Informatik III

(Automatentheorie und formale Sprachen)

Prof. Dr. Jürgen Dix

Zeit und Ort:

Vorlesung: Montag und Dienstag, jeweils 10–12 im T1.
Übung: Di, vierzehntäglich (**Tobias Ahlbrecht**, Hiwis: A. Mantel/K. Böhm). **Anmeldung unter StudIP**.

Vorlesungsunterlagen

`https://studip.tu-clausthal.de/...`

Regelmäßig besuchen! Dort finden Sie alles Wichtige über die Vorlesung, Dokumente, Übungen et cetera.

Prüfungsklausur: tba

Zulassung zur Klausur: >50% der Übungspunkte, und bei maximal einem Hausübungsblatt weniger als 20%.

Es gibt **Bonuspunkte** für die Klausur, wenn deutlich mehr als die notwendigen Punkte erzielt wurden.

Outline

Danksagung

Historie

Diese Vorlesung orientiert sich stark an "Theoretische Informatik" von Katrin Erk und Lutz Priese.

Vielen Dank an Katrin Erk, die mir vor vielen Jahren ihr gesamtes Material zur Verfügung gestellt hat. Dank auch an Michaela Huhn, die weitere Folien beigesteuert hat.

Der Teil über P/NP stammt im Wesentlichen aus Hopcroft/Ullmann und wurde von mir mehrmals in Manchester gelesen.

Organisatorisches

- Zulassung zur Klausur: (1) 50 % der Übungspunkte (jedes Übungsblatt muss mit mindestens 20 % bestanden werden), und (2) jeder Student muss mindestens zweimal an der Tafel eine Aufgabe erfolgreich vorgerechnet haben.
- Übungen können zu Gruppen von maximal zwei Studenten abgegeben werden. Bei Plagiaten werden alle identischen Blätter mit 0 Punkten bewertet.
- Zur Klausur sind keine Bücher, Folien oder Mitschriften zugelassen. **Lediglich ein handgeschriebenes Din A4 Blatt ist erlaubt.**
- Die Übungszettel sind jeweils bis Montags, 13 Uhr in den Zettelkasten zu legen.
- Für diejenigen die deutlich mehr als die 50 % schaffen, gibt es Bonuspunkte. Diese werden bei der Klausur angerechnet: falls nah an der Grenze zur besseren Note steht, bekommt man sie (man muss allerdings bestanden haben).

Worum geht es eigentlich? (1)

Um Grundbegriffe der theoretischen Informatik:

Sprachen: das sind Mengen von Zeichenketten (eine Zeichenkette heißt auch Wort),

Grammatiken: das sind Kalküle, um Sprachen zu erzeugen (jedes Wort wird erzeugt),

Automaten: das sind Maschinen, die Sprachen definieren oder transformieren (indem sie testen, ob ein Wort zu einer Sprache gehört, Sprachen erzeugen, oder Eingabe-Sprachen in Ausgabe-Sprachen übersetzen).

Warum? (1)

Sprachen und ihre Verarbeitung sind ein zentrales Thema der Informatik.

- Programmiersprachen und ihre Compiler;
- Skriptsprachen und Interpreter, Protokolle;
- Eingabesprachen für Webdienste, mobile Dienste, . . . ;
- Werkzeuge für die Textverarbeitung, Engineering-Aufgaben oder domänenspezifische Sprachen;
- . . .

Sprachen und Automaten gehören in den Werkzeugkoffer jedes Informatikers!

Worum geht es eigentlich? (2)

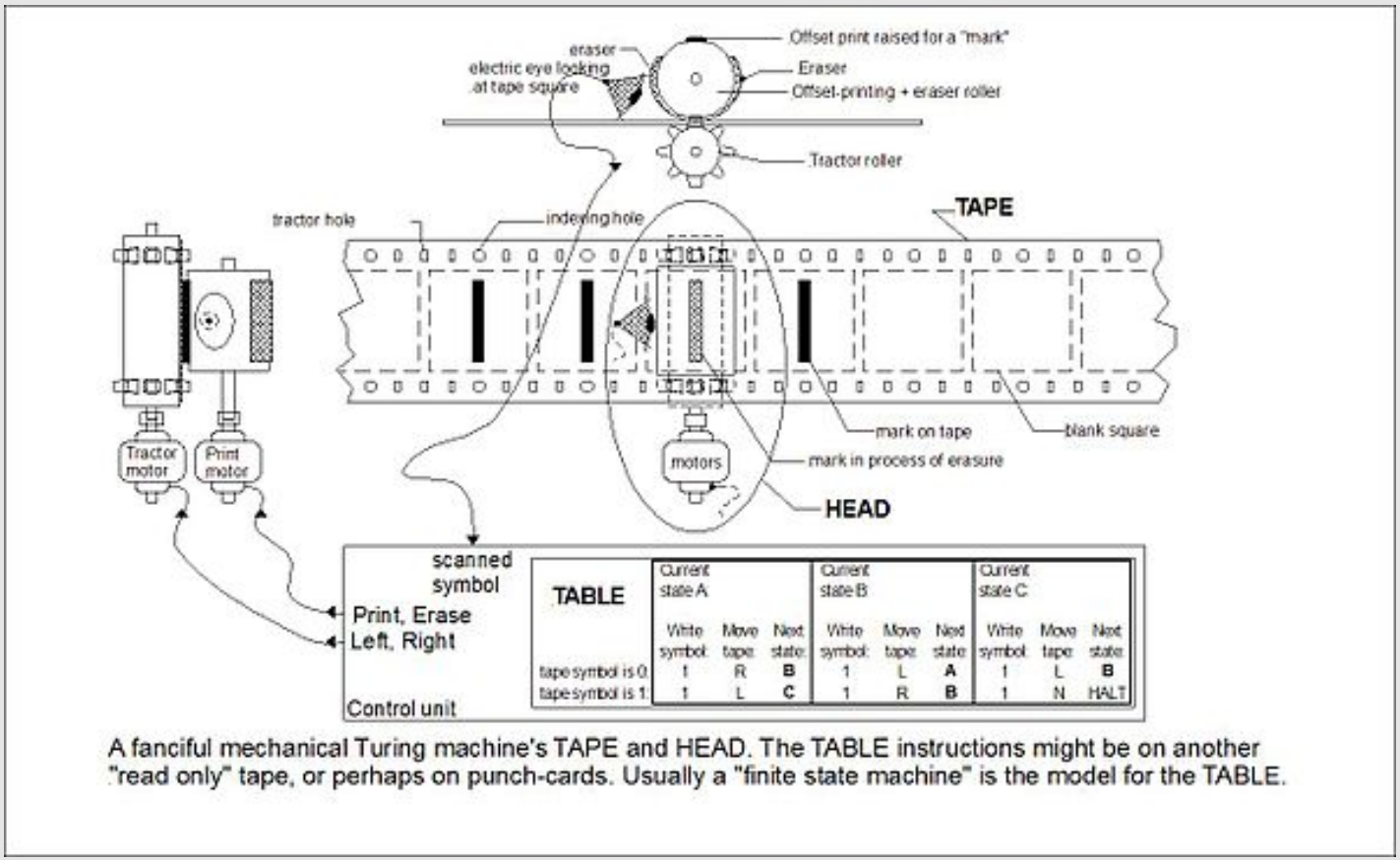
Präzise Definition des Algorithmus-Begriffs: ein Algorithmus ist der Programmcode einer Turingmaschine.

Der Algorithmusbegriff ist universell, d.h. auch mit anderen Berechnungsmodellen wie While- oder GoTo- Programmen, μ-rekursiven Funktionen oder Termersetzungssystemen funktioniert es.

Wir zeigen, daß das Halte- und viele weitere Probleme unentscheidbar sind, d.h. sie können grundsätzlich nicht berechnet werden.

Warum? (2)

Berechenbarkeitstheorie hat die Mathematik in der ersten Hälfte des 20. Jahrhunderts erschüttert!



A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

Abbildung 1: Turing-Maschine ©Wikipedia

Worum geht es eigentlich? (3)

Auf Basis einer präzisen Definition des Algorithmus-Begriffs, des Programmcodes einer Turingmaschine, kann die Komplexität von Problemen untersucht werden:

Wir führen Komplexitätsklassen ein. Die wichtigsten sind die Klassen $\mathbf{P}$ und $\mathbf{NP}$.

Wir zeigen von vielen Problemen, daß sie dieselbe Komplexität haben, ohne diese Komplexität genau zu kennen (sie sind $\mathbf{NP}$ vollständig).

Warum? (3)

Wie entwickeln sich Speicherplatz und Rechenzeit bei großen Probleminstanzen?

Abbildung 2: Traveling Salesperson Instanz in Google Maps

Lernstrategie

JedeR[a] kann 10 km Laufen[b]!

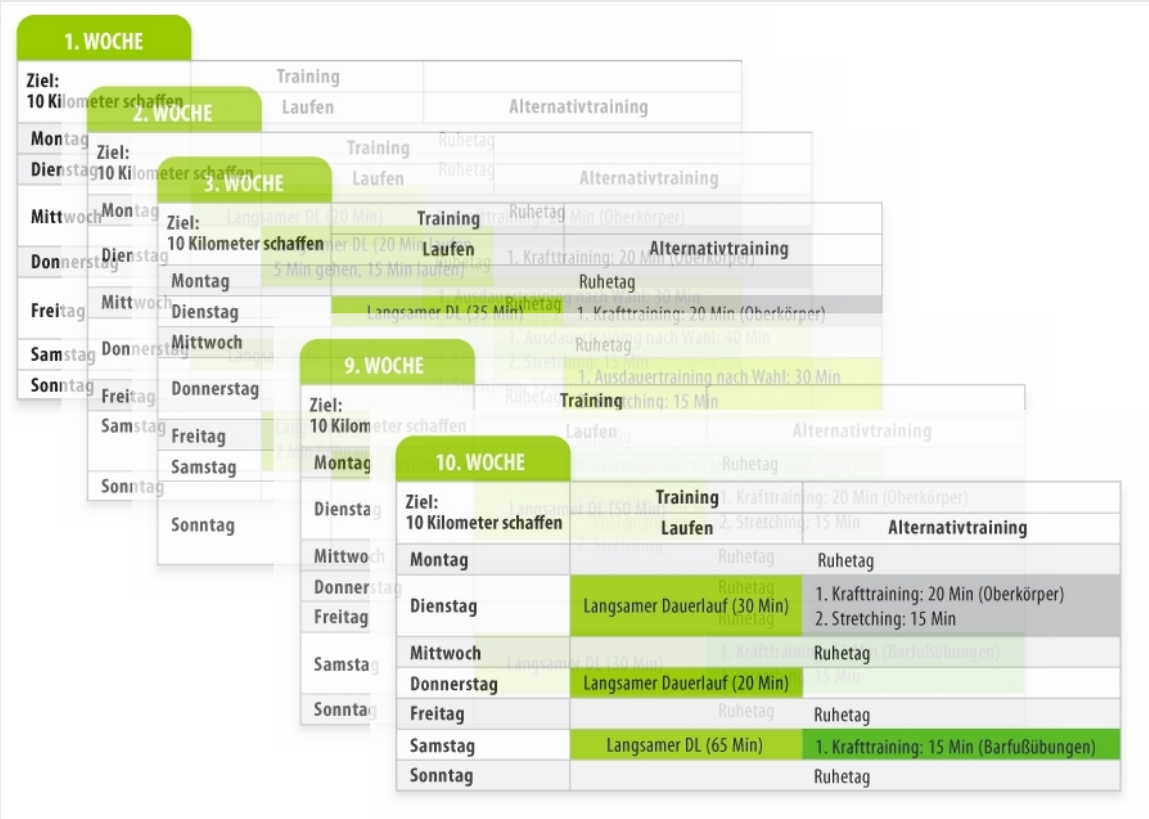
Abbildung 3: ©www.lauf-infos.de

wenn...

[a] gesund, Alter zwischen 15 und 50,...
[b] Ziel: 10 km schaffen, nicht U60', U50' oder gar U35'

Zur Lernstrategie

Wenn er/sie 10 - 12 Wochen konsequent trainiert!



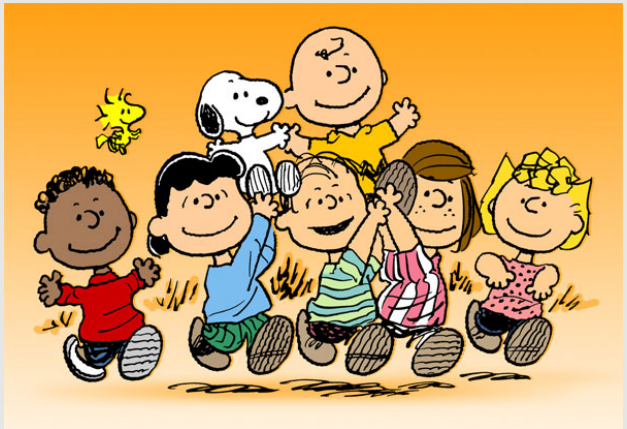
10. WOCHE

Ziel: 10 Kilometer schaffen	Training	
	Laufen	Alternativtraining
Montag		Ruhetag
Dienstag	Langsamer Dauerlauf (30 Min)	1. Krafttraining: 20 Min (Oberkörper) 2. Stretching: 15 Min
Mittwoch		Ruhetag
Donnerstag	Langsamer Dauerlauf (20 Min)	
Freitag		Ruhetag
Samstag	Langsamer DL (65 Min)	1. Krafttraining: 15 Min (Barfußübungen)
Sonntag		Ruhetag

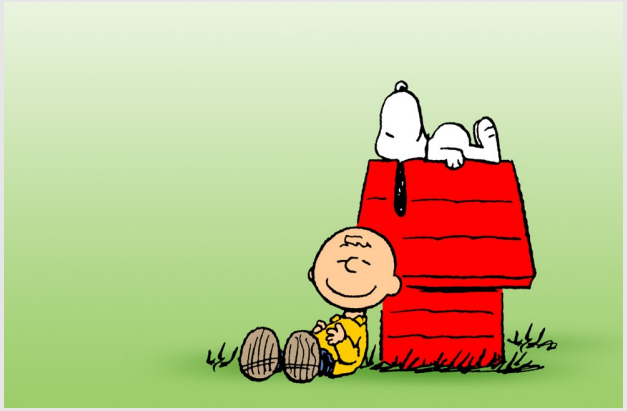
Abbildung 4: ©www.joggen-online.de

Gründe für Erfolg oder Mißerfolg:

Konsequente Teilnahme am Training $\Rightarrow$ 10 km Lauf $\surd$



Einstieg verpasst, Trainingslücken,. . . $\Rightarrow$ ✘



Für einen 10 km Lauf und für Info III gilt:

Trainieren **Sie** - jetzt!

- **Das erste Übungsblatt ist schon in zwei Wochen fällig.**
- **Ebenso die erste Übung.**

Literatur I

Erk, Katrin and Priese, Lutz (2001).
Theoretische Informatik: Eine umfassende Einführung.
Springer.

Hoffmann, Dirk (2011).
Theoretische Informatik.
Carl Hanser.

Hopcroft, J. and J. Ullman (1979).
Introduction to Automata Theory, Languages, and Computation.
Reading, MA: Addison Wesely.

Lin, S. and T. Rado (1965).
Computer studies of turing machine problems.
Journal of the ACM 12(2), 196–212.

Literatur II


Papadimitriou, C. (1994).
Computational Complexity.
Addison-Wesley.


Rado, T. (1962).
On non-computable functions.
The Bell System Technical Journal XLI(3), 877–884.


Turing, A. M. (1936).
On Computable Numbers with an Application to the Entscheidungsproblem.
Proc. London Mathematical Soc., series 2 42, 230–265.
corrections ibid., 43:544–546.

1. Terminologie

Motivation

In diesem Kapitel führen wir die Begriffe

- Sprache und
- Grammatik

ein.

In der gesamten Vorlesung geht es darum zu untersuchen, welche Sprachen man durch welche Grammatiken ausdrücken kann (und welche nicht).

Inhalt

Wir untersuchen insbesondere:

1. Wie man Probleme aus der Mathematik (Graphentheorie, Logik) als Probleme über Sprachen formulieren kann.
2. Wie man Klassen von Grammatiken von steigendem Schwierigkeitsgrad definiert: Chomsky-Hierarchie.
3. Wieviele Grammatiken und Sprachen gibt es überhaupt (soviele wie natürliche Zahlen, reelle Zahlen oder komplexe Zahlen)?

1.1 Sprache, Grammatik

Grundlage einer Sprache ist das *Alphabet*. Es legt die zur Verfügung stehenden Zeichen (auch **Buchstaben** genannt) fest. Ein Alphabet ist oft **endlich**.
Eine Sprache besteht aus *Wörtern*: endliche Sequenzen von Buchstaben (Zeichenreihen). Es gibt mehrere Operationen auf Wörtern und auf Sprachen.

Alphabet: Σ, endlich;

Wort: w, auch **String** oder **Zeichenreihe** genannt, ist immer über einem Alphabet Σ definiert: es ist eine **endliche Folge** von Symbolen aus Σ. Die Länge eines Wortes (bezeichnet mit $|w|$) ist die Anzahl seiner Buchstaben. ε bezeichnet das leere Wort: es hat als einziges die Länge 0.
Achtung: ε ist ein **Metasymbol** für ein Wort, kein Symbol aus dem Alphabet Σ.

Sprache: **Eine Sprache L ist eine Menge von Wörtern** über einem Alphabet Σ. Auf der Menge aller Sprachen $\mathrm{FS}(\Sigma)$ definieren wir folgende Operationen: (sei $L, M \in \mathrm{FS}(\Sigma)$)

1. **Konkatenation ($\circ$):**
   $L \circ M := \{w \circ w' : w \in L,\ w' \in M\}$,
   oft läßt man "$\circ$" weg: LM, ww'.
2. **i-te Potenz von L (^{i}):** $L^0 := \{\varepsilon\}$, $L^{i+1} := LL^i$,
3. **Kleene-Hülle von L (*):** $L^* := \bigcup_{i=0,1,2,\ldots} L^i$,
4. **Variante der Kleene-Hülle von L (L^+):**
   $L^+ := LL^*$. Man zeigt leicht, daß
   $L^+ = \bigcup_{i=1,2,\ldots} L^i$.
5. **Reverse (^{R}):** w^R bezeichnet das Wort w rückwärts gelesen: $(abbcd)^R := dcbba$. Analog ist $L^R := \{w^R : w \in L\}$.
   Dabei ist $\varepsilon^R := \varepsilon$.

Beispiel 1.1 (Sprache der gerade Zahlen)

Σ	$\{0, 1, 2, 3, \ldots, 9\}$
L	Menge der gerade Zahlen
z.B.	$2, 4, 6, 8, 10, 12, 14, \ldots$
aber nicht	$1, 3, 5, \ldots, 111, 113, \ldots$

Tabelle 1: Gerade Zahlen

Beispiel 1.2 (Aussagenlogische Ausdrücke)

Σ	$\{x, y, z, \wedge, \vee, \neg, (,)\}$
L	Menge der aussagenlogischen Ausdrücke über den Variablen x, y und z
z.B.	$x \wedge y$, $\neg y$, $(x \vee y) \vee (z \wedge x), \ldots$
aber nicht	$xx, \neg yz, \wedge\vee, \neg, \ldots$

Tabelle 2: Sprache: Aussagenlogische Ausdrücke

Σ^* ist also die **Menge aller Wörter über Σ**.
$\Sigma^+ := \Sigma^* \setminus \{\varepsilon\}$ ist die Menge aller Wörter über Σ der Länge echt größer als 0.

Mithilfe der obigen Operationen bildet man **reguläre Ausdrücke**, z.B.:

$$(aba)^*bb^* \ + \ (aba)^*a \ + \ (ba).$$

Definition 1.3 (Reguläre Ausdrücke $\mathfrak{Reg}_\Sigma$)

Reguläre Ausdrücke sind wie folgt definiert:

1. 0 ist ein **regulärer Ausdruck**.
2. Für jedes $a \in \Sigma$ ist a ein **regulärer Ausdruck**.
3. Sind r und s **reguläre Ausdrücke**, so auch
   - $(r + s)$ (Vereinigung),
   - (rs) (Konkatenation),
   - (r^*) (Kleene Stern).

Wir lassen die Klammern weg und vereinbaren, daß * Vorrang vor der Konkatenation hat, die wiederum Vorrang vor $+$ hat.

Definition 1.4 (Semantik regulärer Ausdrücke)

Ein **regulärer Ausdruck** r stellt eine Sprache $\mathfrak{I}(r)$ über Σ wie folgt dar:

- $\mathfrak{I}(0) := \emptyset$,
- $\mathfrak{I}(a) := \{a\}$, für $a \in \Sigma$,
- $\mathfrak{I}(r+s) := \mathfrak{I}(r) \cup \mathfrak{I}(s)$,
- $\mathfrak{I}(rs) := \mathfrak{I}(r)\mathfrak{I}(s)$,
- $\mathfrak{I}(r^*) := \mathfrak{I}(r)^*$.

Oft benutzen wir auch a^+ in einem regulären Ausdruck (beachte, daß dies ein **Makro** ist). Wir benutzen "1", definiert durch

$$1 := 0^*$$

Man rechnet leicht nach, daß $\mathfrak{I}(1) = \{\varepsilon\}$. **Vorsicht:** Falls $0, 1 \in \Sigma$, kann es zu Verwechslungen kommen.

Übung:

Welche Sprachen stellen die folgenden regulären Ausdrücke dar?

- aa,
- $(a+b)^*$,
- $aa^* + bb^*$.

Beachten Sie, daß gilt:

- $\{abb\}c^*\{b\} = abbc^*b$,
- $\{aba\}^*\{a\} \cup \{ba\} = (aba)^*a + ba$.

Nur die Ausdrücke auf den rechten Seiten sind reguläre Ausdrücke. Wir benutzen im folgenden beide Schreibweisen, um Sprachen bzw. Mengen auszudrücken.

Beispiel 1.5 (Sprache der gerade Zahlen)

Σ	$\{0, 1, 2, 3, \ldots, 9\}$
L	Menge der gerade Zahlen
z.B.	$2, 4, 6, 8, 10, 12, 14, \ldots$
aber nicht	$1, 3, 5, \ldots, 111, 113, \ldots$
reg. Ausdruck	$(1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^*(0+2+4+6+8)+(0+2+4+6+8)$

Tabelle 3: Gerade Zahlen als regulärer Ausdruck

Eine Grammatik ist ein Kalkül um eine Sprache zu definieren. Also eine Menge von Regeln, mit deren Hilfe man Wörter ableiten kann.
Die zu einer Grammatik gehörende **Sprache** besteht aus den **ableitbaren, terminalen Wörtern**.

Definition 1.6 (Σ-Grammatik)

Eine **Grammatik** G über einem Alphabet Σ (kurz **Σ-Grammatik**) ist ein Tupel $G = (V, T, R, S)$. Dabei ist

- V eine endliche Menge von **Variablen**;
- $T \subseteq \Sigma$ eine endliche Menge von **Terminalen** mit $V \cap T = \emptyset$;
- R eine endliche Menge von **Regeln**. Eine Regel ist ein Element (P, Q) aus $\left((V \cup T)^* \, V \, (V \cup T)^*\right) \times (V \cup T)^*$. Das heißt,
  - P (die Prämisse) ist ein Wort über $(V \cup T)$, das mindestens eine Variable aus V enthält,
  - Q (die Conclusio) ist ein beliebiges Wort über $(V \cup T)$.

  Für eine Regel $(P, Q) \in R$ schreibt man üblicherweise auch $P \to_G Q$ oder nur $P \to Q$.
- S das Startsymbol, $S \in V$.

Oft sagen wir einfach "Grammatik G", da Σ vorher festgelegt wurde und sich nicht ändert.

Beispiel 1.7

$S \to B \qquad B \to do\ begin\ B\ end \qquad B \to A$
$A \to nop\ A \qquad A \to \varepsilon$

- Wir schreiben normalerweise Variablen als Großbuchstaben,
- Terminale als Kleinbuchstaben.
- Wenn es mehrere Regeln mit derselben Prämisse gibt, also z.B. $P \to Q_1$, $P \to Q_2$, $P \to Q_3$, so schreibt man dafür auch kurz

$$P \to Q_1 \mid Q_2 \mid Q_3.$$

Rechnung einer Grammatik:

- Beginn beim Startsymbol, einer Variablen.
  Dies ist ein Wort.
- Eine Grammatikregel $P \to Q$ sagt, daß ein Vorkommen von P **durch** Q **ersetzt** werden darf:
  Wenn das aktuelle Wort die Prämisse P enthält, dann kann man P durch die dazugehörige Conclusio Q ersetzen. Das Ergebnis ist ein neues aktuelles Wort.
  Eine Prämisse muss eine Variable enthalten.
  Umgekehrt ist ein Wort, das noch eine Variable enthält, nicht “fertig”.
- Am Ende steht ein terminales Wort: also eines, in dem keine Variablen vorkommen. Es besteht nur aus Terminalen.

Beispiel 1.8 (Einfache Grammatiken)

- Sei $G_a = (\{S\}, \{a\}, \{R_1, R_2\}, S)$ eine Grammatik mit $R_1 = S \to aS$, $R_2 = S \to \epsilon$.
- Sei $G_{ab} = (\{S\}, \{a, b\}, \{R_1, R_2\}, S)$ eine Grammatik mit $R_1 = S \to aSb$, $R_2 = S \to \epsilon$.
- Sei $G_{gerade} = (\{S, S_0\}, \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \{R_1, R_2\}, S)$ eine Grammatik mit $R_1 = S \to 1S \,|\, 2S_0 \,|\, 3S \,|\, 4S_0 \,|\, 5S \,|\, 6S_0 \,|\, 7S \,|\, 8S_0 \,|\, 9S$, $R_2 = S_0 \to S \,|\, \epsilon$.

Welche Wörter kann man ableiten?

Definition 1.9 (Ableitung, Rechnung)

- Ist $G = (V, T, R, S)$ eine Grammatik, w, w' Wörter aus $(V \cup T)^*$,

so gilt "$\mathbf{w \Longrightarrow w'}$", $\mathbf{w}$ geht über in $\mathbf{w'}$, falls gilt: es gibt $u, v \in (V \cup T)^*$, und $P \to Q \in R$ mit

$$\big(w = uPv \text{ und } w' = uQv\big).$$

Für "w geht über in w' ...

- ... mit der Grammatik G" schreibt man auch: $w \Longrightarrow_G w'$
- ... durch die Regel $P \to Q$" schreibt man auch: $w \Longrightarrow_{P \to Q} w'$.

Falls es Wörter $w_0, \ldots, w_n \in (V \cup T)^*$ gibt mit $w = w_0$, $w_n = w'$ und $w_i \Longrightarrow_G w_{i+1}$ für $0 \leq i < n$, so schreiben wir dafür auch $w \Longrightarrow_G^* w'$.

$n = 0$ ist erlaubt, das heißt, $w \Longrightarrow_G^* w$ gilt stets.

Die Folge $w_0, \ldots, w_n$ heißt **Ableitung** oder **Rechnung** (von w_0 nach w_n in G) der Länge n.

Vorsicht Indeterminismus:

Es kann mehrere Wege geben, um zum gleichen Wort zu kommen.

Beispiel 1.10 (Indeterminismus)

Wir betrachten die Grammatik

$G = (\{S, B\}, \{a, b, c\}, \{R_0, R_1, R_2, R_3\}, S)$

$$
\begin{aligned}
R_0 &= S \to aBBc \\
R_1 &= B \to b \\
R_2 &= B \to ba \\
R_3 &= BB \to bBa
\end{aligned}
$$

Drei Möglichkeiten, das Wort $abbac$ zu erzeugen:

$$
\begin{array}{llllllll}
S & \Longrightarrow_{R_0} & aBBc & \Longrightarrow_{R_1} & abBc & \Longrightarrow_{R_2} & abbac \\
S & \Longrightarrow_{R_0} & aBBc & \Longrightarrow_{R_2} & aBbac & \Longrightarrow_{R_1} & abbac \\
S & \Longrightarrow_{R_0} & aBBc & \Longrightarrow_{R_3} & abBac & \Longrightarrow_{R_1} & abbac
\end{array}
$$

Wenn ein Wort w mehrere Regel-Prämissen enthält, gibt es mehrere mögliche weitere Rechnungen.

Warum ist das ein *feature* und kein *bug*?

- Erlaubt oft einfachere Definitionen von Grammatiken.
- Manchmal gibt es keine eindeutige Grammatiken.
- Eine Grammatik beschreibt die Struktur der Wörter.
  Diese Struktur kann etwas bedeuten.
  Wenn dann die Grammatik nicht eindeutig ist, heißt das, daß manche Wörter mehrere mögliche Strukturen haben.
- Für **natürliche Sprachen** braucht man das unbedingt.
  Manche Sätze sind mehrdeutig, also müssen auch die Grammatiken mehrdeutig sein!

Time flies like an arrow.
Fruit flies like a banana.

Definition 1.11 (Erzeugte Sprache $L(G)$, Äquivalenz)

Die von einer Grammatik G **erzeugte Sprache** $\boldsymbol{L(G)}$ ist die Menge aller **terminalen** Wörter, die durch G vom Startsymbol S aus erzeugt werden können:

$$L(G) := \{w \in T^* \mid S \Longrightarrow_G^* w\} \tag{1}$$

Zwei Grammatiken G_1, G_2 heißen **äquivalent**, falls $L(G_1) = L(G_2)$.

Übung

Wir betrachten die Grammatik

$$G_2 = (\{S\}, \{a, b\}, \{R_1, R_2\}, S)$$

mit $R_1 = S \to aSb$, $R_2 = S \to \varepsilon$ von Beispiel 1.8. Um ein Wort zu erzeugen, starten wir mit dem Startsymbol S:

$$S \Longrightarrow_{R_1} aSb \Longrightarrow_{R_1} aaSbb \Longrightarrow_{R_1} aaaSbbb \Longrightarrow_{R_2} aaabbb$$

Damit haben wir mit G_2 das Wort a^3b^3 erzeugt. Es lassen sich durch mehr oder weniger Anwendungen von R_1 auch andere Wörter hervorbringen.

Lemma 1.12

G_{ab}, die Grammatik aus Beispiel 1.8, erzeugt die Sprache $L(G_{ab}) = \{a^n b^n \mid n \in \mathbb{N}_0\}$.

Beweis

Daß G_{ab} tatsächlich genau diese Sprache erzeugt, zeigen wir allgemein, indem wir alle möglichen Ableitungen von G_{ab} betrachten.

$\subseteq$: zu zeigen: **Jedes terminale Wort, das G_{ab} erzeugt, hat die Form $a^n b^n$.**
Wir zeigen für alle $w \in (V \cup T)^*$: Falls $S \Longrightarrow^*_{G_{ab}} w$, dann gilt entweder $w = a^n S b^n$ oder $w = a^n b^n$ für ein $n \in \mathbb{N}_0$.

Dazu verwenden wir eine Induktion über die Länge einer Ableitung von S nach w.

Induktionsanfang: $w = S = a^0Sb^0$

Induktionsschritt: Es gelte $S \Longrightarrow^*_{G_{ab}} w \Longrightarrow_{G_{ab}} w'$, und für w gelte nach der Induktionsvoraussetzung bereits $w = a^nb^n$ oder $w = a^nSb^n$. Außerdem sei $w \Longrightarrow_{G_{ab}} w'$ eine Ableitung in einem Schritt. Nun ist zu zeigen: $w' = a^mb^m$ oder $w' = a^mSb^m$ für irgendein m.

Fall 1: $w = a^n b^n$. Dann konnte keine Regel angewandt werden, da w schon terminal ist, also tritt dieser Fall nie auf.

Fall 2: $w = a^n S b^n$. Dann wurde von w nach w' entweder Regel R_1 oder R_2 angewandt.
Falls R_1 angewandt wurde, dann gilt
$w = a^n S b^n \Longrightarrow_{R_1} a^n a S b b^n = a^{n+1} S b^{n+1} = w'$.
Falls R_2 angewandt wurde, dann gilt
$w = a^n S b^n \Longrightarrow_{R_2} a^n \varepsilon b^n = w'$. Dies Wort ist terminal und hat die geforderte Form $a^n b^n$.

$\supseteq$: zu zeigen: Für alle n kann $a^n b^n$ von G_{ab} erzeugt werden:
$S \Longrightarrow^*_{G_{ab}} a^n b^n \; \forall n \in \mathbb{N}_0$.
Um $a^n b^n$ zu erzeugen, wende man auf S n-mal die Regel R_1 und dann einmal die Regel R_2 an. $\square$

Definition 1.13 (Dycksprache D_k)

Für $k \in \mathbb{N}$ sei $V_k := \{x_1, \overline{x_1}, x_2 \ldots, x_k, \overline{x_k}\}$ ein Alphabet über den $2k$ Symbolen $x_1, \overline{x_1}, \ldots, x_k, \overline{x_k}$. Die Dycksprache D_k ist die kleinste Menge die folgende Bedingungen erfüllt:

1. $\epsilon \in D_k$,
2. Falls $w \in D_k$, so auch $x_i w \overline{x_i}$.
3. Falls $u, v \in D_k$, so auch uv.

Interpretiert man die x_i als öffnende, die $\overline{x_i}$ als zugehörige schließende Klammern, so kann man die Dycksprache als die Menge aller korrekten Klammerausdrücke sehen.

1.2 Warum Sprachen?

Einige interessante Mengen wurden schon als Sprachen beschrieben, z.B. gerade Zahlen, Klammersprachen.
In der Einleitung wurde versprochen:

Relevante Probleme der Mathematik und Informatik!

Diesem Anspruch versuchen wir jetzt gerecht zu werden:

Fakt 1.14

Viele Fragestellungen der Informatik und der Mathematik können als Probleme über Sprachen formalisiert werden.

Die folgenden Beispiele zeigen, wie man Probleme als Sprachen formalisieren kann.

Primzahlen

Kann man die Menge der Primzahlen als formale Sprache über einem Alphabet sehen?

Beispiel 1.15 (Primzahlen)

Σ	$\{\,\vert\,\}$
L_{prim}	Menge der Primzahlen
z.B.	$\vert\vert, \vert\vert\vert, \vert\vert\vert\vert\vert, \vert\vert\vert\vert\vert\vert\vert, \vert\vert\vert\vert\vert\vert\vert\vert\vert\vert, \ldots$
aber nicht	$\vert, \vert\vert\vert\vert, \vert\vert\vert\vert\vert\vert, \vert\vert\vert\vert\vert\vert\vert, \vert\vert\vert\vert\vert\vert\vert\varepsilon, \vert, \ldots$

Tabelle 4: Primzahlen in Unärdarstellung

Statt | können wir natürlich auch 0, 1, l oder ★ benutzen.

Wichtig:

Gebraucht wird ein Eingabealphabet $\Sigma := \{0, 1, \ldots, n-1\}$, um eine Ganzzahl zur Basis n darzustellen ($n > 1$). Zum Beispiel wird die Zahl 5 binär durch 101 dargestellt. Für die unäre Darstellung benutzt man $|||||$ oder auch 11111 (man könnte auch 00000 benutzen um es mit der Darstellung oben konsistent zu machen).

Anmerkung:

Die n-äre Darstellung ($n > 1$) einer Zahl m führt zu einer Speicherersparnis: Statt m Zellen eines Speicherbandes zu benutzen (in Unärdarstellung) werden nur $\log_n m$ benötigt.

Trotzdem ist nur der Schritt von der unären zur binären Darstellung – also von m nach $\lg m$ – eine wirkliche Verbesserung.

Der Umstieg auf $\log_n m$ lässt nur die Einsparung eines (linearen) Faktors zu.

- Wie kann man die Menge aller **booleschen Ausdrücke** der Form $x_3 \vee x_1 \wedge x_2 \vee \neg x_1$ als Sprache darstellen?
- Wie kann man die **Länge der Formel** $x_3 \vee x_1 \wedge x_2 \vee \neg x_1$ definieren?

Boolesche Ausdrücke

Idee: Eine Variable x_i wird durch das Symbol x gefolgt vom Index i in Binärnotation kodiert. x_{11} ist damit $\rightarrow x11$.

Beispiel 1.16 (Boolesche Audrücke)

Σ_{sat}	$\{\wedge, \vee, \neg, (,), x, 0, 1\}$
L_{bool}	Menge der Booleschen Ausdrücke
z.B.	$x1 \vee x101, (x11 \vee x1) \wedge (x10 \vee \neg x1), \ldots$
aber nicht	$\wedge, \neg\vee, x(), x101x10x11, \ldots$

Tabelle 5: Boolesche Ausdrücke mit Variablenindizes in Binärdarstellung

Boolesche Ausdrücke

Welches Alphabet benötigen wir um boolesche Formeln (wie etwa $x_1 \vee (x_2 \wedge x_4)$) zu kodieren?

$\Sigma_{\text{sat}} := \{\wedge, \vee, \neg, (,), x, 0, 1\}$. Dies genügt, um einen booleschen Ausdruck w mit den Variablen $\{x_1, x_2, \ldots\}$ zu definieren. Eine Variable x_i wird durch das Zeichen x gefolgt von der Zahl i in binärer Notation kodiert.
Wie üblich wird ein ganzer Ausdruck abhängig von der Belegung der Variablen als **t** (true) or **f** (false) ausgewertet.

Ein wichtiges Problem ist es, von einem booleschen Ausdruck zu entscheiden, ob er erfüllbar ist: satisfiability-problem (SAT).

Frage

Existiert eine Belegung der Variablen im Ausdruck w derart, daß w als **t** ausgewertet wird?

Wie kann man dies als Sprache (oder Grammatik) darstellen?

Frage

Angenommen, es läge ein boolescher Ausdruck (in der üblichen Form) mit n Symbolen vor. Wie lang ist die kodierte Version in Σ_{sat}?

Es können nur $\lceil \frac{n}{2} \rceil$ Variablen vorkommen, jede benötigt nicht mehr als $1 + \lceil \lg n \rceil$ Symbole: also höchstens

$$\begin{aligned} &\leq \lceil \tfrac{n}{2} \rceil (1 + \lceil \lg n \rceil) + \lceil \tfrac{n}{2} \rceil \\ &\leq \lceil \tfrac{n}{2} \rceil (2 + \lceil \lg n \rceil) \\ &\leq \lceil \tfrac{n}{2} \rceil (2 \lceil \lg n \rceil) \\ &\leq n \lceil \lg n \rceil \end{aligned}$$

Fakt 1.17

Alle späteren Komplexitäts-Betrachtungen werden unabhängig davon sein, **ob nun n oder $n \lg n$ als Länge der Eingabe** *betrachtet wird.*

Wir definieren:

Definition 1.18 (Satisfiability)

$$L_{\text{sat}} := \{w \in \Sigma^*_{sat} : \text{es gibt Belegungen für } x_i \text{ so, daß die Formel } w \text{ wahr ist}\}$$

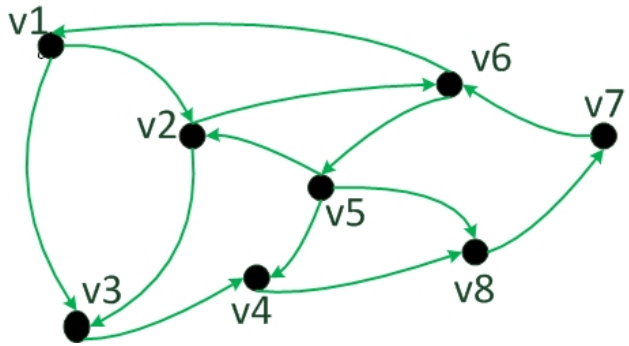
Graphen

Definition 1.19 (Graph)

Ein **Graph** $\mathcal{G} = \langle V, E \rangle$ besteht aus

- einer Menge von Knoten V, und
- $E \subseteq V \times V$, der Kantenmenge.

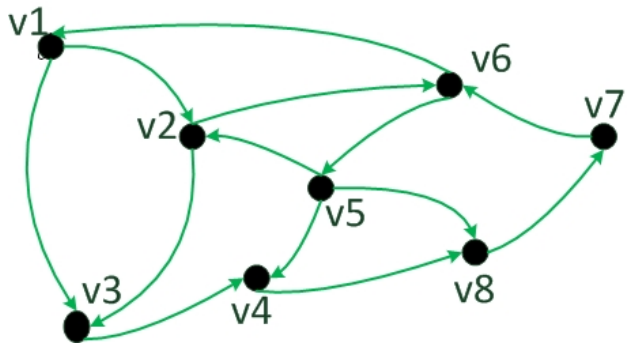
Wenn für alle $u, v \in V$ gilt: $(u, v) \in E \Leftrightarrow (v, u) \in E$, dann heißt $\mathcal{G}$ ein ungerichteter Graph, sonst heißt $\mathcal{G}$ gerichtet.



Frage: Erreichbarkeitsproblem

Gibt es einen Weg in $\mathcal{G}$ von v_i zu v_j?

Graphen



Beispiel 1.20 (Graphsprache)

Alphabet: $\Sigma_{\text{graph}} := \{v, e, 0, 1, (,), \#\}$.

v_i wird repräsentiert durch die Zeichenkette v gefolgt vom Index binär kodiert.
Eine gerichtete Kante $e_{i,j}$ wird kodiert als Zeichenkette $(\text{string1}\#\text{string2})$, wobei $string1$ die binäre Darstellung von i und $string2$ die binäre Darstellung von j ist.

Prüfen Sie Ihr Verständnis: Ergänzen Sie fehlende Details zur Darstellung von Graphen, und schreiben Sie den obigen Graphen als Wort über Σ_{graph} auf!

Definition 1.21 (Erreichbarkeitsproblem L_{reach})

Das Erreichbarkeitsproblem ist wie folgt als formale Sprache definiert:

$$L_{\text{reach}} := \{w \in \Sigma^*_{\text{graph}} : \begin{array}{l}\text{es gibt einen Weg in } w \\ \text{von der ersten Ecke } v_1 \\ \text{zur letzten Ecke } v_n\}\end{array}$$

Ganzzahlige lineare Programmierung

Gegeben sei eine $m \times n$ Matrix $\mathbf{A}$ über $\mathbb{Z}$ und ein Spaltenvektor $\mathbf{b}$.

Frage:

Gibt es einen Spaltenvektor $\mathbf{x}$, der $\mathbf{Ax} \geq \mathbf{b}$ erfüllt?

Repräsentation: Wörter der Sprache sind die Einträge von $\mathbf{A}$ und $\mathbf{b}$: alle werden binär dargestellt.

Definition 1.22

$$L_{\text{ILP}} := \{w \in \Sigma^*_{\text{ILP}} : \quad \text{w repräsentiert ein ILP-Problem } \langle \mathbf{A}, \mathbf{b} \rangle \text{ so, daß es ein } \mathbf{x} \text{ mit } \mathbf{Ax} \geq \mathbf{b} \text{ gibt.}\}$$

Die Beispiele dienen als Beleg für Fakt 1.14.[1]

Fakt 1.14

Viele Fragestellungen der Informatik und der Mathematik können als Probleme über Sprachen formalisiert werden.

Die Darstellung als formale Sprache bringt uns aber nur weiter, wenn wir die aufgeworfenen Probleme mit Info-III-Mitteln auch lösen können!

[1] Zweifler formalisieren noch einige weitere Probleme, wie "n ist die Summe zweier Quadratzahlen".

Was haben wir bis jetzt erreicht?

- Definition von **formalen Sprachen**.
- Reguläre Ausdrücke sind ein endliches Beschreibungsmittel für manche Sprachen.
- Eine Grammatik ist ein Kalkül zur Erzeugung formaler Sprachen: ein weiteres endliches Beschreibungsmittel.
- Ein Wort wird durch Ableitung von einer Grammatik erzeugt. Damit kann man — oft ganz einfach — zeigen, daß ein Wort zu einer Sprache gehört.
- Beweise über Ableitungsfolgen (strukturelle Induktion). Damit kann man — oft mit etwas Aufwand — zeigen, daß bestimmte Wörter nicht von einer Grammatik erzeugt werden.

Beispiel 1.23 (Grammatik-Ausschnitt: Dangling Else)

$S \to \texttt{if}\, E\, \texttt{then}\, S\, \texttt{else}\ S$
$S \to \texttt{if}\, E\, \texttt{then}\, S$
$S \to \texttt{other\_stat}$ `//sonstige Anweisungen`
$E \to \texttt{b}$ `//boolescher Ausdruck`
`//Terminale: if, then, else, b, other_stat`

Leiten Sie das folgende Wort ab

`ifbthenifbthenother_statelseother_stat`

- Stellen Sie Ihre Ableitung als Strukturbaum dar.
- Wieviele Ableitungen gibt es?
- Warum ist das ein Problem?

Grammatiken sind die Grundlage für das Parsen von (Programmier-) Sprachen.

1.3 Die Chomsky-Hierarchie

Unsere Grammatik-Definition ist sehr allgemein!

Einzige Einschränkungen:

- nur endlich viele Regeln und
- jede Regelprämisse muss mindestens eine Variable enthalten.

Das lässt viel Flexibilität, führt aber leicht zu Wildwuchs![2]

Daher wollen wir jetzt die Regeln, die in einer Grammatik zulässig sind, beschränken.

Dann erhält man Grammatiktypen und damit auch Sprachklassen von verschiedenen Schwierigkeitsgraden.

[2] z.B.: Ein Wort kann im Lauf der Ableitung beliebig in der Länge wachsen und wieder schrumpfen.

Definition 1.24 (Sprachklassen)

Eine Grammatik $G = (V, T, R, S)$ heißt

rechtslinear, falls für alle $P \to Q \in R$:
$\big(P \in V$ und $Q \in T^* \cup T^+V\big)$.

Das heißt:

- Es wird eine einzelne Variable ersetzt.
- Mit einer Regelanwendung wird höchstens eine Variable erzeugt.
- Wenn eine Regelanwendung eine Variable erzeugt, steht sie ganz rechts im Wort.

kontextfrei (cf) falls für alle $P \to Q \in R$:
$\big(P \in V$ und $Q \in (V \cup T)^*\big)$.

