Graphen sind NP-vollständig, z.B. auch das folgende:

Definition 19.10 (Clique)
Gegeben: ein gerichteter Graph $G = (V, E)$ und eine Zahl $k \in \mathbb{N}$
Frage: Besitzt G eine k-Clique, d.h. eine Teilmenge $U \subseteq V$ mit

- *$|U| = k$ und*
- *für alle $u \neq v$ in U gilt: $(u, v) \in E$.*

Satz 19.11
Clique ist NP-vollständig.

Beweis:

1) In NP: Rate eine Teilmenge der Größe k und teste, ob sie eine k-Clique ist.

2) NP-hart: Wir reduzieren $3SAT$ auf Clique.

Sei also $F = \overbrace{(l_{11} \vee l_{12} \vee l_{13})}^{K_1} \wedge \ldots \wedge \overbrace{(l_{m1} \vee l_{m2} \vee l_{m3})}^{K_m}$ mit $l_{ij} \in \{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$.
Der Graph G wird nun wie folgt definiert:

- $V := \{(i, j) \mid 1 \leq i \leq m \text{ und } 1 \leq j \leq 3\}$
- $E := \{((i, j), (p, q)) \mid i \neq p \text{ und } l_{ij} \neq \overline{l}_{pq}\}$, wobei $\overline{l} = \begin{cases} \neg x & \text{falls} \quad l = x \\ x & \text{falls} \quad l = \neg x \end{cases}$.
- $k = m$

Es gilt nun:
F 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 $l_{1j_1}, \ldots, l_{mj_m}$, die paarweise nicht komplementär sind
gdw.
es gibt Knoten $(1, j_1), \ldots, (m, j_m)$, die paarweise miteinander verbunden sind
gdw.
es gibt eine k-Clique in G $\square$

Bemerkung:
Entsprechend zur Definition von NP-vollständig kann man auch PSPACE-vollständig definieren. Das Äquivalenzproblem für NEAs/reguläre Ausdrücke ist PSPACE-vollständig.

Abkürzungsverzeichnis

Abb.	Abbildung
bzgl.	bezüglich
bzw.	beziehungsweise
ca.	circa
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
ggf.	gegebenenfalls
HP	Halteproblem
i.a.	im allgemeinen
kf	kontextfrei
ks	kontextsensitiv
LBA	linear beschränkter Automat
MPKP	modifiziertes Postsches Korrespondenzproblem
NEA	nichtdeterministischer endlicher Automat
NP	nichtdeterministisch polynomiell
NTM	nichtdeterministische Turingmaschine
o.a.	oben angegeben
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
usw.	und so weiter
vgl.	vergleiche
z.B.	zum Beispiel
$\square$	was zu beweisen war

Literaturverzeichnis

[Schö_97] U. Schöning. *Theoretische Informatik – kurzgefaßt.* Spektrum Akademischer Verlag Berlin, Heidelberg, 1997

[Thom_89] W. Thomas. *Grundzüge der Theoretischen Informatik.* Skript zur Vorlesung an der RWTH Aachen, 1989

[Wege_93] I. Wegener. *Theoretische Informatik.* Teubner-Verlag, 1993