Das heißt:

- Es wird eine einzelne Variable ersetzt.
- Was links und rechts von der Variablen steht, kann man in der Prämisse nicht kontrollieren.
- Das Wort in der Conclusio kann Variablen und Terminale in beliebiger Mischung enthalten.

kontextsensitiv (cs), falls für alle $P \to Q \in R$ gilt

- **entweder** $\Big(\exists u, v, \alpha \in (V \cup T)^* \; \exists A \in V$ $\big(P = uAv$ und $Q = u\alpha v$ mit $|\alpha| \geq 1\big)\Big)$, **oder** die Regel hat die Form $S \to \varepsilon$
- S kommt in keiner Regelconclusio vor.

Das heißt:

- Eine Variable A wird in einen String α mit einer Länge von mindestens 1 überführt. Das Wort wird nicht kürzer, außer $\varepsilon \in L$.
- Diese Ersetzung von A durch α findet aber nur statt, wenn der in der Regel geforderte **Kontext**, links u und rechts v, im Wort **vorhanden** ist.

beschränkt, falls für alle $P \to Q \in R$ gilt:

- **entweder** $\left(|P| \leq |Q|\right)$ **oder** die Regel hat die Form $S \to \varepsilon$
- S kommt in keiner Regelconclusio vor.

Das heißt:

- Die Conclusio jeder Regel ist mindestens so lang wie die Prämisse (außer falls $\varepsilon \in L$).
- Das heißt, das Wort kann im Lauf der Ableitung nur wachsen, nie kürzer werden.

Aufbauend auf diesen Grammatikarten kann man nun Sprachklassen definieren:

Definition 1.25 (Sprachklassen)

Klasse	definiert als	Sprache heißt
$\mathbf{L_3}$, RAT	$\{L(G) \mid G$ ist rechtslinear$\}$	Typ 3, regulär
$\mathbf{L_2}$, CFL	$\{L(G) \mid G$ ist cf$\}$	Typ 2, kontextfrei
$\mathbf{L_1}$, CSL	$\{L(G) \mid G$ ist cs$\}$	Typ 1, kontextsensitiv
	$\{L(G) \mid G$ ist beschränkt$\}$	beschränkt
$\mathbf{L_0}$, r.e.	$\{L(G) \mid G$ beliebig$\}$	Typ 0, aufzählbar
$\mathbf{L}$	$\{L \mid L \subseteq \Sigma^* \}$	beliebige Sprache

Grammatiken können kompliziert sein!

Beispiel 1.26 ($a^n b^n c^n$)

Sei $G_{abc} = (\{S, X_1, X_2\}, \{a, b, c\}, \{R_1, \ldots, R_5\}, S)$ eine Grammatik mit

$$\begin{array}{lllll} R_1 = & S & \to & abc \mid aX_1bc \\ R_2 = & X_1b & \to & bX_1 \\ R_3 = & X_1c & \to & X_2bcc \\ R_4 = & bX_2 & \to & X_2b \\ R_5 = & aX_2 & \to & aa \mid aaX_1 \end{array}$$

- Ist diese Grammatik **kontextsensitiv**?
- Ist sie **beschränkt**?

1.4 Probleme über Sprachen

Oft interessiert uns die Frage, ob ein gegebenes Wort in einer Sprache (definiert durch eine Grammatik) enthalten ist. Oder ob zwei Grammatiken dieselbe Sprache erzeugen. Natürlich interessiert dann auch, ob es einen Algorithmus gibt, um solch ein Problem zu lösen. Das führt uns dazu, eine vorläufige, intuitive Definition für die beiden Begriffe Problem und Algorithmus zu geben.

Definition 1.27 (Problem, Algorithmus)

Ein **Problem P** ist eine Frage, ob eine bestimmte Eigenschaft auf gegebene Objekte zutrifft.
Die Objekte sind aus einer bekannten, abzählbaren Grundmenge. Ferner gilt für jedes Objekt o: die Eigenschaft trifft auf o zu oder nicht.
Ein **Algorithmus** für ein Problem P ist eine Vorschrift (ein Programm), die zu beliebigem Objekt o berechnet, ob die Eigenschaft für o zutrifft oder nicht.

Beispiel 1.28 (Einige Probleme)

- Ob ein Wort aus einer Grammatik ableitbar ist (Wortproblem), oder ob die zu einer Grammatik gehörende Sprache leer (endlich, unendlich) ist.
- Ob eine Zahl prim ist. Ob für gegebenes n die Gleichung $a^n + b^n = c^n$ eine Lösung in den natürlichen Zahlen hat.
- Ob ein Java-Programm auf eine Eingabe terminiert (Halteproblem).

1.5 Endlich, unendlich und dann?

Wie charakterisiert man die Mächtigkeit einer Menge?

- Abzählen der Elemente funktioniert gut bei endlichen Mengen.
- Nur was machen wir mit $\mathbb{N}^2$ oder $\mathbb{R}$?

Definition 1.29 (Mächtigkeit von Mengen, Kardinalität)

Zwei nichtleere Mengen M sind N sind **gleichmächtig** (haben die gleiche Kardinalität), geschrieben $|N| = |M|$, wenn es eine **bijektive Abbildung** $f : M \to N$ gibt.

Eine Menge M ist **mindestens so mächtig** wie eine Menge N, geschrieben $|N| \leq |M|$, wenn es eine **surjektive** Abbildung $f : M \to N$ gibt.

- Offensichtlich gilt auch mit dieser Definition: $|\{1, 2, 3\}| < |\{a, b, c, d\}|$ und $|\{a, aa, aaa, aaaa\}| < |\mathbb{N}|$

Mächtigkeit von Mengen

Was nützt uns die Funktion f?

- f systematisiert das Aufzählen der Elemente (s. Satz: 1.30)
- f hilft, wenn Aufzählen nicht funktioniert.
- f hilft in Fällen, wo $|N| \neq |M|$ gezeigt werden muss.

Betrachten Sie die beiden Intervalle von reellen Zahlen $[1, 4]$ und $[1, 100]$ graphisch in der Ebene. **Können Sie eine explizite Bijektion zwischen beiden Mengen angeben?**

Obwohl eine Menge eine echte Teilmenge der anderen ist, haben beide gleich viele Elemente!!!

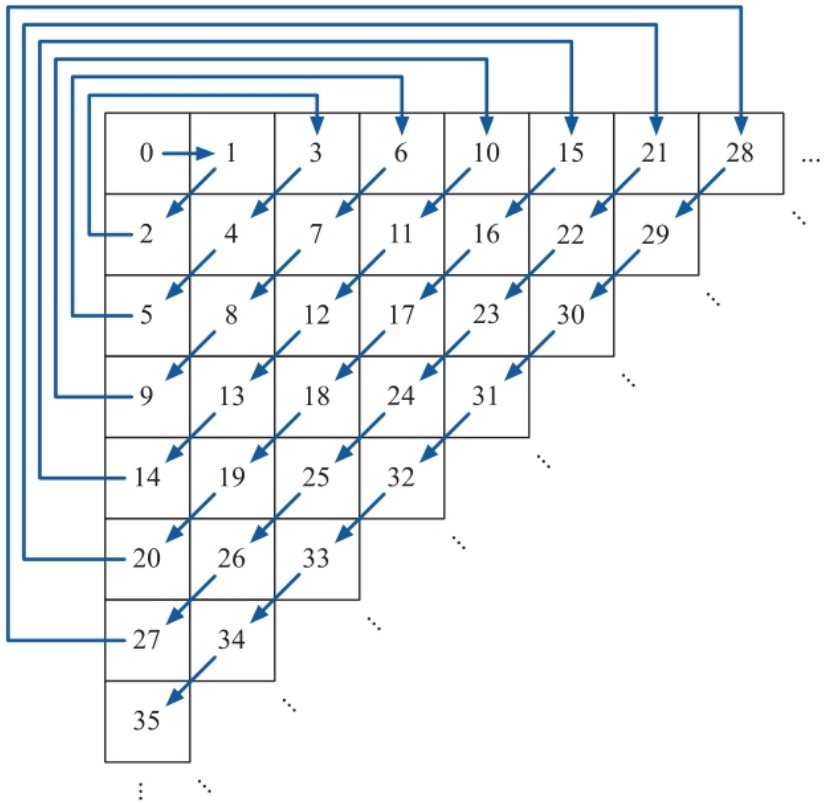


Abbildung 5: Cantorsche Paarung für $\mathbb{N}^2$, ©[Hoffmann2011]

Satz 1.30

$|\mathbb{N} \times \mathbb{N}| = |\mathbb{N}|$

Beweisidee: $\pi(x, y) = y + \frac{(x+y)(x+y+1)}{2}$

Wie charakterisiert man eine **endliche** Menge?

Definition 1.31 (Endlich, unendlich)

Dedekind: Eine Menge heißt **endlich**, wenn sie keiner ihrer echten Teilmengen gleichmächtig ist.

$\mathbb{N}_0$: Eine Menge heißt **endlich**, wenn sie zu einer natürlichen Zahl n gleichmächtig ist.

Hierbei ist $\mathbb{N}_0$ wie folgt definiert:

$$0 := \emptyset,\ 1 := \{0\},\ \ldots n := \{0, 1, \ldots, n-1\}, \ldots$$

Eine Menge die nicht endlich ist, heißt **unendlich**.

- Man betrachte Funktionen

$$f : \mathbb{N}_0 \to \mathbb{N}_0$$

  Welche sind, jedenfalls prinzipiell, berechenbar?
- Welche sind **schnell und effizient** zu berechnen?
- Was genau ist ein *Algorithmus*?

Wieviele Grammatiken (Algorithmen, Sprachen) gibt es überhaupt?

Mögliche Antworten:

1. Endlich viele.
2. Unendlich viele.
3. Soviele wie es natürliche Zahlen gibt.
4. Soviele wie es reelle Zahlen gibt.
5. Nicht klar für Algorithmen, da dieser Begriff nicht genau definiert wurde.

Σ^*: Sei Σ endlich oder abzählbar unendlich. Dann ist Σ^* abzählbar unendlich. Jede endliche Zeichenreihe kann als natürliche Zahl repräsentiert werden (Primfaktorzerlegung).

Algorithmen: Es gibt soviele Algorithmen wie natürliche Zahlen.

$f : \mathbb{N}_0 \to \{0, 1\}$: Es gibt überabzählbar (mehr als abzählbar) viele solche Funktionen.

Zu gegebenen $f_1, \ldots, f_n, \ldots$ sei

$$C : \mathbb{N}_0 \to \mathbb{N}_0; n \mapsto \begin{cases} 1, & \text{falls } f_n(n) = 0; \\ 0, & \text{sonst.} \end{cases}$$

C ist von allen f_i verschieden!

Teilmengen von $\mathbb{N}_0$: Es gibt überabzählbar viele, denn man kann sie identifizieren mit Funktionen $f : \mathbb{N}_0 \to \{0, 1\}$.

Also kann nicht jede Sprache durch eine Grammatik dargestellt werden.

Hilbert's Hotel: Hilbert's Hotel hat abzählbar unendlich viele Zimmer. Alle sind belegt.

$+1$: Können Sie noch einen weiteren Gast unterbringen?

$+\mathbb{N}$: Können Sie auch abzählbar viele zusätzliche Gäste unterbringen?

$+\mathbb{N} \times \mathbb{N}$: Nun kommen abzählbar viele Busse mit jeweils abzählbar vielen Fahrgästen. Wie bringen Sie diese unter?

https://www.youtube.com/watch?v=Uj3_KqkI9Zo

Lemma 1.32

Für abzählbare Alphabete Σ ist Σ^* abzählbar.

Beweis

Sei $\Sigma = \{a_1, a_2, a_3, \dots\}$ abzählbar unendlich.

Dann ordnen wir jedem (endlichen!!) Wort $a_{i_1} a_{i_2} a_{i_3} a_{i_4} \cdots \in \Sigma^*$ die folgende Zahl zu

$$2^{i_1} \cdot 3^{i_2} \cdot 5^{i_3} \cdot 7^{i_4} \dots \ .$$

Offenbar ist diese Abbildung injektiv (eindeutige Darstellung einer Zahl in ihre Primfaktoren), d.h. $|\Sigma^*| \leq |\mathbb{N}|$. $\square$

Lemma 1.33

1. $\mathbb{N}$ *ist* abzählbar.
2. *Die Menge aller Teilmengen von* $\mathbb{N}$*, bezeichnet mit* $2^{\mathbb{N}}$*, ist* überabzählbar *(und lässt sich nicht auf* $\mathbb{N}$ *bijektiv abbilden).*
3. *Die Menge aller reellen Zahlen ist überabzählbar.*
4. *Die Menge aller Sprachen ist überabzählbar.*
5. *Folgende Mengen sind* abzählbar*:*
   - *Menge aller Grammatiken,*
   - *Menge aller Algorithmen,*
   - *Menge aller aus einem abzählbaren Alphabet gebildeten endlichen Zeichenreihen.*

Satz 1.34 (Cantor, 1874: $|\mathbb{R}| > |\mathbb{N}|$)

$[0,1] = \{r : 0 \leq r \leq 1, r \in \mathbb{R}\}$ *ist überabzählbar.*[a]

[a] Schon Euklid (ca. 300 B.C.) wußte, daß nicht-rationale Zahlen existieren. Lindemann zeigte dass π nicht algebraisch ist, aber erst Cantor zeigte, daß es echt mehr reelle als natürliche Zahlen gibt.

Beweis: (Cantor, 1877)

Diagonalisierung

Widerspruchsbeweis: Annahme: $[0,1]$ ist abzählbar.

Dann gibt es eine bijektive Funktion $f : \mathbb{N} \to [0,1]$. f bildet jede natürliche Zahl n eineindeutig auf eine reelle Zahl r in $[0,1]$ ab.

Wir zählen die Funktionswerte von f auf, beschränken uns dabei aber auf die Nachkommastellen:

Beweis: (Cantor, 1877)

f(0) = 0,	0	1	3	4	5	4	6	1	0	2	9	...
f(1) = 0,	2	1	2	5	8	7	9	9	2	4	0	...
f(2) = 0,	3	4	1	2	4	3	8	3	3	2	8	...
f(3) = 0,	2	1	4	4	5	4	5	2	0	3	5	...
f(4) = 0,	0	1	3	6	5	7	6	1	0	0	0	...
f(5) = 0,	2	1	5	5	8	6	5	9	2	4	0	...
f(6) = 0,	7	4	8	4	3	6	8	5	3	3	5	...

...

Beweis: (Cantor, 1877)

Diagonalisieren: Betrachte $r_{diag} = 0, d_0 d_1 d_2 d_3 d_4 d_5 \ldots$, wobei die Ziffer $d_i \in \{0, \ldots, 8\}$ jeweils so gewählt wird, daß sie **nicht** mit der $i+1$-ten Ziffer von $f(i)$ übereinstimmt [a]:

f(0) = 0,	**0**	1	3	4	5	4	6	1	0	2	...	$\mathbf{d_0 = 2}$
f(1) = 0,	2	**1**	2	5	8	7	9	9	2	4	...	$\mathbf{d_1 = 2}$
f(2) = 0,	3	4	**1**	2	4	3	8	3	3	2	...	$\mathbf{d_2 = 2}$
f(3) = 0,	2	1	4	**4**	5	4	5	2	0	3	...	$\mathbf{d_3 = 3}$
f(4) = 0,	0	1	3	6	**5**	7	6	1	0	0	...	$\mathbf{d_4 = 3}$
f(5) = 0,	2	1	5	5	8	**6**	5	9	2	4	...	$\mathbf{d_5 = 3}$
f(6) = 0,	7	4	8	4	3	6	**8**	5	3	3	...	$\mathbf{d_6 = 4}$
...												

[a] Ein(!) mögliches Beispiel in rot neben der Tabelle

Beweis: (Cantor, 1877)

f(0) = 0,	**0**	1	3	4	5	4	6	1	0	2	...	$\mathbf{d_0 = 2}$
f(1) = 0,	2	**1**	2	5	8	7	9	9	2	4	...	$\mathbf{d_1 = 2}$
f(2) = 0,	3	4	**1**	2	4	3	8	3	3	2	...	$\mathbf{d_2 = 2}$
f(3) = 0,	2	1	4	**4**	5	4	5	2	0	3	...	$\mathbf{d_3 = 3}$
f(4) = 0,	0	1	3	6	**5**	7	6	1	0	0	...	$\mathbf{d_4 = 3}$
f(5) = 0,	2	1	5	5	8	**6**	5	9	2	4	...	$\mathbf{d_5 = 3}$
f(6) = 0,	7	4	8	4	3	6	**8**	5	3	3	...	$\mathbf{d_6 = 4}$

...

$\mathbf{r_{diag} = 0, d_0 d_1 d_2 d_3 d_4 d_5 \ldots}$ kommt nicht in dieser Aufzählung nicht vor, formal: $r_{diag} \neq f(n)$ für alle $n \in \mathbb{N}$, **aber es gilt $\mathbf{r_{diag} \in [0,1]}$!**

Dann ist f nicht surjektiv! Also $|\mathbb{N}| < |[0,1]|$. $\square$

Anmerkung für genaue Beobachter:

Genau genommen, müssen wir noch gewährleisten, daß die nicht eindeutige Zahlendarstellung uns den Beweis nicht kaputt macht:

Problem: Reelle Zahlen mit abbrechender Dezimaldarstellung können alternativ als nichtabbrechende Dezimalzahl, die mit einer unendlichen Folge von Neunen enden, geschrieben werden. z.B. $0.5 = 0.4999999\ldots$.
In dem Beweis argumentieren wir über die Darstellung von r_{diag} und schließen, daß - weil die Darstellung von r_{diag} in der Auflistung fehlt - auch der Wert r_{diag} fehlt.

Also jetzt ganz genau: Wir machen unsere Liste der Funktionswerte von f eindeutig: Dort wird jede reelle Zahl auf die "kürzestmögliche" Art aufgeschrieben und für die Konstruktion der Zahl r_{diag} gegebenenfalls mit Nullen verlängert. Das ist - ohne Beschränkung der Allgemeinheit - möglich.

Bei der Wahl der Ziffern für r_{diag} verbieten wir die 9 als Ziffer zu wählen. Das ist - dank des Dezimalsystems mit 10 verschiedenen Ziffern - immer möglich, und garantiert, daß keine unendliche Neunerfolge entstehen kann, d.h. r_{diag} ist keine dieser "alternativen" Darstellungen (mit Neunerfolge) einer abbrechenden Dezimalzahl.

Damit gilt endgültig: Wenn die Darstellung von r_{diag} in der Liste fehlt, dann kommt der Wert r_{diag} nicht als Funktionswert von f vor.

Gleich nochmal:

Satz 1.35 (Cantor: $|2^M| > |M|$)

Für jede nichtleere Menge M ist die Potenzmenge[a] , genannt 2^M, mächtiger als M.

[a] Die Potenzmenge von M ist die Menge aller Teilmengen von M. Sie kann auch als die Menge aller charakteristischen Funktionen $f : M \to \{0, 1\}$ verstanden werden.

Beweis: (Cantor)

Annahme: Nehmen wir an, daß $|2^M| = |M|$ für eine Menge M gilt.

Dann gibt es eine bijektive Funktion $f : M \to 2^M$. f bildet jedes Element $m \in M$ eineindeutig auf eine Teilmenge $f(m) \subseteq M$ ab.

Beweis: (Cantor)

Diagnonalisieren:

Nun gibt es sicher m, für die $m \in f(m)$ gilt und solche, für die $m \notin f(m)$ gilt. Sei

$$Diag\_Set = \{m \in M \mid m \notin f(m)\} \subseteq M.$$

f war bijektiv, d.h. auch für $Diag\_Set$ gibt es ein m_{diag}, sodaß $f(m_{diag}) = Diag\_Set$ gilt.
Es stellt sich die Frage: $m_{diag} \in Diag\_Set$?

$$m_{diag} \in Diag\_Set \Rightarrow m_{diag} \notin f(m_{diag}) \Rightarrow m_{diag} \notin Diag\_Set$$
$$m_{diag} \notin Diag\_Set \Rightarrow m_{diag} \in f(m_{diag}) \Rightarrow m_{diag} \in Diag\_Set$$

Es gibt also kein bijektives $f : M \to 2^M$. Also $|2^M| > |M|$. □

Konsequenz: Es gibt keine "maximale" Unendlichkeit.

Diagonalisieren

(engl.: dovetailing - verzahnen)

Beschreibung eines Elements, das

- auf die Definition der Funktion f Bezug nimmt
- aber bei jedem Vergleichselement "auf der Diagonale" abweicht.

Verzahnung von

- **Referenz auf die Definition**
- **und Anwendung der Definition**

= Selbstreferenz.

Konsequenz:

Satz 1.36 (Existenz nichtberechenbarer Funktionen)

Es muss also Sprachen geben, die nicht mit einer Grammatik erzeugt werden können.

Es muss auch Probleme und Funktionen geben, für die es keinen Algorithmus gibt, die also nicht berechenbar sind.

Die Beweise beruhen (nur) auf Kardinalitätsargumenten.
D.h. noch haben wir keine nichtberechenbare Funktion "gesehen", dürfen also noch hoffen, daß uns die nichtberechenbaren Probleme auch nicht interessieren.

2. Endliche Automaten (reg)

Inhalt (1)

In diesem Kapitel führen wir den endlichen Automaten ein: ein vereinfachtes Modell eines Computers. Wir zeigen,

1. daß die von endlichen Automaten erkannten Sprachen genau die vom Grammatik-Typ 3 (rechtslinear) sind;
2. daß deterministische und indeterminierte endliche Automaten äquivalent sind;

Inhalt (2)

3. ein Kriterium, das Pumping Lemma, das es erlaubt, eine Sprache als nicht rational nachzuweisen;
4. daß es Algorithmen gibt, um Probleme über endlichen Automaten, bzw. Typ 3 Sprachen zu lösen;
5. daß Typ 3 Sprachen genau die sind, die durch reguläre Ausdrücke beschrieben werden können.

2.1 DEAen (e.a.)

- Die Sprache $L = \{aa\}\{ab\}^*\{c\}$ ist regulär und wird erzeugt von der Grammatik

$$G = (\{S, A\}, \{a, b, c\}, R, S),$$

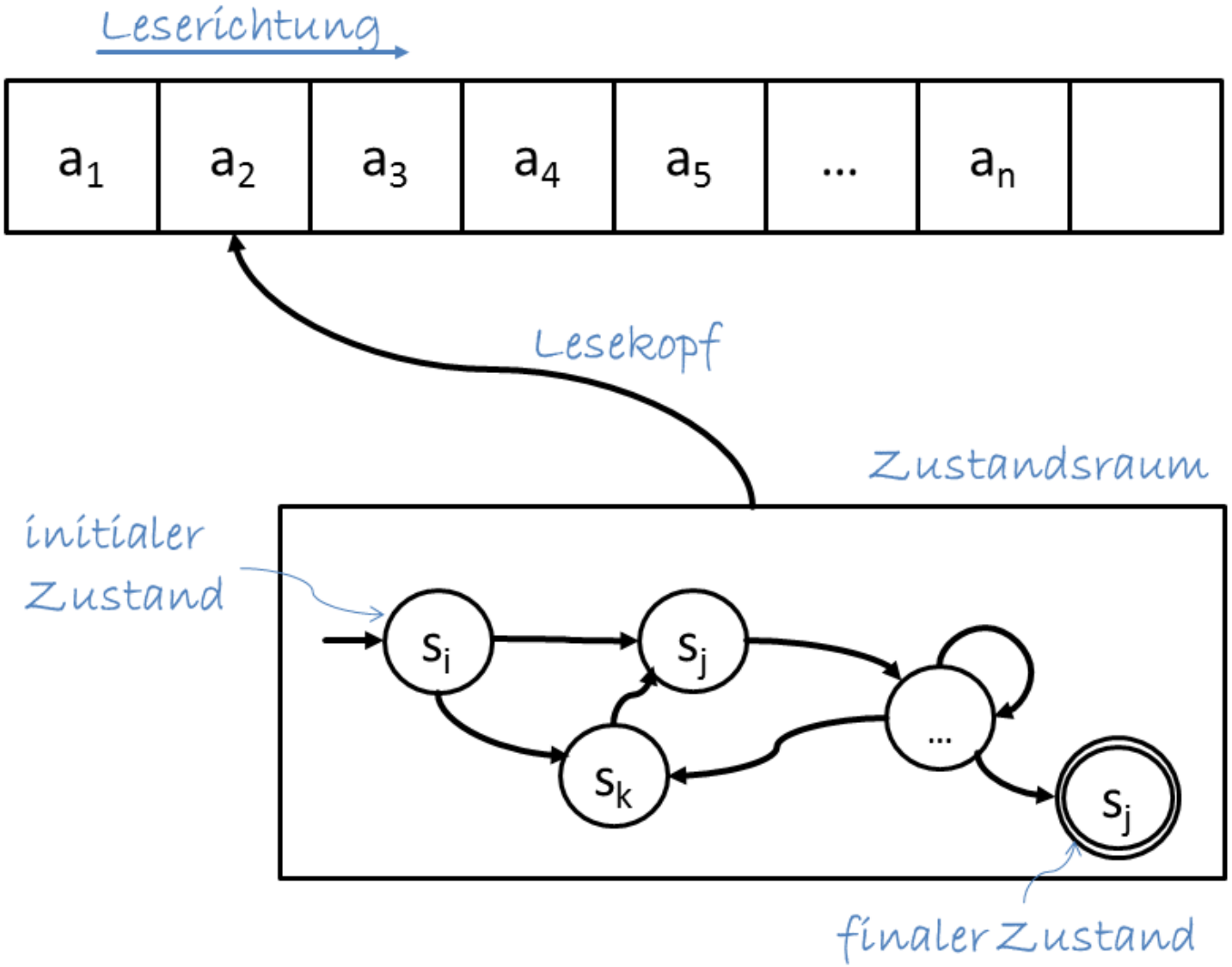
wobei R aus folgenden Regeln besteht:

$$S \to aaA$$
$$A \to abA \mid c$$

- Auch die Sprache aller durch 3 teilbaren Dezimalzahlen ist regulär. Eine erzeugende Grammatik ist $G = (\{S, S_0, S_1, S_2\}, \{0, \ldots, 9\}, R, S)$ mit der folgenden Regelmenge R:

$$
\begin{aligned}
S &\to 3S_0 \mid 6S_0 \mid 9S_0 \mid 1S_1 \mid 4S_1 \mid 7S_1 \mid 2S_2 \mid 5S_2 \mid 8S_2 \mid 0\\
S_0 &\to 0S_0 \mid 3S_0 \mid 6S_0 \mid 9S_0 \mid 1S_1 \mid 4S_1 \mid 7S_1 \mid 2S_2 \mid 5S_2 \mid 8S_2 \mid \varepsilon\\
S_1 &\to 0S_1 \mid 3S_1 \mid 6S_1 \mid 9S_1 \mid 1S_2 \mid 4S_2 \mid 7S_2 \mid 2S_0 \mid 5S_0 \mid 8S_0\\
S_2 &\to 0S_2 \mid 3S_2 \mid 6S_2 \mid 9S_2 \mid 1S_0 \mid 4S_0 \mid 7S_0 \mid 2S_1 \mid 5S_1 \mid 8S_1
\end{aligned}
$$

Ohne das ε in der zweiten Regel wäre nur die "0" als Terminalwort herleitbar. In der ersten Zeile könnte man auch $0S_0$ schreiben (an der letzten Stelle).


Leserichtung
a1
a2
a3
a4
a5
...
an
Lesekopf
Zustandsraum
initialer Zustand
si
sj
sk
...
sj
finaler Zustand

Ein endlicher Automat:

- Gegeben $w \in \Sigma^*$, ein endlicher Automat testet, ob $w \in L$ gilt.
- Ein endlicher Automat hat:
  - einen Lesekopf, um w zu lesen. Den kann er nur von links nach rechts bewegen.
  - einen internen Zustand, mit endlich vielen Werten
- Er fängt an in einem **initialen Zustand**.
- Bei jedem Buchstaben, den er liest, ändert er seinen Zustand.
- Wenn er am Ende von w in einem **finalen Zustand** ist, sagt er "ja". Das heißt, er **akzeptiert** w: w ist in L. Sonst sagt er "nein".
- Er stoppt auf jeden Fall nach $|w|$ Schritten.

Darstellung als Graph:

- ein **Knoten** für jeden möglichen **Zustand**,
- **Kanten** sind mit Buchstaben beschriftet: Sie beschreiben **Zustandsänderungen**.
- **Initiale** Zustände werden mit einem **Pfeil** gekennzeichnet,
- **finale Zustände** mit einem **doppelten Kreis**.

Bemerkung zur Definition des endlichen Automaten

In einem ea ist δ eine totale Funktion von $K \times \Sigma \to K$, also ist für jedes Paar aus Zustand und Symbol, ein Nachfolgezustand definiert.

Wenn δ eine beliebige, d.h. möglicherweise partielle Funktion von $K \times \Sigma \to K$ ist, dann kann es Paare (q, a) geben, für die kein Nachfolger definiert ist, also $\delta(q, a) = \perp$ (undefiniert). Wir setzen $\delta^*(\perp, a) = \perp$ für alle $a \in \Sigma$.

Wenn man beim Lesen eines Wortes w in einen Zustand kommt, in dem kein Nachfolgezustand für den aktuellen Buchstaben a definiert ist, also der Fall $\delta(q, a) = \perp$, dann kann das Wort nicht zu Ende gelesen werden, und die Verarbeitung im Endzustand enden, also $w \notin L(\mathrm{A})$. **Solche Automaten behandeln wir später: indeterminierte ea'en.**

Definition 2.1 (Endlicher Automat)

Ein endlicher Automat (e.a. oder auch DEA) (finite automaton) ist ein Tupel $A = (K, \Sigma, \delta, s_0, F)$. Dabei ist

- K eine endliche Menge von **Zuständen**,
- Σ ein **endliches Alphabet** (aus dessen Buchstaben die Eingabewörter bestehen können),
- $\delta : K \times \Sigma \to K$ die (totale) Übergangsfunktion,
- $s_0 \in K$ der **Startzustand**, und
- $F \subseteq K$ die Menge der **finalen Zustände**.

$\delta(q, a) = q'$ bedeutet:

- Der Automat ist im Zustand q,
- liest ein a und
- geht in den Zustand q' über.

Wir erweitern δ zu δ^*:
$\delta^* : K \times \Sigma^* \to K$ ist strukturell rekursiv über Σ^* definiert:

$$\begin{aligned} \delta^*(q, \varepsilon) &:= q \\ \delta^*(q, wa) &:= \delta(\delta^*(q, w), a) \end{aligned}$$

Wenn klar ist, was gemeint ist, wird δ^* auch einfach als δ geschrieben.

Beispiel 2.2

Die Sprache $\{a^{2n} \mid n \in \mathbb{N}_0\}$ über dem Alphabet $\{a, b\}$ wird akzeptiert von dem endlichen Automaten $A = (\{s_0, s_1, s_2\}, \{a, b\}, \delta, s_0, \{s_0\})$ mit

$$\delta(s_0, a) = s_1 \qquad \delta(s_1, a) = s_0 \qquad \delta(s_2, a) = s_2$$
$$\delta(s_0, b) = s_2 \qquad \delta(s_1, b) = s_2 \qquad \delta(s_2, b) = s_2$$

Ein Beispiel für δ^*:

$$
\begin{aligned}
\delta^*(s_0, aab) &= \delta\big(\delta^*(s_0, aa), b\big) \\
&= \delta\big(\delta\big(\delta^*(s_0, a), a\big), b\big) \\
&= \delta\big(\delta\big(\delta(\delta^*(s_0, \varepsilon), a\big), a\big), b\big) \\
&= \delta\big(\delta\big(\delta(s_0, a), a\big), b\big) \\
&= \delta\big(\delta(s_1, a), b\big) \\
&= \delta(s_0, b) \\
&= s_2
\end{aligned}
$$

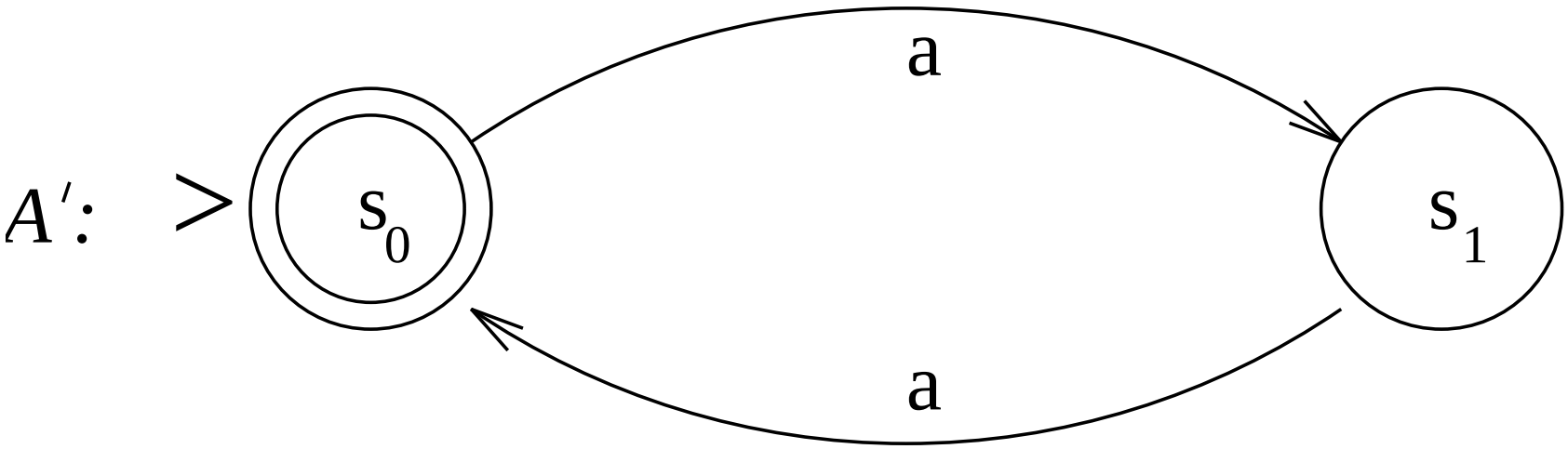


Abbildung 6: Eine Variante des Automaten aus Beispiel 2.2 mit gleicher Sprache, aber kleinerem Alphabet

Definition 2.3 (Von einem e.a. akzeptierte Sprache)

$L(A)$, die von einem Automaten A akzeptierte Sprache, ist definiert als

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

Die Menge der von endlichen Automaten akzeptierten Sprachen ist

RAT $:= \{L \mid$ es gibt endlichen Automaten A, mit $L = L(A)\}$

und heißt die Menge der rationalen Sprachen.

Wir zeigen bald, daß dies genau die Menge der durch rechtslineare Grammatiken dargestellten Sprachen ist. Am Ende dieses Kapitels zeigen wir:

- **RAT** besteht genau aus den regulären Sprachen, also aus den durch reguläre Ausdrücke dargestellten Sprachen.

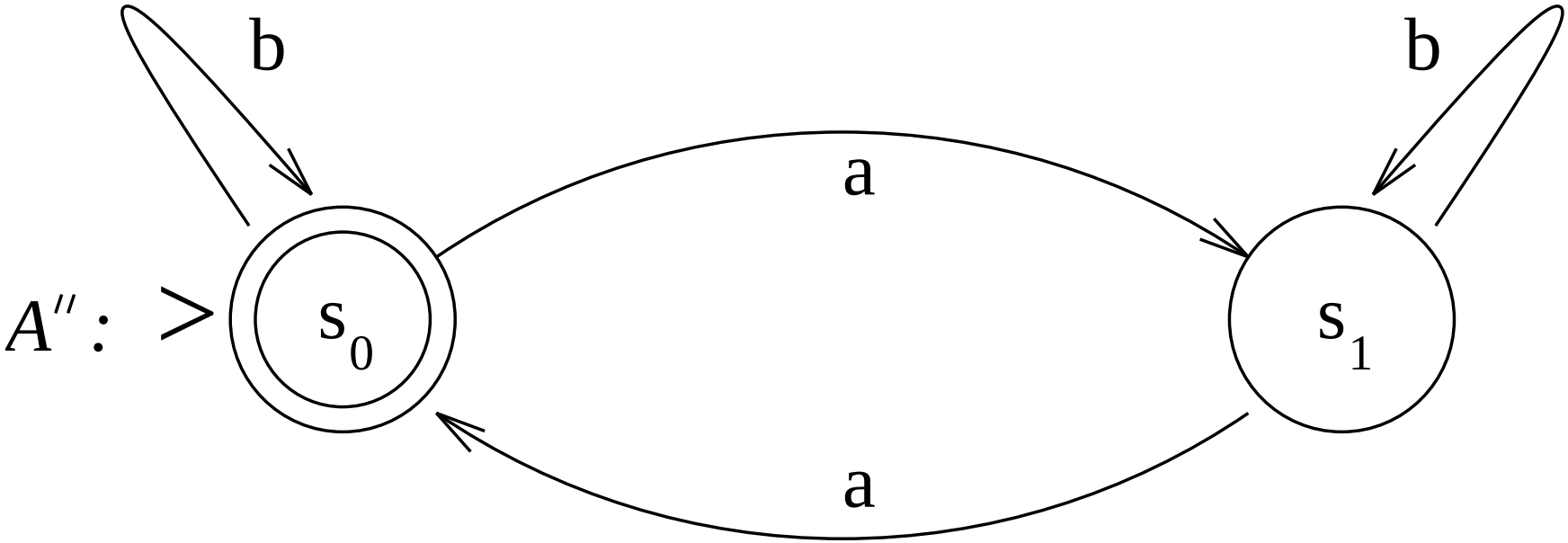


Abbildung 7: A'' akzeptiert Wörter, die eine gerade Anzahl von "a"s haben

Beispiel 2.4

Der Automat A'' in Abb. 7 akzeptiert $L(A'') = \{w \in \{a,b\}^* \mid \exists n : \ \#_a(w) = 2n\}$. Dabei ist $\#_a(w)$ die Anzahl der "a"s in w.

Zum Beispiel würde A'' das Wort $bbaabaab$ wie folgt akzeptieren:

$$\underset{\in F}{s_0} \xrightarrow{b} \underset{\in F}{s_0} \xrightarrow{b} \underset{\in F}{s_0} \xrightarrow{a} s_1 \xrightarrow{a} \underset{\in F}{s_0} \xrightarrow{b} \underset{\in F}{s_0} \xrightarrow{a} s_1 \xrightarrow{a} \underset{\in F}{s_0} \xrightarrow{b} \underset{\in F}{s_0}$$

Beispiel 2.5

$L = \{w \in \{0,1\}^* \mid w$ enthält genau zwei Einsen $\}$ wird akzeptiert von dem endlichen Automaten in Abb. 8.

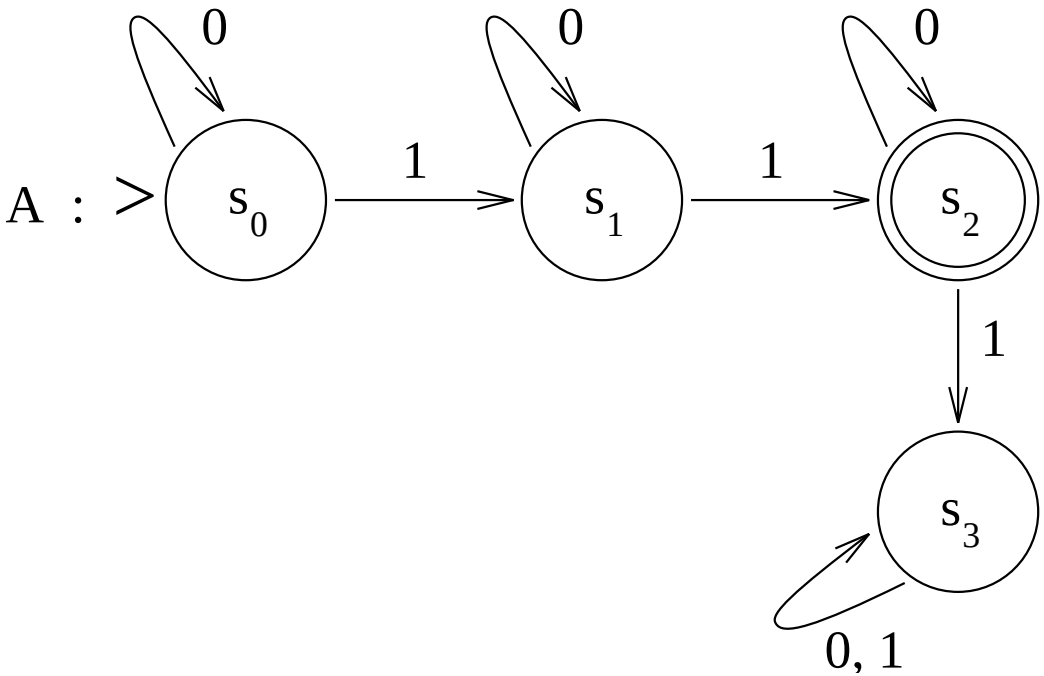


Abbildung 8: Dieser Automat akzeptiert Wörter, mit genau 2 Einsen

Beispiel 2.6

Der endliche Automat in Abb. 9 akzeptiert die Sprache aller durch 3 teilbaren Dezimalzahlen.

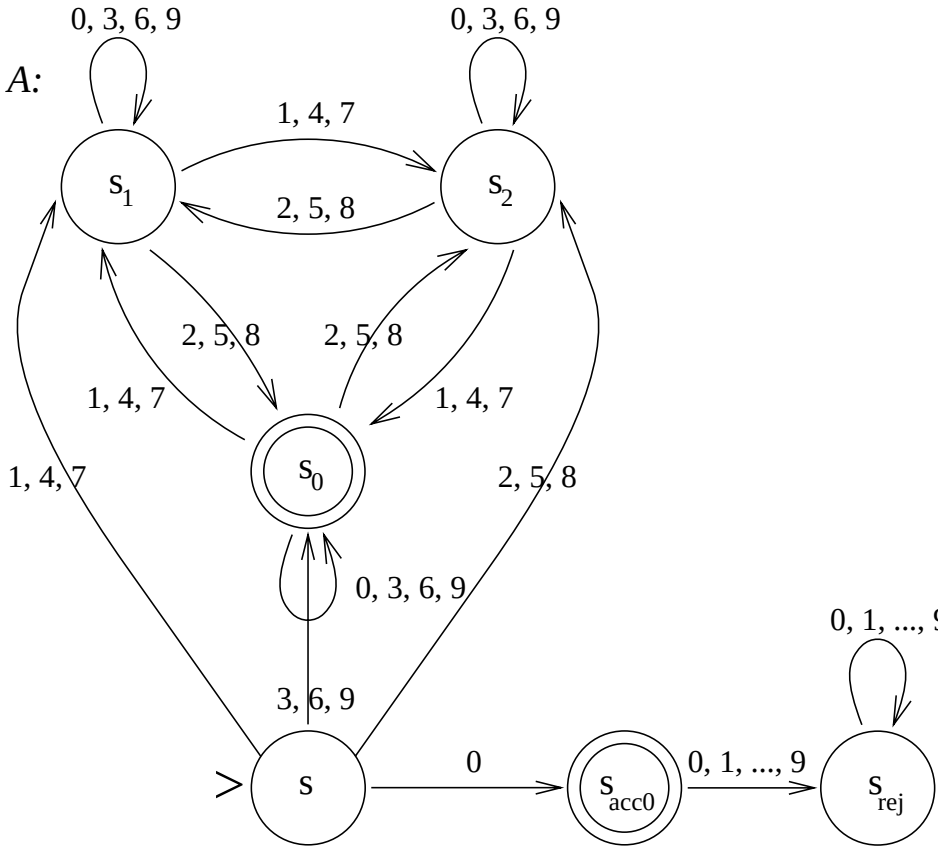


Abbildung 9: Ein endlicher Automat für die durch 3 teilbaren Dezimalzahlen

2.2 NEAs (nd e.a.)

Determinierte endliche Automaten:

- in Zustand q bei Eingabe a **ein einziger** Nachfolgezustand
- festgelegt durch Übergangs**funktion** δ.

Indeterminierter endlicher Automat:

- in Zustand q bei Eingabe a evtl. mehrere Nachfolgezustände – oder gar keiner.
- Übergangsrelation Δ.

Definition 2.7 (Indeterminierter endlicher Automat)

Ein indeterminierter endlicher Automat (nd e.a.) A ist ein Tupel $A = (K, \Sigma, \Delta, I, F)$. Dabei ist

- K eine endliche Menge von Zuständen,
- Σ ein endliches Alphabet,
- $\Delta \subseteq (K \times \Sigma) \times K$ eine Übergangsrelation,
- $I \subseteq K$ eine Menge von Startzuständen und
- $F \subseteq K$ eine Menge von finalen Zuständen.

$\Delta^* \subseteq (K \times \Sigma^*) \times K$ ist definiert wie folgt:

$$(q, \varepsilon) \quad \Delta^* \quad q' \quad \underline{\text{gdw}} \quad q' = q$$

$$(q, wa) \quad \Delta^* \quad q' \quad \underline{\text{gdw}} \quad \exists q'' \in K \left((q, w)\ \Delta^*\ q'' \text{ und } (q'', a)\ \Delta\ q' \right)$$

Schreibweisen:

- statt $(q, w)\ \Delta^*\ q'$ auch $\big((q, w), q'\big) \in\ \Delta^*$ oder $q' \in\ \Delta^*(q, w)$.
- Δ^* wird auch als Δ abgekürzt, wenn es im Kontext nicht zu Verwechslungen führt.

Wann akzeptiert ein indeterminierter Automat ein Wort?

Ein nd e.a. A akzeptiert ein Wort w, wenn

- es mindestens einen Weg mit der **Beschriftung** w durch A gibt,
- der in einem **finalen** Zustand endet.

Definition 2.8 (Von nd e.a. akzeptierte Sprache)

Die von einem indeterminierten endlichen Automaten A akzeptierte Sprache ist

$$L(A) := \{w \in \Sigma^* \mid \exists s_0 \in I \;\; \exists q \in F \;\; (s_0, w)\Delta^* q\}$$

Ein nd e.a. kann lügen: er kann ein Wort nicht akzeptieren (also "nein" sagen), obwohl es einen anderen Weg gibt, der es akzeptiert. Wenn er allerdings "ja" sagt, dann lügt er nicht.

Wie kann man sich Indeterminismus vorstellen?

- Determinierte endliche Automaten:
  Algorithmus für das Wortproblem ablesbar.
- Indeterminierte endliche Automaten:
  Alg.= Automat plus **Suchstrategie**.
- Um zu prüfen, ob ein Wort w akzeptiert wird, müssen alle Wege mit **Beschriftung** w durch den Automaten getestet werden.
- Der ganze Suchbaum muss durchlaufen werden.

- Zwei Sichtweisen auf indet. Automaten:
  - Der Automat durchläuft alle Wege **parallel**.
  - Der Automat **rät**, welchen von mehreren möglichen Folgezuständen er wählt.

Automat $A = (\{s_0, s_1, s_2\}, \{a, b\}, \Delta, s_0, \{s_0\})$ mit

$$\begin{aligned} \Delta(s_0, a) &= \{s_1\} \\ \Delta(s_1, b) &= \{s_0, s_2\} \\ \Delta(s_2, a) &= \{s_0\} \end{aligned}$$

Darstellung als Graph:

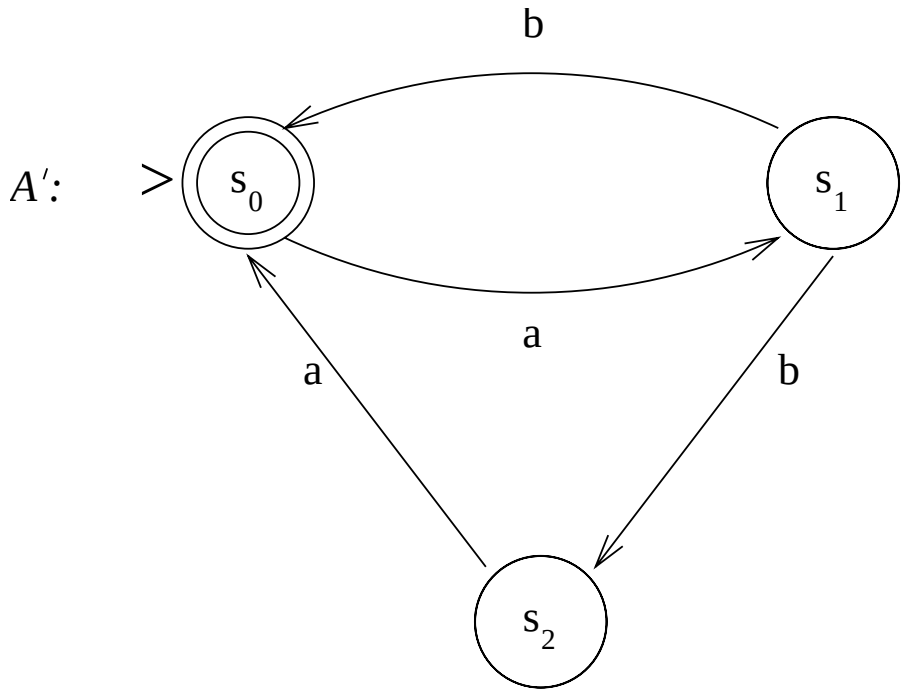


Abbildung 10: Ein indeterminierter Automat

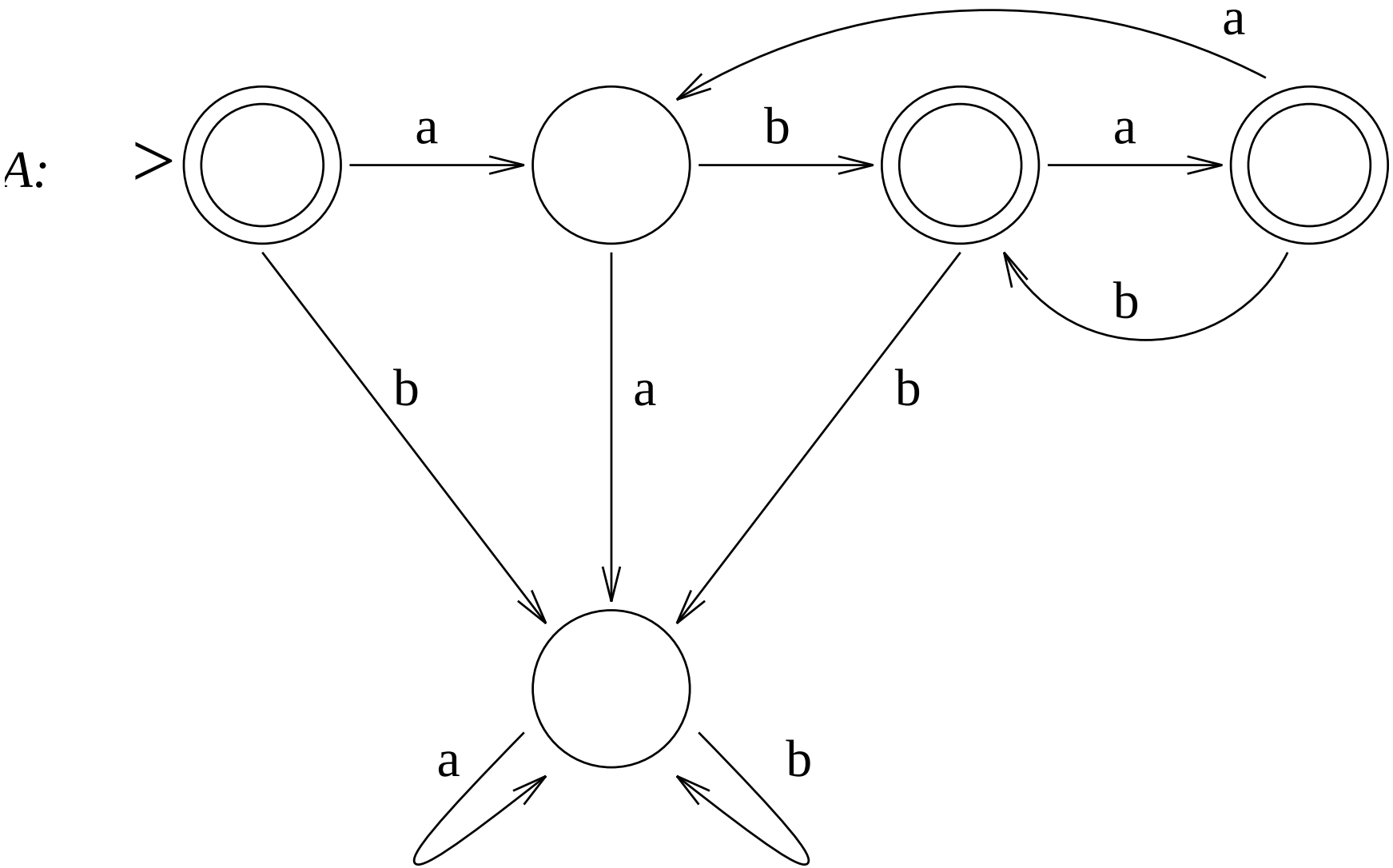


Abbildung 11: Ein deterministischer Automat für $L = \{ab, aba\}^*$

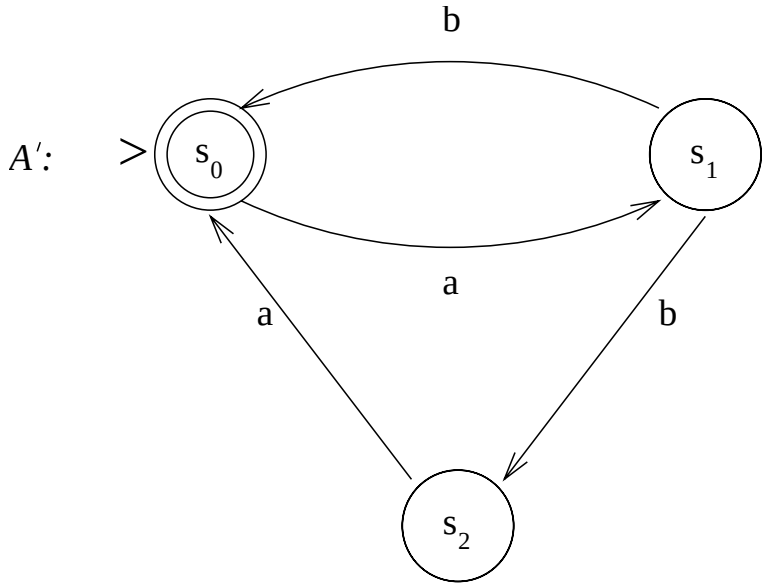


Abbildung 12: Ein indeterminierter Automat für $L = \{ab, aba\}^*$

Die Sprache $L = \{ab, aba\}^*$ wird

- von dem deterministischen Automaten A aus Abb. 11 und
- von dem indeterminierten Automaten A' aus Abb. 12

akzeptiert.

$$L = \{a, b\}^*\{a\}\{a, b\}$$

ist die Sprache aller Wörter über $\{a, b\}$, deren zweitletzter Buchstabe ein a ist.

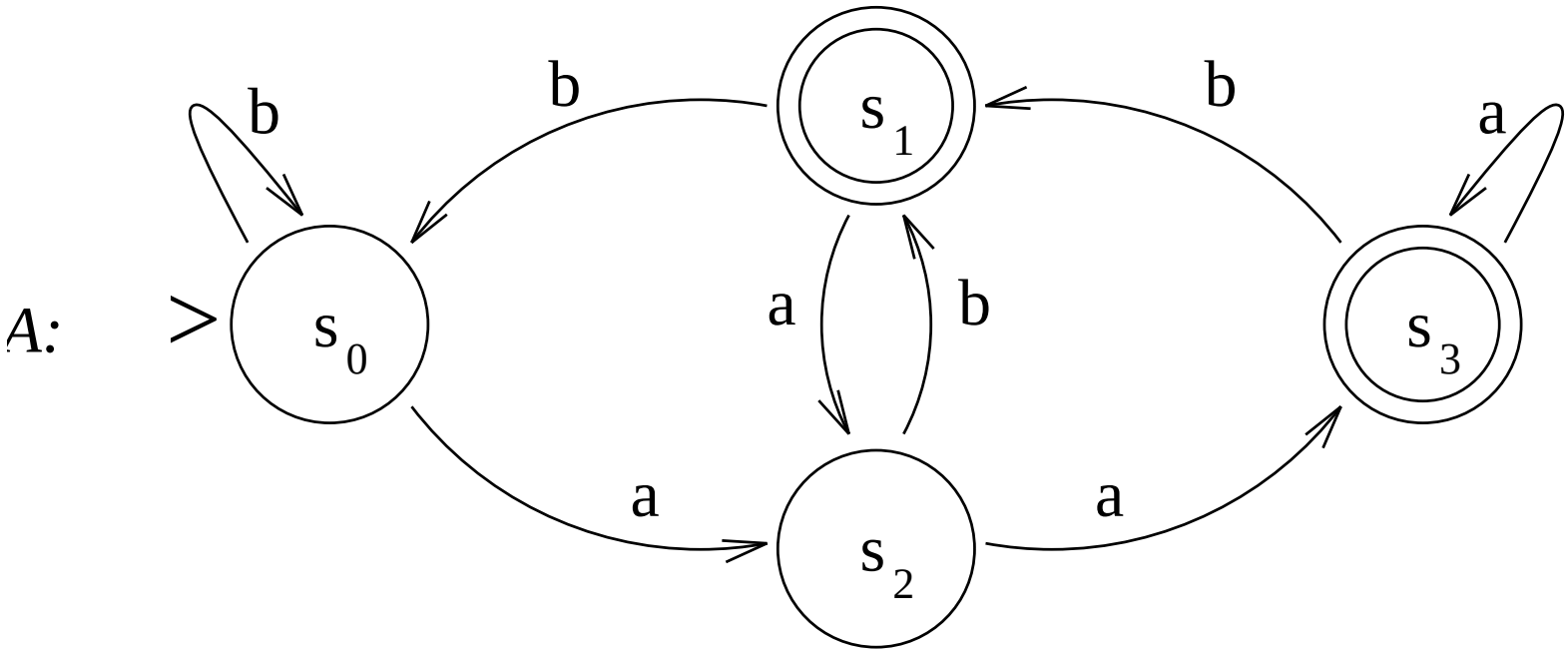


Abbildung 13: Ein deterministischer Automat für $L = \{a, b\}^*\{a\}\{a, b\}$

Idee:

- im Zustand jeweils die letzten zwei Buchstaben merken
- wichtig: **war der vorletzte Buchstabe ein “a”?**
- endlich viele Zustände:
  Die Anzahl der Buchstaben “im Speicher” steht fest—es sind immer höchstens 2.
  Deshalb kann ein endlicher Automat die Sprache L akzeptieren.

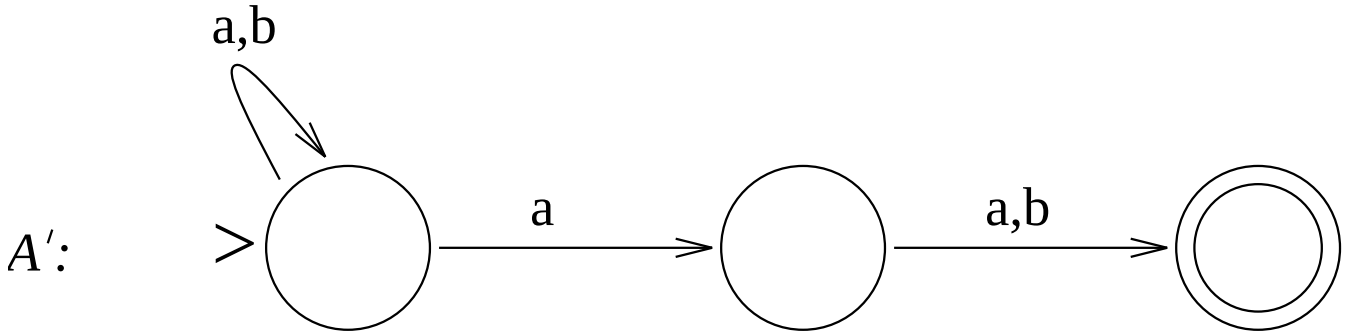


Abbildung 14: Ein indeterminierter Automat für $L = \{a, b\}^*\{a\}\{a, b\}$

Größenvergleich: Sprache über $\{a, b\}$ der Wörter, deren n-t-letzter Buchstabe ein "a" ist:

- **deterministischer** Automat: $\mathbf{2^n}$ Zustände, einen für jede Buchstabenkombination der Länge n
- **indeterminierter** Automat: $\mathbf{n + 1}$ Zustände

Ausführliche Darstellung an der Tafel.

Ein indeterminierter endlicher Automat heißt...

deterministisch gdw $\forall a \in \Sigma \, \forall q \in K \; |\Delta(q,a)| \leq 1$

vollständig gdw $\forall a \in \Sigma \, \forall q \in K \; |\Delta(q,a)| \geq 1$

Lemma 2.9

Ein endlicher Automat (e.a.) ist ein vollständiger und deterministischer indeterminierter endlicher Automat (nd. e.a.).

Oft sagt man auch **nichtdeterministischer** Automat. Damit ist aber **nicht** gemeint, daß der Automat auf jeden Fall nicht deterministisch ist: es wird nur nicht gefordert, daß er deterministisch sein muss.

Theorem 2.10 (ea gleich mächtig wie nd ea)

Eine Sprache ist rational (wird von einem ea akzeptiert) gdw sie von einem indeterminierten ea akzeptiert wird.

Beweis.

"$\Rightarrow$"

- Sei L eine rationale Sprache.
- Dann gibt es laut Definition einen endlichen Automaten A mit $L = L(A)$.
- Jeder endliche Automat ist aber schon ein (vollständiger und deterministischer) indeterminierter endlicher Automat.

□

"$\Leftarrow$"

- Sei $A = (K, \Sigma, \Delta, I, F)$ ein indeterminierter endlicher Automat, der die Sprache $L(A)$ akzeptiert.
- Dann kann man aus A einen deterministischen Automaten A' konstruieren mit $L(A) = L(A')$ mit Hilfe einer **Potenzmengenkonstruktion**:
  In A kann es zu einem Zustand q und einem gelesenen Zeichen a mehrere mögliche Folgezustände geben, die alle quasi parallel beschritten werden.
- Wir konstruieren einen **Zustand von A'** als eine **Menge von Zuständen von A**:
  - Gelangt man mit einem Eingabewort w in A indeterminiert in einen der Zustände q_1 bis q_n,
  - so gelangt man mit w in A' in einen Zustand $q' = \{q_1, \ldots, q_n\}$.

Zunächst ein konkretes Beispiel:

$L_{aab/aba} = \{w \in \{a,b\}^* \mid w$ hat aab oder aba als Teilwort $\}$
wird akzeptiert von folgendem nd. e.a.:

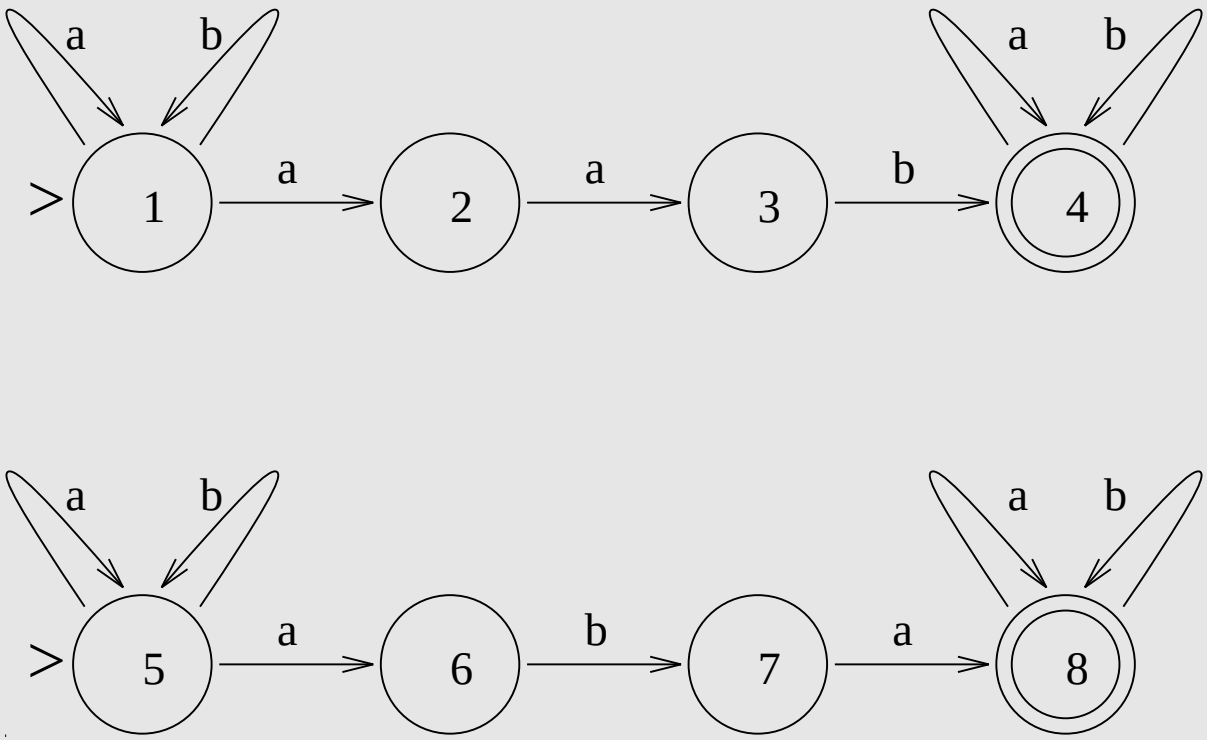


Abbildung 15: Indeterminierter endlicher Automat für $L_{aab/aba}$

Neuer Startzustand: Menge der alten Startzustände, also $\{1, 5\}$.
Nächster Schritt: Übergang von $\{1, 5\}$ mit a.

$$\begin{aligned}\Delta(1, a) &= \{1, 2\}\\ \Delta(5, a) &= \{5, 6\}\end{aligned}$$

Also, neuer Zustand $\{1, 2, 5, 6\}$ mit

$$\delta_{A'}(\{1, 5\}, a) = \{1, 2, 5, 6\}$$

Nächster Schritt: Übergang von $\{1, 5\}$ mit b.

$$\begin{aligned}\Delta(1, b) &= \{1\}\\ \Delta(5, b) &= \{5\}\end{aligned}$$

Für den Eingabebuchstaben b bleibt A' also im Startzustand.
Nächster Schritt: Übergang von $\{1,2,5,6\}$ mit a.

$$\begin{aligned}\Delta(1,a) &= \{1,2\}\\ \Delta(2,a) &= \{3\}\\ \Delta(5,a) &= \{5,6\}\\ \Delta(6,a) &= \emptyset\end{aligned}$$

Also, neuer Zustand $\{1,2,3,5,6\}$ mit

$$\delta_{A'}(\{1,2,5,6\},a) = \{1,2,3,5,6\}$$

Finale Zustände von A' sind die, die mindestens einen finalen Zustand von A enthalten.

Damit gilt:

- Wenn man mit dem Eingabewort w in einen Zustand von A' kommt, der einen finalen Zustand von A enthält,
- dann gibt es in A eine Rechnung, so daß mit w ein finaler Zustand erreicht wird.
- Damit ist $w \in L(A)$ nach der Definition der Sprache eines indeterminierten endlichen Automaten.

Für den nd e.a. aus dem Beispiel ergibt sich im Endeffekt:

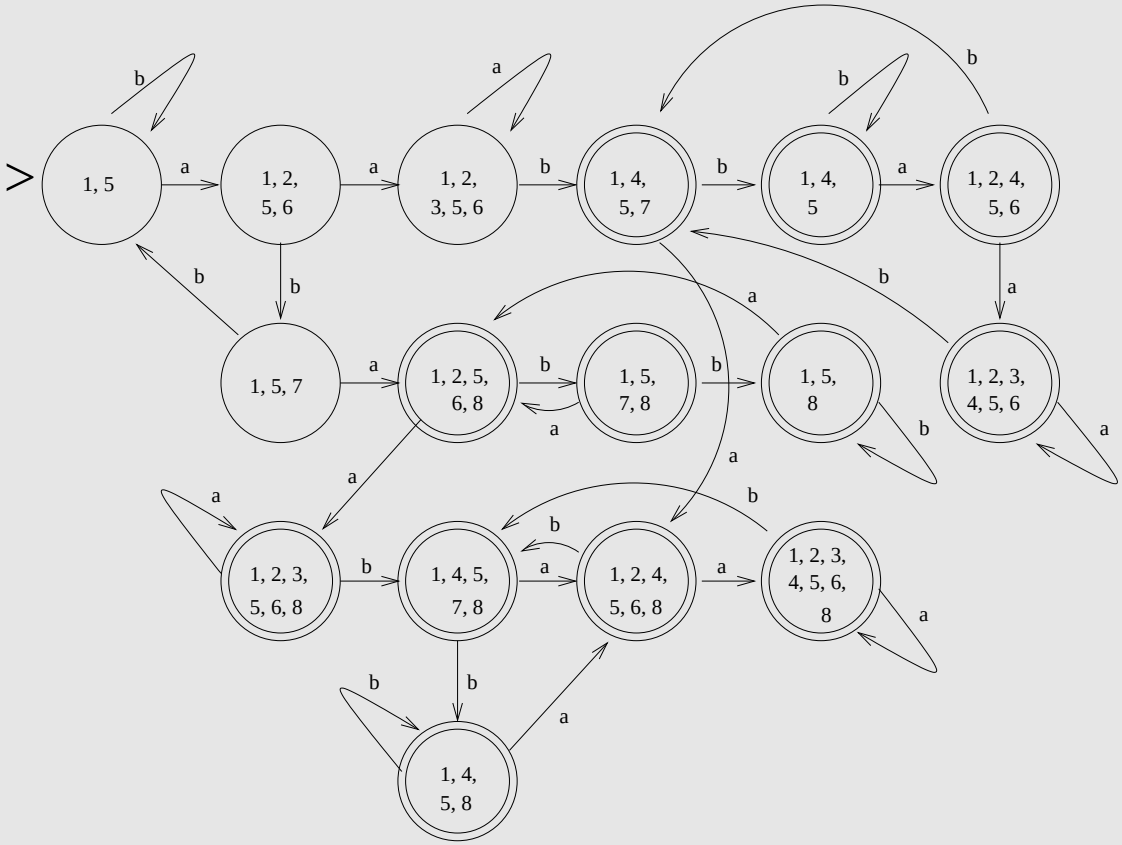


Abbildung 16: Determinierter endlicher Automat für $L_{aab/aba}$

Diesen Automaten kann man noch vereinfachen.

Wenn $\mathcal{A}'$ einen Zustand $\{q_1, \ldots, q_n\}$ hat, so daß es für a keinen Übergang in $\mathcal{A}$ gibt, für keinen der Zustände q_1 - q_n:

- Das Eingabewort wird von hier aus auf keinen Fall mehr akzeptiert.
- $\mathcal{A}'$ enthält dann einen Zustand $\emptyset$, einen **Abseitszustand**.
- Alle Übergänge von dort führen wieder in den Zustand $\emptyset$ zurück.

Konstruktion des deterministischen endlichen Automaten A' formal:

- Sei $A = (K, \Sigma, \Delta, I, F)$ ein indeterminierter endlicher Automat, der $L(A)$ akzeptiert.
- Dann gibt es einen deterministischen endlichen Automaten A' mit $L(A) = L(A')$ ist.
- Die Übergangsfunktion von A' ist Abbildung $\hat{\Delta} : 2^K \times \Sigma \to 2^K$ mit $\hat{\Delta}(M, a) := \bigcup_{q \in M} \Delta(q, a)$.

  $\hat{\Delta}^*$ sei als Erweiterung von $\hat{\Delta}$ auf mehrere Schritte definiert gemäß der allgemeinen Definition von δ^*.

Hilfsüberlegung: Es ist $\hat{\Delta}^*(M,w) = \bigcup_{q\in M} \Delta^*(q,w)$. Das beweisen wir durch Induktion über die Länge von w:

Induktionsanfang: $\hat{\Delta}^*(M,\varepsilon) = M = \bigcup\limits_{q\in M} \{q\} = \bigcup\limits_{q\in M} \Delta^*(q,\varepsilon)$

Induktionsschritt:

$$
\begin{array}{lll}
 & \hat{\Delta}^*(M,wa) & \\
= & \hat{\Delta}(\hat{\Delta}^*(M,w),a) & \text{allg. Def. von } \hat{\Delta}^* \text{ aus 2.1} \\
= & \bigcup_{p\in\hat{\Delta}^*(M,w)} \Delta(p,a) & \text{Definition von } \hat{\Delta} \\
= & \bigcup_{p\in \bigcup\limits_{q\in M} \Delta^*(q,w)} \Delta(p,a) & \text{Ind.-Vor. für } \hat{\Delta}(M,w) \\
= & \{q' \mid \exists q\in M\ \exists p\in\Delta^*(q,w)\ q'\in\Delta(p,a)\} & \\
= & \{r \mid \exists q\in M\ r\in\Delta^*(q,wa)\} & \text{allg. Def. von } \Delta^* \text{ aus 2.1} \\
= & \bigcup_{q\in M} \Delta^*(q,wa) &
\end{array}
$$

Sei nun $A' = (K', \Sigma, \delta', s_0', F')$ mit

- $K' := 2^K$, $\delta' := \hat{\Delta}$,
- $s_0' := I$, und $F' := \{M \subseteq K \mid M \cap F \neq \emptyset\}$.

Dann gilt: $w \in L(A')$

gdw $(\delta')^*(s_0', w) \in F'$ (Definition der Sprache eines Automaten)

gdw $\hat{\Delta}^*(I, w) \in F'$ (Definition von δ' und da $s_0' = I$)

gdw $\hat{\Delta}^*(I, w) \cap F \neq \emptyset$ (Definition von F')

gdw $\bigcup_{q \in I} \Delta^*(q, w) \cap F \neq \emptyset$ (nach Hilfsüberlegung)

gdw $\exists q \in I \; \exists q' \in F \; \left(q' \in \Delta^*(q, w)\right)$

gdw $w \in L(A)$

2.3 Automaten mit ε-Kanten (ε nd. e.a.)

- bisher: Kanten eines Automaten waren jeweils mit einem *Buchstaben* beschriftet
- jetzt: Kanten, die mit einem Wort beschriftet sind.
- Es darf auch das leere Wort ε sein!
- Ein Automat mit ε-Kanten:
  - kann in einem Schritt ein ganzes Wort verarbeiten,
  - kann einen Zustandsübergang machen, ohne dabei einen Eingabebuchstaben zu lesen.

Definition 2.11 (Automat mit ε-Kanten)

Ein Automat mit ε-Kanten (ε-nd e.a., oder kurz NEA) A ist ein Tupel $A = (K, \Sigma, \Delta, I, F)$. Dabei ist

- K eine endliche Menge von Zuständen,
- Σ ein endliches Alphabet,
- Δ eine endliche Teilmenge von $(K \times \Sigma^*) \times K$,
- $I \subseteq K$ die Menge von Startzuständen, und
- $F \subseteq K$ die Menge der finalen Zustände

Wir erweitern Δ zu $\Delta^* \subseteq (K \times \Sigma^*) \times K)$ wie folgt:

$$(q, \varepsilon)\ \Delta^*\ q' \quad :\Longleftrightarrow_{def} \quad q' = q \text{ oder } ((q, \varepsilon), q') \in \Delta$$

$$(q, w_1 w_2)\ \Delta^* q' \quad :\Longleftrightarrow_{def} \quad \exists q'' \in K \left(\left((q, w_1), q''\right) \in \left(\Delta \cup \Delta^*\right) \right.$$
$$\left. \text{und } \left((q'', w_2), q'\right) \in \left(\Delta \cup \Delta^*\right) \right)$$

Wie verarbeitet ein ε-nd e.a. ein Wort $w \in \Sigma^*$?

- In einem Schritt von Δ oder
- in mehreren Schritten von Δ^*.

Statt Δ^* schreibt man auch kurz Δ.

Definition 2.12 (ε-nd e.a. akzeptierte Sprache)

Die von einem ε-nd e.a. $A = (K, \Sigma, \Delta, I, F)$ akzeptierte Sprache ist

$$L(A) := \{w \in \Sigma^* \mid \exists s_0 \in I \; \exists q \in F \; \big((s_0, w) \; \Delta^* \; q\big)\}$$

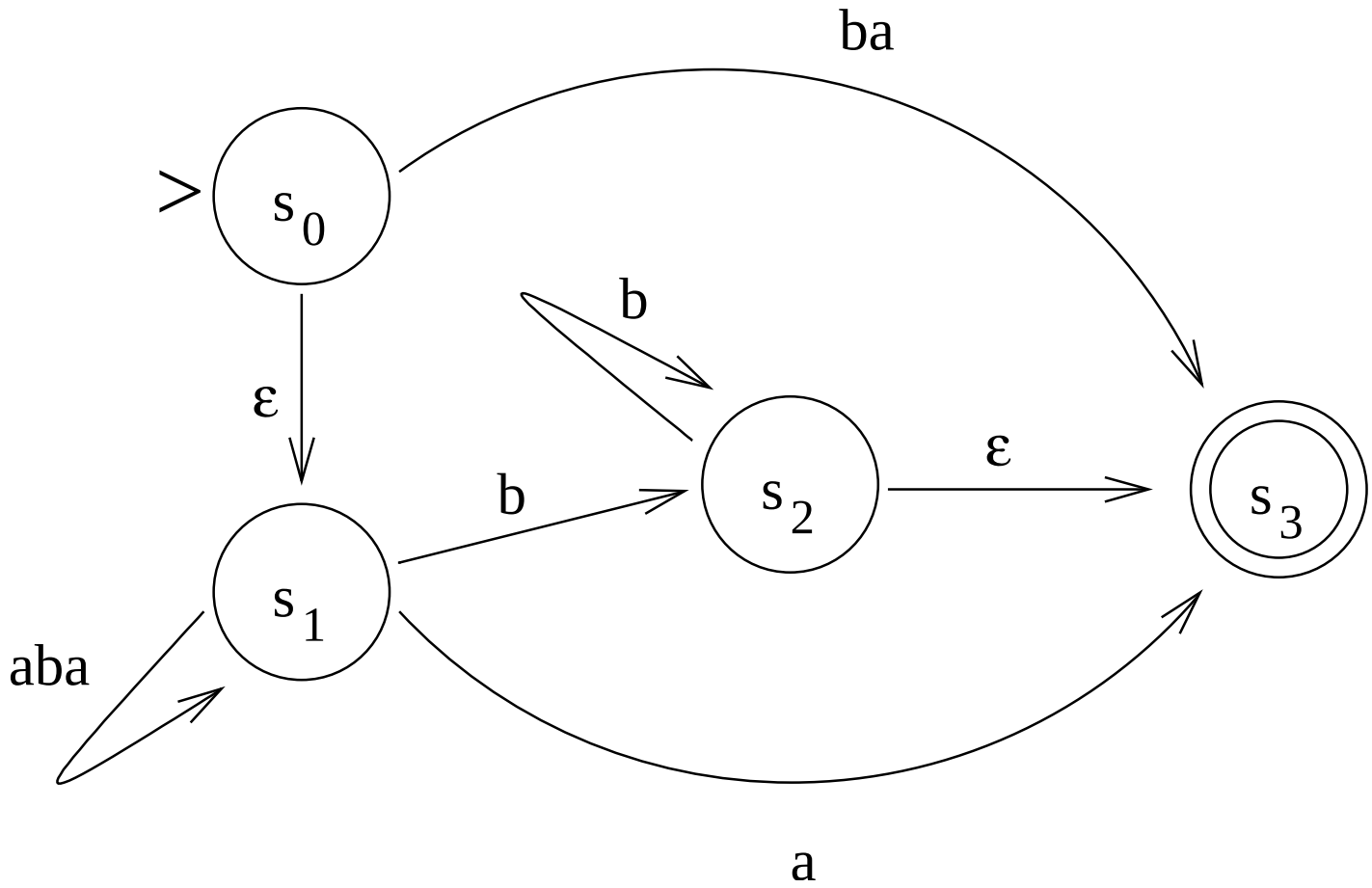


Abbildung 17: Ein Automat mit ε–Kanten für
$\{aba\}^*\{b\}\{b\}^* + \{aba\}^*\{a\} + \{ba\}$

Noch ein Beispiel:

$L = \{abc\}^*\{b\}^* + \{abc\}^*\{bab\}\{b\}^*$

Theorem 2.13 (ε-nd e.a. gleich mächtig wie nd e.a.)

Zu jedem ε-nd e.a. A existiert ein nd e.a. A' mit $L(\mathrm{A}) = L(\mathrm{A}')$.

Beweis.

Idee: Ersetze Übergänge aus A:

- nur mit einem Buchstaben markiert: beibehalten
- mit n Buchstaben markiert: ersetzen durch n Übergänge mit einem Buchstaben
- ε-Übergänge: Wenn in A gilt: $(q, a)\ \Delta\ q'$ und $(q', \varepsilon)\ \Delta\ q''$, dann ersetze $(q', \varepsilon)\ \Delta\ q''$ durch $(q, a)\ \Delta\ q''$.

□

A' enthalte also die Zustände von A und

- für jeden Übergang $(q_1, a)\ \Delta_A\ q_2$ mit $a \in \Sigma$ den Übergang $(q_1, a)\ \Delta_{A'}\ q_2$,
- für jeden Übergang $(q_1, w)\ \Delta_A\ q_2$ für Wörter $w = a_1 \ldots a_n$ einer Länge $n \geq 2$ (wobei $a_i \in \Sigma$ gilt für $1 \leq i \leq n$) neue Zustände $p_{(w,1)}, \ldots, p_{(w,n-1)}$ und die Übergänge

$$\begin{array}{rcl} (q_1, a_1) & \Delta_{A'} & p_{(w,1)} \\ (p_{(w,i)}, a_{i+1}) & \Delta_{A'} & p_{(w,i+1)} \text{ für alle } i < n-1 \\ (p_{(w,n-1)}, a_n) & \Delta_{A'} & q_2 \end{array}$$

- für jeden Übergang $(q_1, \varepsilon)\ \Delta_A^*\ q_2$ im alten und jeden Übergang $(q_0, a)\ \Delta_{A'}\ q_1$ im neuen Automaten auch den Übergang $(q_0, a)\ \Delta_{A'}\ q_2$.

Es sei $F_{A'} := F_A$ und
$I_{A'} := I_A \cup \{q \in K_A \mid \exists s \in I_A \left((s, \varepsilon)\ \Delta_A^*\ q\right)\}$.

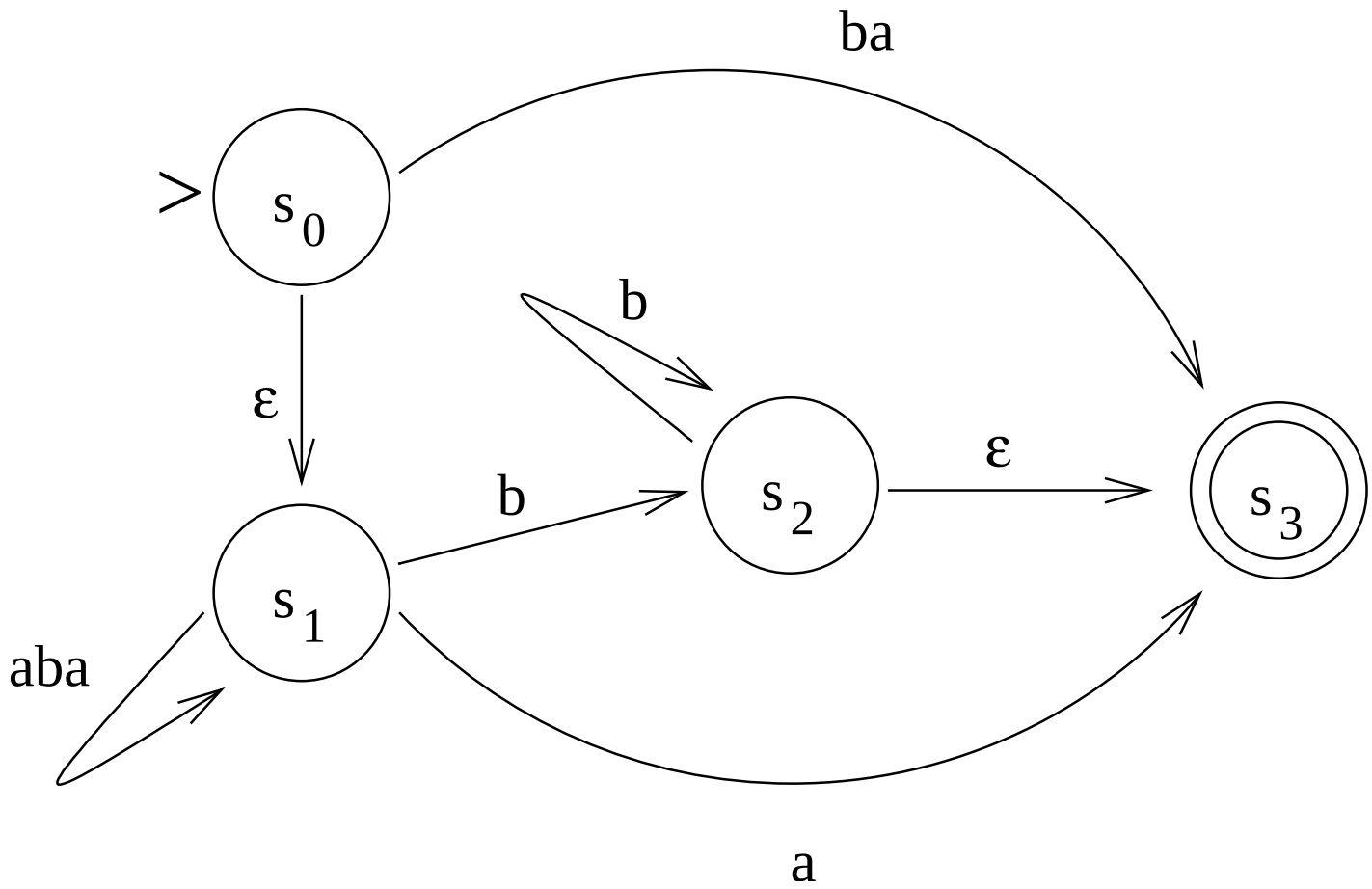


Abbildung 18: Ein Automat mit ε–Kanten für $\{aba\}^*\{b\}\{b\}^* + \{aba\}^*\{a\} + \{ba\}$

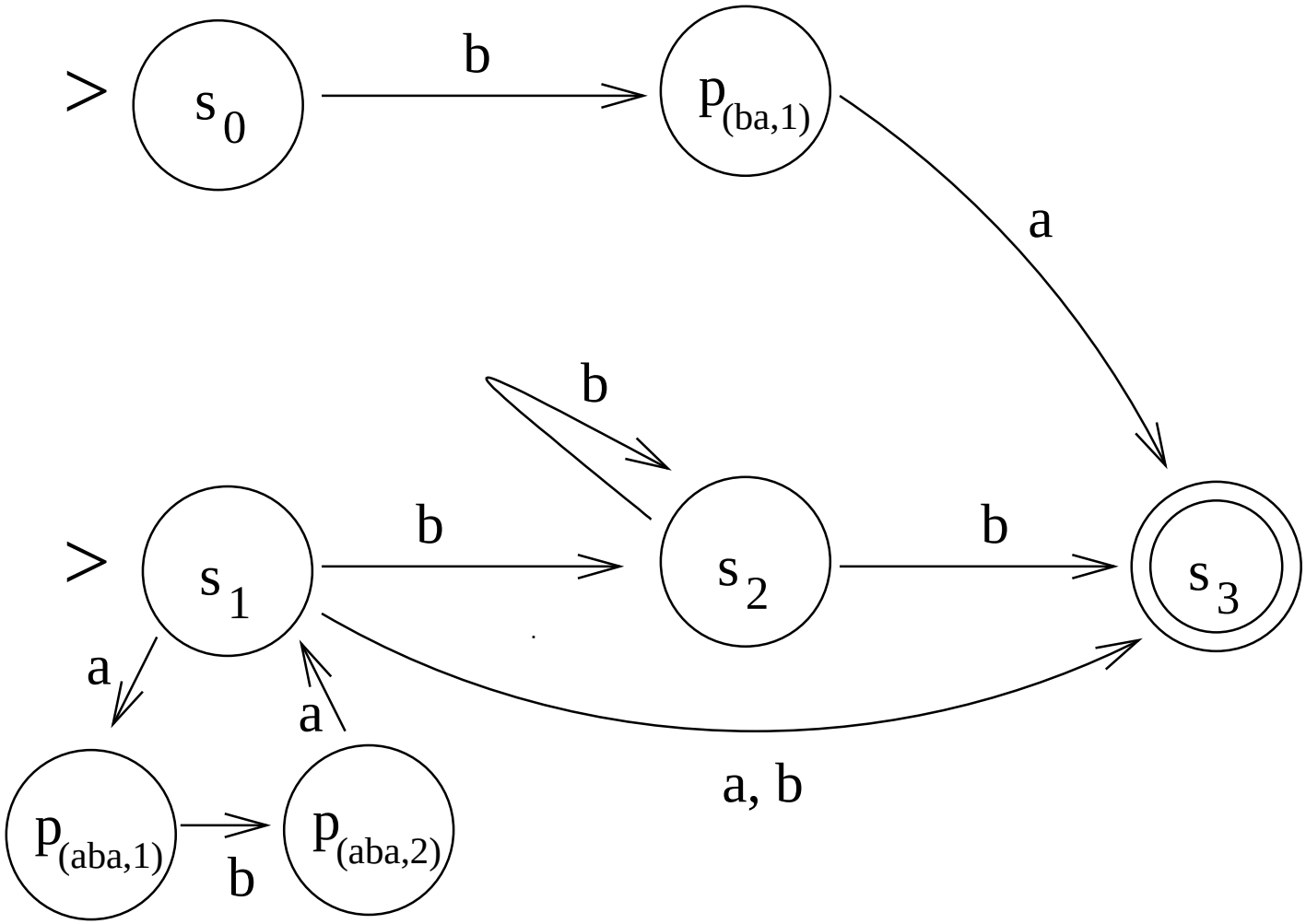


Abbildung 19: Ein indeterminierter Automat, der die gleiche Sprache akzeptiert wie der aus Abb. 18

2.4 rational = Typ 3

Theorem 2.14 (Satz von Kleene: RAT = $\mathbf{L_3}$)

Eine Sprache L ist rational gdw $L \in \mathbf{L}_3$.

Beweis

"$\Rightarrow$" Zu zeigen: **Wenn eine Sprache L von einem endlichen Automaten A akzeptiert wird, kann sie durch eine rechtslineare Grammatik dargestellt werden.**

- Sei also $L = L(A)$,
- A sei ein endlicher Automat mit $A = (K, \Sigma, \delta, s_0, F)$.
- Dazu konstruieren wir eine Grammatik $G = (V, T, R, S)$:

Automat A:	in **Zustand q**,	**liest a**,	geht in **Zustand q'**
Grammatik:	**Endvariable q**,	**erzeugt a**	neue **Endvariable q'**

$V := K$,
$T := \Sigma$,
$S := s_0$, und
$q \to aq' \in R$ falls $\delta(q, a) = q'$,
$q \to \varepsilon \in R$ falls $q \in F$

Mit Induktion über die Länge eines Wortes w kann man nun zeigen, daß gilt: $S \Longrightarrow_G^* wq$ gdw $\delta^*(s_0, w) = q$.

Daraus wiederum folgt

$$\begin{aligned} S \Longrightarrow_G^* w \quad & \textbf{gdw} \quad \exists q \in F \left(S \Longrightarrow_G^* wq \Longrightarrow w \right) \\ & \textbf{gdw} \quad \exists q \in F \left(\delta(s_0, w) = q \right) \\ & \textbf{gdw} \quad w \in L(\mathrm{A}) \end{aligned}$$

"$\Leftarrow$" Zu zeigen: **Wenn eine Sprache L durch eine rechtslineare Grammatik dargestellt werden kann, dann gibt es einen endlichen Automaten, der sie akzeptiert.**

- Sei $G = (V, T, R, S)$ eine rechtslineare Grammatik mit $L = L(G)$.
- Wir konstruieren zu G einen ε-nd e.a. A mit $L = L(A)$.
- Sei $A = (K, \Sigma, \Delta, I, F)$ mit

$$K := V \cup \{q_{stop}\}$$
$$I := \{S\}$$
$$\Sigma := T$$
$$F := \{q_{stop}\}$$

Dabei sei q_{stop} neu, d.h. $q_{stop} \notin V$.

Für Δ definieren wir

$$(X, u)\ \Delta\ X' \quad :\Longleftrightarrow_{def} \quad X \to uX' \in R$$
$$(X, u)\ \Delta\ q_{stop} \quad :\Longleftrightarrow_{def} \quad X \to u \in R$$

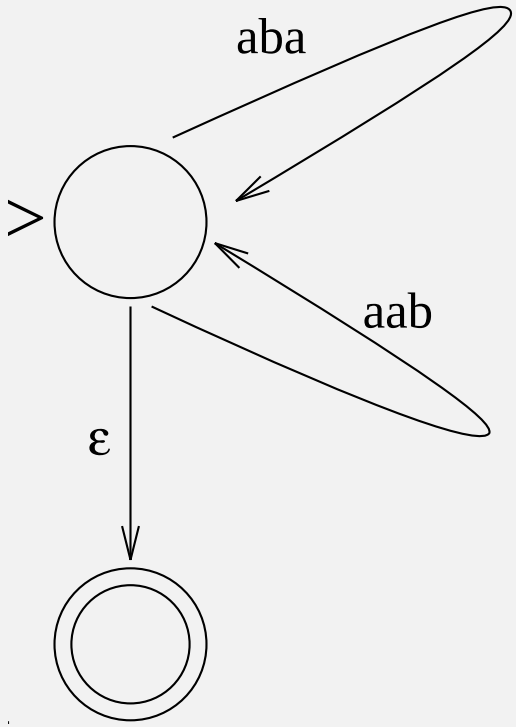
mit $X, X' \in K$ und $u \in \Sigma^*$.

Damit gilt:
$\big(S \Longrightarrow_G^* w\big)$ gdw $\big((S, w)\ \Delta^*\ q_{stop}\big)$ gdw $\big(w \in L(\mathrm{A})\big)$.

Daß dies so ist, kann man durch Induktion über die Länge einer Ableitung in G zeigen. □

Beispiel 2.15

Die Grammatik mit den Regeln $S \to abaS \mid aabS \mid \varepsilon$ wird zu diesem Automaten mit ε-Kanten:



2.5 Pumping Lemma

Wie kann man feststellen, ob eine Sprache regulär ist?

- Rationale Sprachen sind die, die man mit regulären Ausdrücken beschreiben kann (Beweis später).
- **Wie kann dann eine rationale Sprache mit unendlich vielen Wörtern aufgebaut sein?** Es gibt nur einen Operator, um mit einem regulären Ausdruck unendlich viele Wörter zu beschreiben: den Kleene-Stern. *Also müssen sich in den Wörtern einer unendlichen regulären Sprache bestimmte Buchstabenketten beliebig oft wiederholen.*

- Genauso: Wie kann ein endlicher Automat eine unendliche Sprache akzeptieren?
- Er hat nur **endlich** viele Zustände. *Also muss er Schleifen enthalten.*

Ziel: Zeigen, daß manche Sprachen nicht regulär sind.

- Wenn eine Sprache nicht das einfache Schema von regulären Zeichen-Wiederholungen hat,
- dann kann kein endlicher Automat sie akzeptieren.
- Also ist die Sprache nicht regulär.

Theorem 2.16 (Pumping-Lemma für L_3-Sprachen)

*Sei $L \in$ **RAT**. Dann existiert ein $n \in \mathbb{N}$, so daß gilt:*
Für alle $x \in L$ *mit* $|x| \geq n$ **existieren** $u, v, w \in \Sigma^*$ *mit*

- $x = uvw$,
- $1 \leq |v| < n$, *und*
- $uv^m w \in L$ für alle $m \in \mathbb{N}_0$.

- In einer regulären Sprache L
- gibt es zu jedem Wort x in L ab einer bestimmten Länge n neue Wörter uv^iw die auch wieder in der Sprache liegen,
- dabei ist v ein Mittelteil des Wortes x.

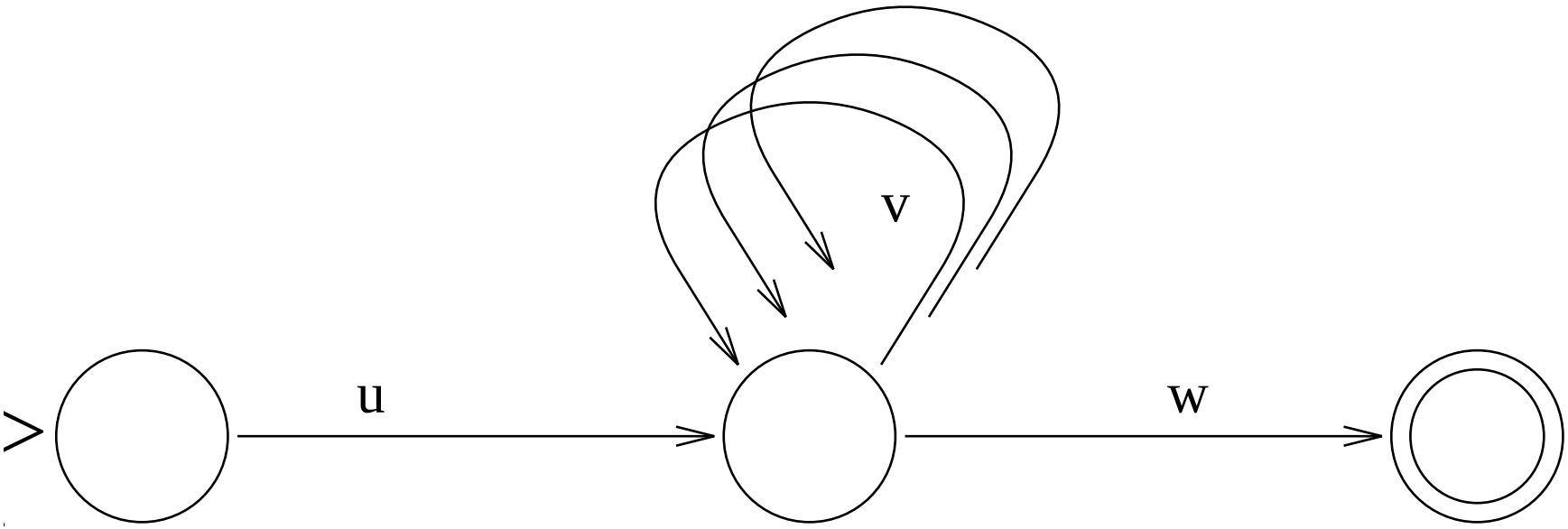


Abbildung 20: Ein Automat, der erst u liest, dann beliebig oft v und dann w

Beweis

Sei L eine reguläre Sprache.

1. Fall: L ist endlich. Sei w_{max} das längste Wort in L. Dann setzen wir n, die Konstante aus dem Satz, auf $n = |w_{max}| + 1$. Dann gibt es keine Wörter $x \in L$, für die $|x| \geq n$ gilt und für die die Bedingungen des Pumping-Lemmas erfüllt sein müßten.

2. Fall: L ist unendlich. Dann gilt:

- in L: beliebig lange Wörter
- erkennender Automat: nur **endlich** viele Zustände.
- Wort länger als Anzahl der Zustände: Dann muss der Automat mindestens einen Zustand q mehrmals durchlaufen.
- Die Schleife, die bei q beginnt und endet, kann der Automat beliebig oft durchlaufen.

Wir wählen folgende Werte:

- Sei $L = L(\mathrm{A})$ und $\mathrm{A} = (K, \Sigma, \delta, s_0, F)$ ein endlicher Automat.
- Für die Konstante n: Wir wählen $n := |K| + 1$.
- Wir betrachten ein beliebiges Wort $x \in L$ mit

$$|x| = t \geq n,$$

$$x = x_1 x_2 \ldots x_t$$

für $x_i \in \Sigma$.

- Seien $q_0, q_1, \ldots, q_t \in K$ die Zustände, die beim Akzeptieren von x durchlaufen werden, mit

$$q_0 = s_0, q_t \in F, \text{ und}$$

$$\delta(q_i, x_{i+1}) = q_{i+1} \quad \forall 0 \leq i \leq t-1$$

Damit gilt: Da $t \geq |K| + 1$ ist, gibt es zwei Werte i und $j \in \{0, \ldots, t\}$, so daß $i \neq j$ und $q_i = q_j$.

Falls $|j - i| \geq |K| + 1$ ist, ist das gleiche Argument wiederum anwendbar, und es gibt zwei weitere, näher beieinander liegende Zustände $q_{i'}, q_{j'}$ im Intervall zwischen q_i und q_j mit $q_{i'} = q_{j'}$.

Also finden wir Werte $i, j \in \{0, \ldots, t\}$ mit $0 < j - i < |K| + 1$ und $q_i = q_j$.

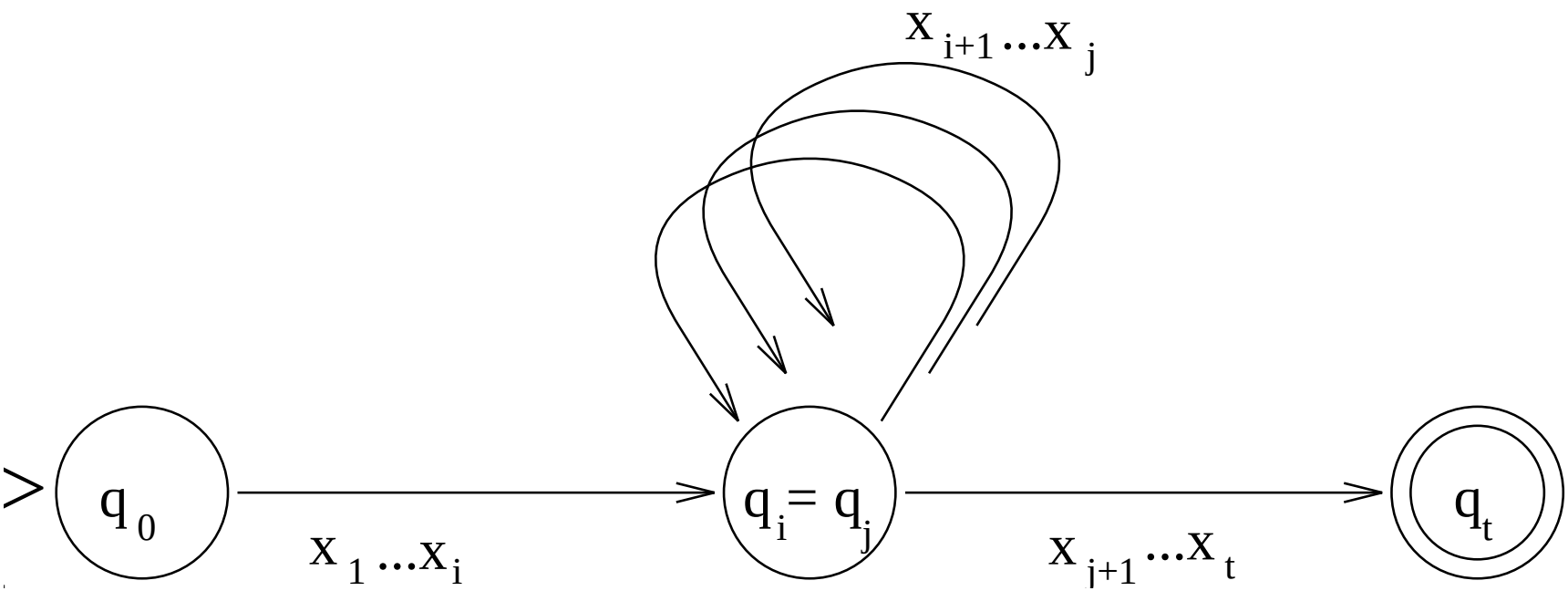


Abbildung 21: Akzeptanz eines **pumpbaren** Wortes x durch einen Automaten

Wir wählen nun

$$\left.\begin{aligned} u &:= x_1 \dots x_i \\ v &:= x_{i+1} \dots x_j \\ w &:= x_{j+1} \dots x_t \end{aligned}\right\} x = uvw \text{ mit } 1 \leq |v| < n.$$

Damit haben wir das gewünschte Ergebnis:

- **Für alle $m \geq 0$ gibt es Wege von q_0 zu q_i mit Beschriftung uv^m, und somit Wege von q_0 nach q_t mit Beschriftung $uv^m w$.**
- **Also gilt $\forall m \in \mathbb{N}_0$, daß $uv^m w \in L$.**
- **Wegen $j - i < |K| + 1 = n$ ist außerdem $|v| < n$.** □

Aus $A \Rightarrow B$ folgt nicht $B \Rightarrow A$!

Das Pumping-Lemma ist **nur** eine Implikation:

Vorsicht: Es gibt Sprachen, die

- nicht rational sind,
- aber für die die Aussage des $\mathbf{L}_3$-Pumping-Lemmas gilt.

Also kann man **nicht** schliessen: Wenn für eine Sprache die Aussage des $\mathbf{L}_3$-Pumping-Lemmas gilt, ist sie rational.

Aber es gilt: Aus $A \Rightarrow B$ folgt $\neg B \Rightarrow \neg A$!

Wenn für eine Sprache die **Aussage des L_3-Pumping-Lemmas nicht gilt**, *dann ist die Sprache* **nicht regulär**.

Wie zeigt man mit dem Pumping-Lemma, daß eine Sprache nicht regulär ist?

Man zeigt, daß es für jedes $n \in \mathbb{N}$ ein Wort der Sprache gibt, das wenn man es entsprechend dem Pumping-Lemma aufpumpt, Wörter hervorbringt, die nicht zu der Sprache gehören.

Beispiel 2.17 (Anwendung Pumping-Lemma)

Folgende Sprachen sind nicht rational

1. $L_1 := \{a^i b a^i \mid i \in \mathbb{N}_0\}$
2. $L_2 := \{a^p \mid p \text{ ist Primzahl}\}$

Beweis

(1) $L_1 := \{a^i b a^i \mid i \in \mathbb{N}\}$.

Es gibt beliebig lange Wörter in L_1. Angenommen, L_1 ist rational. Dann existiert eine Zahl n mit den Eigenschaften des Pumping-Lemmas.

Betrachte das Wort $a^n b a^n \in L_1$. Nach dem Pumping-Lemma existieren Teilwörter u, v, w mit $a^n b a^n = uvw$, so daß die restlichen Aussagen des Pumping-Lemmas gelten.

Die Frage ist nun, wie "liegt" das Teilwort v in $a^n b a^n$?

Es gibt 3 Möglichkeiten, wie sich $a^n b a^n$ auf die Teilwörter u, v, w verteilen kann:

1. $u = a^k, v = a^j, w = a^i b a^n$ mit $i, k \geq 0$, $j > 0$ und $k + j + i = n$.
   Wenn wir nun v einmal aufpumpen, erhalten wir $uv^2w = a^k a^{2j} a^i b a^n = a^{k+2j+i} b a^n = a^{n+j} b a^n \notin L_1$, ein Widerspruch.
2. $u = a^n b a^i, v = a^j, w = a^k$ führt analog zu 1. zum Widerspruch.
3. $u = a^k, v = a^j b a^i, w = a^l$ mit $k + j = i + l = n$ und $i, j, k, l \geq 0$
   Pumpen wir nun wieder v einmal auf.
   $uv^2w = a^k a^j b a^i a^j b a^i a^l = a^{k+j} b a^{i+j} b a^{i+l} \notin L_1$, schon wegen der zwei "$b$" ein Widerspruch.

Also ist L_1 nicht rational.

Beweisschema zur Anwendung des PL's

Behauptung: L ist nicht rational.

- Beweis durch Widerspruch: **Annahme: L ist rational.**
- Für rationale Sprachen, also auch L, gilt das PL.
- Es gibt also ein $n \in \mathbb{N}$, sodaß ...
  // ... die Details des Pumping Lemmas gelten.
- Nun suchen wir ein **"kritisches" Wort** $x \in L$, d.h.
  - $|x| > n$, *d.h. x ist "lang genug" zum Pumpen*
  - *egal (=Fallunterscheidung) wie die Zerlegung $x = uvw$ auch zu liegen kommt, wir geben für jeden Fall ein* **böses m an, sodaß $uv^m w \notin L$.**
- Also gilt für L das $\mathbf{L_3}$-Pumping-Lemma nicht: ein Widerspruch[3]

[3] Der kreative Teil des Beweises ist das kritische Wort in Abhängigkeit von n anzugeben und die Fallunterscheidung sauber durchzuführen., der Rest ist immer gleich!

Theorem 2.18 (Erweitertes Pumping-Lemma für L_3)

*Sei $L \in$ **RAT**. Dann existiert ein $n \in \mathbb{N}$, so daß für alle Wörter $x = abc \in L$ mit $|x| \geq n$ und $|b| = n$ gilt*

- $b = uvw$,
- $|v| \geq 1$,
- für alle $i \in \mathbb{N}_0 : auv^iwc \in L$.

Anschaulich bedeutet dies, daß der zu pumpende Teil v des Wortes x beliebig gewählt werden kann (solange er in einem Teilwort einer gewissen Länge vorkommt).

2.6 Wortprobleme

Kann man bestimmen,

- ob eine gegebene **rechtslineare Grammatik** leer ist?
- ob **zwei rechtslineare Grammatiken** äquivalent sind?
- ob ein **Wort** w in $L(G)$ enthalten ist?

Definition 2.19 (erreichbar, co-erreichbar, trim)

Sei $A = (K, \Sigma, \Delta, I, F)$ ein indeterminierter endlicher Automat. Ein Zustand $q \in K$ heißt

- **erreichbar** $:\Longleftrightarrow_{def} \exists s \in I \, \exists w \in \Sigma^* \, (s, w)\Delta^* q.$
  (Es gibt ein Wort, mit dem q vom Startzustand aus erreicht wird.)
- **co-erreichbar** $:\Longleftrightarrow_{def} \exists w \in \Sigma^* \, \exists f \in F \, (q, w) \, \Delta^* f.$
  (Es gibt ein Wort, mit dem von q aus ein finaler Zustand erreicht wird.)
- **trim** $:\Longleftrightarrow_{def} q$ ist erreichbar und co-erreichbar.

Analog heißt der Automat A **erreichbar**, falls alle $q \in K$ erreichbar sind. A heißt **co-erreichbar**, falls alle $q \in K$ co-erreichbar sind, und A heißt **trim**, falls alle $q \in K$ trim sind.

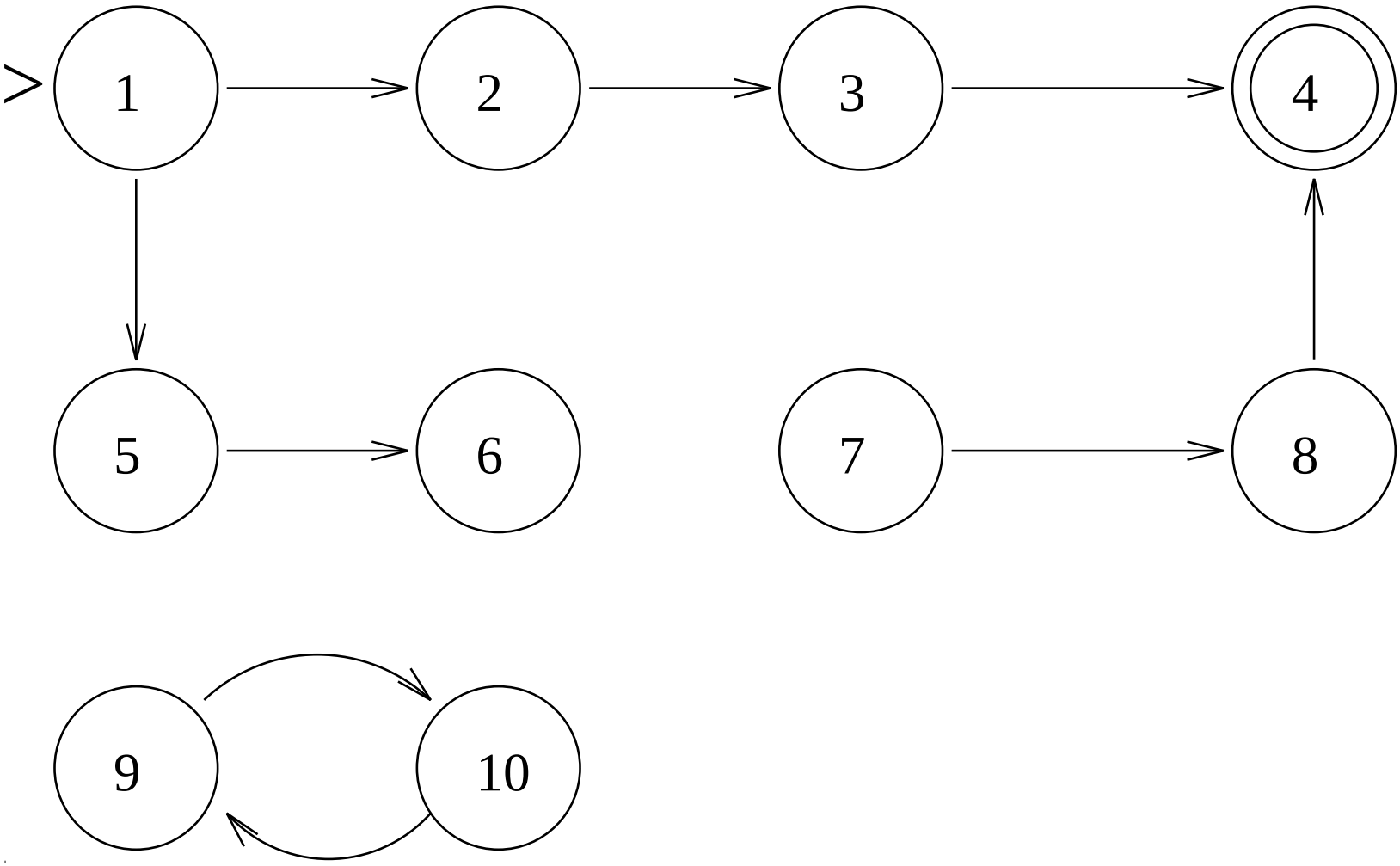


Abbildung 22: In diesem Automaten sind die Zustände 1 – 6 erreichbar, 1 – 4, 7 und 8 co-erreichbar, und die Zustände 1 – 4 sind trim

Kann man bei gegebenem endlichen Automaten feststellen, welche Zustände **erreichbar**, **co-erreichbar** oder **trim** sind?

Definition 2.20 (Teilautomaten)

Seien $A = (K_A, \Sigma_A, \Delta_A, I_A, F_A)$, $A' = (K_{A'}, \Sigma_{A'}, \Delta_{A'}, I_{A'}, F_{A'})$ Automaten. A' heißt Teilautomat von A gdw

$$K_{A'} \subseteq K_A, \quad \Sigma_{A'} \subseteq \Sigma_A, \quad \Delta_{A'} \subseteq \Delta_A, \quad I_{A'} \subseteq I_A, \quad F_{A'} \subseteq F_A$$

A' heißt der von K' erzeugte Teilautomat von A gdw

- $K' \subseteq K_A$,
- $A' = (K', \Sigma_A, \Delta_A \cap ((K' \times \Sigma_A) \times K'), I_A \cap K', F_A \cap K')$.

Diese Teilautomaten kann man leicht konstruieren.

Definition 2.21 ($A^{\mathbf{err}}$, $A^{\mathbf{co-e}}$, $A^{\mathbf{trim}}$)

$A^{\mathbf{err}}$ ist der von den erreichbaren Zuständen von A erzeugte Teilautomat von A.

$A^{\mathbf{co-e}}$ ist der von den co-erreichbaren Zuständen von A erzeugte Teilautomat von A.

$A^{\mathbf{trim}}$ ist der von den trimmen Zuständen von A erzeugte Teilautomat von A.

Lemma 2.22

1. $A^{trim} = (A^{err})^{co-e} = (A^{co-e})^{err}$.
2. $L(A) = L(A^{err}) = L(A^{co-e}) = L(A^{trim})$.

Lemma 2.23

Sei A ein endlicher Automat. Es ist entscheidbar, *ob die Menge*

1. $L(A)$ *leer ist.*
2. $L(A)$ *unendlich ist.*

Beweis

Zu (i): Um festzustellen, ob $L(A) = \emptyset$ ist, berechnet man zu dem e.a. A den erreichbaren nd. e.a. A^{err} (man eliminiert also alle die Zustände von A, die nicht vom Startzustand aus erreichbar sind). $L(A) = L(A^{err})$ ist genau dann nicht leer, wenn A^{err} noch finale Zustände hat.

Zu (ii): Um festzustellen, ob $L(A)$ unendlich ist, eliminiert man aus A^{err} alle nicht co-erreichbaren Zustände und erhält den nd. e.a. A^{trim}. $L(A) = L(A^{trim})$ ist offensichtlich genau dann unendlich, wenn in der graphischen Darstellung von A^{trim} ein Kreis enthalten ist. □

Dies funktioniert auch für nd e.a.'en.

Lemma 2.24

Seien A_1, A_2, A_3 endliche Automaten. Es ist entscheidbar, ob folgendes gilt:

(i) $L(A_1) \cap L(A_2) = \emptyset$.

(ii) $L(A_1) \cup L(A_2) = L(A_3)$.

(iii) $L(A_1) = L(A_2)$.

Beweis

Seien A_1, A_2 endliche Automaten.

Zu (i): Es gibt einen endlichen Automaten $A_\cap$ mit $L(A_\cap) = L(A_1) \cap L(A_2)$ (**warum?**).
Nach dem obigen Lemma ist die Frage, ob $L(A_\cap) = \emptyset$ ist, entscheidbar. Also ist auch die Frage, ob $L(A_1) \cap L(A_2) = \emptyset$ ist, entscheidbar.

Zu (ii): Man kann zu A_1 und A_2 einen ea A_3' effektiv konstruieren, mit: $L(A_3') = L(A_1) \cup L(A_2)$ (wie in (i)). Von diesem Automaten konstruiert man effektiv den Komplement-Automaten A_3''. Nun wendet man (1) auf A_3'' und A_3: dies gilt genau dann wenn $L(A_1) \cup L(A_2) = L(A_3)$.

Zu (iii): Man kann zu A_1 und A_2 einen endlichen Automaten $A_=$ konstruieren mit $L(A_=) = (L(A_1) \cap \overline{L(A_2)}) \cup (\overline{L(A_1)} \cap L(A_2))$. (**Warum?**).
Wenn man nun nachrechnet, stellt man fest, daß $\big(L(A_1) = L(A_2)$ gdw $L(A_=) = \emptyset\big)$ ist. Die Frage, ob $L(A_=) = \emptyset$ ist, ist nach dem obigen Lemma entscheidbar, also gilt Gleiches für die Frage, ob $L(A_1) = L(A_2)$ ist. □

Korollar

An diesen Beweisen kann man sehen, daß $A_\cap$, $A_\cup$ und $A_=$ effektiv aus A_1 und A_2 konstruierbar sind.

Angenommen, für zwei e.a.'en gilt $L(A_1) = L(A_2)$. Muss dann auch $A_1{}^{\mathbf{trim}} = A_2{}^{\mathbf{trim}}$ gelten?

2.7 Rational = Regulär

Bisher haben wir oft von **regulären Sprachen** gesprochen, wenn wir **Sprachen vom Typ 3**, also Sprachen die rechtslinearen Grammatiken entsprechen, meinten. Richtiger hätten wir von **rationalen** Sprachen sprechen müssen.

Dieser Sprachgebrauch hat sich eingebürgert. Allerdings kann man unter **regulären Sprachen** auch solche Sprachen verstehen, die durch **reguläre Ausdrücke** definiert werden können.

Wir zeigen nun, daß Sprachen die durch reguläre Ausdrücke definiert werden können, genau die **rationalen Sprachen** sind, also die, die durch einen **endlichen Automaten** charakterisiert werden können.

Theorem 2.25 (Hauptsatz von Kleene)

Die durch endliche Automaten akzeptierten Sprachen (also die rationalen) sind genau die, die man durch reguläre Ausdrücke beschreiben kann.

Beweis

"$\Rightarrow$" Dies zeigen wir durch Induktion über den Weg während des Akzeptierens eines Wortes im Automaten A. OBdA numerieren wir die Zustände durch und setzen: $s_0 = q_1$.

$$R^k_{i,j} := \{w \in \Sigma^* : \delta^*(q_i, w) = q_j \text{ und für alle Präfixe } u \text{ von } w \text{ mit } \varepsilon \neq u \neq w \text{ gilt } \delta^*(q_i, w) \in \{q_1, q_2, \ldots, q_k\}\}$$

Beweis: "$\Rightarrow$"

$R^k_{i,j}$ beschreibt alle Wörter, die zwischen q_i und q_j in A durchlaufen werden können, mit der Einschränkung, daß nur $\{q_1, q_2, \ldots, q_k\}$ als Zwischenzustände aufgesucht werden dürfen.

Offensichtlich ist

$$L(\mathrm{A}) = \bigcup_{q_f \in F} R^n_{1,f}$$

Es genügt daher zu zeigen, daß alle Mengen $R^n_{1,f}$ durch reguläre Ausdrücke beschrieben werden können.
Dazu zeigen wir, durch Induktion über k:

$$R^k_{i,j} \in \mathfrak{Reg}_\Sigma \text{ für alle } i, j \leq n$$

Beweis: "$\Rightarrow$"

Induktionsanfang: $k = 0$

$$R^0_{i,j} = \begin{cases} \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{falls } i \neq j \\ \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(q_i, a) = q_j\} & \text{falls } i = j \end{cases}$$

$R^0_{i,j}$ kann man offensichtlich als reguläre Ausdrücke darstellen.

Induktionsschritt: $k \to k+1$: Induktionsvoraussetzung: $R^k_{i,j} \in \mathfrak{Reg}_\Sigma$ für alle $i, j \leq n$. Solange wir also nur $q_1, ...q_k$ als Zwischenzustände zulassen, können wir einen regulären Ausdruck angeben.
Nun erlauben wir zusätzlich q_{k+1} als Zwischenzustand:

Beweis: "$\Rightarrow$"

Welche Wege gibt es von q_i nach q_j, falls $\{q_1, \ldots, q_k, q_{k+1}\}$ als Zwischenzustände erlaubt sind?

- q_{k+1} wird gar nicht angelaufen, d.h. der Weg ist schon in $R^k_{i,j}$ beschrieben.
- Wir laufen von q_i zu q_{k+1} (erster Besuch!), laufen dann beliebig oft eine Schleife, d.h. von q_{k+1} nach q_{k+1} (letzter Besuch!), und von dort zu q_j. Formal: $R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}$

macht insgesamt:

$$R^{k+1}_{i,j} = R^k_{i,j} \cup R^k_{i,k+1}(R^k_{k+1,k+1})^* R^k_{k+1,j}$$

Beweis: "$\Rightarrow$"

$$R_{i,j}^{k+1} = R_{i,j}^{k} \cup R_{i,k+1}^{k}(R_{k+1,k+1}^{k})^* R_{k+1,j}^{k}$$

Für $R_{i,j}^{k+1}$ verwenden wir nur Vereinigung, Konkatenation und den Kleene-Stern.

Wir erinnern uns an Definition 1.3 (Reguläre Ausdrücke) und schließen, da - gemäß der Induktionsvoraussetzung - alle Mengen auf der rechten Seite als reguläre Ausdrücke beschreibbar sind, gilt das auch für den Gesamtausdruck $R_{i,j}^{k+1}$. □

Beweis: "$\Leftarrow$"

Wir zeigen nun, daß jeder reguläre Ausdruck auch durch einen endl. Automaten akzeptiert wird. Dies geht, wie üblich, durch Induktion über den Aufbau regulärer Ausdrücke. Es genügt zu zeigen, daß dies für ε-NEAs gilt[a].

Induktionsanfang: Offensichtlich gibt es NEAs, die den regulären Ausdrücken "0" und "a" entsprechen: Startzustand und (davon verschiedener) Finalzustand. Im Fall "0" gibt es keine Übergänge, im Fall "a" gibt es einen Übergang für a.

[a] denn aus diesen lassen sich ja äquivalente DEAs konstruieren

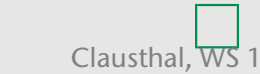
Beweis: "$\Leftarrow$"

Induktionsschritt: Wir müssen zeigen, daß es ε-NEAs zu $R + S$, RS und zu R^* gibt, unter der Voraussetzung, daß es ε-NEAs A_R und A_S gibt.

Für A_{R+S}: siehe Beweis von Theorem 2.24.

Für A_{RS} schalten wir die Automaten A_R und A_S hintereinander (und ergänzen ε-Übergänge von den Finalzuständen von A_R in die Startzustände von A_S).

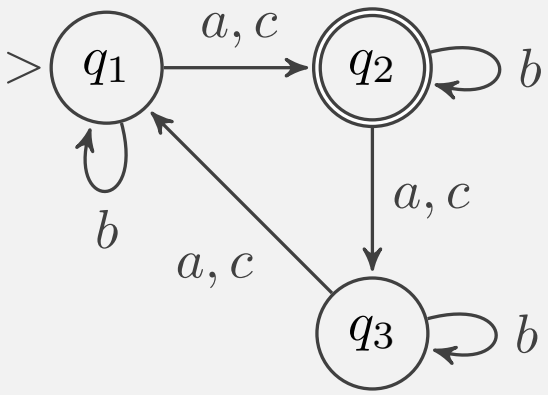
A_{R^*} konstruiert man, indem man zunächst einen neuen akzeptierenden Startzustand einführt und von diesem zu jedem ehemaligen Startzustand einen ε-Übergang. Zusätzlich wird von jedem alten Finalzustand ein ε-Übergang zu allen ehemaligen Startzuständen hinzugefügt.



Aus dem Beweis ergeben sich sofort folgende Konstruktionsverfahren:

- regulärer Ausdruck $\rightarrow$ e.a;
- e.a. $\rightarrow$ regulärer Ausdruck.

Beispiel 2.26 (e.a. $\rightarrow$ reg. Ausdruck)



$$R^0_{1,1} = R^0_{2,2} = R^0_{3,3} = \{b\} \cup \{\varepsilon\}, r^0_{1,1} = r^0_{2,2} = r^0_{3,3} = b + 1$$
$$R^0_{1,2} = R^0_{2,3} = R^0_{3,1} = \{a, c\}, r^0_{1,2} = r^0_{2,3} = r^0_{3,1} = a + c$$
$$R^1_{1,1} = R^0_{1,1} \cup R^0_{1,1}(R^0_{1,1})^* R^0_{1,1}, r^1_{1,1} = \cdots = b^*$$
$$R^1_{2,2} = R^0_{2,2} \cup R^0_{2,1}(R^0_{1,1})^* R^0_{1,2}, r^1_{2,2} = r^0_{2,2} = a + c$$
$$R^1_{3,3} = R^0_{3,3} \cup R^0_{3,1}(R^0_{1,1})^* R^0_{1,3}, r^1_{3,3} = r^0_{3,3} = a + c$$
$$R^1_{1,2} = R^0_{1,2} \cup R^0_{1,1}(R^0_{1,1})^* R^0_{1,2},$$
$$r^1_{1,2} = a + c + (b + 1)b^*(a + c) = b^*(a + c)$$
$$R^2_{1,2} = \ldots, r^2_{1,2} = \cdots = b^*(a + c)b^*$$
$$R^3_{1,2} = \ldots, r^3_{1,2} = \cdots = ((b^*(a + c))^3)^* b^*(a + c)b^*$$

Entscheidungsprobleme für reguläre Sprachen

Sei $L, L_1, L_2 \in \mathbf{L_3}$ gegeben, d.h. wir haben Automaten A, A_1, A_2 mit $L = L(A)$, $L_1 = L(A_1)$ und $L_2 = L(A_2)$

Problem	Input	Frage	Entscheidbar?
Wortproblem	$w \in \Sigma^*$	$w \in L$?	**Ja**
Leerheitsproblem		$L = \emptyset$?	**Ja**
Endlichkeitsproblem		$\|L\|$ endlich?	**Ja**
Äquivalenzproblem		$L_1 = L_2$?	**Ja**

Abschlusseigenschaften für reguläre Sprachen

Sei $L, L_1, L_2 \in \mathbf{L_3}$ gegeben, d.h. wir haben Automaten A, A_1, A_2 mit $L = L(A)$, $L_1 = L(A_1)$ und $L_2 = L(A_2)$

Operation	Frage	Konstruktion	Antwort
Vereinigung	$L_1 \cup L_2 \in \mathbf{L_3}$?	Produkt-Automat	**Ja**
Schnitt	$L_1 \cap L_2 \in \mathbf{L_3}$?	Produkt-Automat	**Ja**
Konkatenation	$L_1 \circ L_2 \in \mathbf{L_3}$?	ε-Kanten zw. A_1, A_2	**Ja**
Stern	$L^* \in \mathbf{L_3}$?	ε-Kanten von F zu I	**Ja**
Negation	$\overline{L} \in \mathbf{L_3}$?	e.a. negieren	**Ja**

3. Kellerautomaten (cf)

Inhalt

In diesem Kapitel erweitern wir e.a.'en zu Kellerautomaten (Push-Down Automata (PDA)). Sie erfahren,

1. daß die von PDA'en erkannten Sprachen genau die vom Typ 2 (kontextfrei) sind;
2. daß kontextfreie Sprachen in verschiedene Normalformen (Chomsky, Greibach) gebracht werden können;
3. daß es Kriterien (Pumping Lemma, Ogden's Lemma) gibt, um eine Sprache als nicht kontextfrei nachzuweisen;
4. daß es Algorithmen gibt, um Probleme über PDA'en, bzw. Typ 2-Sprachen zu lösen (z.B. den Cocke-Younger-Kasami Algorithmus).

Zur Erinnerung: kontextfreie Grammatiken

- kontextfreie Regel: Eine Variable wird durch ein Wort ersetzt, egal in welchem Kontext die Variable steht
- Grammatik $G = (V, T, R, S)$ heißt kontextfrei (cf) gdw für alle $P \to Q \in R$ $\big(P \in V$ und $Q \in (V \cup T)^*\big)$.
  Es wird eine einzelne Variable ersetzt. Das Wort in der Conclusio kann Variablen und Terminale in beliebiger Mischung enthalten.

Zum Vergleich: rechtslineare Grammatiken

Eine Grammatik $G = (V, T, R, S)$ heißt rechtslinear gdw für alle $P \to Q \in R$ $\big(P \in V$ und $Q \in T^* \cup T^+V\big)$.

Es wird eine einzelne Variable ersetzt. Mit einer Regelanwendung wird jeweils höchstens eine Variable erzeugt, die, wenn sie auftritt, ganz rechts im Wort steht.

Zur Erinnerung: kontextfreie Sprachen

- $L_1 = \{a^n b^n \mid n \in \mathbb{N}_0\}$
- $L_2 = \{ww^R \mid w \in \{a,b\}^*\}$

Beispiel:

L_1: Grammatik $S \to aSb \mid \varepsilon$

- Ableitung $S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aabb$

L_2: Grammatik $S \to aSa \mid bSb \mid \varepsilon$

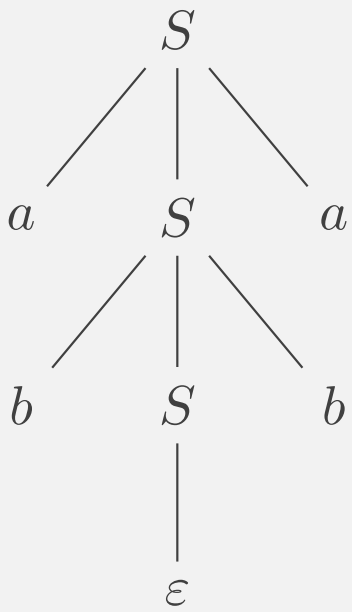
- Ableitung $S \Longrightarrow aSa \Longrightarrow abSba \Longrightarrow abba$

Was ist mit $L_2 = \{ww \mid w \in \{a,b\}^*\}$?

3.1 Ableitungsbäume

Beispiel 3.1

- $L_{Pal} = \{ww^R \mid w \in \{a,b\}^*\}$
- Grammatik: $S \to aSa \mid bSb \mid \varepsilon$



Definition 3.2 (Ableitungsbaum zu einer Grammatik)

Sei $G = (V, T, R, S)$ eine cf-Grammatik. Ein Ableitungsbaum (parse tree) zu G ist ein angeordneter, knotengewichteter Baum $B = (W, E, v_0)$, für den gilt:

- Jeder Knoten $v \in W$ ist mit einem Symbol aus $V \cup T \cup \{\varepsilon\}$ markiert.
- Die Wurzel v_0 ist mit S markiert.
- Jeder innere Knoten ist mit einer Variablen aus V markiert.

- Jedes Blatt ist mit einem Symbol aus $T \cup \{\varepsilon\}$ markiert.
- Ist $v \in W$ ein innerer Knoten mit Kindern $v_1, \ldots, v_k$ in dieser Anordnung, ist A die Markierung von v und A_i die Markierung von v_i, so ist $A \to A_1 \ldots A_k \in R$.
- Ein mit ε markiertes Blatt hat keine Geschwister (denn das entspräche einer Ableitung wie $A \to ab\varepsilon Bc$).

Ablesen eines Wortes vom Ableitungsbaum:

- Ableitung $S \overset{*}{\underset{G}{\Longrightarrow}} w$ erzeugt Wort w.
- Wo findet sich das im Ableitungsbaum?
- Blätter von links nach rechts durchlesen.

Was heißt dabei „von links nach rechts“ genau?

Idee:

- Ableitungsbäume sind **angeordnet**, d.h. es gibt eine Ordnung unter den Kindern eines Knotens.
- Daraus definieren wir eine Ordnung unter den Blättern, die ja nur entfernt „verwandt“ sein müssen.

Für Blätter $b_1, b_2 \in W$ definieren wir:

$b_1 < b_2$ gdw b_1, b_2 sind Geschwister, und b_1 liegt „links“ von b_2,
oder $\exists v, v_1, v_2 \in W \;\; v \to v_1, v \to v_2, v_1 < v_2$
und v_i ist Vorfahre von b_i für $i \in \{1, 2\}$.

Zwei weitere Begriffe zu Ableitungsbäumen:

- Sei $\{b_1, \ldots, b_k\}$ die Menge aller Blätter in B mit $b_1 < \ldots < b_k$, und sei A_i die Markierung von b_i, so heißt das Wort $A_1 \ldots A_k$ die **Front** von B.
- Ein **A-Baum** in B ist ein Unterbaum von B, dessen Wurzel mit A markiert ist.

Theorem 3.3

Sei $G = (V, T, R, S)$ eine cf-Grammatik. Dann gilt für $w \in T^*$:
$(S \Longrightarrow_G^* w)$ gdw Es gibt Ableitungsbaum in G mit Front w.

Beweis.

Kurze Skizze, da beide Beweisrichtungen offensichtlich sind.
Wir beweisen eine stärkere Aussage (w ist aus $(V \cup T)^*$ und S eine beliebige Variable A):
Zu G und w existiert ein Ableitungsbaum B, so daß

$(A \Longrightarrow_G^* w)$ gdw es gibt einen A-Baum in B mit Front w

„$\Rightarrow$“ durch Induktion über die Länge von Ableitungen
„$\Leftarrow$“ durch Induktion über die Tiefe von A-Bäumen

□

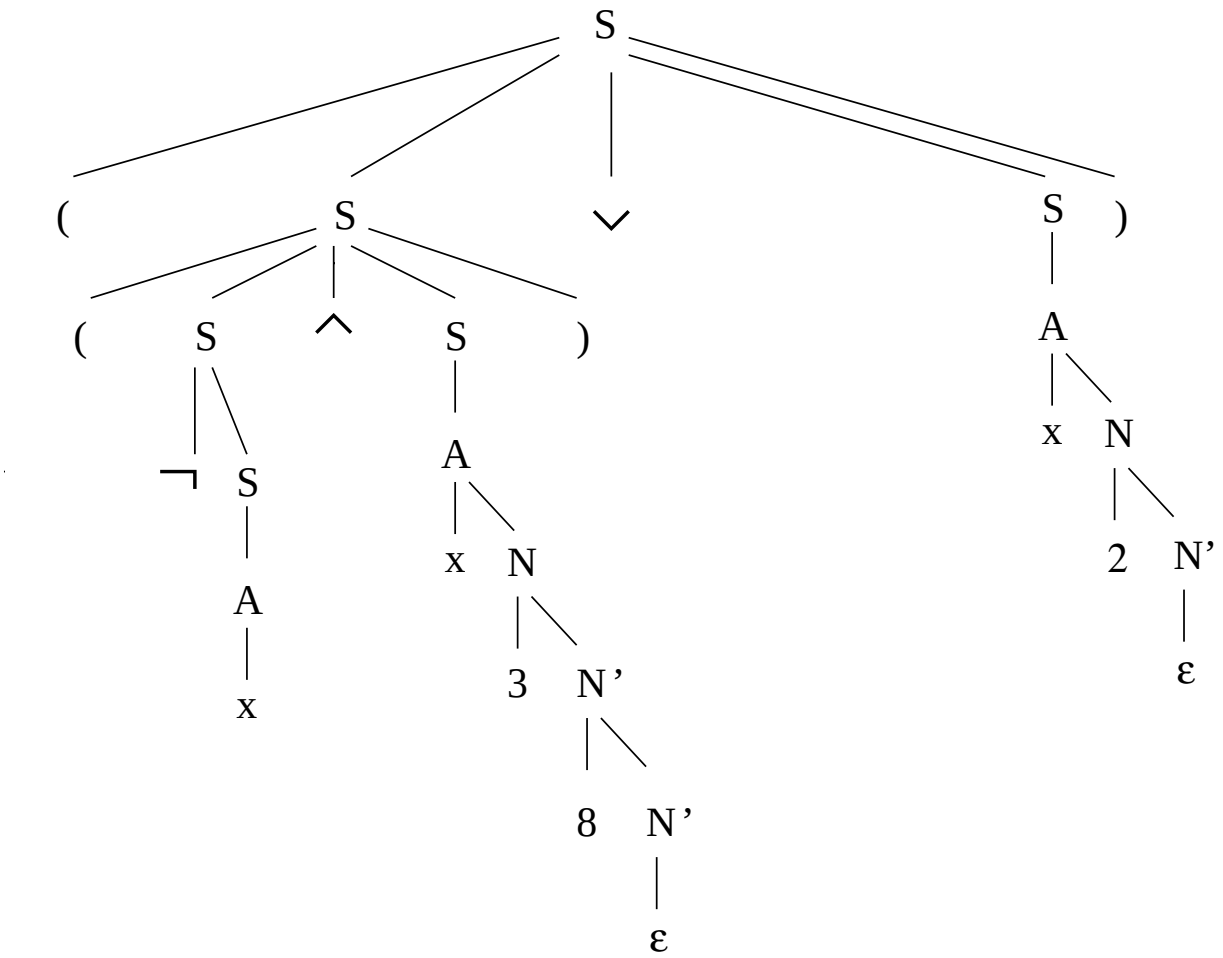


Abbildung 23: Ableitungsbaum für $((\neg x \wedge x38) \vee x2)$

Beispiel 3.4

Die Menge aller aussagenlogischen Formeln über den Variablen $\{x, x_0, x_1, x_2, \ldots\}$:

- Beispiel: $((x_1 \vee x_5) \wedge (\neg x_{791} \wedge x_{47}))$
- erzeugende Grammatik: $G = (\{S, A, N, N'\}, \{x, 0, \ldots, 9, (,), \wedge, \vee, \neg\}, R, S)$ mit der Regelmenge

$$\begin{aligned} R = \{S &\to (S \wedge S) \mid (S \vee S) \mid \neg S \mid A \\ A &\to x \mid xN \\ N &\to 1N' \mid 2N' \mid \ldots \mid 9N' \mid 0 \\ N' &\to 0N' \mid 1N' \mid \ldots \mid 9N' \mid \varepsilon\} \end{aligned}$$

Abbildung 23 zeigt den Ableitungsbaum für $((\neg x \wedge x38) \vee x2)$.

Der Ableitungsbaum steht für viele **äquivalente** Ableitungen, darunter diese:

$$
\begin{array}{rcl}
S & \Rightarrow & (S \vee S) \Rightarrow \\
((S \wedge S) \vee S) & \Rightarrow & ((\neg S \wedge S) \vee S) \Rightarrow \\
((\neg A \wedge S) \vee S) & \Rightarrow & ((\neg x \wedge S) \vee S) \Rightarrow \\
((\neg x \wedge A) \vee S) & \Rightarrow & ((\neg x \wedge xN) \vee S) \Rightarrow \\
((\neg x \wedge x3N') \vee S) & \Rightarrow & ((\neg x \wedge x38N') \vee S) \Rightarrow \\
((\neg x \wedge x38) \vee S) & \Rightarrow & ((\neg x \wedge x38) \vee A) \Rightarrow \\
((\neg x \wedge x38) \vee xN) & \Rightarrow & ((\neg x \wedge x38) \vee x2N') \Rightarrow \\
((\neg x \wedge x38) \vee x2) & &
\end{array}
$$

Ähnlich: Aufbau arithmetischer Ausdrücke

$$S \to (S + S) \mid (S * S) \mid (S - S) \mid (S/S) \mid Z$$

$\ldots$

Definition 3.5 (Linksableitung)

Eine Ableitung $w_1 \Longrightarrow_G w_2 \Longrightarrow_G \ldots \Longrightarrow_G w_n$ heißt Linksableitung falls w_{i+1} durch Ersetzen der Variable ganz links in w_i entsteht für alle $i < n$.
Die Rechtsableitung ist analog definiert.

Ablesen aus dem Ableitungsbaum:

- **Linksableitung:** Tiefensuche. Der Ast ganz links wird zuerst durchsucht
- **Rechtsableitung:** Tiefensuche, der Ast ganz rechts zuerst.

Definition 3.6 (Mehrdeutigkeit)

Eine cf-Grammatik G heißt **mehrdeutig** gdw es ein Wort $w \in L(G)$ gibt, so daß G **zwei verschiedene Linksableitungen** zu w besitzt.
Eine Sprache $L \in \mathbf{L}_2$ heißt **inhärent mehrdeutig** gdw alle kontextfreien Grammatiken für L mehrdeutig sind.

Eine Grammatik G ist mehrdeutig, wenn es zwei verschiedene Ableitungsbäume in G mit gleicher Front gibt.

Bei Programmiersprachen möchte man **keine Mehrdeutigkeit**.

Beispiel 3.7

Grammatik von vorhin war eindeutig: Regeln

$$
\begin{aligned}
S &\to (S \wedge S) \mid (S \vee S) \mid \neg S \mid A \\
A &\to x \mid xN \\
N &\to 1N' \mid 2N' \mid \ldots \mid 9N' \mid 0 \\
N' &\to 0N' \mid 1N' \mid \ldots \mid 9N' \mid \varepsilon\}
\end{aligned}
$$

Weitere Grammatik für AL-Formeln:

$$G = (\{K, D, L, A\}, \{v, w, x, y, z, (,), \wedge, \vee, \neg\}, R, K)$$

$$
\begin{aligned}
R = \{\ & K \to K \wedge K \mid D \\
& D \to (D \vee D) \mid L \\
& L \to \neg A \mid A \\
& A \to v \mid w \mid x \mid y \mid z\}
\end{aligned}
$$

Klammer-Ersparnisregel!

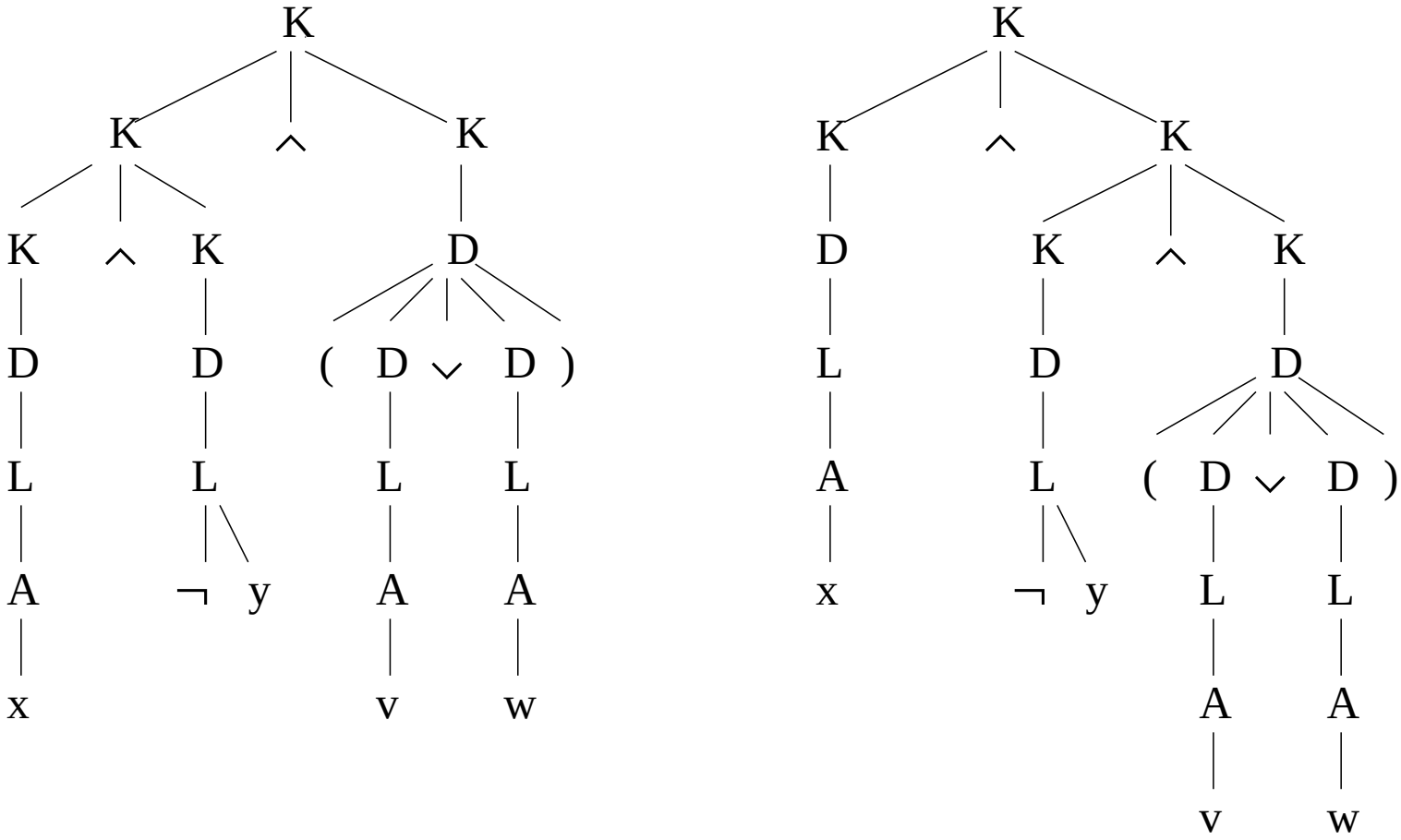


Abbildung 24: Zwei Ableitungsbäume für $x \wedge \neg y \wedge (v \vee w)$

Beispiel 3.8

Die Sprache $L := \{a^i b^j c^k \mid i = j \text{ oder } j = k\}$ ist inhärent mehrdeutig.

Machen Sie sich klar, was man hierzu beweisen muß: **über alle** Grammatiken und **über alle** möglichen Ableitungen in jeder Grammatik.

3.2 Umformungen

Im Folgenden soll für alle cf-Grammatiken gelten:
Das Startsymbol S kommt nie auf einer rechten Regelseite vor.

Ist das bei einer Grammatik nicht gegeben, kann man einfach

- ein neues Startsymbol S_{neu}, sowie
- die Regel $S_{neu} \to S$

einführen.

Überflüssige Symbole und überflüssige Regeln in cf-Grammatiken

- Variablen,
  - aus denen nie ein terminales Wort werden kann,
  - die man nicht vom Startsymbol S aus ableiten kann,
- Ableitungen $\ldots \Longrightarrow wAv \Longrightarrow wv$ mit $A \to \varepsilon$,
- Ableitungen $\ldots \Longrightarrow wAv \Longrightarrow wBv$ mit $A \to B$.

Definition 3.9 ((co-) erreichbare, nutzlose Symbole)

Sei $G = (V, T, R, S)$ eine Grammatik. Ein Symbol $x \in (V \cup T)$ heißt

erreichbar: Es gibt $\alpha, \beta \in (V \cup T)^*$: $S \Longrightarrow_G^* \alpha x \beta$
(x kommt in einem Wort vor, das von S aus erzeugt werden kann.)

co-erreichbar: Es gibt $w \in T^*$: $x \Longrightarrow_G^* w$
(Aus x kann eine Terminalkette werden.)

nutzlos: x ist nicht erreichbar oder nicht co-erreichbar.

Theorem 3.10 (cf-Grammatik ohne nutzlose Symbole)

Ist $G = (V, T, R, S)$ eine cf-Grammatik mit $L(G) \neq \emptyset$, dann existiert eine cf-Grammatik $G' = (V', T', R', S')$ mit:

- *G' ist äquivalent zu G.*
- *Jedes $x \in (V \cup T)$ ist erreichbar und co-erreichbar.*
- *Man kann G' aus G effektiv konstruieren.*

Beweis

Eine Variable A ist **co-erreichbar**, wenn gilt:

- Entweder es gibt eine Regel $A \to w$, w Kette von Terminalen
- Oder es gibt eine Regel $A \to w$, w Kette von Terminalen und co-erreichbaren Variablen

Der folgende Algorithmus sammelt alle co-erreichbaren Variablen einer Grammatik $G = (V, T, R, S)$ in Neu_1:

$Alt_1 := \emptyset$
$Neu_1 := \{A \in V \mid \exists w \in T^* \ (A \to w \in R)\}$

while $Alt_1 \neq Neu_1$
{
$\quad Alt_1 := Neu_1$
$\quad Neu_1 := Alt_1 \cup \{A \in V \mid \exists \alpha \in (T \cup Alt_1)^* \ (A \to \alpha \in R)\}$
}

- Bestimmung einer Grammatik $G'' = (V'', T'', R'', S'')$ nur mit diesen co-erreichbaren Variablen:

if $S \in Neu_1$ $\quad /* S\ ist\ co-erreichbar */$
{
$V'' := Neu_1$
$T'' := T$
$R'' := R \cap \left(V'' \times (V'' \cup T'')^*\right)$
$S'' := S$
}
else return $\quad /* L(G) = \emptyset */$

- Ein Symbol x ist **erreichbar**, wenn gilt:
  - Entweder es gibt eine Regel $S \to wxv$.
  - Oder es gibt eine Regel $A \to wxv$ für eine erreichbare Variable A.

Der folgende Algorithmus sammelt in $Neu2$ alle erreichbaren Symbole von G'':

$$
\begin{aligned}
&Alt_2 := \emptyset \\
&Neu_2 := \{S\} \\
&\textbf{while } Alt_2 \neq Neu_2 \\
&\{ \\
&\quad Alt_2 := Neu_2 \\
&\quad Neu_2 := Alt_2 \cup \{x \in (V'' \cup T'') \mid \exists A \in Alt_2 \\
&\qquad\qquad\qquad\qquad \exists \alpha, \beta \in (V'' \cup T'')^* \\
&\qquad\qquad\qquad\qquad (A \to \alpha x \beta \in R)\} \\
&\}
\end{aligned}
$$

- Bestimmung der Grammatik G' ohne **nutzlose Symbole**:

$$G' := (V', T', R', S') \texttt{ mit}$$
$$V' := Neu_2 \cap V''$$
$$T' := Neu_2 \cap T$$
$$R' := R'' \cap \left(V' \times (V' \cup T')^*\right)$$
$$S' := S$$

Damit gilt dann: $L(G') = L(G)$, und G' enthält keine nutzlosen Symbole. □

Einschränkung des Formats von Grammatiken

Theorem 3.11 (Normalform)

Für die Sprachklassen $\mathbf{L}_0$ bis $\mathbf{L}_3$ gilt: Zu jeder Grammatik G existiert eine äquivalente Grammatik G', bei der für alle Regeln $P \to Q \in R'$ gilt:

- $Q \in V^*$ *(ohne Beschränkung für P), oder*
- $Q \in T$, *und* $P \in V$.

Für alle Sprachklassen außer $\mathbf{L}_3$ hat G' denselben Typ wie G.

Beweis.

Für jedes Terminal $t \in T$ erzeuge man eine neue Variable V_t.

- $V' = V \cup \{V_t \mid t \in T\}$
- R' entsteht aus R, indem für jede Regel $P \to Q \in R$ in Q alle Vorkommen eines Terminals t durch die zugehörige Variable V_t ersetzt werden. Außerdem enthält R' für jedes $t \in T$ eine neue Regel $V_t \to t$.

Also $L(G') = L(G)$, und für alle Sprachklassen außer $\mathbf{L}_3$ hat G' denselben Typ wie G. □

Nullbare Variablen

- nullbare Variable: ε in mehreren Schritten ableitbar
- **Idee:** Nullbarkeit vermeiden, lieber gleich nur Variablen erzeugen, aus denen nichtleere Teilwörter werden.

Definition 3.12 (ε-Regel, nullbare Variablen)

Eine Regel der Form $P \to \varepsilon$, wobei P Variable ist, heißt ε-Regel. Eine Variable A heißt nullbar, falls $A \Longrightarrow^* \varepsilon$ möglich ist.

Theorem 3.13 (ε-Regeln sind eliminierbar)

Zu jeder cf-Grammatik G existiert eine äquivalente cf-Grammatik G'

- *ohne ε-Regeln und nullbare Variablen, falls $\varepsilon \notin L(G)$,*
- *mit der einzigen ε-Regel $S \to \varepsilon$ und der einzigen nullbaren Variablen S, falls $\varepsilon \in L(G)$ und S das Startsymbol ist.*

Beweis

Sei $G = (V, T, R, S)$ eine kontextfreie Grammatik, und S komme o.E. in keiner Regelconclusio vor. Wir stellen zunächst mit folgendem Algorithmus fest, welche Variablen aus V nullbar sind.

$Alt := \emptyset$
$Neu := \{A \in V \mid A \to \varepsilon \in R\}$
while $Alt \neq Neu$
{ $Alt := Neu$
for all $(P \to Q) \in R$ **do**
{ **if** $Q = A_1 \ldots A_n$ **and** $A_i \in Neu$ für $1 \leq i \leq n$
and $P \notin Neu$,
then return $Neu := Neu \cup \{P\}$
}
}

Jetzt:

- Vorhanden: Menge $Alt \subseteq V$ der nullbaren Variablen
- Daraus konstruieren: Regelsatz R' einer Grammatik $G' = (V, T, R', S)$ ohne nullbare Variablen (außer eventuell S).

Ausgangsgrammatik G habe die Form aus Satz 3.11: Für jede Regel $P \to Q$ gelte $Q \in V^*$ oder $Q \in T$.

$R' := R$
for all $P \to A_1 \dots A_n \in R \, mit \, P, A_i \in V$:
{
$Generiere \, alle \, Regeln \, P \to \alpha_1 \dots \alpha_n \, mit$
if $A_i \Longrightarrow^* \varepsilon$
$\alpha_i := \varepsilon$ **oder** $\alpha_i := A_i$
else return
$\alpha_i := A_i$
if $\alpha_1 \dots \alpha_n \neq \varepsilon$
$R' = R' \cup \{P \to \alpha_1 \dots \alpha_n\}$
}
$Streiche \, aus \, R' \, alle \, RegelnA \to \varepsilon \, mit \, A \neq S.$

Zu zeigen: $L(G') = L(G)$
Vorgehen:

- G hat die Form von Satz 3.11:
  Für jede Regel $P \to Q$ gilt $Q \in V^*$ oder $Q \in T$.
- Wir beweisen die etwas stärkere Behauptung

$$\text{für alle } A \in V \text{ für alle } w \in (V \cup T)^* - \{\varepsilon\}$$
$$\big((A \Longrightarrow_G^* w) \quad \underline{\text{gdw}} \quad (A \Longrightarrow_{G'}^* w)\big),$$

- Daraus folgt sofort $L(G') = L(G)$.

„$\Rightarrow$“ Wir zeigen: Aus $A \Longrightarrow_G^* w$ folgt $A \Longrightarrow_{G'}^* w$ (Induktion über die Länge einer Ableitung von A nach w in G).

Induktionsanfang: Länge = 0.

Dann ist $w = A$, und $A \Longrightarrow_{G'}^* A$ gilt immer.

Induktionsschritt: Es sei schon gezeigt: Wenn in G in n Schritten eine Ableitung $B \Longrightarrow_G^* u$ durchgeführt werden kann, dann folgt, daß in G' die Ableitung $B \Longrightarrow_{G'}^* u$ möglich ist.

Außerdem gelte in der Ausgangsgrammatik G:
$A \Longrightarrow_G^* w \neq \varepsilon$ in $n+1$ Schritten.
Dann gilt:

- $A \Longrightarrow_G w' \Longrightarrow_G^* w$,
- $w' = A_1 \dots A_\ell \Longrightarrow_G^* w_1 \dots w_\ell = w$,
- und es wird jeweils A_i zu w_i in höchstens n Schritten für geeignete $w', A_1, \dots, A_\ell, w_1, \dots, w_\ell$.
- Per Induktionsvoraussetzung gilt also schon:
  - Entweder $A_i \Longrightarrow_{G'}^* w_i$
  - oder $w_i = \varepsilon$ für $1 \leq i \leq \ell$.

Fall 1: $w_i = \varepsilon$, A_i ist nullbar.
Dann gibt es in G' eine Regel $A \to A_1 \dots A_{i-1} A_{i+1} \dots A_\ell$ nach der obigen Konstruktionsvorschrift für G', falls $A_1 \dots A_{i-1} A_{i+1} \dots A_\ell \neq \varepsilon$. Das ist der Fall, denn sonst hätten wir: $A \Longrightarrow w' = \varepsilon \Longrightarrow^* w = \varepsilon$ (aus nichts wird nichts), aber $w = \varepsilon$ ist ausgeschlossen.

Fall 2: $w_i \neq \varepsilon$. Dann gilt nach Induktionsvoraussetzung $A_i \Longrightarrow^*_{G'} w_i$.

Wir haben also Folgendes gezeigt:

Sei $I = \{i \in \{1 \dots \ell\} \mid w_i \neq \varepsilon\} \neq \emptyset$.

Dann gibt es in R' eine Regel $A \to A_{i_1} \dots A_{i_m}$ mit $I = \{i_1, \dots, i_m\}$, und die A_i sind so angeordnet wie in der ursprünglichen Regel $A \to A_1 \dots A_\ell$.

Mit dieser neuen Regel können wir w so ableiten:

$$A \Longrightarrow_{G'} A_{i_1} \dots A_{i_m} \Longrightarrow^*_{G'} w_{i_1} \dots w_{i_m} = w$$

„$\Leftarrow$“ Wir zeigen: Aus $A \Longrightarrow^*_{G'} w$ folgt $A \Longrightarrow^*_{G} w$ (Induktion über die Länge einer Ableitung von A nach w in G'):

Induktionsanfang: Länge ist 0. Dann ist $w = A$, und $A \Longrightarrow^*_{G} A$ gilt immer.

Induktionsschritt: Es gelte für alle Ableitungen $A \Longrightarrow^*_{G'} w$ einer Länge von höchstens n, daß $A \Longrightarrow^*_{G} w$. Ist $A \Longrightarrow^*_{G'} w$ eine Ableitung der Länge $n+1$, so gibt es ein ℓ, Wörter $w_1, \ldots, w_\ell$ und Variablen $A_1, \ldots, A_\ell$ mit $A \Longrightarrow_{G'} A_1 \ldots A_\ell \Longrightarrow^*_{G'} w = w_1 \ldots w_\ell$. Es gilt jeweils $A_i \Longrightarrow^*_{G'} w_i$ in höchstens n Schritten, und $w_i \neq \varepsilon$.

Nach der Induktionsvoraussetzung folgt daraus:

- für die Originalgrammatik G gibt es Ableitungen $A_i \Longrightarrow_G^* w_i$
- damit gibt es auch eine Ableitung $A_1 \ldots A_\ell \Longrightarrow_G^* w$.

Da es in G' eine Ableitung $A \Longrightarrow_{G'} A_1 \ldots A_\ell$ gibt, gibt es in R' eine Regel $A \to A_1 \ldots A_\ell$. Wie ist diese Regel aus R entstanden?
Eine Regel in R' entsteht aus einer Regel in R, indem einige nullbare Variablen gestrichen werden. Es gab also in G nullbare Variablen B_1 bis B_m, so daß R die Regel

$$A \to A_1 \ldots A_{\ell_1} B_1 A_{\ell_1+1} \ldots A_{\ell_2} B_2 \ldots A_m B_m A_{m+1} \ldots A_\ell$$

enthält. (m kann auch 0 sein, dann war die Regel selbst schon in R.)

Also gilt in G:

$$A \Longrightarrow_G A_1 \dots A_{\ell_1} B_1 A_{\ell_1+1} \dots A_{\ell_2} B_2 \dots A_m B_m A_{m+1} \dots A_\ell$$
$$\Longrightarrow_G^* A_1 \dots A_{\ell_1} A_{\ell_1+1} \dots A_{\ell_2} \dots A_m A_{m+1} \dots A_\ell \Longrightarrow_G^* w$$

da ja $B_i \Longrightarrow_G^* \varepsilon$ möglich ist. □

Beispiel 3.14

Für die Regelmenge R in der linken Spalte sind die Variablen A, B, C nullbar.
Der obige Algorithmus erzeugt aus R die rechts aufgeführte Regelmenge R'.

R:	R':
$S \to ABD$	$S \to ABD \mid AD \mid BD \mid D$
$A \to ED \mid BB$	$A \to ED \mid BB \mid B$
$B \to AC \mid \varepsilon$	$B \to AC \mid A \mid C$
$C \to \varepsilon$	
$D \to d$	$D \to d$
$E \to e$	$E \to e$

Beobachtung:

- Der Algorithmus lässt Variablen zurück, die nicht in Prämissen auftauchen (und deshalb nicht co-erreichbar sind).
  Hier: C.
- Der Algorithmus lässt nutzlose Regeln zurück.
  Hier: $B \to AC \mid C$.

Lemma 3.15

Mit Hilfe des letzten Satzes sieht man, daß $\mathbf{L}_2 \subseteq \mathbf{L}_1$ gilt.

Beweis

Regeln einer kontextsensitiven Grammatik:

- entweder $uAv \to u\alpha v$
  mit $u, v, \alpha \in (V \cup T)^*, |\alpha| \geq 1, A \in V$
- oder $S \to \varepsilon$
  und S kommt in keiner Regelconclusio vor.

Ergebnis des vorigen Satzes:

zu jeder kontextfreien Grammatik G gibt es eine äquivalente Grammatik G' mit

- G' enthält keine ε-Regeln, mit der Ausnahme $S \to \varepsilon$ für den Fall, daß $\varepsilon \in L(G)$.
- Außerdem kommt S in keiner Regelconclusio vor, wie wir o.E. am Anfang dieses Abschnitts angenommen haben.

Damit ist G' kontextfrei und kontextsensitiv.

Also ist die von der kontextfreien Grammatik G erzeugte Sprache $L(G)$ auch kontextsensitiv. $\square$

Definition 3.16 (Kettenproduktion)

Eine Regel der Form $A \to B$ mit $A, B \in V$ heißt Kettenproduktion.

Theorem 3.17 (Kettenproduktionen sind eliminierbar)

Zu jeder cf-Grammatik existiert eine äquivalente cf-Grammatik ohne Kettenproduktionen.

Beweis

Sei $G = (V, T, R, S)$ eine kontextfreie Grammatik ohne ε-Regeln, außer ggf. $S \to \varepsilon$.
Der folgende **Algorithmus** generiert einen neuen **Regelsatz R' ohne Kettenproduktionen**.

$R' = R$
for all $A \in V$
  for all $B \in V, B \neq A$
    if test$(A \Longrightarrow^* B)$
      then return for all $rules\ B \to \alpha \in R,\ \alpha \notin V$
        $R' := R' \cup \{A \to \alpha\}$
$Streiche\ alle\ Kettenproduktionen\ in\ R'.$

$$
\begin{array}{l}
\mathit{procedure}\ \textbf{test}(A \Longrightarrow^* B) \\
\{ \\
\quad Alt\ :=\ \emptyset \\
\quad Neu\ :=\ \{C \in V \mid C \to B \in R\} \\
\\
\quad \textbf{while}\ Alt\ \neq\ Neu\ \textbf{do} \\
\quad \{ \\
\qquad Alt\ :=\ Neu \\
\qquad Neu\ :=\ Alt\ \cup \{D \in V \mid \exists C \in Alt\ (D \to C \in R)\} \\
\quad \} \\
\quad \textbf{if}\ A \in Neu\ \textbf{then return}\ TRUE\ \textbf{else return}\ FALSE \\
\}
\end{array}
$$

Die Funktion test, die auf $A \Longrightarrow^* B$ prüft, geht genauso vor wie der Algorithmus zur Ermittlung co-erreichbarer Variablen.
Die Ausgangsgrammatik $G = (V, T, R, S)$ und die neue Grammatik $G' = (V, T, R', S)$ sind offensichtlich äquivalent. □

Theorem 3.18 (Normalform für cf. Grammatiken)

Zu jeder cf-Grammatik existiert eine äquivalente cf-Grammatik

- *ohne ε-Regeln (bis auf $S \to \varepsilon$, falls ε zur Sprache gehört; in diesem Fall darf S in keiner Regelconclusio vorkommen),*
- *ohne nutzlose Symbole,*
- *ohne Kettenproduktionen,*
- *so daß für jede Regel $P \to Q$ gilt: entweder $Q \in V^*$ oder $Q \in T$.*

Beweis

1. Man teste zunächst, ob S nullbar ist. Falls ja, dann verwende man S_{neu} als neues Startsymbol und füge die Regeln $S_{neu} \to S \mid \varepsilon$ zum Regelsatz hinzu.
2. Man eliminiere nutzlose Symbole.
3. Man eliminiere alle ε-Regeln außer $S_{neu} \to \varepsilon$.
4. Man bringe die Grammatik in die Form des Satzes 3.11.
5. Man eliminiere Kettenproduktionen.
6. Zum Schluss eliminiere man noch einmal alle nutzlosen Symbole (wg. 2. und 3.).

Der letzte Schritt führt keine neuen Regeln ein, also ist die resultierende Grammatik immer noch in der Form des Satzes 3.11.

3.3 Normalformen

Grammatiktypen und Normalformen

- Grammatiktypen **rechtslinear, kontextfrei, kontextsensitiv** sind definiert als
  Einschränkungen der Form, die Prämisse P und Conclusio Q einer Grammatikregel $P \to Q$ haben dürfen.

- **Normalformen** von Grammatiken sind definiert als
  Einschränkungen der Form, die Prämisse P und Conclusio Q einer Grammatikregel $P \to Q$ haben dürfen.

  Eine **Normalform für cf-Grammatiken** z.B. schränkt die Form von cf-Grammatikregeln weiter ein.

Was ist der Unterschied?

Die unterschiedlichen Grammatik-Klassen der Chomsky-Hierarchie unterscheiden sich darin, welche Sprachklassen sie generieren können.

cf-Grammatiken in Normalform generieren dieselben Sprachen wie die Ausgangsgrammatik.

Aber man kann mit ihnen besser Beweise führen und interessante Eigenschaften zeigen. Dies nutzen wir später beim CYK Algorithmus aus.

Wozu dann Normalformen?

Weniger Variationsmöglichkeiten in der Form einer Regel $\rightarrow$ weniger Fallunterscheidungen bei

- Algorithmen, die mit Grammatiken arbeiten.
- Beweisen allgemein über alle Grammatiken eines bestimmten Typs

- **Chomsky**-Normalform: baut auf den Umformungen auf, die wir eben gesehen haben.
- **Greibach**-Normalform: ähnlich den rechtslinearen Grammatiken.

Definition 3.19 (Chomsky-Normalform)

Eine cf-Grammatik $G = (V, T, R, S)$ ist in **Chomsky-Normalform (CNF)**, wenn gilt:

- G hat nur Regeln der Form
  $A \to BC$ mit $A, B, C \in V$ und
  $A \to a$ mit $A \in V, a \in T$
  Ist $\varepsilon \in L(G)$, so darf G zusätzlich die Regel $S \to \varepsilon$ enthalten. In diesem Fall darf S in keiner Regelconclusio vorkommen.
- G enthält keine nutzlosen Symbole.

Theorem 3.20 (Chomsky-Normalform)

Zu jeder cf-Grammatik existiert eine äquivalente cf-Grammatik in Chomsky-Normalform.

Beweis

- Schritt 1: Wir wenden auf G die Umformungen von Satz 3.18 an.
  Ergebnis: G hat keine nutzlosen Symbole, und alle Regeln haben die Form
  1. $A \to \alpha$ mit $A \in V$ und $\alpha \in V^*$, $|\alpha| \geq 2$, und
  2. $A \to a$ mit $A \in V$, $a \in T$

- Regeln von Typ (I): weil die Grammatik in der Form aus Satz 3.11 ist und somit nach Satz 3.18 auch keine Kettenproduktionen enthält.
- Regeln vom Typ (II): in Chomsky-Normalform erlaubt.

- Schritt 2: Regeln vom Typ (I) so umformen, so daß keine Conclusio eine Länge größer 2 hat.
  Wir ersetzen also jede Regel

$$A \to A_1 \dots A_n \text{ mit } A, A_i \in V, n \geq 3$$

durch die folgenden neuen Regeln:

$$\begin{aligned} A &\to A_1 C_1 \\ C_1 &\to A_2 C_2 \\ &\vdots \\ C_{n-2} &\to A_{n-1} A_n \end{aligned}$$

Dabei sind die C_i neue Variablen in V. □

Definition 3.21 (Greibach-Normalform)

Eine cf-Grammatik $G = (V, T, R, S)$ ist in Greibach-Normalform (GNF), wenn gilt:

- G hat nur Regeln der Form

  $$A \to a\alpha \text{ mit } A \in V \text{ und } a \in T \text{ und } \alpha \in V^*$$

  Ist $\varepsilon \in L(G)$, so darf G zusätzlich die Regel $S \to \varepsilon$ enthalten. In diesem Fall darf S in keiner Regelconclusio vorkommen.
- G enthält keine nutzlosen Symbole.

3.4 Pumping-Lemma für cf

Pumping-Lemma für $\mathbf{L}_3$

- L regulär und unendlich,
- dann enthalten alle ihre Wörter, die länger sind als eine Grenzlänge n, ein Infix v einer Länge $< n$,
- dies v kann man beliebig oft wiederholen und erhält wieder Wörter aus L.
- Das Pumping-Lemma für $\mathbf{L}_3$ ist eine notwendige, aber nicht hinreichende Bedingung für $\mathbf{L}_3$.
- Man benutzt das $\mathbf{L}_3$-Pumping-Lemma, um zu zeigen, daß Sprachen nicht regulär sind.

Ähnlich: Das Pumping-Lemma für $\mathbf{L}_2$.

Theorem 3.22 (Pumping Lemma für cf Sprachen)

Zu jeder kontextfreien Sprache $L \subseteq \Sigma^*$ existiert ein $n \in \mathbb{N}$, so daß gilt: Für alle $z \in L$ mit $|z| \geq n$ existieren $u, \boldsymbol{v}, w, \boldsymbol{x}, y, \in \Sigma^*$, so daß

- $z = u\boldsymbol{v}\,w\,\boldsymbol{x}y$,
- $\boldsymbol{v}\boldsymbol{x} \neq \varepsilon$,
- $|\boldsymbol{v}w\,\boldsymbol{x}| < n$ *und*
- $u\boldsymbol{v}^i w\boldsymbol{x}^i y \in L$ *für alle* $i \in \mathbb{N}_0$

Beweis

Beweisidee: Argument über die Form kontextfreier Grammatikregeln und über die Form von Ableitungsbäumen: **was passiert, wenn in einer Ableitung dieselbe Variable zweimal auftritt**?

Wir wenden das PL auf folgende Sprachen an

- $\{a^p : p \text{ ist Primzahl}\}$,
- $\{a^n b^n c^n : n \in \mathbb{N}\}$,
- $\{zzz : z \in \{a,b\}^*\}$.

Theorem 3.23 (Ogden's Lemma)

Zu jeder kontextfreien Sprache $L \subseteq \Sigma^*$ existiert ein $n \in \mathbb{N}$, so daß gilt: Für alle $z \in L$ mit $|z| \geq n$ und für jede Markierung von mindestens n Positionen in z (zusammenhängend oder nicht) existieren $u, \boldsymbol{v}, w, \boldsymbol{x}, y, \in \Sigma^*$, so daß

- $z = u\boldsymbol{v}\,w\,\boldsymbol{x}y$,
- $\boldsymbol{v}$ und $\boldsymbol{x}$ enthalten zusammen mindestens eine markierte Position,
- $\boldsymbol{v}w\,\boldsymbol{x}$ enthält höchstens n markierte Positionen,
- $u\boldsymbol{v}^i w\boldsymbol{x}^i y \in L$ für alle $i \in \mathbb{N}_0$.

3.5 PDAs

Erzeugende Grammatiken – akzeptierende Automaten

- Erinnerung: reguläre Sprachen
  - werden erzeugt von rechtslinearen Grammatiken
  - werden akzeptiert von endlichen Automaten
- jetzt: kontextfreie Sprachen
  - werden erzeugt von kontextfreien Grammatiken
  - werden akzeptiert von **Pushdown-Automaten**

Rechtslineare Grammatiken $\leftrightarrow$ e.a.

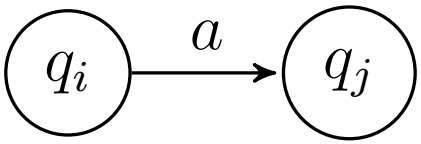
Eine Grammatik $G = (V, T, R, S)$ heißt rechtslinear gdw

für alle $P \to Q \in R$ gilt $\big(P \in V$ und $Q \in T^* \cup T^+V\big)$.

Es wird eine einzelne Variable ersetzt. Mit einer Regelanwendung wird jeweils höchstens eine Variable erzeugt, die, wenn sie auftritt, ganz rechts im Wort steht.

Die Korrespondenz zwischen endlichen Automaten und rechtslinearen Gramatiken, wurde über die Ähnlichkeit von Zustandsübergängen und rechtslinearen Regeln bewiesen.

Zustandsübergang



Regel

$$Q_i \to aQ_j$$

Erinnerung: Kontextfreie Grammatiken

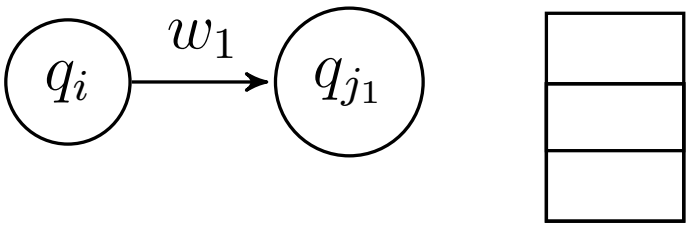
Eine Grammatik $G = (V, T, R, S)$ heißt kontextfrei (cf)

gdw für alle $P \to Q \in R$ gilt $\big(P \in V$ und $Q \in (V \cup T)^*\big)$.

Eine einzelne Variable wird - unabhängig vom Kontext - in der Conclusio durch ein Wort aus Variablen und Terminale in beliebiger Mischung ersetzt.

Ziel: Beweisidee recyclen! Dazu notwendig: mehrere Nichtterminale in einem Zustandsübergang verarbeiten[4].

Zustandsübergang



Regel

$$Q_i \to w_1 Q_{j_1} \dots Q_{j_n} w_{n+1}$$

[4] Die Eichhörnchen-Lösung wird sein: "Pack alles Zusätzliche in den Keller"

Beispiel 3.24 (Kontextfreie Sprachen und PDAs)

- $L_1 = \{a^n b^n \mid n \in \mathbb{N}_0\}$
- Grammatik. $S \to aSb \mid \varepsilon$
- Ableitung $S \Longrightarrow aSb \Longrightarrow aaSbb \Longrightarrow aabb$

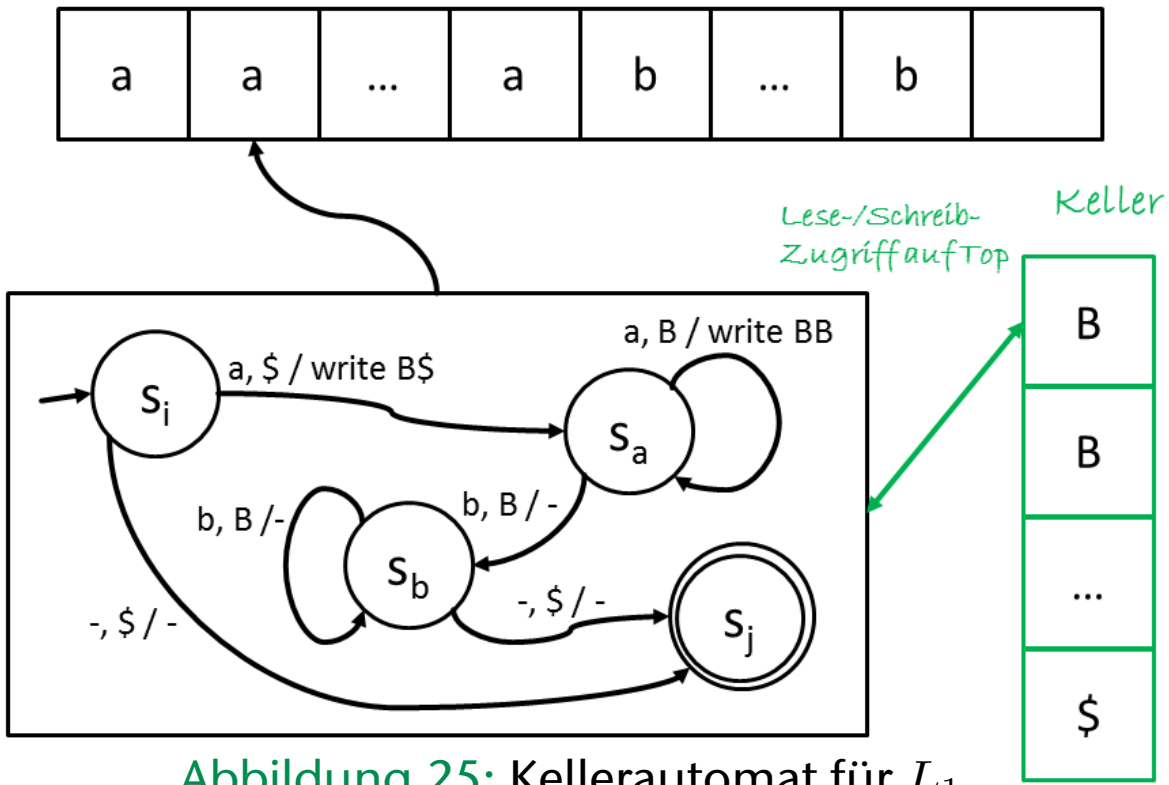
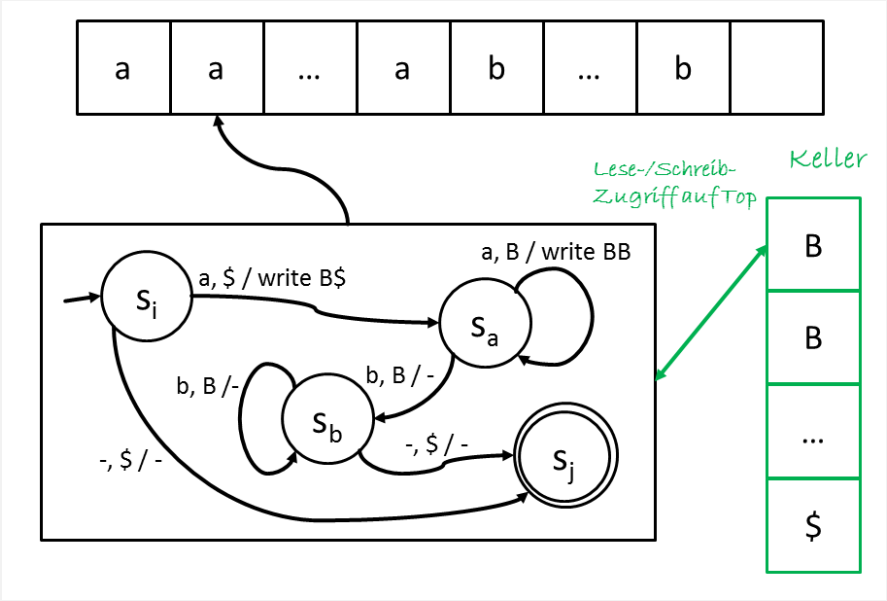


Abbildung 25: Kellerautomat für L_1

Beispiel 3.25 (Funktionsweise PDA)

$$L_1 = \{a^n b^n \mid n \in \mathbb{N}_0\}$$



Funktionsweise:

1. Der Kellerautomat liest (maximal) ein Zeichen des Eingabewortes, und die oberste Kellerzelle[a],
2. macht einen Zustandsübergang und beschreibt dabei eine oder mehrere Kellerzellen,
3. wiederholt (1) und (2), bis er am Ende des Wortes ist,
4. und akzeptiert, wenn er dann in einem Finalzustand ist.[b]

[a] dabei wird der Inhalt konsumiert (Eichhörnchen essen die Vorräte!)
[b] Alternativ kann man auch bei leerem Keller akzeptieren.

Beispiel 3.26 (Palindrome)

- $L_{cPal} = \{wcw^R \mid w \in \{a,b\}^*\}$ // c markiert die "Mitte"
- $L_{Pal} = \{ww^R \mid w \in \{a,b\}^*\}$
- Grammatik G_{cPal}: $S \to aSa \mid bSb \mid c$
- Grammatik G_{Pal}: $S \to aSa \mid bSb \mid \varepsilon$
- Ableitung für G_{Pal}: $S \Longrightarrow aSa \Longrightarrow abSba \Longrightarrow abba$

Wie sehen PDAs für die beiden Varianten der Palindrom-Sprache aus?

PDA$(cPal)$ kann man einfach deterministisch halten.

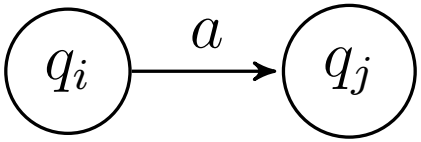
Was ist mit PDA(Pal)?

Erinnerung: Rechtslineare Grammatik und e.a.

Es wird eine einzelne Variable ersetzt. Mit einer Regelanwendung wird jeweils höchstens eine Variable erzeugt, die, wenn sie auftritt, ganz rechts im Wort steht.

Die Korrespondenz zwischen e.a.'en und rechtslinearen Gramatiken, wurde über die Ähnlichkeit von Zustandsübergängen und rechtslinearen Regeln bewiesen (siehe Theorem 2.14)

Zustandsübergang



Regel

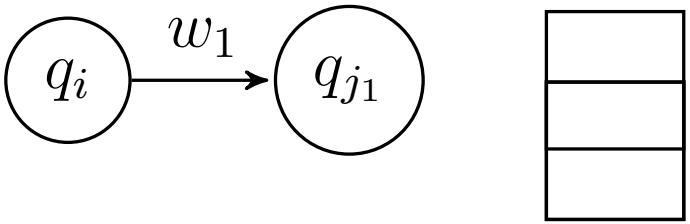
$$Q_i \to aQ_j$$

Erinnerung: Kontextfreie Grammatiken

Eine einzelne Variable wird - unabhängig vom Kontext - in der Conclusio durch ein Wort aus Variablen und Terminale in beliebiger Mischung ersetzt.

Ziel: Beweisidee recyclen! Dazu notwendig: mehrere Nichtterminale in einem Zustandsübergang verarbeiten[5].

Zustandsübergang



Regel

$$Q_i \to w_1 Q_{j_1} \dots Q_{j_n} w_{n+1}$$

[5] Die Eichhörnchen-Lösung wird sein: "Pack alles Zusätzliche in den Keller"

- Beispiel: die „prototypische cf-Sprache“
  $\{a^n b^n \mid n \in \mathbb{N}_0\}$
  Wie kann man diese Sprache akzeptieren?
  Endliche Automaten reichen nicht aus: Sie können sich nicht merken, wie oft sie einen Zustand durchlaufen haben. Für $a^n b^n$ muss man aber mitzählen.
- Idee: Aufruf von Prozeduren:
  - Rücksprungadresse, weitere Informationen auf dem Stack sichern
  - und wieder zurückholen, wenn die aufgerufene Prozedur beendet ist.

  Wenn die aufgerufene Prozedur weitere Prozeduren aufruft: kein Problem, $\hookrightarrow$ Stack.

Idee des **Stack, Stapels, Kellers**: **Last in, first out**; beliebig viel Information speichern; die zuletzt gespeicherte Information liegt immer „obenauf“.

- Eine Grammatikregel wie $S \to aAb$ kann man wie einen Aufruf einer Prozedur für das A betrachten.
  Was man braucht: einen Automaten, der sich merkt, daß nach Abarbeiten der A–„Prozedur“ noch ein b zu erwarten ist.
- **Push-Down-Automat**: wie endlicher Automat, aber hat **zusätzlich** einen Stack.
  Übergangsrelation:
  - bezieht oberstes Stacksymbol in den Übergang ein,
  - kann nicht nur den Zustand ändern, sondern auch **auf dem Stack lesen und schreiben**

Definition 3.27 (Push-Down-Automat)

Ein Push-Down-Automat (PDA) ist ein Tupel $\mathrm{M} = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$. Dabei ist

K	eine endliche Menge von Zuständen
Σ	das Eingabealphabet
Γ	das Stack- oder Kelleralphabet
$s_0 \in K$	der Startzustand
$Z_0 \in \Gamma$	das Anfangssymbol im Keller
$F \subseteq K$	eine Menge von finalen Zuständen
Δ	die Zustandsübergangsrelation, eine endliche Teilmenge von $(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma) \times (K \times \Gamma^*)$

Was macht die Übergangsrelation?

In einem Arbeitsschritt eines PDA soll

- in Abhängigkeit vom aktuellen Zustand aus K,
- in Abhängigkeit vom nächsten Eingabezeichen oder auch unabhängig vom Eingabezeichen, und
- in Abhängigkeit vom obersten Kellersymbol

Folgendes geschehen:

- das nächste **Eingabezeichen** wird **gelesen oder nicht** (bei ε),
- das oberste **Kellersymbol** wird **entfernt**,
- der **Zustand** wird **geändert**, und
- es werden null oder mehr **Zeichen auf den Keller** geschoben. Bei einem neuen Keller-Wort $\gamma = A_1 \dots A_n$ wird A_n zuerst auf den Keller geschoben usw., so daß am Schluss A_1 obenauf liegt.

Verwendete Symbole:

- Symbole a, b, c für Buchstaben aus Σ,
- u, v, w für Wörter aus Σ^*,
- A, B für Stacksymbole aus Γ,
- γ, η für Stackinhalte aus Γ^*.

Was macht die Übergangsrelation – formal?

- Begriff der **Konfiguration**: beschreibt die aktuelle Situation des PDA **komplett**
- Bestandteile:
  - **aktueller Zustand**,
  - **noch zu lesendes Restwort**,
  - **kompletter Stackinhalt**
- Relation $\vdash$ zwischen Konfigurationen:
  Für Konfigurationen C_1, C_2 bedeutet

$$C_1 \vdash C_2$$

daß der PDA in einem Schritt von C_1 nach C_2 gelangen kann.

Definition 3.28 (Konfiguration eines PDA, $\vdash$)

Eine Konfiguration C eines PDA $\mathrm{M} = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$ ist ein Tripel

$$C = (q, w, \gamma) \in K \times \Sigma^* \times \Gamma^*.$$

(w ist dabei der noch nicht bearbeitete Teil des Eingabewortes, γ ist der komplette Stackinhalt, und q ist der aktuelle Zustand.)

(s_0, w, Z_0) heißt Startkonfiguration bei Input w.

C_2 heißt Nachfolgekonfiguration von C_1, oder $C_1 \vdash C_2$, falls $\exists a \in \Sigma\ \exists A \in \Gamma\ \exists w \in \Sigma^*\ \exists \gamma, \eta \in \Gamma^*$, so daß

entweder $C_1 = (q_1, aw, A\gamma)$, $C_2 = (q_2, w, \eta\gamma)$, und $(q_1, a, A)\ \Delta\ (q_2, \eta)$,

oder $C_1 = (q_1, w, A\gamma)$, $C_2 = (q_2, w, \eta\gamma)$, und $(q_1, \varepsilon, A)\ \Delta\ (q_2, \eta)$,

Eine Rechnung eines PDA ist eine Folge von Konfigurationen, bei denen jeweils die $(i+1)$-te Nachfolgekonfiguration der i-ten ist.

Definition 3.29 (Rechnung)

Sei A ein Push-Down-Automat. Man schreibt

$$C \vdash_A^* C'$$

gdw es eine Reihe $C_0, C_1, \ldots, C_n$ von Konfigurationen gibt (mit $n \geq 0$), so daß

- $C = C_0$,
- $C' = C_n$,
- und für alle $i < n$ gilt, daß $C_i \vdash_A C_{i+1}$.

In diesem Fall heißt $C_0, C_1, \ldots, C_n$ eine Rechnung (Berechnung) von A der Länge n (von C_0 nach C_n).

Definition 3.30 (von PDA akzeptierte Sprache)

Ein PDA M kann auf 2 verschiedene Weisen eine Sprache akzeptieren, über finale Zustände oder über den leeren Keller:

$$L_f(\mathrm{M}) = \{w \in \Sigma^* \mid \exists q \in F\ \exists \gamma \in \Gamma^*\ \big((s_0, w, Z_0) \vdash_{\mathrm{M}}^* (q, \varepsilon, \gamma)\big)\}$$
$$L_l(\mathrm{M}) = \{w \in \Sigma^* \mid \exists q \in K\ \big((s_0, w, Z_0) \vdash_{\mathrm{M}}^* (q, \varepsilon, \varepsilon)\big)\}$$

Anmerkungen:

- Wenn aus dem Kontext klar ersichtlich ist, ob $L_f(\mathrm{M})$ oder $L_l(\mathrm{M})$ gemeint ist, schreibt man auch einfach $L(\mathrm{M})$.
- Das zu akzeptierende Wort w muss von M ganz gelesen werden:

  $$(s_0, w, Z_0) \vdash^* (q, \varepsilon, \cdot)$$

  ist gefordert.

Man beachte, daß das unterste Symbol im Keller zwar gelöscht werden kann, daß der PDA dann aber hängt: er kann nicht mehr weiter rechnen (es gibt keine Nachfolgekonfiguration).

Das entspricht etwa dem Stehenbleiben eines nd e.a. falls die Übergangsfunktion nicht definiert ist (was auch bei einem PDA sein kann).

Beispiel 3.31

$L = \{w \in \{a,b\}^* \mid w = w^R\}$: Palindrome über $\{a,b\}$.
L wird über leeren Keller akzeptiert von dem PDA
$\mathrm{M} := (\{s_0, s_1\}, \{a,b\}, \{Z_0, A, B\}, \Delta, s_0, Z_0, \emptyset)$ mit

$$\left.\begin{array}{ll} (s_0, \varepsilon, Z_0) & \Delta\,(s_1, \varepsilon) \end{array}\right\} \varepsilon \text{ akzeptieren}$$

$$\left.\begin{array}{ll} (s_0, a, Z_0) & \Delta\,(s_0, A) \\ (s_0, a, A) & \Delta\,(s_0, AA) \\ (s_0, a, B) & \Delta\,(s_0, AB) \\ (s_0, b, Z_0) & \Delta\,(s_0, B) \\ (s_0, b, A) & \Delta\,(s_0, BA) \\ (s_0, b, B) & \Delta\,(s_0, BB) \end{array}\right\} \text{Stack aufbauen}$$

$$\left.\begin{array}{ll}(s_0, \varepsilon, A) & \Delta\,(s_1, \varepsilon) \\ (s_0, \varepsilon, B) & \Delta\,(s_1, \varepsilon)\end{array}\right\}$$ Richtungswechsel für Palindrome mit ungerader Buchstabenanzahl

$$\left.\begin{array}{ll}(s_0, a, A) & \Delta\,(s_1, \varepsilon) \\ (s_0, b, B) & \Delta\,(s_1, \varepsilon)\end{array}\right\}$$ Richtungswechsel für Palindrome mit gerader Buchstabenanzahl

$$\left.\begin{array}{ll}(s_1, a, A) & \Delta\,(s_1, \varepsilon) \\ (s_1, b, B) & \Delta\,(s_1, \varepsilon)\end{array}\right\}$$ Stack abbauen

Idee:

- Ein Palindrom $w = w^R$ hat die Form vv^R oder vav^R oder vbv^R für irgendein $v \in \{a, b\}^*$.
- Der Automat M liest v und merkt sich jeden Buchstaben.
- Er rät indeterminiert die Wortmitte.
  Falls das Wort eine ungerade Anzahl von Buchstaben hat, also $w = vav^R$ oder $w = vbv^R$, dann muss dabei ein Buchstabe überlesen werden.
- Der Stack enthält nun v^R, also muss M jetzt nur noch jeden weiteren gelesenen Buchstaben mit dem jeweils obersten Kellersymbol vergleichen.

Für das Eingabewort $abbabba$ rechnet M so:

$$\begin{aligned}
&(s_0, abbabba, Z_0) \vdash (s_0, bbabba, A) \vdash (s_0, babba, BA) \vdash \\
&(s_0, abba, BBA) \vdash (s_0, bba, ABBA) \vdash (s_1, bba, BBA) \vdash \\
&(s_1, ba, BA) \vdash (s_1, a, A) \vdash (s_1, \varepsilon, \varepsilon)
\end{aligned}$$

Beispiel 3.32

Die Sprache $L = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$ wird über finalen Zustand akzeptiert von dem PDA $\mathrm{M} = (\{s_0, s_1\}, \{a,b\}, \{Z_0, \underline{A}, A, \underline{B}, B\}, \Delta, s_0, Z_0, \{s_0\})$ mit den folgenden Regeln:

$$(s_0, a, Z_0)\ \Delta\ (s_1, \underline{A}) \qquad (s_0, b, Z_0)\ \Delta\ (s_1, \underline{B})$$
$$(s_1, a, \underline{A})\ \Delta\ (s_1, A\underline{A}) \qquad (s_1, b, \underline{B})\ \Delta\ (s_1, B\underline{B})$$
$$(s_1, a, A)\ \Delta\ (s_1, AA) \qquad (s_1, b, B)\ \Delta\ (s_1, BB)$$
$$(s_1, a, \underline{B})\ \Delta\ (s_0, Z_0) \qquad (s_1, b, \underline{A})\ \Delta\ (s_0, Z_0)$$
$$(s_1, a, B)\ \Delta\ (s_1, \varepsilon) \qquad (s_1, b, A)\ \Delta\ (s_1, \varepsilon)$$

Idee:

- auf dem Stack mitzählen, wieviel A-Überhang oder B-Überhang momentan besteht
- Der Stack enthält zu jedem Zeitpunkt
  - entweder nur $A/\underline{A}$ (A-Überhang)
  - oder nur $B/\underline{B}$ (B-Überhang)
  - oder nur das Symbol Z_0 (Gleichstand).
- Das unterste A bzw. B auf dem Stack ist durch einen Unterstrich gekennzeichnet.
  So weiß M, wenn er dies Stacksymbol löscht, daß dann bis zu diesem Moment gleichviel as wie bs gelesen wurden.

Akzeptieren über finale Zustände und Akzeptieren über leeren Keller sind gleichmächtig.

Wir behandeln das nicht formal, sondern erwähnen es nur.

Theorem 3.33 (finale Zustände $\rightarrow$ leerer Keller)

Zu jedem PDA M_1 existiert ein PDA M_2 mit $L_f(\mathrm{M}_1) = L_l(\mathrm{M}_2)$.

Idee: Wir simulieren die Maschine M_1, die über finale Zustände akzeptiert, durch die Maschine M_2, die über leeren Keller akzeptiert.

M_2 arbeitet wie M_1, mit dem Unterschied: Wenn ein Zustand erreicht wird, der in M_1 final war, kann M_2 seinen Keller leeren.

Theorem 3.34 (leerer Keller $\rightarrow$ finale Zustände)

Zu jedem PDA M_1 existiert ein PDA M_2 mit $L_l(M_1) = L_f(M_2)$.

Idee: Wir simulieren die Maschine M_1, die über leeren Keller akzeptiert, durch die Maschine M_2, die über finale Zustände akzeptiert.

M_2 arbeitet wie M_1, legt aber ein zusätzliches Symbol ganz unten in den Keller. Wenn M_1 seinen Keller geleert hätte (also das neue unterste Symbol sichtbar wird), kann M_2 in einen finalen Zustand gehen.

Theorem 3.35 (PDA akzeptieren L_2)

Die Klasse der PDA-akzeptierten Sprachen ist $\mathbf{L}_2$.

Dazu beweisen wir die folgenden zwei Lemmata, die zusammen die Aussage des Satzes ergeben.

Lemma 3.36 (cf-Grammatik $\rightarrow$ PDA)

Zu jeder kontextfreien Grammatik G gibt es einen PDA M mit $L(\mathrm{M}) = L(G)$.

Beweis

Sei $G = (V, T, R, S)$ eine kontextfreie Grammatik in Greibach-Normalform: Alle Grammatikregeln haben die Form

$$A \rightarrow a\alpha \text{ mit } A \in V, a \in T, \alpha \in V^*$$

Wir konstruieren zu G einen PDA M, der $L(G)$ akzeptiert.

Idee: M

- vollzieht die Grammatikregeln nach, die angewendet worden sein könnten, um das aktuelle Eingabewort zu erzeugen und
- merkt sich das aktuelle Wort in der Ableitung bzw. dessen Rest
- merkt sich auf dem Keller alle Variablen, die im gedachten Ableitungswort noch vorkommen und noch ersetzt werden müssen.
- Die Variable ganz links liegt zuoberst: M arbeitet mit der Linksableitung.

Genauer:

- Die Erzeugung eines Wortes mit G beginnt beim Startsymbol S.
  Deshalb liegt bei M in der Startkonfiguration oben auf dem Keller ein S.
- Angenommen, G hat 2 Regeln mit S auf der linken Seite: $S \to aA_1A_2$ und $S \to bB_1B_2$
  Angenommen, der erste gelesene Buchstabe des Input-Wortes w ist ein a.
  Wenn w von G erzeugt wurde, hat G die erste der zwei S-Produktionen angewendet. Schieben wir also A_1A_2 auf den Stack.

- Der zweite Buchstabe des Eingabeworts muss durch Anwendung einer Regel $A_1 \to a_1\alpha$ erzeugt worden sein. Angenommen, der zweite Buchstabe des Eingabeworts ist a_1, dann müssen die nächsten Buchstaben des Wortes aus den Variablen in α entstehen.
  Wir entfernen A_1 vom Stack und legen α auf den Stack.
- Was, wenn es zwei Regeln $A_1 \to a_1\alpha_1$ und $A_1 \to a_1\alpha_2$ gibt? M **wählt indeterminiert eine der Regeln aus.**
- Der PDA hat nur einen einzigen Zustand und akzeptiert über den leeren Keller:
  Am Ende des Wortes dürfen auf dem Keller keine Variablen mehr übrig sein, aus denen noch mehr Buchstaben hätten erzeugt werden können.

Formal ist $\mathrm{M} = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$ mit
$K := \{s_0\}$, $\Sigma := T$, $\Gamma := V$, $Z_0 := S$, $F := \emptyset$,
$\Delta := \{((s_0, a, A), (s_0, \alpha)) \mid A \to a\alpha \in R\}$.

Bei den Regeln $(s_0, a, A)\ \Delta\ (s_0, \alpha)$ der Übergangsrelation gilt $\alpha \in \Gamma^*$, da G in GNF ist, und $a \in \Sigma \cup \{\varepsilon\}$ (ε, da die Regel $S \to \varepsilon$ in R sein kann).

Wir zeigen nun, daß mit dieser Definition von M insgesamt gilt:

Es gibt eine Linksableitung $S \underset{G}{\Longrightarrow}^* x\alpha$ mit $x \in T^*, \alpha \in V^*$

gdw

M rechnet $(s_0, x, S) \vdash_{\mathrm{M}}^* (s_0, \varepsilon, \alpha)$

Daraus folgt dann unmittelbar, daß $L(G) = L_\ell(\mathrm{M})$ gilt.

"$\Leftarrow$" Wir zeigen, daß es zu **jeder Rechnung von M eine entsprechende Ableitung in G** gibt.

Verfahren: Induktion über die Länge n einer Rechnung von M der Form $C_0 = (s_0, x, S) \vdash C_1 \vdash \ldots \vdash C_n = (s_0, \varepsilon, \alpha)$.

$n = 0$: Dann ist $x = \varepsilon, \alpha = S$ und $S \Longrightarrow^0 S$

$n \to n+1$: Es gelte $C_0 = (s_0, xa, S) \vdash C_1 \vdash \ldots \vdash C_n = (s_0, a, \beta) \vdash C_{n+1} = (s_0, \varepsilon, \alpha)$ mit $a \in \Sigma, \beta \in V^*$ und $x \in V^*$.
Dann kann der PDA a auch ignorieren und rechnen

$$(s_0, x, S) \vdash \ldots \vdash C'_n = (s_0, \varepsilon, \beta)$$

Daraus folgt nach Induktionsvoraussetzung, daß $S \Longrightarrow^* x\beta$.
Der Schritt von C_n nach C_{n+1} ist $(s_0, a, \beta) \vdash (s_0, \varepsilon, \alpha)$.

Da β der Stack vor dem Rechenschritt ist und α der Stack danach, gilt:

- $\beta = A\eta$ und
- $\alpha = \gamma\eta$
- für ein $A \in V$ und $\eta, \gamma \in V^*$.

Es muss somit für diesen Konfigurationsübergang des PDA eine Regel

$$(s_0, a, A)\ \Delta\ (s_0, \gamma)$$

verwendet worden sein.
Daraus folgt, daß $A \to a\gamma \in R$ wegen der Konstruktion des PDA, und damit

$$S \Longrightarrow^* x\beta = xA\eta \Longrightarrow xa\gamma\eta = xa\alpha$$

"$\Rightarrow$" Wir zeigen, daß es zu jeder Ableitung in G eine entsprechende Rechnung in M gibt.

Verfahren: Induktion über die Länge n einer Ableitung in G der Form

$$w_0 = S \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_n = x\alpha$$

$n = 0$: Dann ist $x = \varepsilon, \alpha = S$, und $(s_0, x, S) \vdash^0 (s_0, \varepsilon, \alpha)$ gilt ebenfalls in M.

$n \to n+1$: Es gelte $S \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_n = zA\gamma \Longrightarrow x\alpha$. Dann rechnet der PDA nach der Induktionsvoraussetzung

$$(s_0, z, S) \vdash^* (s_0, \varepsilon, A\gamma)$$

Für den Schritt von w_n nach w_{n+1} muss A ersetzt worden sein: Linksableitung!
Also $\exists a \in \Sigma\, \exists \eta \in V^*$ ($A \to a\eta \in R$ wurde angewandt). Damit ist dann $x = za$ und $\alpha = \eta\gamma$.
Wenn $A \to a\eta \in R$, gibt es im PDA auch einen Übergang $(s_0, a, A)\ \Delta\ (s_0, \eta)$.
Dann kann der PDA aber so rechnen:
$(s_0, x, S) = (s_0, za, S) \vdash^* (s_0, a, A\gamma) \vdash (s_0, \varepsilon, \eta\gamma) = (s_0, \varepsilon, \alpha)$. □

Beispiel 3.37

Die Sprache $L = \{ww^R \mid w \in \{a,b\}^+\}$ wird generiert von der GNF-Grammatik $G = (\{S,A,B\},\{a,b\},R,S)$ mit

$$R = \{\ S \to aSA \mid bSB \mid aA \mid bB$$
$$A \to a$$
$$B \to b\}$$

Daraus kann man mit dem gerade vorgestellten Verfahren einen PDA mit den folgenden Regeln konstruieren:

$(s_0,a,S)\ \Delta\ (s_0,SA)$
$(s_0,a,S)\ \Delta\ (s_0,A)$
$(s_0,b,S)\ \Delta\ (s_0,SB)$
$(s_0,b,S)\ \Delta\ (s_0,B)$
$(s_0,a,A)\ \Delta\ (s_0,\varepsilon)$
$(s_0,b,B)\ \Delta\ (s_0,\varepsilon)$

Lemma 3.38 (PDA $\rightarrow$ cf-Grammatik)

Zu jedem Push-Down-Automaten M gibt es eine kontextfreie Grammatik G mit $L(G) = L(\mathrm{M})$.

Beweis

Sei M ein PDA, der eine Sprache L **über leeren Keller** akzeptiert.
Wir konstruieren aus dem Regelsatz von M eine kontextfreie Grammatik, die L erzeugt. Wie akzeptiert M ein Wort?

- Jedes Symbol muss aus dem Keller genommen werden: Akzeptanz über leeren Keller!

Pro Δ-Schritt kann der Keller um höchstens 1 Symbol schrumpfen, nämlich

- entweder mit einem Schritt $(q, a, A)\ \Delta\ (q', \varepsilon)$
- oder mit einem Schritt $(q, \varepsilon, A)\ \Delta\ (q', \varepsilon)$

Für jedes Symbol, das irgendwann auf den Keller gelegt wird, muss der Automat also

- entweder einen Buchstaben lesen und das Symbol löschen
- oder einen ε-Übergang machen und das Symbol löschen

Wir konstruieren aus möglichen Rechenschritten von M Variablen unserer neuen Grammatik: Die Variablen sind 3-Tupel der Form

$$[q, A, p]$$

Solch eine Variable bedeutet: M kann vom Zustand q in den Zustand p übergehen und dabei A vom Keller entfernen.

Was ist mit Übergängen, die den Stack wachsen lassen?

$$(q, a, A)\ \Delta\ (q_1, B_1 \dots B_m)$$

- Automat liest ein a, also muss Grammatik ein a erzeugen.
- Es werden neue Symbole $B_1 \dots B_m$ auf den Keller geschoben.
  Also noch mind. m Schritte, um diese zu entfernen.
- Welche Zustände kann M dabei einnehmen?
  Grundsätzlich erst einmal beliebige.

- Wir machen aus einem Übergang $(q, a, A)\ \Delta\ (q_1, B_1 \ldots B_m)$ des Automaten die Grammatik-Regeln

$$[q, A, q_{m+1}] \to a[q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_m, B_m, q_{m+1}]$$

  für alle Kombinationen beliebiger $q_2 \ldots q_{m+1} \in K_{\mathrm{M}}$.
- Der Automat entfernt im Endeffekt das A vom Keller, d.h.: Wenn vorher der Stack $A\gamma$ war, kommt er in eine Konfiguration, in der der Stack $B_1, \ldots, B_m\gamma$ ist.
  Er hat das A durch $B_1, \ldots, B_m$ ersetzt. **Was muss er also jetzt tun, um in eine Konfiguration zu kommen, in der der Stack γ ist?**
  Er muss B_1 entfernen, dann B_2, usw., und schließlich B_m.

- Welche Zustände werden dabei durchlaufen?
  Bevor der Automat B_1 entfernt, ist er im Zustand q_1, da der Rechenschritt ja $(q, a, A)\ \Delta\ (q_1, B_1 \ldots B_m)$ ist.
  Nachdem der Automat B_1 entfernt hat, ist er in irgendeinem Zustand, nennen wir ihn q_2.
  Nachdem der Automat schließlich B_m entfernt hat, ist er in irgendeinem Zustand, nennen wir ihn q_{m+1}.
- Wenn wir die 3-Tupel jetzt als Variablen sehen, was wird dann aus ihnen?
  Oben haben wir gesagt:
  - Jedes Symbol muss aus dem Keller genommen werden.
  - Pro Δ-Schritt kann der Keller um höchstens 1 Symbol schrumpfen.
  - Für jedes Symbol, das irgendwann auf den Keller gelegt wird, muss der Automat also entweder einen Buchstaben lesen und das Symbol löschen, oder einen ε-Übergang machen und das Symbol löschen

In unserem (einen) Rechenschritt $(q, a, A)\ \Delta\ (q_1, B_1 \dots B_m)$ werden m neue Symbole $B_1 \dots B_m$ auf den Keller des PDA geschoben.
Also sind noch mindestens m Schritte des PDA nötig, um sie zu entfernen. In jedem dieser Schritte wird ein ε oder ein Buchstabe gelesen.
Wir haben m Variablen $[q_1, B_1, q_2], \dots, [q_m, B_m, q_{m+1}]$ erzeugt.
Aus jeder dieser Variablen wird entweder ε oder ein Buchstabe (plus eventuell weitere Variablen, falls der Automat mehr als m Schritte macht, um $B_1, \dots, B_m$ zu enfernen).

Sei $M = (K, \Sigma, \Gamma, \Delta, s_0, Z_0, F)$ ein PDA, so konstruiert man daraus die Grammatik $G = (V, T, R, S)$ mit
$V := \{[q, A, p] \mid q, p \in K, A \in \Gamma\} \cup \{S\}$
$T := \Sigma$
R enthält die Regeln

- $S \to [s_0, Z_0, q]$ für alle $q \in K$,
- $[q, A, q_{m+1}] \to a\ [q_1, B_1, q_2][q_2, B_2, q_3] \ldots [q_m, B_m, q_{m+1}]$ für jeden Δ-Übergang $(q, a, A)\ \Delta\ (q_1, B_1 \ldots B_m)$ und für alle Kombinationen beliebiger $q_2, \ldots, q_{m+1} \in K$, und
- $[q, A, q_1] \to a$ für $(q, a, A)\ \Delta\ (q_1, \varepsilon)$

Dabei ist wieder $a \in \Sigma \cup \{\varepsilon\}$.

Es ist nun noch zu beweisen, daß damit gilt:

$$\left([q, A, p] \Longrightarrow^* x\right) \underline{\text{gdw}} \left((q, x, A) \vdash^* (p, \varepsilon, \varepsilon)\right)$$

für $p, q \in K, A \in \Gamma, x \in \Sigma^*$, woraus sofort $L_\ell(\mathrm{M}) = L(G)$ folgt.

"$\Leftarrow$" M rechne
$(q, x, A) = C_0 \vdash C_1 \vdash \ldots \vdash C_n = (p, \varepsilon, \varepsilon)$.
Wir zeigen, daß es eine entsprechende Grammatik-Ableitung gibt, per Induktion über die Länge n der Rechnung von M.

$n = 1$: Es ist $x = \varepsilon$ oder $x = a \in \Sigma$.
Das heißt, es wurde eine Δ-Regel $(q, x, A)\ \Delta\ (p, \varepsilon)$ angewendet.
Nach der Konstruktion von G ist damit aber $[q, A, p] \to x \in R$.
Also gilt $[q, A, p] \Longrightarrow^* x$.

$n \to n + 1$: Sei $x = ay$ mit $y \in \Sigma^*, a \in \Sigma \cup \{\varepsilon\}$.
Dann rechnet der Automat $C_0 = (q, ay, A) \vdash (q_1, y, B_1 B_2 \dots B_m) = C_1 \vdash^* C_{n+1} = (p, \varepsilon, \varepsilon)$ für eine Δ-Regel $(q, a, A)\ \Delta\ (q_1, B_1 B_2 \dots B_m)$

Sei nun $y = y_1 \dots y_m, y_i \in \Sigma^*$.

- Es sind im Moment $B_1 \dots B_m$ auf dem Keller. Diese Variablen müssen wieder entfernt werden.
- Sei y_i das Teilstück von y, während dessen Abarbeitung B_i vom Stack entfernt wird.
- Während diese Stacksymbole entfernt werden, ist der PDA in irgendwelchen Zuständen.
  Er sei in q_i, bevor y_i abgearbeitet und B_i entfernt wird.

Dann gilt also:

$$\exists q_2, \dots, q_{m+1} \left((q_i, y_i, B_i) \vdash^* (q_{i+1}, \varepsilon, \varepsilon), 1 \leq i \leq m \right)$$

Nach der Induktionsvoraussetzung kann man in diesem Fall aber mit der cf-Grammatik Ableitungen $[q_i, B_i, q_{i+1}] \Longrightarrow^* y_i$ durchführen.

Damit und wegen des Schritts von C_0 nach C_1 gibt es eine Ableitung

$$[q, A, p] \Longrightarrow a[q_1, B_1, q_2] \dots [q_m, B_m, q_{m+1}] \Longrightarrow^* ay_1 \dots y_m = ay = x$$

"$\Rightarrow$" Es gebe in G eine Ableitung der Form
$w_0 = [q, A, p] \Longrightarrow w_1 \Longrightarrow \ldots \Longrightarrow w_n = x$.

Wir zeigen, daß es eine entsprechende Rechnung in M gibt, per Induktion über die Länge n der Ableitung.

$n = 1$: Wenn G $[q, A, p] \Longrightarrow x$ ableitet, dann gibt es nach der Konstruktion von G eine Δ-Regel

$$(q, x, A)\ \Delta\ (p, \varepsilon)$$

Also rechnet der PDA

$$(q, x, A) \vdash^* (p, \varepsilon, \varepsilon)$$

$n \to n+1$: Angenommen, G erlaubt die Ableitung

$$[q, A, q_{m+1}] \Longrightarrow a\,[q_1, B_1, q_2] \ldots [q_m, B_m, q_{m+1}] =$$
$$w_1 \Longrightarrow \ldots \Longrightarrow w_{n+1} = x$$

Dann läßt sich x schreiben als

$$x = a x_1 \ldots x_m$$

Dabei ist

$$[q_i, B_i, q_{i+1}] \Longrightarrow^* x_i$$

jeweils eine Ableitung in $\leq n$ Schritten für $1 \leq i \leq m$.

Der erste Schritt dieser Ableitung in G muss entstanden sein aus einem PDA-Übergang $(q, a, A)\ \Delta\ (q_1, B_1 \dots B_m)$.

Für die weiteren Schritte kann nach Induktionsvoraussetzung der PDA $(q_i, x_i, B_i) \vdash^* (q_{i+1}, \varepsilon, \varepsilon)$ rechnen. Insgesamt rechnet der PDA also

$$\begin{aligned}(q, x, A) = (q, ax_1 \dots x_m, A) &\vdash (q_1, x_1 \dots x_m, B_1 \dots B_m) \\ &\vdash^* (q_i, x_i \dots x_m, B_i \dots B_m) \vdash^* (q_{m+1}, \varepsilon, \varepsilon).\end{aligned}$$

□

Beispiel 3.39

Sprache: $L_{ab} = \{a^n b^n \mid n \in \mathbb{N}_0\}$
L_{ab} wird über leeren Keller akzeptiert von dem PDA
$\mathrm{M} = (\{s_0, s_1\}, \{a, b\}, \{Z_0, A\}, s_0, Z_0, \emptyset)$ mit den Regeln

1. $(s_0, \varepsilon, Z_0)\ \Delta\ (s_0, \varepsilon)$
2. $(s_0, a, Z_0)\ \Delta\ (s_0, A)$
3. $(s_0, a, A)\ \Delta\ (s_0, AA)$
4. $(s_0, b, A)\ \Delta\ (s_1, \varepsilon)$
5. $(s_1, b, A)\ \Delta\ (s_1, \varepsilon)$

Entsprechend Lemma 3.38 transformieren wir M in eine cf-Grammatik G.

Die Transformation ergibt folgende Grammatik-Regeln:

$$S \rightarrow [s_0, Z_0, s_0] \mid [s_0, Z_0, s_1]$$

1. $[s_0, Z_0, s_0] \rightarrow \varepsilon$
2. $[s_0, Z_0, s_0] \rightarrow a[s_0, A, s_0]$
   $[s_0, Z_0, s_1] \rightarrow a[s_0, A, s_1]$
3. $[s_0, A, s_0] \rightarrow a[s_0, A, s_0][s_0, A, s_0]$
   $[s_0, A, s_0] \rightarrow a[s_0, A, s_1][s_1, A, s_0]$
   $[s_0, A, s_1] \rightarrow a[s_0, A, s_0][s_0, A, s_1]$
   $[s_0, A, s_1] \rightarrow a[s_0, A, s_1][s_1, A, s_1]$
4. $[s_0, A, s_1] \rightarrow b$
5. $[s_1, A, s_1] \rightarrow b$

Lesbarer haben wir damit folgende Grammatik:

$S \to A \mid B$
$A \to aC \mid \varepsilon$
$B \to aD$
$C \to aCC \mid aDE$
$D \to aCD \mid aDF \mid b$
$F \to b$

Man sieht jetzt:

- Variable E ist nutzlos, und damit auch die Variable C.
- Die Grammatik enthält Kettenproduktionen und nullbare Variablen.

Wenn wir die Grammatik von diesen überflüssigen Elementen befreien, bleiben folgende Regeln übrig:

$S \to \varepsilon \mid aD$

$D \to aDF \mid b$

$F \to b$

Mit dieser Grammatik kann man z.B. folgende Ableitung ausführen:

$S \Longrightarrow aD \Longrightarrow aaDF \Longrightarrow aaaDFF \Longrightarrow aaabFF$

$\Longrightarrow aaabbF \Longrightarrow aaabbb$

3.6 Der CYK-Algorithmus

Gegeben: eine cf-Grammatik G, so daß $L(G)$ eine Sprache ist über Σ, und ein Wort $w \in \Sigma^*$

Frage: Ist $w \in L(G)$?

Eine **Lösung** eines solchen Problems ist ein *Algorithmus*, der diese Frage für jede Probleminstanz beantwortet, also für jede cf-Grammatik G und jedes Wort w.

Das Wortproblem und $\mathbf{L}_3$:

- Gegeben eine rechtslineare Grammatik G, so daß $L(G)$ eine Sprache ist über Σ, und ein Wort $w \in \Sigma^*$.
- Konstruiere aus G einen ε-nd e.a. A_1.
- Konstruiere aus A_1 einen nd e.a. A_2.
- Konstruiere aus A_2 einen e.a. A_3.
- Probiere aus, ob A_3 das Wort w akzeptiert. Dazu braucht der Automat A_3 genau $|w|$ Schritte.

Das Wortproblem und $\mathbf{L}_2$:

- Wir wissen, wie man zu jeder cf-Grammatik G einen Pushdown-Automaten konstruiert.
- Aber ein Pushdown-Automat kann ε-Übergänge machen, in denen er das Wort nicht weiter liest.
- **Wie kann man dann garantieren, daß der Automat in endlich vielen Schritten das Wort w zu Ende gelesen hat?**
- Ein Verfahren ist nur dann ein **Algorithmus**, wenn garantiert ist, daß es für jede beliebige Eingabe nach endlich vielen Schritten hält.
- Deshalb verwenden wir jetzt ein ganz anderes Verfahren, ohne Pushdown-Automaten: den **Cocke-Younger-Kasami (CYK) - Algorithmus**.
- Die Grundidee des Algorithmus ist auch unter dem Namen Chart-Parsing bekannt.

Chart-Parsing

- Gegeben eine cf-Grammatik G, die eine Sprache über Σ erzeugt, und ein Wort $w \in \Sigma^*$.
- Wir versuchen, von w aus Grammatikregeln **rückwärts** anzuwenden, um im Endeffekt beim Startsymbol S von G anzukommen.
- Die wichtigste Frage lautet also:
  Welche Regeln können angewendet worden sein bei der Erzeugung von w?

Beispiel 3.40

Die Grammatik $G = (\{S\}, \{a, b\}, R, S)$ mit

$$R = \{\ S \to aSa \mid bSb \mid aa \mid bb\ \}$$

erzeugt die Sprache $\{vv^R \mid v \in \{a, b\}^+\}$

Betrachten wir das Wort $w = abbaabba$. **Was sind mögliche letzte Schritte von Ableitungen, die zu w geführt haben können?**

- $aSaabba \Longrightarrow abbaabba$
- $abbSbba \Longrightarrow abbaabba$
- $abbaaSa \Longrightarrow abbaabba$

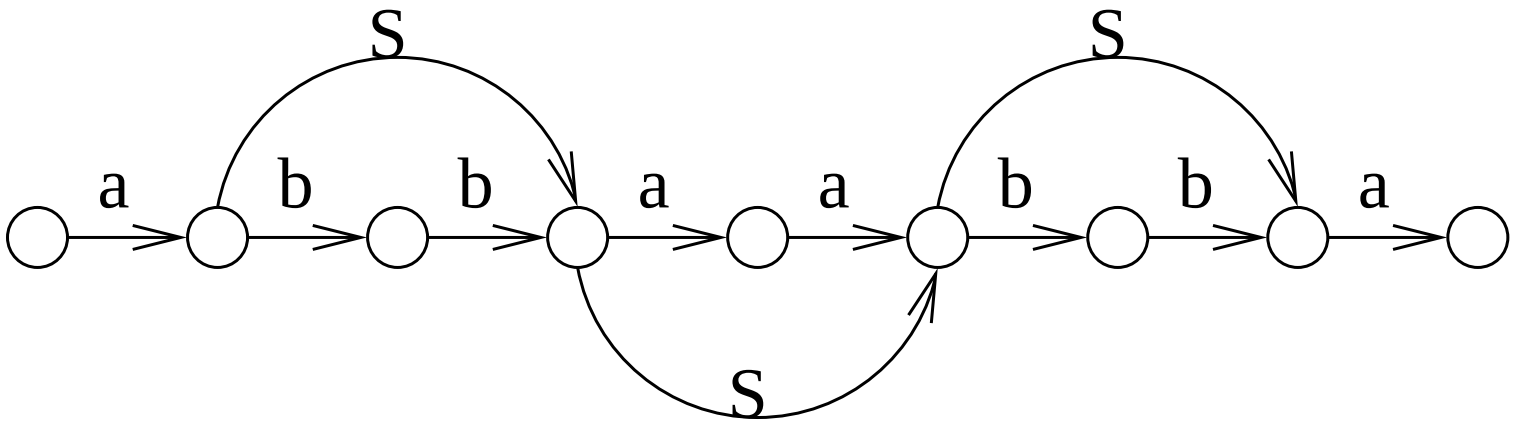
Nehmen wir noch weitere Schritte dazu. Möglich wäre z.B.

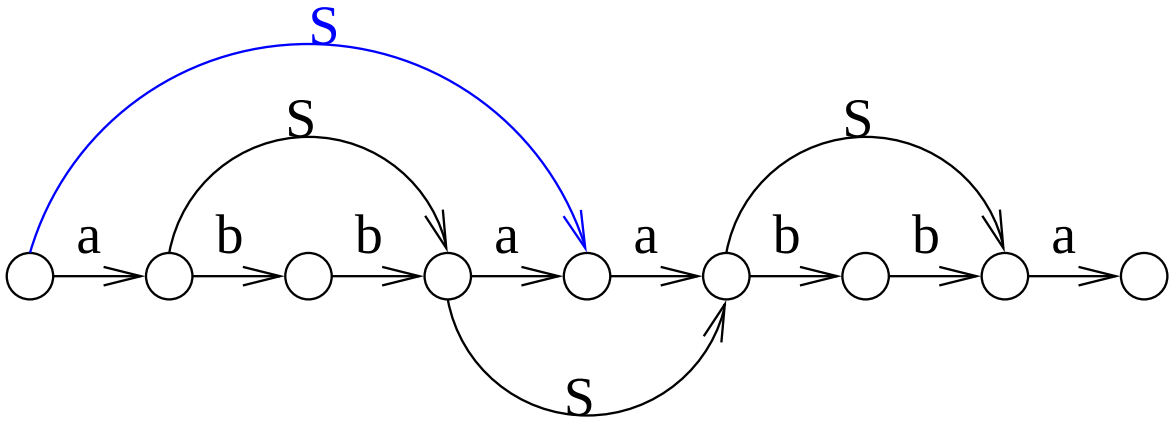
- $aSSSa \Longrightarrow aSaaSa \Longrightarrow aSabba \Longrightarrow aaSaabba \Longrightarrow aabbaabba$
- $S \Longrightarrow aSa \Longrightarrow abSba \Longrightarrow abbSbba \Longrightarrow abbaabba$

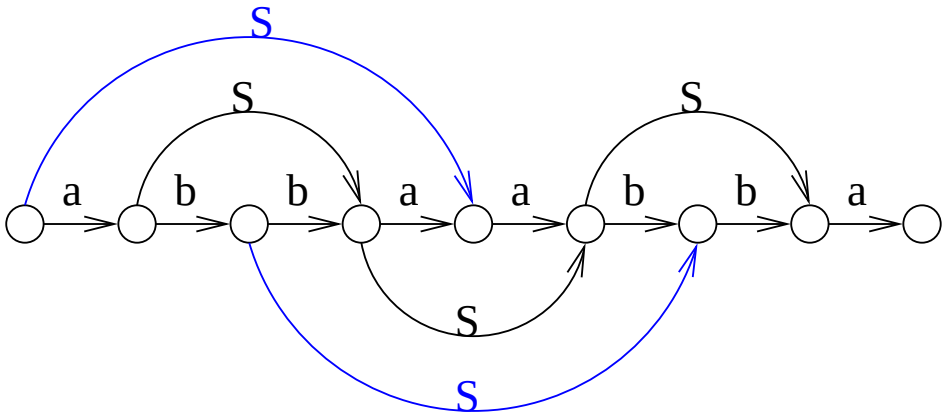
Die erste Rückwärts-Ableitung hat nicht zum Ziel geführt, die zweite schon.

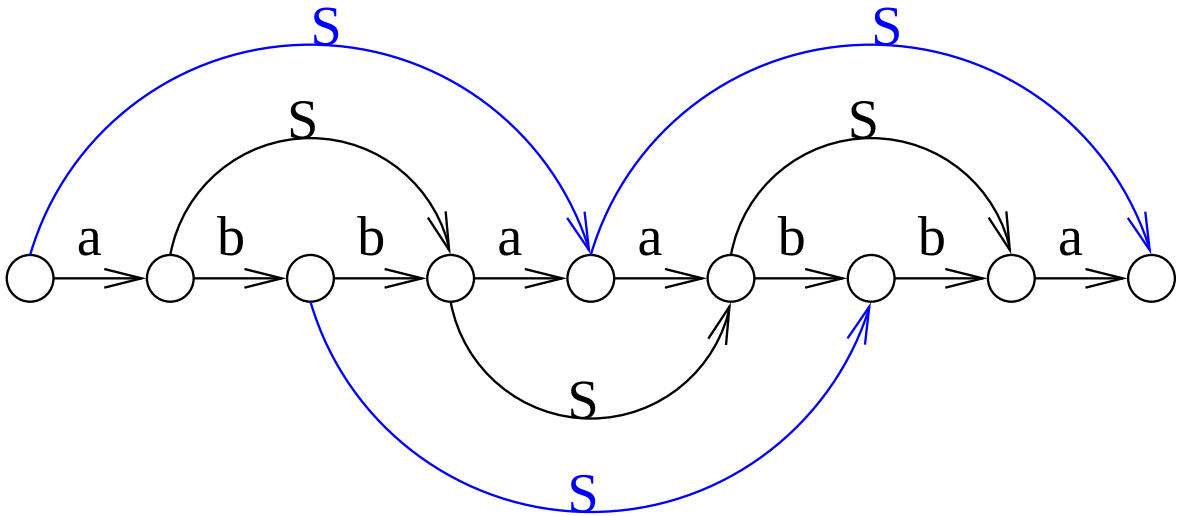
Bemerkung:

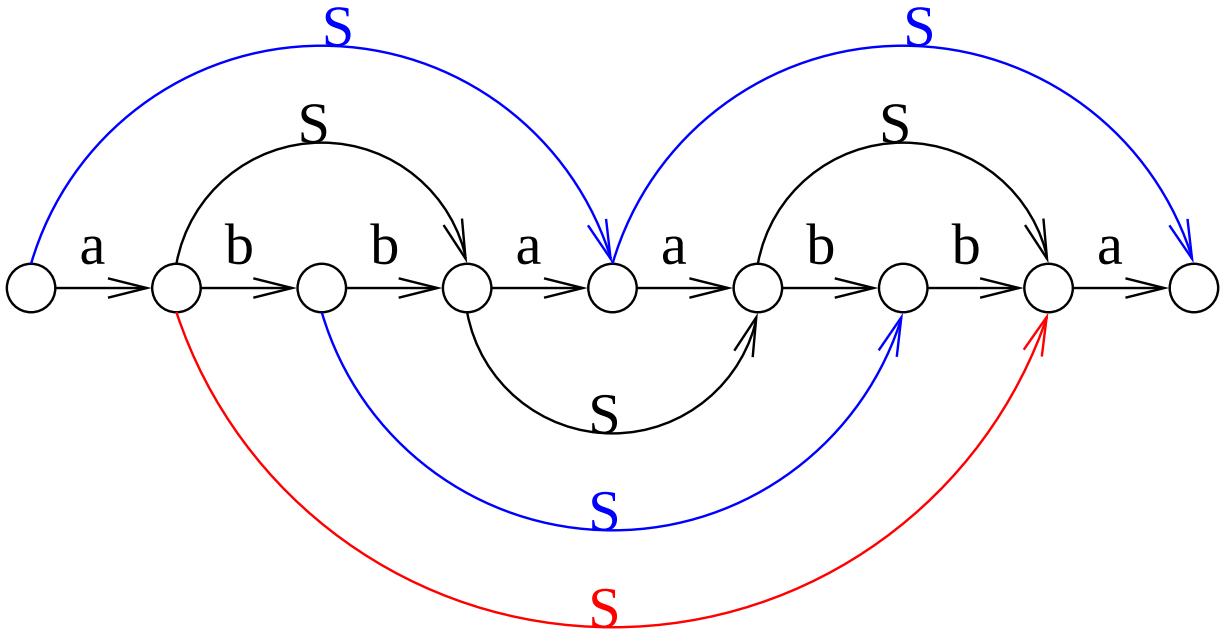
- Wir könnten jede mögliche Ableitung rückwärts von w bis zu S einzeln verfolgen.
- Dabei würden wir viel Arbeit mehrmals machen.
  Z.B. haben wir bei beiden Rückwärts-Ableitungen, die wir gerade berechnet haben, festgestellt, daß die mittleren **aa** in abb**aa**bba nur aus S entstanden sein können.
- Wir merken uns alle möglichen einzelnen Ableitungsschritte in einer **Chart**, um Mehrfacharbeit zu vermeiden.

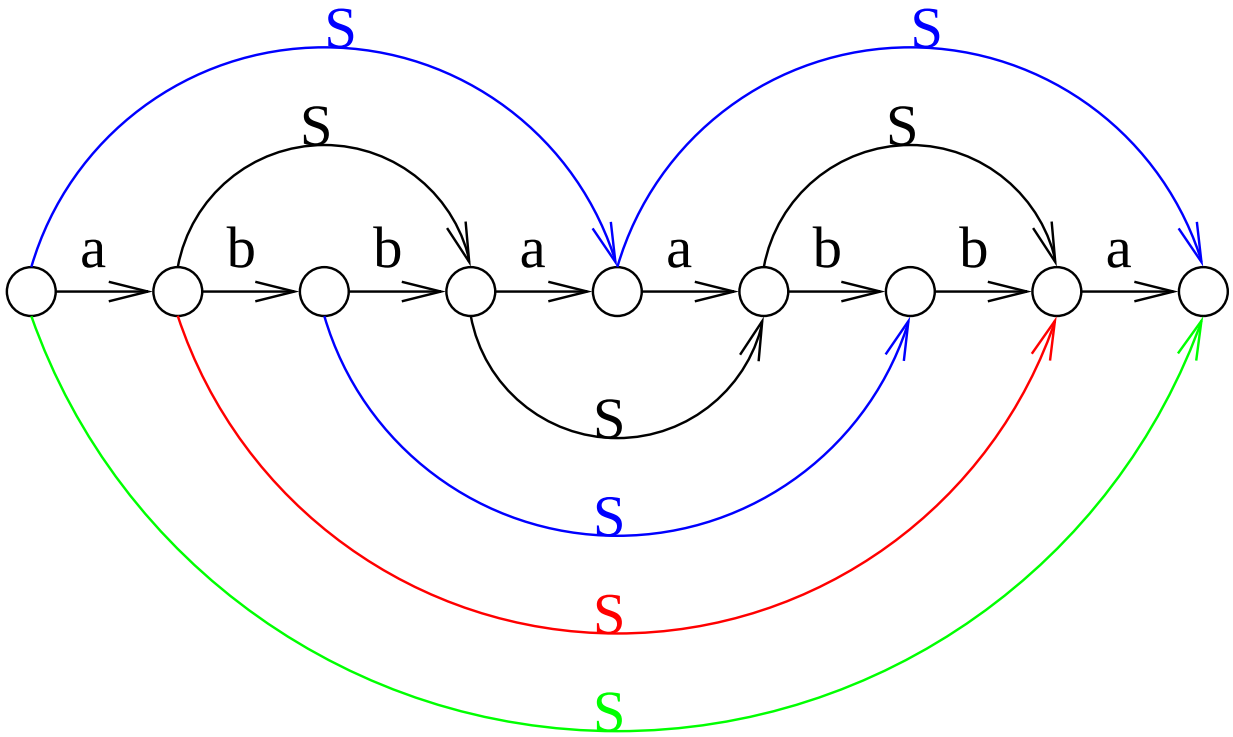

S
S
a
b
b
a
a
b
b
a
S


S
S
S
a
b
b
a
a
b
b
a
S


S
S
S
a
b
b
a
a
b
b
a
S
S


S
S
S
S
a
b
b
a
a
b
b
a
S
S


S
S
S
S
a
b
b
a
a
b
b
a
S
S
S



Beobachtung: Wenn das Wort w in der Sprache $L(G)$ ist, haben wir am Ende der Berechnung eine mit S markierte Kante in der Chart, die vom ersten bis zum letzten Knoten reicht.

In welcher Reihenfolge bearbeiten wir die Chart?

- Die rechte Seite einer cf. Regel kann beliebig lang und komplex sein. Es ist einfacher, wenn man nur Regeln von normierter Länge betrachtet. Wir fordern:

  Grammatik ist in **Chomsky- Normalform**.

Damit haben wir nur noch zwei Regelformen:

- $A \to AB$ und
- $A \to a$.

Also müssen wir immer nur **zwei** benachbarte Kanten betrachten, um herauszufinden, ob darüber eine neue Kante eingefügt werden kann.

Berechnungsreihenfolge:

- Zunächst berechnen wir die Kanten, die einen Buchstaben überspannen.
- Dann die, die zwei Buchstaben überspannen.
- Und so weiter, zum Schluss prüfen wir, ob wir eine Kante in die Chart eintragen können, die alle $|w|$ Buchstaben überspannt.

Beispiel

Eine Grammatik in CNF, die auch $L(G) = \{vv^R \mid v \in \{a,b\}^+\}$ erzeugt: $G = (\{S, S_a, S_b, A, B\}, \{a,b\}, R, S\}$ mit

$$
\begin{aligned}
R = \{ \quad & S \to AS_a \mid BS_b \mid AA \mid BB \\
& S_a \to SA \\
& S_b \to SB \\
& A \to a \\
& B \to b\}
\end{aligned}
$$

Darstellung der Chart:

- als ein Array der Größe $|w| \times |w|$.
- Für eine Kante, die i. bis j. Leerstelle überspannt und mit A markiert ist, steht im $[i,j]$-Element des Arrays die Eintragung A.

Chart-Einträge: Prinzip

und nun in der Chart :

	a_1	a_2	a_3	a_4	$\ldots$	$\ldots$	$\ldots$	a_n
a_1	A_1							
a_2		A_2						
a_3			A_3					
a_4		$B \in V_{i,j}$		A_4				
$\ldots$					$\ldots$			
$\ldots$		$\downarrow$				$\ldots$		
$\ldots$		$A \in V_{i,k}$	$\leftarrow$		$C \in V_{j+1,k}$		$\ldots$	
a_n								A_n

Jetzt formaler:

Definition 3.41 ($M * N$)

Sei $L = L(G)$ kontextfrei, und $G = (V, T, R, S)$ in Chomsky-Normalform. Mit $M, N \subseteq V$ sei

$$M * N := \{A \in V \mid \exists B \in M, \exists C \in N : \ A \to BC \in R\}$$

- Sei M die Menge aller Kanten, die den i-ten bis j-ten Buchstaben überspannen, und
- sei N die Menge aller Kanten, die den $j+1$-ten bis k-ten Buchstaben überspannen,

dann ist $M * N$ die Menge aller neuen Kanten, die man über den i-ten bis k-ten Buchstaben spannen kann.

Definition 3.42 ($w_{i,j}$, $V^w_{i,j}$)

Es sei $w = a_1 \dots a_n$ mit $a_i \in \Sigma$, so ist $w_{i,j} := a_i \dots a_j$ das Infix von w vom i-ten bis zum j-ten Buchstaben, und es sei

$$V^w_{i,j} := \{A \in V \mid A \Longrightarrow^*_G w_{i,j}\}$$

Frage: Wie finden wir alle neuen Kanten, die den i. bis k. Buchstaben überspannen?

Indem wir alle Paare von zwei alten Kanten betrachten, die zusammen *genau* den i-ten bis k-ten Buchstaben überspannen.

Lemma 3.43

Sei $w = a_1 \dots a_n$, $a_i \in \Sigma$, *d.h.* $|w| = n$. *Dann gilt:*

1. $V_{i,i} = \{A \in V \mid A \to a_i \in R\}$
2. $V_{i,k} = \bigcup_{j=i}^{k-1} V_{i,j} * V_{j+1,k}$ *für* $1 \leq i < k \leq n$

Beweis

1. $V_{i,i} = \{A \in V \mid A \Longrightarrow_G^* a_i\} = \{A \in V \mid A \to a_i \in R\}$, da G in CNF ist.

2. $A \in V_{i,k}$ mit $1 \leq i < k \leq n$

   gdw $A \Longrightarrow_G^* a_i \dots a_k$

   gdw $\exists j,\ i \leq j < k :\ \exists B, C \in V : A \Longrightarrow BC$, und
   $B \Longrightarrow_G^* w_{i,j} \neq \varepsilon$
   und $C \Longrightarrow_G^* w_{j+1,k} \neq \varepsilon$ (da G in CNF ist)

   gdw $\exists j,\ i \leq j < k :\ \exists B, C \in V : A \Longrightarrow BC$
   und $B \in V_{i,j}$ und $C \in V_{j+1,k}$

   gdw $\exists j,\ i \leq j < k : A \in V_{i,j} * V_{j+1,k}$

CYK steht für Cocke-Younger-Kasami.

Input sei eine Grammatik G in CNF und ein Wort $w = a_1 \dots a_n \in \Sigma^*$.

(i) **for** $i := 1$ *to* n **do** $\quad$ /* Regeln A → a eintragen */

$$\mathtt{V}_{i,i} := \{A \in V \mid A \to a_i \in R\}$$

(ii) **for** $h := 1$ *to* $n-1$ **do**

for $i := 1$ *to* $n-h$ **do**

$$\mathtt{V}_{i,i+h} = \bigcup_{j=i}^{i+h-1} V_{i,j} * V_{j+1,i+h}$$

(iii) **if** $S \in V_{1,n}$ **then return** Ausgabe $w \in L(G)$

else return Ausgabe $w \notin L(G)$

Eigenschaften:

- Die Korrektheit des Algorithmus folgt aus Lemma 3.43.
- Für Wörter der Länge $|w| = n$ entscheidet der CYK-Algorithmus in der Größenordnung von n^3 Schritten, ob $w \in L(G)$ ist.

Zusammenfassung des CYK-Algorithmus:

- Es geht um das **Wortproblem** für $\mathbf{L}_2$: Gegeben eine cf-Grammatik G und ein Wort w, ist $w \in L(G)$?
- Vorgehensweise: Wir rechnen von w aus rückwärts: Welche Teilworte von w könnten durch welche Regelanwendungen entstanden sein?
  Wir versuchen, rückwärts bis zum Startsymbol zu rechnen.
- Dazu legen wir alle Zwischenergebnisse in einer Chart ab – alle, auch die, die im Endeffekt nicht zu einer Lösung führen.

- Das Wort w liegt in $L(G)$, wenn der Algorithmus eine S-Kante findet, die die ganze Chart überspannt.
- Die Eingabegrammatik G soll in Chomsky-Normalform vorliegen.
- Berechnungsreihenfolge: Anfangend bei Kanten, die einen Buchstaben überspannen, bis zu der (möglichen) Kante, die $|w|$ Buchstaben überspannt.

Beispiel : Leere Chart

	a	b	b	a	a	b	b	a
a								
b								
b								
a								
a								
b								
b								
a								

Beispiel: Chart nach Eintrag der Regeln $A \to a$

	a	b	b	a	a	b	b	a
a	A							
b		B						
b			B					
a				A				
a					A			
b						B		
b							B	
a								A

Beispiel: Chart nach Eintrag der Regeln $A \to a$ und erstem Lauf durch die For-Schleife $h = 1$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b		S	B					
a			-	A				
a				S	A			
b					-	B		
b						S	B	
a							-	A

Beispiel: Chart nach Eintrag der Regeln $A \to a$ und 2.tem Lauf durch die For-Schleife $h = 2$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b	-	S	B					
a		S_a	-	A				
a			-	S	A			
b				S_b	-	B		
b					-	S	B	
a						S_a	-	A

Beispiel: Chart nach Eintrag der Regeln $A \to a$ und 3.tem Lauf durch die For-Schleife $h = 3$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b	-	S	B					
a	S	S_a	-	A				
a		-	-	S	A			
b			S	S_b	-	B		
b				-	-	S	B	
a					S	S_a	-	A

Beispiel: Chart nach Eintrag der Regeln $A \to a$ und 4.tem Lauf durch die For-Schleife $h = 4$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b	-	S	B					
a	S	S_a	-	A				
a	-	-	-	S	A			
b		-	S	S_b	-	B		
b			S_b	-	-	S	B	
a				-	S	S_a	-	A

Beispiel: Chart nach Eintrag der Regeln $A \to a$ und 5.tem Lauf durch die For-Schleife $h = 5$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b	-	S	B					
a	S	S_a	-	A				
a	-	-	-	S	A			
b	-	-	S	S_b	-	B		
b		S	S_b	-	-	S	B	
a			-	-	S	S_a	-	A

Beispiel: Chart nach Eintrag der Regeln $A \to a$ und 6.tem Lauf durch die For-Schleife $h = 6$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b	-	S	B					
a	S	S_a	-	A				
a	-	-	-	S	A			
b	-	-	S	S_b	-	B		
b	-	S	S_b	-	-	S	B	
a		S_a	-	-	S	S_a	-	A

Beispiel: Fertige Chart: $h = 7$

	a	b	b	a	a	b	b	a
a	A							
b	-	B						
b	-	S	B					
a	S	S_a	-	A				
a	-	-	-	S	A			
b	-	-	S	S_b	-	B		
b	-	S	S_b	-	-	S	B	
a	S	S_a	-	-	S	S_a	-	A

In dem Feld $[1, n]$ steht S, also ist $abbaabba \in L(G)$.

Zusammenfassung des CYK-Algorithmus:

- Der CYK-Algorithmus löst das **Wortproblem** für $\mathbf{L}_2$: Gegeben eine cf-Grammatik G und ein Wort w: ist $w \in L(G)$?
- Vorgehensweise: Wir rechnen von w aus rückwärts: Welche Teilworte von w könnten durch welche Regelanwendungen entstanden sein?
  Wir versuchen, rückwärts bis zum Startsymbol zu rechnen.
- Dazu legen wir alle Zwischenergebnisse in einer Chart ab – alle, auch die, die im Endeffekt nicht zu einer Lösung führen.

Zusammenfassung des CYK-Algorithmus:

- Das Wort w gehört zu $L(G)$, wenn der Algorithmus eine S-Kante findet, die die ganze Chart überspannt.
- Die Eingabegrammatik G soll in **Chomsky-Normalform** vorliegen.
- Berechnungsreihenfolge: Anfangend bei Kanten, die einen Buchstaben überspannen, bis zu der (möglichen) Kante, die $|w|$ Buchstaben überspannt.

3.7 Entscheidungsprobleme

Entscheidungsprobleme für $\mathbf{L_2}$

Sei $L, L_1, L_2 \in \mathbf{L_2}$ gegeben, d.h. wir haben PDAs A, A_1, A_2 oder kontextfreie Grammatiken G, G_1, G_2 für L, L_1 und L_2

Problem	Input	Frage	Entscheidbar?
Wortproblem	$w \in \Sigma^*$	$w \in L$?	Ja[6]
Leerheitsproblem		$L = \emptyset$?	Ja[7]
Endlichkeitsproblem		$\lvert L \rvert$ endlich?	Ja [8]
Äquivalenzproblem		$L_1 = L_2$?	Nein[9]

[6] CYK-Algorithmus

[7] Ist das Startsymbol S co-erreichbar?

[8] Grammatik in CNF bringen, (keine nutzlosen Symbole):

- Ist das Startsymbol S co-erreichbar?
- Gibt es eine Variable A, aus der ein Wort $\alpha A \beta \in (V \cup T)^*$ ableitbar ist?

[9] Nicht gezeigt

3.8 Abschlusseigenschaften

Theorem 3.44 (Abschlusseigenschaften von L_2)

$\mathbf{L}_2$ *ist abgeschlossen gegen* $\cup, \circ$ *und* *.

Beweis

Seien $G_i = (V_i, T_i, R_i, S_i)$ mit $i \in \{1, 2\}$ zwei cf-Grammatiken, $L_i = L(G_i)$ und $V_1 \cap V_2 = \emptyset$.

zu $\cup$: $G := (V_1 \cup V_2 \cup \{S_{neu}\}, T_1 \cup T_2, R_1 \cup R_2 \cup \{S_{neu} \to S_1 \mid S_2\}, S_{neu})$ erzeugt gerade $L_1 \cup L_2$.

zu $\circ$: $G := (V_1 \cup V_2 \cup \{S_{neu}\}, T_1 \cup T_2, R_1 \cup R_2 \cup \{S_{neu} \to S_1 S_2\}, S_{neu})$ erzeugt gerade $L_1 \circ L_2$.

zu *: $G := (V_1 \cup \{S_{neu}\}, T_1, R_1 \cup \{S_{neu} \to S_1 S_{neu} \mid \varepsilon\}, S_{neu})$ erzeugt gerade L_1^*. □

Theorem 3.45 ($\cap$, $\neg$)

$\mathbf{L}_2$ ist nicht abgeschlossen gegen $\cap$ und $\neg$.

Beweis

zu ∩: $L_1 = \{a^n b^n c^m \mid n, m \in \mathbb{N}_+\}$ wird erzeugt von $G_1 = (\{S, S', T\}, \{a, b, c\}, R_1, S)$ mit

$$R_1 = \{ \quad S \to S'T$$
$$S' \to aS'b \mid ab$$
$$T \to cT \mid c\}$$

$L_2 = \{a^m b^n c^n \mid n, m \in \mathbb{N}_+\}$ wird erzeugt von $G_2 = (\{S, S', T\}, \{a, b, c\}, R_2, S)$ mit

$$R_2 = \{ \quad S \to TS'$$
$$S' \to bS'c \mid bc$$
$$T \to aT \mid a\}$$

Sowohl L_1 als auch L_2 sind cf, aber $L_1 \cap L_2 = L_{abc} = \{a^n b^n c^n \mid n \in \mathbb{N}_+\} \notin \mathbf{L_2}$.

zu $\neg$: Nehmen wir an, $\mathbf{L_2}$ wäre abgeschlossen gegen $\neg$. Aber $L_1 \cap L_2 = \neg(\neg L_1 \cup \neg L_2)$, damit wäre $\mathbf{L_2}$ auch abgeschlossen gegen $\cap$ – ein Widerspruch. □

Abschlusseigenschaften für kontextfreie Sprachen

Sei $L, L_1, L_2 \in \mathbf{L_2}$ gegeben, d.h. wir haben PDAs A, A_1, A_2 oder kontextfreie Grammatiken G, G_1, G_2 für L, L_1 und L_2. Sei $M \in \mathbf{L_3}$.

Operation	Frage	Konstruktion	Antwort
Vereinigung	$L_1 \cup L_2 \in \mathbf{L_2}$?	Grammatik-Vereinigung	**Ja**
Schnitt	$L_1 \cap L_2 \in \mathbf{L_2}$?		**Nein**
Konkatenation	$L_1 \circ L_2 \in \mathbf{L_2}$?	Startregel	**Ja**
Stern	$L^* \in \mathbf{L_2}$?	Startregeln	**Ja**
Negation	$\overline{L} \in \mathbf{L_2}$?		**Nein**
Schnitt mit $\mathbf{L_3}$-Sprache	$M \cap L \in \mathbf{L_2}$?	Produkt-Konstruktion	**Ja**

4. Turing Maschinen (re)

Der Inhalt dieses Kapitels:

1. Was ist eine berechenbare Funktion?
2. Wie schreibt man ein Programm für eine DTM?
3. Tabellen- versus Flußdiagramm- Darstellung.
4. Modifikationen von DTMn: (mehrere) Halbbänder, zweiseitig unbeschränkte Bänder.
5. DTM versus NTM.
6. Gödelisieren: Programme als Wörter in Σ^*.
7. Aufzählbar versus entscheidbar.
8. Unentscheidbarkeit, Reduktionen.

Berechenbarkeit: Betrachtet werden Abbildungen über den natürlichen Zahlen $\mathbb{N}$.

Frage:

Welche davon sollen berechenbar genannt werden?

Komplexität: Um die Komplexität eines Algorithmus' zu messen braucht man ein Maschinenmodell zum Vergleich!

Frage:

Welches Modell wird gewählt?

Robustheit: Das Modell soll nicht von **einfachen Modifikationen** abhängig sein.

Anspruch:

Es muss robust gegenüber kleinen Veränderungen sein.

Die Maschine unter den Maschinen:

(One ring to rule them all.)

Dieser ring ist die Turing-Maschine, benannt nach **Alan Mathison Turing**.

Church's These

Church's These besagt:

> Der intuitive Begriff von Berechenbarkeit findet seine formale Ensprechung im mathematischen Begriff der Turingmaschine.
>
> *Etwas ist algorithmisch lösbar, soll bedeuten, es kann mit einer Turingmaschine gelöst werden.*

- Die Church'sche These kann **nicht formal bewiesen werden**.
- Sie könnte aber **widerlegt** werden, wenn es jemandem gelänge, ein Problem zu konstruieren, das intuitiv berechenbar ist, aber nicht durch eine DTM gelöst werden kann.
- Dies ist jedoch **sehr** unwahrscheinlich.

Vom ea. über den PDA zur Turingmaschine

Endliche Automaten akzeptieren reguläre Sprachen.

- Ein Speicher: der Zustand. Es wird jeweils ein Zustand aus einer endlichen Zustandsmenge eingenommen.
- Das Eingabewort wird nur einmal gelesen, und zwar von links nach rechts.
- Die Zustandsmenge kann sehr groß sein, ist aber endlich.

  Endliche Automaten können nicht einmal alle kontextfreien Sprachen akzeptieren.

Vom ea. über den PDA zur Turingmaschine

Keller-Automaten akzeptieren kontextfreie Sprachen.

- Erster Speicher: der Zustand. Es wird jeweils ein Zustand aus einer endlichen Zustandsmenge eingenommen.
- Zweiter Speicher: ein Keller (*stack*), der beliebig viele Zeichen aus einem endlichen Keller-Alphabet aufnehmen kann. Zugriffsart: *LIFO*
- Das Eingabewort wird nur einmal gelesen, und zwar von links nach rechts.
- Der Keller kann beliebig groß werden, nur die Zugriffsart ist beschränkt.
  Diese Beschränkung der Automaten sieht nicht sehr groß aus – ist es aber doch: Keller-Automaten können nicht einmal alle kontextsensitiven Sprachen akzeptieren.

Vom ea. über den PDA zur Turingmaschine

Was ist mit PDA's mit 2 Kellern?

Als dritten Automatentyp behandeln wir jetzt Turing-Maschinen.

Ausblick:

Die von Turing-Maschinen akzeptierten Sprachen sind genau die in $\mathbf{L}_0$: Sprachen vom Typ 0.

Eigenschaften von Turing-Maschinen:

- Erster Speicher: der Zustand. Es wird jeweils ein Zustand aus einer endlichen Zustandsmenge eingenommen.
- Zweiter Speicher: ein beliebig großer Speicher, der Zugriff an beliebigen Stellen erlaubt.

Vom ea. über den PDA zur Turingmaschine

- Zusätzlich sind Turing-Maschinen sehr *einfach* aufgebaut. Vorteile:
  - Man kann leichter etwas über Turing-Maschinen beweisen als über komplexer aufgebaute Maschinen – weniger Fallunterscheidungen.
  - Aussagen über diese einfach aufgebauten Automaten lassen sich gut verallgemeinern auf komplexer aufgebaute Rechensysteme.
- Form dieses zweiten Speichers: ein unendlich langes Band. Die Turing-Maschine hat einen Schreib-/Lesekopf, die sie über diesem Band in einem Rechenschritt um ein Feld nach rechts oder links bewegen kann.
- Das Eingabewort steht auf dem unendlich langen Band geschrieben. Die Maschine kann es damit **beliebig oft lesen.**

Vom ea. über den PDA zur Turingmaschine

Ausblick:

Wir werden sehen, daß Turing-Maschinen deutlich mächtiger sind als Keller-Automaten. Sie können sogar Sprachen akzeptieren, die über die Klasse der Sprachen vom Typ 1 hinausgehen.

4.1 DTM'n

Definition 4.1 (Turing-Maschine (DTM))

Eine deterministische Turing-Maschine (DTM) M ist ein Tupel $\mathrm{M} = (\ K, \Sigma, \delta, s\ )$. Dabei ist

- K eine endliche Menge von Zuständen mit $h \notin K$ (h ist der **Haltezustand**),
- Σ ein Alphabet mit $L, R \notin \Sigma, \# \in \Sigma$,
- $\delta : K \times \Sigma \to (K \cup \{h\}) \times (\Sigma \cup \{L, R\})$ eine Übergangsfunktion, und
- $s \in K$ ein Startzustand.

Anzahl der Zustände: $|K| - 1$ (s nicht gezählt).

In einem δ-Übergangsschritt $\delta(q, a) = \langle q', x \rangle$ kann eine Turing-Maschine

- in Abhängigkeit vom aktuellen **Zustand** $q \in K$ und
- in Abhängigkeit von dem **gelesenen Zeichen** $a \in \Sigma$ (das momentan unter dem Schreib-/Lesekopf steht),

folgendes tun:

- Sie kann entweder einen Schritt nach links tun, falls $x = L$ ist,
- oder einen Schritt nach rechts tun, falls $x = R$ ist,
- oder das Zeichen a, das momentan unter dem Schreib-/Lesekopf steht, durch $b \in \Sigma$ überschreiben, falls $x = b \in \Sigma$ gilt.

Zusätzlich geht sie in einen Zustand $q' \in K \cup \{h\}$ über.
Da $h \notin K$ ist, kann es nie einen δ-Übergang von h ausgeben.

Wichtig:

Das spezielle Zeichen # (*blank*) ist das **Leerzeichen**. Es ist **nie Teil des Eingabeworts**; man kann es u.a. dazu benutzen, (Eingabe) Wörter voneinander abzugrenzen.

Frage:

Wie sieht eine Turing-Maschine aus?

- Band einer DTM ist einseitig unbeschränkt:
  - Nach rechts ist es unendlich lang.
  - Nach links hat es ein Ende.
  - Wenn eine DTM versucht, das Ende zu überschreiten, bleibt sie hängen. In diesem Fall hält sie nicht.

- Zu Beginn der Rechnung sieht das Band so aus:
  - Ganz links steht ein Blank.
  - Direkt rechts davon steht das Eingabewort.
  - Wenn eine DTM mehrere Eingabewörter hintereinander bekommt, sind sie durch Blanks getrennt.
  - Rechts vom letzten Eingabewort stehen nur noch Blanks.
- Zu Beginn der Rechnung steht der Schreib-/Lesekopf der DTM auf dem Blank direkt rechts neben dem (letzten) Eingabewort.
- **Das Band enthält immer nur endlich viele Symbole, die keine Blanks sind.**

Beispiel 4.2 (Replace(a): a's durch b's ersetzen)

Die folgende Turing-Maschine $\mathtt{Replace}(a)$ erwartet *ein* Eingabewort. Sie liest es von rechts nach links einmal durch und macht dabei jedes a zu einem b. Es ist $\mathtt{Replace}(a) = (\ \{s, q_1\}, \{a, b, \#\}, \delta, s\ )$ mit folgender δ-Funktion:

$$\begin{array}{llll} \langle s, \# \rangle \mapsto \langle q_1, L \rangle & & \langle q_1, \# \rangle \mapsto \langle h, \# \rangle \\ \langle s, a \rangle \mapsto \langle s, a \rangle & & \langle q_1, a \rangle \mapsto \langle q_1, b \rangle \\ \langle s, b \rangle \mapsto \langle s, b \rangle & & \langle q_1, b \rangle \mapsto \langle q_1, L \rangle \end{array}$$

Wir notieren eine Rechnung dieser DTM zunächst einmal informell.

Beispiel 4.3 ($L_{\#}$)

Die folgende Turing-Maschine $L_{\#}$ läuft zum ersten blank links von der momentanen Position.
Es ist $L_{\#} = (\{s, q_1\}, \{a, b, \#\}, \delta, s)$ mit folgender δ-Funktion:

$$\begin{array}{ll} \langle s, \# \rangle \mapsto \langle q_1, L \rangle & \langle q_1, \# \rangle \mapsto \langle h, \# \rangle \\ \langle s, a \rangle \mapsto \langle q_1, L \rangle & \langle q_1, a \rangle \mapsto \langle q_1, L \rangle \\ \langle s, b \rangle \mapsto \langle q_1, L \rangle & \langle q_1, b \rangle \mapsto \langle q_1, L \rangle \end{array}$$

Wir wollen nun ein DTM Copy realisieren.
Eingabe ist ein String bestehend aus Einsen.
Dieser String soll einfach kopiert werden.
D.h. falls n Einsen auf dem Band stehen, sollen nach Ausführung von Copy $2n$ Einsen auf dem Band stehen (getrennt durch ein Blank $\#$).

Beispiel 4.4 (Copy)

Die Maschine Copy benutzt 8 Zustände:

state	$\#$	1	c
s	$\langle q_1, c\rangle$	$-$	$-$
q_1	$\langle q_2, R\rangle$	$\langle q_1, L\rangle$	$\langle q_1, L\rangle$
q_2	$-$	$\langle q_3, \#\rangle$	$\langle q_7, \#\rangle$
q_3	$\langle q_4, R\rangle$	$-$	$-$
q_4	$\langle q_5, 1\rangle$	$\langle q_4, R\rangle$	$\langle q_4, R\rangle$
q_5	$\langle q_6, 1\rangle$	$\langle q_5, L\rangle$	$\langle q_5, L\rangle$
q_6	$-$	$\langle q_2, R\rangle$	$-$
q_7	$\langle q_8, R\rangle$	$-$	$-$
q_8	$\langle h, \#\rangle$	$\langle q_8, R\rangle$	$-$

Für Copy haben wir die Übergangsfunktion δ nicht überall definiert. Man beachte allerdings, daß es niemals eine Situation für Copy gibt, in dem etwa in q_6 ein # gelesen werden kann. Daher kann man irgendwelche Werte einsetzen, um δ zu einer Funktion zu machen.

Fakt 4.5 (Übergangsfunktion δ nicht überall definiert)

Man läßt in der Literatur auch zu, daß δ nicht überall definiert ist. Falls die DTM dann in einen solchen nichtdefinierten Zustand kommt, sagt man auch die DTM hängt. Sie hält also nicht. Für uns ist allerdings die Übergangsfunktion überall definiert.

Für jedes $n \in \mathbb{N}$ konstruieren wir eine Maschine, die genau n Einsen auf das leere Band schreibt: **mit möglichst wenig Zuständen**.

Beispiel 4.6 (Print_n)

Für gegebenes n, schreiben wir zunächst $\lfloor \frac{n}{2} \rfloor$ viele Einsen auf das Band (höchstens $\lfloor \frac{n}{2} \rfloor$ Zustände). Dieser String wird kopiert (8 Zustände). Dann ersetzen wir das trennende # mit einer 1. Falls n gerade ist, ersetzen wir noch die letzte 1 durch # (2 neue Zustände).

Insgesamt können wir n Einsen mit höchstens $\lfloor \frac{n}{2} \rfloor + 10$ Zuständen konstruieren.

Konfigurationen:

Den Begriff haben wir auch schon für Keller-Automaten eingeführt.

- Eine **Konfiguration** beschreibt die *komplette* aktuelle Situation der Maschine in einer Rechnung.
- Eine **Rechnung** ist dann einfach eine Folge von Konfigurationen, wobei immer von einer Konfiguration zu einer Nachfolgekonfiguration übergegangen wird.

Die Konfiguration einer DTM besteht aus 4 Elementen:

- dem aktuellen Zustand q,
- dem Wort w links vom Schreib-/Lesekopf,
- dem Zeichen a, auf dem der Kopf gerade steht, und
- dem Wort u rechts von der aktuellen Kopfposition.

Wir wissen folgendes über die 4 Elemente:

- w enthält das komplette Anfangsstück des Bandes bis zur aktuellen Kopfposition. **Links ist das Band endlich!**
  $w = \epsilon$ bedeutet, daß der Kopf die linkeste Position auf dem Band einnimmt.
- u enthält den Bandinhalt rechts vom Schreib-/Lesekopf bis zum letzten Zeichen, das kein Blank ist. **Nach rechts ist das Band unendlich, aber es enthält nach rechts von einer bestimmten Bandposition an nur noch Blanks**.
  $u = \epsilon$ bedeutet, daß rechts vom Schreib-/Lesekopf nur noch Blanks stehen. u darf also überall Blanks besitzen, nur nicht als letzten Buchstaben.
  Konfiguration ist endlich: Die endliche Beschreibung des unendlichen Bandes endet da, wo nur noch Blanks kommen.

Definition 4.7 (Konfiguration einer DTM)

Eine Konfiguration C einer DTM $\mathrm{M} = (\ K, \Sigma, \delta, s\ )$ ist ein Wort der Form $C = q, w\underline{a}u$. Dabei ist

- $q \in K \cup \{h\}$ der aktuelle Zustand,
- $w \in \Sigma^*$ der Bandinhalt links des Kopfes,
- $a \in \Sigma$ das Bandzeichen unter dem Schreib-/Lesekopf.
  Die Position des Schreib-/Lesekopfes ist durch einen Unterstrich gekennzeichnet.
- $u \in \Sigma^*(\Sigma - \{\#\}) \cup \{\epsilon\}$ der Bandinhalt rechts des Kopfes.

Definition 4.8 (Nachfolgekonfiguration)

Eine Konfiguration C_2 heißt Nachfolgekonfiguration von C_1, in Zeichen

$$C_1 \vdash_M C_2$$

falls gilt:

- $C_i = q_i, w_i \underline{a_i} u_i$ für $i \in \{1, 2\}$, und
- es gibt einen Übergang $\delta(q_1, a_1) = \langle q_2, b \rangle$ wie folgt:

  Fall 1: $b \in \Sigma$. Dann ist $w_1 = w_2,\ u_1 = u_2,\ a_2 = b$.

  Fall 2: $b = L$. Dann gilt für w_2 und a_2: $w_1 = w_2 a_2$.
  Für u_2 gilt: Wenn $a_1 = \#$ und $u_1 = \epsilon$ ist, so ist $u_2 = \epsilon$, sonst ist $u_2 = a_1 u_1$.

  Fall 3: $b = R$. Dann ist $w_2 = w_1 a_1$.
  Für a_2 und u_2 gilt: Wenn $u_1 = \epsilon$ ist, dann ist $u_2 = \epsilon$ und $a_2 = \#$, ansonsten ist $u_1 = a_2 u_2$.

Definition 4.9 (Eingabe)

w heißt Eingabe (*input*) für M, falls M mit der Startkonfiguration

$$C_0 = s, \#w\underline{\#}$$

startet.

$\langle w_1, \ldots, w_n \rangle$ heißt Eingabe für M, falls M mit der Startkonfiguration

$$C_0 = s, \#w_1\# \ldots \#w_n\underline{\#}$$

startet.

Eine Turing-Maschine **hält**, wenn sie den Haltezustand h erreicht.

Sie **bleibt hängen**, wenn sie versucht, das Bandende nach links zu überschreiten. Wenn sie einmal hängengeblieben ist, rechnet sie nicht mehr weiter.

Definition 4.10 (Halten, Hängen)

Sei M eine Turing-Maschine.

- M hält in $C = q, w\underline{a}u$ **gdw.** $q = h$.
- M hängt in $C = q, w\underline{a}u$ **gdw.** $w = \epsilon \wedge \exists q' \; \delta(q, a) = \langle q', L \rangle$.

Definition 4.11 (Rechnung)

Sei M eine Turing-Maschine. Man schreibt

$$C \vdash_M^* C'$$

gdw. es eine Reihe $C_0, C_1, \ldots, C_n$ von Konfigurationen gibt (mit $n \geq 0$), so daß $C = C_0$ und $C' = C_n$ ist und für alle $i < n$ gilt, daß $C_i \vdash_M C_{i+1}$.
Dann heißt $C_0, C_1, \ldots, C_n$ eine **Rechnung** (Berechnung) von M der Länge n von C_0 nach C_n.
Eine Rechnung **hängt** (bzw. **hält**), falls ihre letzte Konfiguration hängt (bzw. hält).

Beispiel 4.12 (Replace(a))

Wir betrachten noch einmal die DTM von eben und notieren ihre Rechnung jetzt in Form einer Sequenz von Konfigurationen.
Es ist Replace(a)= $(\ \{s, q_1\}, \{a, b, \#\}, \delta, s\ )$ mit δ-Funktion:

$$\begin{array}{llll} \langle s, \# \rangle \mapsto \langle q_1, L \rangle & & \langle q_1, \# \rangle \mapsto \langle h, \# \rangle \\ \langle s, a \rangle \mapsto \langle q_0, a \rangle & & \langle q_1, a \rangle \mapsto \langle q_1, b \rangle \\ \langle s, b \rangle \mapsto \langle q_0, b \rangle & & \langle q_1, b \rangle \mapsto \langle q_1, L \rangle \end{array}$$

Analog kann man Maschinen Replace(a, b) für ein beliebiges Alphabet Σ mit $a, b \in \Sigma$ definieren: diese ersetzen jedes a durch ein b und lassen alle anderen Symbole unverändert.

Zum Beispiel rechnet sie auf $w = abbab$ so:

$$s, \#abbab\underline{\#} \vdash q_1, \#abba\underline{b} \vdash q_1, \#abb\underline{a}b \vdash q_1, \#ab\underline{b}bb \vdash$$
$$q_1, \#a\underline{b}bbb \vdash q_1, \#\underline{a}bbbb \vdash q_1, \#\underline{b}bbbb \vdash$$
$$q_1, \underline{\#}bbbbb \vdash h, \underline{\#}bbbbb$$

Die δ-Übergänge $\delta(s, a)$ und $\delta(s, b)$ werden nie gebraucht. **Da die δ-Funktion aber vollständig definiert sein muß, haben wir sie hier trotzdem angegeben.**

TMn können Funktionen berechnen!

Betrachte eine Funktion $f : \{a, b\}^* \to b^*$, die jedes Wort $w \in \{a, b\}^*$ abbildet auf ein gleichlanges Wort $w' \in b^*$.

Definition 4.13 (TM-berechenbar)

Sei Σ_0 ein Alphabet mit $\# \notin \Sigma_0$. Eine (partielle) Funktion $f : ({\Sigma_0}^*)^m \to ({\Sigma_0}^*)^n$ heißt DTM-berechenbar, falls eine deterministische Turing-Maschine $\mathrm{M} = (\ K, \Sigma, \delta, s\ )$ existiert

- mit $\Sigma_0 \subseteq \Sigma$,
- so daß $\forall w_1, \ldots, w_m \forall u_1, \ldots, u_n \in {\Sigma_0}^*$ gilt:
  - $f(w_1, \ldots, w_m) = \langle u_1, \ldots, u_n \rangle$ gdw
    M rechnet $s, \#w_1\# \ldots \#w_m\underline{\#} \vdash_{\mathrm{M}}^* h, \#u_1\# \ldots \#u_n\underline{\#}$, und
  - $f(w_1, \ldots, w_m)$ ist undefiniert gdw
    M gestartet mit $s, \#w_1\# \ldots \#w_m\underline{\#}$ hält nicht (läuft unendlich oder hängt).

Vorsicht:

Wir betrachten Turing-Maschinen hier unter einem anderen Aspekt als alle bisherigen Automaten:

- Bei endlichen Automaten und Pushdown-Automaten haben wir untersucht, **welche Sprachen sie akzeptieren**.
- Bei Turing-Maschinen untersuchen wir,
  - welche Sprachen sie akzeptieren und
  - **welche Funktionen sie berechnen**.

Definition 4.14 (Von einer DTM akzeptierte Sprache)

Ein Wort w wird akzeptiert von einer DTM M, falls M auf Eingabe von w (wobei der Kopf auf dem ersten Blank rechts von w steht) hält.
Eine Sprache $L \subseteq \Sigma^*$ wird akzeptiert von einer DTM M, wenn genau die Wörter aus L aus M und keine anderen akzeptiert werden.

Achtung:

Bei nicht akzeptierten Wörtern muss die DTM nicht halten (**sie darf es sogar nicht**)!

Frage:

Wie ist es mit Funktionen auf natürlichen Zahlen? Wie stellen wir natürliche Zahlen für Turing-Maschinen am besten dar?

Wir verwenden die **Unärdarstellung**: Eine Zahl n wird auf dem Band der Maschine durch n senkrechte Striche dargestellt.

Eine Turing-Maschine M berechnet eine Funktion $f : \mathbb{N}^k \to \mathbb{N}^n$ in Unärdarstellung also wie folgt:

- Wenn $f(i_1, \ldots, i_k) = \langle j_1, \ldots, j_n \rangle$ ist, dann rechnet M

$$s, \#|^{i_1}\# \ldots \#|^{i_k}\underline{\#} \vdash_{\mathrm{M}}^* h, \#|^{j_1}\# \ldots \#|^{j_n}\underline{\#}.$$

- Ist $f(i_1, \ldots, i_k)$ undefiniert, dann hält M bei Input $\#|^{i_1}\# \ldots \#|^{i_k}\#$ nicht.

Definition 4.15 (TM ^{part}, TM)

TM^{part} ist die Menge der partiellen TM-berechenbaren Funktionen $f : \mathbb{N}^k \to \mathbb{N}$. **TM** ist die Menge der totalen TM-berechenbaren Funktionen $\mathbb{N}^k \to \mathbb{N}$.

Frage:

Welche Funktionen betrachten wir?

- Man könnte Funktionen mit Werten über einem **beliebigen Alphabet Σ** betrachten.
- Man könnte **Funktionen mit n Parametern und m Ausgabewerten** betrachten.
- In der Definition von DTM und TM^{part} haben wir uns aber in beiden Fällen eingeschränkt: Wir betrachten
  - nur Funktionen über natürliche Zahlen,
  - nur Funktionen mit einstelligem Wertebereich.

Frage:

Sind das wirklich Einschränkungen?

- Man kann ein Eingabe-Alphabet in einem anderen **kodieren**. ASCII-Zahlen kodieren Buchstaben.
- So kann man auch einen ganzen Text in einer Zahl kodieren (Kapitel 1).

  Das heißt: Man kann eine Funktion f_1 mit Definitionsbereich Σ_0^k kodieren in eine Funktion f_2 mit Definitionsbereich $\mathbb{N}^k$.

Kodieren heißt: Um $f_1(a_1, \ldots, a_k)$ zu berechnen,

- kodiert man erst a_1 in n_1, …, a_k in n_k,
- berechnet dann $f_2(n_1, \ldots, n_k) = m$,
- und dekodiert dann m in ein Zeichen b.
- Damit hat man $f_1(a_1, \ldots, a_k) = b$ berechnet.

Eine Folge von Zahlen aus $\mathbb{N}^n$ ist ein String über dem Alphabet $\{0, \ldots, 9, \#\}$. Dann kann man genau so, wie wir es eben für beliebige Alphabete Σ gesagt haben, diesen Text auch mit einer einzigen Zahl ausdrücken.

Das haben wir schon in Kapitel 1 gemacht, um zu zeigen, daß Σ^* abzählbar ist.

Damit haben wir aus einer Funktion mit Wertebereich $\mathbb{N}^n$ eine mit Wertebereich $\mathbb{N}$ gemacht.

4.2 Flußdiagramme

Graphische Darstellung der Übergangsfunktion

In Flußdiagrammen werden

- die **Zustandsnamen** nicht genannt;
- nur die Schritte und die Ausführungsreihenfolge beschrieben.

Folgende Elemente stehen zur Verfügung:

- $\mathbf{L}$ — eine DTM, die nach dem Starten ein Feld nach links geht und danach hält.
- $\mathbf{R}$ — eine DTM, die ein Feld nach rechts geht und danach hält.
- $\mathbf{a}$ **(für** $a \in \Sigma$**)** — eine DTM, die den Buchstaben a auf das aktuelle Bandfeld druckt und danach hält.

- **$M_1 \longrightarrow M_2$ oder abgekürzt M_1M_2 (falls M_1, M_2 die Flußdiagramme zweier DTM sind) —**
  eine DTM, die zuerst wie M_1 arbeitet und dann, falls M_1 hält, wie M_2 weiterarbeitet.

  Direkt aufeinanderfolgende Schritte werden
  - entweder direkt nebeneinander notiert
  - oder durch einen Pfeil verbunden.

  Im Gegensatz zu der Maschine M_1 gilt also für M_1M_2: **Nachdem M_1 seine Arbeit beendet hat, ist M_1M_2 nicht im Haltezustand, sondern im Startzustand von M_2.**

- $>$ — der Startschritt der DTM.
- $\mathbf{M_1} \xrightarrow{a} \mathbf{M_2}$ — nach M_1 wird M_2 nur dann ausgeführt, wenn nach der Beendigung von M_1 der aktuelle Bandbuchstabe (also der, auf dem sich der Lesekopf befindet) a ist.

Sind allgemein $M_0, M_1, \ldots, M_n$ Turing-Maschinen, $a_i \in \Sigma$ für $1 \leq i \leq n$, so ist

$$
\begin{array}{cc}
 & M_1 \\
 & \uparrow a_1 \\
> & M_0 \xrightarrow{a_2} M_2 \\
 & \downarrow a_n \quad \ddots \\
 & M_n
\end{array}
$$

die Turing-Maschine, die zuerst wie M_0 arbeitet und dann, falls M_0 mit dem Buchstaben a_i auf dem Arbeitsfeld hält, wie M_i weiterarbeitet.

- σ — eine Schreibabkürzung für einen beliebigen Buchstaben aus Σ.
  Die Maschine $M \xrightarrow{\sigma} \ldots\sigma$ zum Beispiel ist eine Abkürzung für

$$
\begin{array}{cc}
 & \ldots a_1 \\
 & \uparrow a_1 \\
> & M \xrightarrow{a_2} \ldots a_2 \\
 & \downarrow a_n \qquad \ddots \\
 & \ldots a_n
\end{array}
$$

falls $\Sigma = \{a_1, \ldots, a_n\}$ ist.

Die Maschine $> \mathrm{L} \xrightarrow{\sigma} \mathrm{R}\sigma\mathrm{R}$ für $\Sigma = \{\#, |\}$ macht

- zuerst einen Schritt nach links;
- steht hier ein $\#$ (bzw. ein $|$), so **merkt sie es sich** und geht einen Schritt nach rechts,
- dann druckt sie **das gemerkte Zeichen** (also $\#$, bzw. $|$) und geht ein weiteres Feld nach rechts.

Weitere Schreibabkürzungen sind:

- $\xrightarrow{\sigma \neq a}$ für $\sigma \in \Sigma \setminus \{a\}$
- $\mathbf{M_1} \xrightarrow{a,b} \mathbf{M_2}$, falls nach der Ausführung von M_1 sowohl für den Bandbuchstaben a als auch für b nach M_2 verzweigt werden soll.

Beispiel 4.16

Die DTM $\mathrm{M}^+ = (\{s, q_1, q_2, q_3, q_4\}, \{|, \#\}, \delta, s)$ addiert zwei natürliche Zahlen in Unärdarstellung. Sie rechnet

$$s, \#|^n\#|^m\underline{\#} \vdash^*_{\mathrm{M}^+} h, \#|^{n+m}\underline{\#}$$

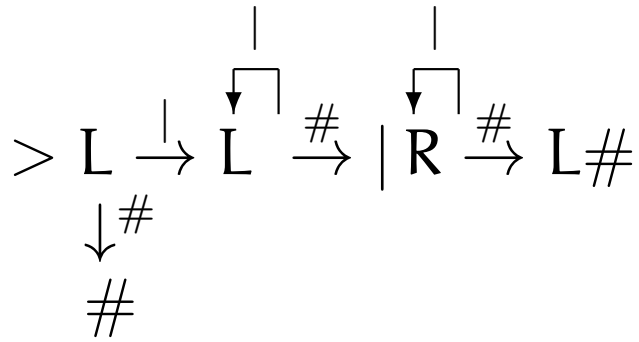
Der Trick: **Sie löscht den letzten Strich von $|^m$ und schreibt ihn in den Zwischenraum zwischen $|^n$ und $|^m$.**

Hier ist zunächst die δ-Funktion:

$$\begin{array}{llllllllll}
\langle s,\#\rangle & \mapsto & \langle q_1,L\rangle & \langle q_2,\#\rangle & \mapsto & \langle q_3,|\rangle & \langle q_3,\#\rangle & \mapsto & \langle q_4,L\rangle \\
\langle q_1,\#\rangle & \mapsto & \langle h,\#\rangle & \langle q_2,|\rangle & \mapsto & \langle q_2,L\rangle & \langle q_4,|\rangle & \mapsto & \langle h,\#\rangle \\
\langle q_1,|\rangle & \mapsto & \langle q_2,L\rangle & \langle q_3,|\rangle & \mapsto & \langle q_3,R\rangle &&&
\end{array}$$

Für $\delta(s,|)$ und $\delta(q_4,\#)$ haben wir keine Werte angegeben; sie sind beliebig, weil M^+ sie nie benötigt.

Das Flußdiagramm zur gleichen DTM ist erheblich leichter zu lesen:

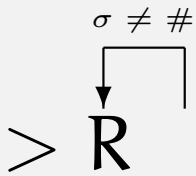


Die folgenden Turing-Maschinen werden wir als Bestandteile von komplexeren Turing-Maschinen später noch häufig benutzen.

Beispiel 4.17 ($R_\#$)

$R_\#$ bewegt nur den Kopf, ohne zu schreiben:

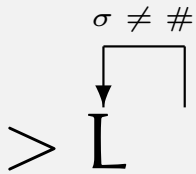
- Sie geht mindestens einmal nach rechts.
- Dann geht sie solange weiter nach rechts, bis sie ein # liest.



Beispiel 4.18 ($L_{\#}$)

Analog funktioniert die DTM $L_{\#}$:

- Sie geht mindestens einmal nach links.
- Dann geht sie solange weiter nach links, bis sie ein # liest.



Beispiel 4.19 (Copy)

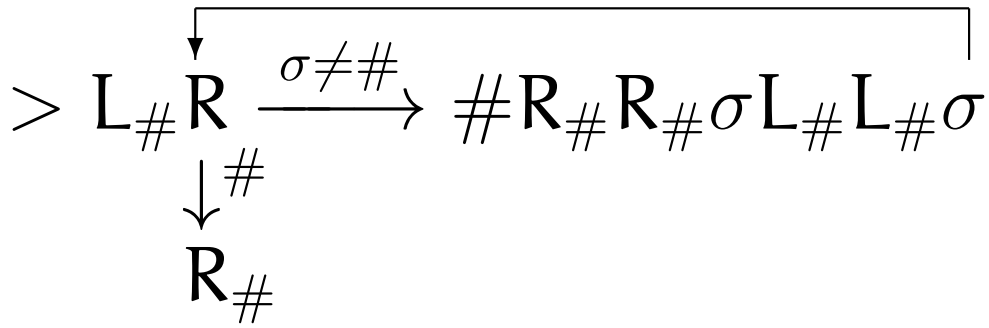
Die DTM Copy kopiert das Eingabewort w einmal rechts neben sich:

$$s, \#w\underline{\#} \vdash^*_{\text{Copy}} h, \#w\#w\underline{\#}$$

für alle $w \in (\Sigma \setminus \{\#\})^*$.

Sie rechnet so:

- Sie bewegt sich nach links auf das Blank vor das Eingabewort,
- geht das Eingabewort von links nach rechts durch,
- merkt sich jeweils ein Zeichen σ von w, markiert die aktuelle Position, indem sie σ mit $\#$ überschreibt,
- und kopiert das Zeichen σ.
- Sie verwendet dabei die Maschinen $L_\#$ und $R_\#$, die wir schon definiert haben.

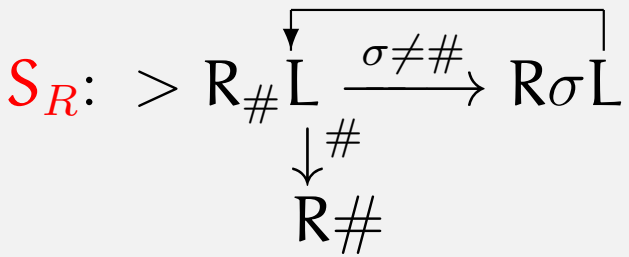


Beispiel 4.20 (S_R)

Die DTM S_R bewirkt eine „Verschiebung nach rechts“, das heißt, wenn S_R das Alphabet Σ besitzt, rechnet sie

$$s, w_1\underline{\#}w_2\#w_3 \vdash_{S_R}^* h, w_1\#\underline{\#}w_2w_3$$

für alle Wörter $w_1, w_3 \in \Sigma^*$ und $w_2 \in (\Sigma \setminus \{\#\})^*$. (Entgegen der Konvention startet sie ausnahmsweise zwischen zwei Eingabewörtern.) Sie arbeitet so:

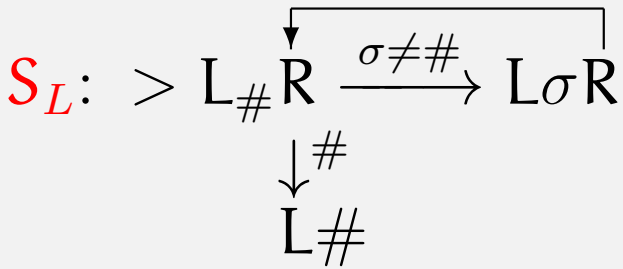


Beispiel 4.21 (S_L)

Dazu invers arbeitet die Maschine S_L, die einen „shift nach links“ bewirkt. (Entgegen der Konvention startet sie ausnahmsweise zwischen zwei Eingabewörtern.) Sie rechnet

$$s, w_1\#w_2\underline{\#}w_3 \vdash_{S_L}^* h, w_1w_2\underline{\#}\#w_3$$

für alle $w_1, w_3 \in \Sigma^*, w_2 \in (\Sigma \setminus \{\#\})^*$. Sie ist definiert als



4.3 Varianten von TMn

Die TM, die wir bisher kennen, . . .

- ist determiniert und
- hat ein einseitig unbeschränktes Band (**Halbband**).
- Ab jetzt heisst sie: **Standard-Turing-Maschine (Standard-DTM) oder kurz (DTM)**.

Variationen:

- **zweiseitig** unbeschränktes Band: zw-DTM;
- **mehrere** Bänder: k-DTM;
- **indeterminierte** Turing-Maschinen: NTM;
- **Kombinationen**: k-NTM, zw-NTM, k-zw-NTM.

Turing-Maschinen, die nie hängenbleiben

- Gegeben: eine Turing-Maschine M, deren Eingabe die normierte Form $\#w\underline{\#}$ hat
- Daraus konstruieren wir M', eine DTM, die
  - dasselbe berechnet wie M und
  - **nie hängt**.

- M' rechnet so:
  - $s, \#w\underline{\#} \vdash^*_{M'} s', \alpha\#w\#$.
  - Das Bandende ist am Anfang ein Zeichen links vom Eingabewort!
  - Die DTM verschiebt die Eingabe ein Zeichen nach rechts.
  - Dann druckt sie ganz links ein **Sonderzeichen** α, das das **Bandende** anzeigt.
  - Ab dann rechnet sie wie M, aber:
    Wenn sie α erreicht, bleibt sie dort stehen und druckt immer wieder α.

Dies ist ein ähnlicher Trick wie wir ihn schon bei der Kellermaschine verwandt haben, um zu verhindern, daß der Keller ganz geleert wird.

Damit gilt:

M' hält für eine Eingabe w
gdw
M hält für w.

M' hängt nie. Wenn M hängt, rechnet M' unendlich lang.

O.E. sollen also alle Turing-Maschinen, die wir von jetzt an betrachten, nie hängen.

zw-DTM: Was ändert sich?

Zweiseitig unbeschränktes Band:

- Die Definition der Maschine bleibt gleich.
- Die Definition der **Konfiguration** ändert sich.
- Sie hat immer noch die Form $q, w\underline{a}u$, aber:
  w umfasst analog zu u alle Zeichen bis zum letzten nicht-Blank links vom Schreib-/Lesekopf.
  $w = \epsilon$ bzw. $u = \epsilon$ bedeutet, daß links bzw. rechts vom Schreib-/Lesekopf nur noch Blanks stehen.

Definition 4.22 (zw-DTM)

Eine Turing-Maschine mit zweiseitig unbeschränktem Band (zw-DTM) ist eine DTM, für die die Begriffe der Konfiguration und der Nachfolgekonfiguration wie folgt definiert sind:
Eine Konfiguration C einer zw-DTM $\mathrm{M} = (K, \Sigma, \delta, s)$ ist ein Wort der Form $C = q, w\underline{a}u$.
Dabei ist

- $q \in K \cup \{h\}$ der aktuelle Zustand,
- $w \in (\Sigma \setminus \{\#\})\Sigma^* \cup \{\epsilon\}$ der Bandinhalt links des Kopfes,
- $a \in \Sigma$ das Zeichen unter dem Kopf, und
- $u \in \Sigma^*(\Sigma \setminus \{\#\}) \cup \{\epsilon\}$ der Bandinhalt rechts des Kopfes.

$C_2 = q_2, w_2\underline{a_2}u_2$ heißt Nachfolgekonfiguration von $C_1 = q_1, w_1\underline{a_1}u_1$, in Zeichen $C_1 \vdash_M C_2$, falls es einen Übergang $\delta(q_1, a_1) = \langle q_2, b\rangle$ gibt, mit:

Fall 1: $b \in \Sigma$. Dann $w_1 = w_2, u_1 = u_2$ und $a_2 = b$.

Fall 2: $b = L$. Für u_2: Wenn $a_1 = \#$ und $u_1 = \epsilon$ ist, dann $u_2 = \epsilon$, sonst $u_2 = a_1u_1$.
Für a_2 und w_2: Wenn $w_1 = \epsilon$ ist, dann $w_2 = \epsilon$ und $a_2 = \#$; sonst $w_1 = w_2a_2$.

Fall 3: $b = R$. Für w_2: Wenn $a_1 = \#$ und $w_1 = \epsilon$ ist, dann $w_2 = \epsilon$, sonst $w_2 = w_1a_1$.
Für a_2 und u_2: Wenn $u_1 = \epsilon$ ist, dann $u_2 = \epsilon$ und $a_2 = \#$; ansonsten $u_1 = a_2u_2$.

Wir übernehmen die Definitionen von *TM-berechenbar*, *akzeptieren* kanonisch für zw-DTM.

Theorem 4.23 (Simulation von zw-DTM durch DTM)

Zu jeder zw-DTM M*, die eine Funktion* f *berechnet oder eine Sprache* L *akzeptiert, existiert eine Standard-DTM* M'*, die ebenfalls* f *berechnet oder* L *akzeptiert.*

Beweis (Anfang)

Sei $w = a_1 \dots a_n$ Input für $M = (K, \Sigma, \delta, s)$. Dann sieht das beidseitig unendliche Band zu Beginn der Rechnung so aus:

$$\dots \# \# \# a_1 \dots a_n \underline{\#} \# \dots$$

Beweis (Fortsetzung)

Idee:

- M hat quasi zwei unendlich lange Halbbänder.
- M mit einer DTM M' mit nur einem Halbband simulieren, also den Inhalt beider Halbbänder von M auf einem unterbringen.
- Den Teil des Bandes, der zwei Zeichen links vom Input w beginnt, umklappen:

$$\begin{array}{ll} \text{Spur 1} & \# \, \# \, \ldots \, \# \, \# \\ \text{Spur 2} & \# \, a_1 \ldots a_n \, \underline{\#} \end{array} \; \cdots$$

- Die DTM M' hat zwei **Spuren**, d.h. zwei Reihen von Zeichen, die auf demselben Band untergebracht sind.
- Das Bandalphabet von M' ist $\Sigma' \supseteq \Sigma \times \Sigma$.

Beweis (Fortsetzung)

Sei $\mathrm{M}' = (\ K', \Sigma', \delta', s\ )$. M' rechnet so:

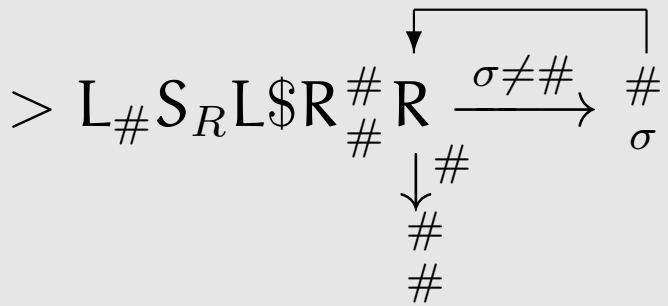
- M legt zunächst eine zweite Spur an,
- simuliert dann die Arbeit von M, und
- transformiert dann das Ergebnis wieder auf nur eine Spur herunter.

Beweis (Fortsetzung)

Erste Phase der Rechnung: M' rechnet

$$s, \#a_1 \dots a_n\underline{\#} \vdash^*_{\mathrm{M}'} q, \$ \begin{array}{c} \# \# \dots \# \; \# \\ \# \, a_1 \dots a_n \, \underline{\#} \end{array} \# \dots$$

Die zweite Spur wird nur so weit wie nötig angelegt.
Mit dem Symbol $\$$ markiert M' das Ende des Halbbands, damit sie nicht hängenbleibt.
Mit der Shift-Right-Maschine aus Beispiel 4.20 kann man diese Phase der Arbeit von M' so formulieren:



Beweis (Fortsetzung)

Zweite Phase der Rechnung: M' simuliert M.
Dabei muß sie sich immer merken, auf welcher der beiden Spuren sie gerade arbeitet.
Deshalb definieren wir $K' \supseteq K \times \{1,2\}$. (q,i) bedeutet, daß die simulierte Maschine M im Zustand q ist und M' auf Spur i arbeitet.
Für die Simulation von M durch M' soll nun gelten:

M erreicht von $s, \# \vdots \#w\underline{\#}$ aus eine Konfiguration $q, u_1 b \vdots a u_2$

($\vdots$ steht in beiden Konfigurationen zwischen denselben zwei Bandpositionen (an denen das Band „umklappt“)).

gdw

M' rechnet $p, \$ \begin{smallmatrix} \# & \cdots & \# \\ \# & w & \underline{\#} \end{smallmatrix} \# \vdash^*_{\mathrm{M}'} p', \$ \begin{smallmatrix} b & u_1^R & \# & & \# \\ a & u_2 & \# & \cdots & \# \end{smallmatrix} \#$

Beweis (Fortsetzung)

M' simuliert M wie folgt:

- Wenn M' das Zeichen $\$$ erreicht, wechselt sie die Spur.
- Wenn die simulierte Maschine M nach rechts (links) geht, geht M' nach rechts (links) auf Spur 2 und nach links (rechts) auf Spur 1.
- Wenn M' ein $\#$ erreicht (d.h. sie erreicht den Bandteil, wo noch nicht zwei Spuren angelegt sind), macht sie daraus $\begin{smallmatrix}\#\\\#\end{smallmatrix}$.

Gilt etwa $\delta_{\mathrm{M}}(q,a) = \langle q', L\rangle$, so muß in M' gelten:

- $\delta_{\mathrm{M}'}\left((q,2), \begin{smallmatrix}x\\a\end{smallmatrix}\right) = \langle (q',2), L\rangle$ für alle möglichen x,
- $\delta_{\mathrm{M}'}\left((q,1), \begin{smallmatrix}a\\x\end{smallmatrix}\right) = \langle (q',1), R\rangle$ (auf der oberen Spur ist der Inhalt des "linken Halbbandes" revers notiert, deshalb muß hier die Laufrichtung entgegengesetzt sein).

Beweis (Fortsetzung)

Außerdem gilt immer:

- $\delta_{M'}\big((q,1),\$\big) = \langle (q,2), R\rangle$,
- $\delta_{M'}\big((q,2),\$\big) = \langle (q,1), R\rangle$,
- $\delta_{M'}\big((q,i),\#\big) = \langle (q,i), \begin{smallmatrix}\#\\\#\end{smallmatrix}\rangle$,
- etc.

Die ersten beiden Gleichungen beschreiben den Spurwechsel beim Überschreiten von $\$$, die dritte das Erzeugen eines neuen Doppelspurstücks.

Beweis (Fortsetzung)

Wenn dann M mit $h, u\underline{\#}$ hält, dann erreicht M' eine der folgenden Konfigurationen:

(i) $\quad (h,1), \$ \begin{matrix} \# & & \# & & u^R & & \# & & \# \\ \# & \dots & \# & \# \dots \# & \underline{\#} & \dots & \# \end{matrix}$ oder

(ii) $\quad (h,2), \$ \begin{matrix} \# & & \# & & \# & & \# \\ \# & \dots & \# & \# \dots \# & \underline{\#} & \dots & \# \end{matrix}$ oder

(iii) $\quad (h,2), \$ \begin{matrix} u_1^R & \# & & \# \\ u_2\underline{\#} & \# & \dots & \# \end{matrix}$ mit $u_1 u_2 = u$.

Bei Konfigurations-Form (iii) kann entweder das u_1^R über das u_2 „hinausragen“ oder umgekehrt.

Beweis (Fortsetzung)

Dritte Phase der Rechnung: Die Simulation von M ist abgeschlossen.

M' muß nun den Bandinhalt von zwei Spuren auf nur eine heruntertransformieren, um danach die Konfiguration $h, \#u\underline{\#}$ zu erreichen.

Beweis (Fortsetzung)

- M' macht zunächst alle $\begin{smallmatrix}\#\\\#\end{smallmatrix}$ rechts vom beschriebenen Bandteil zu $\#$. Für Fall (i) und (ii) löscht sie die $\begin{smallmatrix}\#\\\#\end{smallmatrix}$ links von u^R bzw. u.
- Für Fall (iii) schiebt M' dann die untere Spur nach links, bis sie eine Konfiguration $q, \begin{smallmatrix}u_1^R\\\#\ldots\#\end{smallmatrix} u_2\underline{\#}$ erreicht.
- Für Fall (i) und (iii) muß M' jetzt u_1^R bzw. u^R auf nur eine Spur transformieren und zugleich invertieren, sie muß also für den allgemeineren Fall (iii) $q, \$ \begin{smallmatrix}u_1^R\\\#\ldots\#\end{smallmatrix} u_2\underline{\#} \vdash^*_{M'} q', \$u_1u_2\underline{\#}$ rechnen.
- Danach muß M' nur noch das $\$$ links löschen und nach rechts neben u laufen.

Beweis (Fortsetzung)

Damit hat die Standard-DTM M' die Arbeitsweise der zw-DTM M vollständig simuliert. Also kann man mit zw-Turing-Maschinen nicht mehr berechnen als mit Standard DTM'n. □

k-DTM

Definition 4.24 (DTM mit k Halbbändern, k-DTM)

Eine Turing-Maschine $\mathrm{M} = (\ K, \Sigma_1, \ldots, \Sigma_k, \delta, s\ )$ mit k Halbbändern mit je einem Kopf (**k-DTM**) ist eine Turing-Maschine mit einer Übergangsfunktion

$$\delta : K \times \Sigma_1 \times \ldots \times \Sigma_k \to (K \cup \{h\}) \times (\Sigma_1 \cup \{L, R\}) \times \ldots \times (\Sigma_k \cup \{L, R\})$$

Eine Konfiguration einer k-Turing-Maschine hat die Form

$$C = q, w_1\underline{a_1}u_1, \ldots, w_k\underline{a_k}u_k.$$

- Die Köpfe einer k-DTM können sich unabhängig bewegen (sonst hätten wir nur eine DTM mit k Spuren).
- Die Definition der Nachfolgekonfiguration verläuft analog zu der Definition bei Standard-DTM.
- Für eine k-DTM, die eine Funktion $f : \Sigma_0^m \to \Sigma_0^n$ berechnet, legen wir fest, daß sowohl die m Eingabewerte als auch – nach der Rechnung – die n Ergebniswerte auf dem ersten Band stehen sollen.
- Es **übertragen sich alle Begriffe** wie *berechenbar, entscheidbar* etc. kanonisch auf k-DTM.
- Für die Beschreibung von k-DTMn durch Flußdiagramme vereinbaren wir, daß $\sigma^{(i)}$ bedeuten soll, daß das Zeichen σ auf **Band i steht** bzw. auf **Band i geschrieben** wird.

Theorem 4.25 (Simulation von k-DTM durch DTM)

Zu jeder k-DTM M, die eine Funktion f berechnet (resp. eine Sprache L akzeptiert), existiert eine DTM M', die ebenfalls f berechnet (resp. L akzeptiert).

Beweis (Skizze):

Wir arbeiten mit einer Turing-Maschine mit mehreren Spuren.
Um eine k-**DTM** zu simulieren, verwenden wir $2k$ **Spuren**, also Bandzeichen, die aus $2k$ übereinander angeordneten Buchstaben bestehen.

- In den Spuren mit ungerader Nummer stehen die Inhalte der k Bänder von M.
- Die Spuren mit gerader Nummer verwenden wir, um die Positionen der Köpfe von M zu simulieren: Die $2i$-te Spur enthält an genau einer Stelle ein $\wedge$, nämlich da, wo M gerade seinen i-ten Kopf positioniert hätte, und ansonsten nur Blanks.

Beweis (Fortsetzung)

M' kodiert zunächst die Eingabe von M. Dann simuliert M' die Maschine M. Am Ende der Rechnung wird noch die Ausgabe dekodiert.

Man kann natürlich die DTM-Varianten kombinieren.

4.4 NTM'n

Definition 4.26 (NTM)

Eine indeterminierte Turing-Maschine **(NTM)** M ist ein Tupel $M = (K, \Sigma, \Delta, s)$.
Dabei sind K, Σ, s definiert wie bei determinierten Turing-Maschinen.
$\Delta \subseteq (K \times \Sigma) \times \Big((K \cup \{h\}) \times (\Sigma \cup \{L, R\})\Big)$ ist die Übergangs**relation**.

Schreibweisen:

- Statt $\langle (q,a),(q',b)\rangle \in \Delta$ schreiben wir auch $(q,a)\ \Delta\ (q',b)$.
- Wir betrachten Δ auch als mengenwertige Funktion, hier:

$$\Delta : K \times \Sigma \to 2^{\Big(K \cup \{h\}\Big) \times \Big(\Sigma \cup \{L,R\}\Big)}$$

Damit ist auch $\langle q',b\rangle \in \Delta(q,a)$ eine legitime Schreibweise.

Also genau wie schon bei indeterminierten endlichen Automaten (und bei Kellerautomaten).

Konfigurationen sind definiert wie bei DTMn. **Nun kann eine Konfiguration aber mehrere mögliche Nachfolgekonfigurationen haben.**

Definition 4.27 (Nachfolgekonfiguration, Rechnung)

Eine Konfiguration C_2 heißt Nachfolgekonfiguration von C_1, in Zeichen $C_1 \vdash_{\mathrm{M}} C_2$, falls $C_i = q_i, w_i \underline{a_i} u_i$ für $i \in \{1, 2\}$ gilt und es einen Übergang $\langle q_1, a_1 \rangle \, \Delta \, \langle q_2, b \rangle$ gibt, so daß einer der drei Fälle aus Def. 4.8 gilt.
$C_0 \ldots C_n$ heißt Rechnung einer NTM M, falls für alle $i < n$ gilt $C_i \vdash_{\mathrm{M}} C_{i+1}$.

$C_0 \ldots C_n$ ist also eine Rechnung einer indeterminierten Turing-Maschine, falls jeweils die Konfiguration C_{i+1} eine der Nachfolgekonfigurationen von C_i ist.

Definition 4.28 (NTM: Halten, Hängen, Akzeptieren)

Sei $\mathrm{M} = (K, \Sigma, \Delta, s_0)$ eine indeterminierte Turing-Maschine.

- M **hält** bei Input w, falls es **unter den möglichen Rechnungen**, die M wählen kann, **eine gibt**, so daß M eine Haltekonfiguration erreicht.
- M **hängt** in einer Konfiguration, wenn es keine (durch Δ definierte) Nachfolgekonfiguration gibt.
- M **akzeptiert** ein Wort w, falls sie **von** $s, \#w\underline{\#}$ **aus einen Haltezustand erreichen kann**, und M akzeptiert eine Sprache L, wenn sie genau alle Wörter $w \in L$ akzeptiert.

Wenn es nicht nur darauf ankommt, ob die Maschine hält, sondern auch mit welchem Bandinhalt:

Welche der vielen Haltekonfigurationen sollte dann gelten?

Um dieses Problem zu umgehen, übertragen wir die Begriffe des *Entscheidens* und *Aufzählens* **nicht** auf NTM. Im Allgemeinen verwendet man NTM auch nicht dazu, Funktionen zu berechnen.

Wie rechnet eine indeterminierte Turing-Maschine?

- Die Regeln einer determinierten DTM kann man sich als Programm (aus sehr einfachen Schritten) vorstellen.
- **Bei NTM ist das anders!**
- Eine NTM ist nicht einfach eine Maschine, die immer richtig rät!

Dieselbe Diskussion hatten wir bei indeterminierten endlichen Automaten.

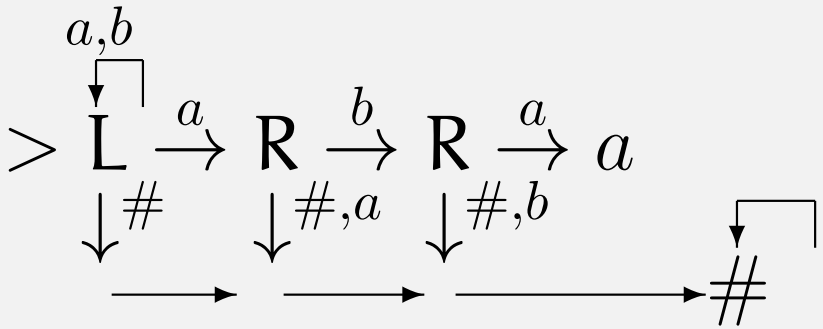
- Korrekte Vorstellung:
  - Übergänge von Konfiguration zu Nachfolgekonfiguration entspr. Regelmenge
  - **plus Suchverfahren!**

  oder: Eine NTM beschreitet alle möglichen Rechenwege parallel.
- Eine NTM akzeptiert ein Wort, wenn es mindestens einen Berechnungsweg gibt, der in einer Haltekonfiguration endet.
- Sprechweise **„Die NTM rät“**: Wir verwenden diese Sprechweise, sie ist aber mit Vorsicht zu genießen!

Beispiel 4.29 (Indeterminierte TM)

Die folgende NTM akzeptiert

$L = \{w \in \{a,b\}^* \mid w$ besitzt aba als Teilwort$\}$.

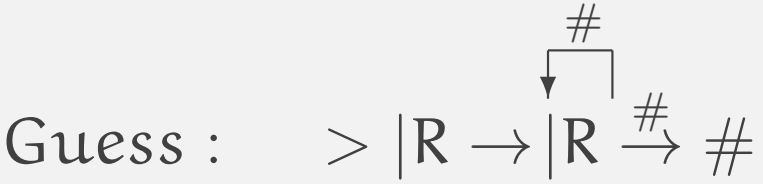


Sei $L = \{|^n \mid n$ ist nicht prim und $n \geq 2\}$.
Wir konstruieren eine Turing-Maschine
Composite, die indeterminiert zwei Zahlen rät,
diese miteinander multipliziert und mit dem
Eingabewort vergleicht.

Beispiel 4.30 (Composite)

Es ist Composite: $>$ R Guess R Guess Mult Compare:

- Guess, eine indeterminierte Turing-Maschine, rät eine Zahl $n \geq 2$, d.h. $s, \#\underline{\#} \vdash^*_{\text{Guess}} h, \#|^n\underline{\#}$ wie folgt:



- Mult multipliziert (determiniert) zwei Zahlen n und m:

$$s, \#|^n\#|^m\underline{\#} \vdash^*_{\text{Mult}} h, \#|^{n*m}\underline{\#}$$

- Compare vergleicht zwei Zahlen n und m und hält nur dann, wenn beide gleich sind:

$$s, \#|^n\#|^m\underline{\#} \vdash^*_{\text{Compare}} h, \#|^n\#|^n\underline{\#} \text{ gdw } n = m$$

Theorem 4.31 (Simulation von NTM durch DTM)

Jede Sprache, die von einer NTM akzeptiert wird, wird auch von einer Standard-DTM akzeptiert.

Beweis (Anfang)

Sei

- L eine Sprache über Σ_0 mit $\# \notin \Sigma_0$;
- $M = (K, \Sigma, \Delta, s)$ eine indeterminierte Turing-Maschine, die L akzeptiert.

Beweis (Fortsetzung)

Wir konstruieren zu M eine Standard-DTM M', die so rechnet:

- M' durchläuft systematisch alle Rechnungen von M und sucht nach einer Haltekonfiguration.
- M' hält dann und nur dann, wenn sie eine Haltekonfiguration von M findet.

Beweis (Fortsetzung)

Suchbaum, Rechnungsbaum: Man kann alle Rechnungen von M von einer Startkonfiguration C_0 aufmalen als einen Baum mit Wurzel C_0. Ein Ast ist eine mögliche Rechnung von M.

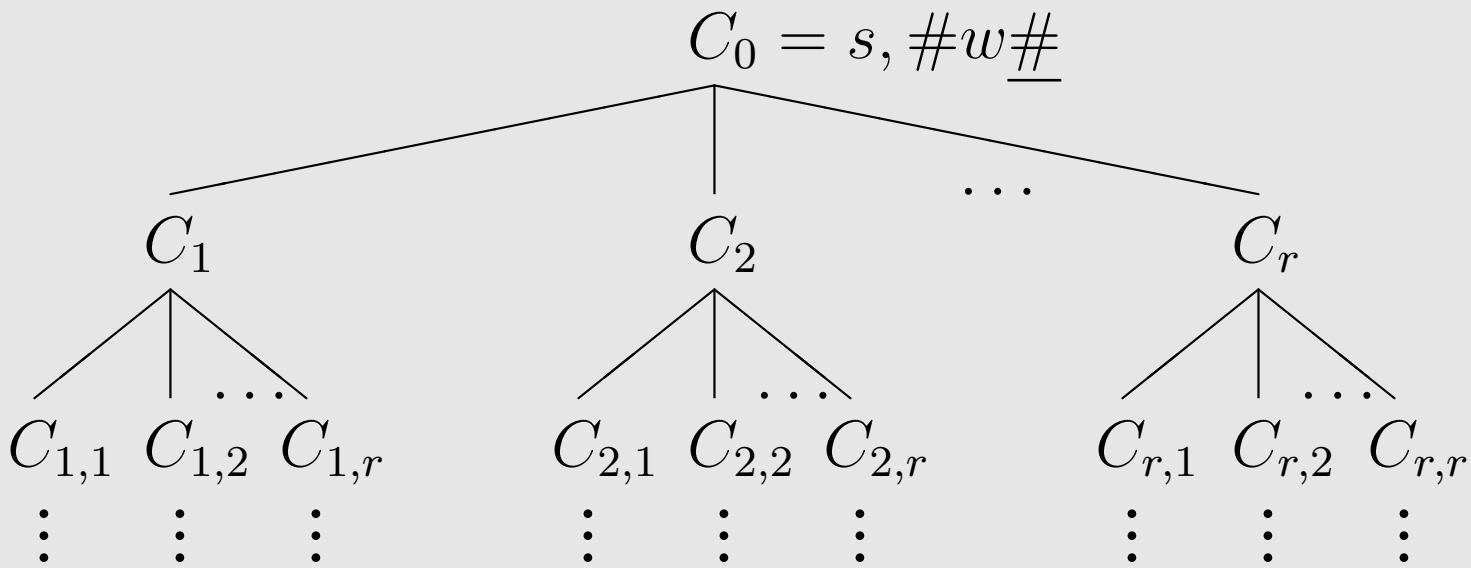


Abbildung 26: R_{C_0}, der Rechnungsbaum von C_0 aus

Beweis (Fortsetzung)

Problem: Es kann zur Startkonfiguration

$$C_0 = s, \#w\underline{\#}$$

unendlich viele Rechnungen von M geben, und jede einzelne von ihnen kann unendlich lang sein. Wir können also nicht erst einen Ast ganz durchlaufen und dann den nächsten Ast durchsuchen.

Beweis (Fortsetzung)

Die Lösung: Wir durchlaufen den Rechnungsbaum nicht depth-first, sondern per **iterative deepening**. Das geht so:

- Wir untersuchen erst alle möglichen Rechnungen, aber nur bis zum ersten Schritt.
- Dann untersuchen wir nochmal alle möglichen Rechnungen, und zwar zwei Schritte weit.
- Dann untersuchen wir nochmal alle möglichen Rechnungen, und zwar drei Schritte weit, etc.

Beweis (Fortsetzung)

Können wir damit denn in endlicher Zeit alle Haltekonfigurationen finden?

Problem: Kann der Rechnungsbaum nicht nur unendlich tief, sondern auch unendlich breit werden?

Beweis (Fortsetzung)

Nein, das kann nicht passieren. Obwohl es von der Startkonfiguration C_0 aus unendlich viele verschiedene Rechnungen geben kann, hat jede Konfiguration nur endlich viele Nachfolger:

- Sei der aktuelle Zustand q, das aktuelle Zeichen unter dem Kopf a.
- Dann gibt es höchstens $|K| + 1$ verschiedene Nachfolgezustände,
- die Maschine kann höchstens $|\Sigma|$ viele verschiedene Zeichen drucken oder nach links oder nach rechts gehen.
- Also: Maximale Anzahl von Nachfolgekonfigurationen

$$r = max\{|\Delta(q, a)| \quad | \quad q \in K, a \in \Sigma\}$$

Beweis (Fortsetzung)

M' kann z.B. als eine 3-DTM gewählt werden:

- Auf dem ersten Band steht immer das Eingabewort w. Da die Rechnung immer wieder neu mit $s, \#w\underline{\#}$ von M beginnt, wird das Eingabewort immer wieder gebraucht.
- Auf dem zweiten Band steht, welcher Weg durch den Rechnungsbaum gerade verfolgt wird.
  Der Einfachheit halber: Wenn eine Konfiguration weniger als r Nachfolgekonfigurationen hat, soll der zugehörige Knoten trotzdem r Söhne haben, und die überzähligen Konfigurationen sind einfach leer.

Beweis (Fortsetzung)

Dann ist der aktuelle Pfad im Rechnungsbaum darstellbar als Zahl im r-adischen System.
Eine Zahl $d_1 \dots d_n$ bedeutet:

- Von der Startkonfiguration C_0 aus ist die d_1-te der r möglichen Nachfolgekonfigurationen gewählt worden, C_{d_1}.
- Von C_{d_1}, einem Knoten der Tiefe 1, aus wurde die d_2-te mögliche Nachfolgekonfiguration gewählt, etc.,
- bis zu einer Tiefe im Baum von n.

Beweis (Fortsetzung)

Dann geht "iterative deepening" so:

- Wir beginnen mit einem leeren zweiten Band. Die Zahl 0 steht für eine Rechnung, die nur die Startkonfiguration C_0 umfaßt.
- Die nächste Rechnung erhalten wir, indem wir zu der Zahl auf Band 2 die Zahl 1 im r-adischen System addieren.

Beweis (Fortsetzung)

- Auf Band 3 wird eine Rechnung von M determiniert simuliert, und zwar entsprechend der Zahl $d_1 \ldots d_n$ auf Band 2.
  Die Endkonfiguration $C_{d_1 \ldots d_n}$ dieser Rechnung steht im Rechnungsbaum an dem Knoten, der das Ende des Pfades $d_1 \ldots d_n$ bildet.
  Ist die Konfiguration $C_{d_1 \ldots d_n}$ eine Haltekonfiguration, so hält M'.
  Sonst wird zu der Zahl auf Band 2 eins addiert und die nächste Rechnungssimulation begonnen.

Beweis (Ende)

Damit gilt:

M' hält bei Input w

gdw

es gibt in R_{C_0} eine Haltekonfiguration.

Das ist genau dann der Fall, wenn M bei Input w hält, das heißt, wenn w in L liegt. □

4.5 Universelle DTMn

- Turing-Maschinen sind sehr mächtig, und wir haben sie bisher sehr flexibel eingesetzt.

  Wie mächtig sind sie wirklich?

- **Vergleich mit normalen Computern:**
  Eine Turing-Maschine hat eine vorgegebene Regelmenge.
  „Normale“ Computer sind flexibel: **Sie können beliebige Programme erhalten und ausführen.**

- Tatsächlich geht das mit Turing-Maschinen auch.

- Man kann eine Turing-Maschine U konstruieren, die beliebige Turing-Maschinen simuliert:

  U bekommt als Eingabe
  - die Regelmenge einer beliebigen Turing-Maschine M und
  - ein Wort w, auf dem M rechnen soll.

  U **simuliert** M, indem sie jeweils nachschlägt, welchen δ-Übergang M machen würde.

  Eine solche Turing-Maschine U heißt universelle Turing-Maschine.

Frage: In welches Format fasst man die Regeln einer DTM M am besten, um sie einer universellen DTM als Eingabe zu geben?

Anders ausgedrückt: Was muss man angeben, um eine DTM komplett zu beschreiben?

- ihr Alphabet,
- ihre Zustände,
- ihre δ-Übergänge und
- ihren Startzustand.

Wenn wir jetzt noch das Alphabet, die Zustände und den Startzustand für alle DTM standardisieren, dann reicht es aus, nur die δ-Übergänge anzugeben. Also:

- Nehmen wir ein unendliches Alphabet $\Sigma_\infty = \{a_0, a_1, \ldots\}$ an, so daß das Alphabet jeder DTM eine Teilmenge von Σ_∞ ist.
- Die Namen der Zustände einer DTM sind ja egal. Nehmen wir also einfach an, daß sie $q_1, \ldots, q_n$ sind – das n kann dabei von DTM zu DTM verschieden sein.
- Außerdem legen wir fest, daß q_1 immer der Startzustand sein soll, und daß wir den Haltezustand mit q_0 bezeichnen.

Damit haben wir erreicht:
Wir können eine DTM komplett beschreiben, indem wir nur ihre δ-Übergänge beschreiben.

Wie kann eine solche Beschreibung aussehen?

Die DTM $\mathrm{L}_{\#}$ hat die Regeln

$$\langle s, \# \rangle \mapsto \langle q_1, L \rangle \qquad \langle q_1, \# \rangle \mapsto \langle h, \# \rangle$$
$$\langle s, | \rangle \mapsto \langle q_1, L \rangle \qquad \langle q_1, | \rangle \mapsto \langle q_1, L \rangle$$

Angenommen, $\#$ ist das Zeichen a_0, und $|$ ist a_1.
Wir können dann die DTM $\mathrm{L}_{\#}$ so beschreiben:

	a_0	a_1
q_1	q_2, L	q_2, L
q_2	q_0, a_0	q_2, L

oder kürzer:

$$Z\ S\ 2\lambda\ S\ 2\lambda$$
$$Z\ S\ 0a0\ S\ 2\lambda$$

Es steht

- Z für „nächste Zeile“,
- S für „nächste Spalte“,
- λ für „links”,
- ρ für „rechts“,
- die Zahl n für den n-ten Zustand und an für das n-te Zeichen von Σ_∞.

Damit wird die DTM durch ein Wort beschrieben: $ZS2\lambda S2\lambda ZS0a0S2\lambda$. Nun ersetzen wir alle vorkommenden Zahlen durch die entsprechende Anzahl von $|$ (0 bleibt) und erhalten

$$ZS||\lambda S||\lambda ZS0a0S||\lambda.$$

Beispiel 4.32 (Gödelisierung: Gödelzahl, Gödelwort)

Als Gödelisierung bezeichnet man ein Verfahren, jeder Turing-Maschine eine natürliche Zahl, eine Gödelzahl, oder ein Wort über einem Alphabet, ein Gödelwort, so zuzuordnen, daß man aus der Zahl bzw. dem Wort die Turing-Maschine effektiv rekonstruieren kann.

Wir haben also gerade ein Verfahren angegeben, einer DTM ein **Gödelwort** zuzuweisen.
Dabei war es wichtig, daß wir Σ_∞ verwendet haben.

Wir müssen, wenn wir das Programm einer DTM als Eingabe für die **universelle DTM** U verwenden wollen, ein **einheitliches** Alphabet haben, mit dem wir **alle** Turing-Maschinen beschreiben können.

Vom Gödelwort zur Gödelzahl

Wie ordnen wir einem Gödelwort eine eindeutige natürliche Zahl zu?

Wir nutzen die Primfaktorzerlegung der natürlichen Zahlen aus: sie ist eindeutig.

- Ein Gödelwort ist endlich lang. Der i-ten Position ordnen wir die i-te Primzahl zu.
- Jede Primzahl bekommt nun noch einen Exponenten und die Gödelzahl ist dann genau das Produkt.

Vom Gödelwort zur Gödelzahl (2)

- Je nachdem was an der i-ten Position im Gödelwort steht, bestimmen wir den Exponenten. Es können ja nur folgende sieben Zeichen vorkommen: $Z, S, \lambda, \rho, 0, |, a$. **Ordnen wir diesen Zeichen also jeweils die Zahlen 1 bis 7 zu.**
- Dann kann von jeder natürlichen Zahl bestimmt werden, ob sie ein Code für eine DTM ist oder nicht. **Und jede Maschine bekommt genau eine Gödelzahl.**

Aus dem Gödelwort $ZS||\lambda S||\lambda ZS0a0S||\lambda$ wird also die natürliche Zahl

$$2^1 3^2 5^6 7^6 11^3 13^2 17^6 19^6 23^3 29^1 31^2 37^5 41^7 43^5 47^6 53^2 59^6 61^3.$$

4.6 Entscheidbar/Aufzählbar

Frage:

Was heißt es, daß eine DTM eine Sprache akzeptiert?

- Ein endlicher Automat akzeptiert ein Wort, wenn er **nach dem Lesen des letzten Buchstabens** in einem **finalen Zustand** ist.
- Ein Pushdown-Automat akzeptiert ein Wort, wenn er **nach dem Lesen des letzten Buchstabens** in einem **finalen Zustand** ist bzw. seinen Keller geleert hat.
- Wann akzeptiert eine DTM ein Wort?

Eine DTM kann auf einem gegebenen Eingabewort . . .

- irgendwann den Haltezustand erreichen und damit anhalten,
- unendlich lang rechnen, oder
- hängenbleiben.

Welcher Fall eintritt wissen wir im Allgemeinen nicht!

„Akzeptieren“ und „Entscheiden“:

- Eine DTM akzeptiert eine Sprache L, wenn sie
  - für jedes Eingabe-Wort $w \in L$ irgendwann hält,
  - für jedes Wort $v \notin L$ aber unendlich lang rechnet oder hängt.
- Eine DTM entscheidet eine Sprache L, wenn sie
  - für jedes Eingabe-Wort $w \in L$ hält mit dem Bandinhalt Y (“Yes”) und
  - für jedes Wort $v \notin L$ hält mit dem Bandinhalt N (“No”).

Definition 4.33 (Entscheidbar)

L sei eine Sprache über Σ_0, $\{\#, N, Y\} \cap \Sigma_0 = \emptyset$.
$M = (\ K, \Sigma, \delta, s\ )$ sei eine DTM mit $\Sigma_0 \subseteq \Sigma$.
M entscheidet L, falls $\forall w \in \Sigma_0^*$ gilt:

$$s, \#w\underline{\#} \vdash_M^* \begin{cases} h, \#Y\underline{\#} & \text{falls } w \in L, \\ h, \#N\underline{\#} & \text{sonst.} \end{cases}$$

L heißt entscheidbar (oder recursive, rec.), falls es eine DTM gibt, die L entscheidet.

Definition 4.34 (Akzeptierbar)

L sei eine Sprache über Σ_0, $\{\#, N, Y\} \cap \Sigma_0 = \emptyset$.
$\mathrm{M} = (\ K, \Sigma, \delta, s\ )$ sei eine DTM mit $\Sigma_0 \subseteq \Sigma$.
M **akzeptiert** ein Wort $w \in \Sigma_0^*$, falls M bei Input w hält. M **akzeptiert die Sprache** L, falls $\forall w \in \Sigma_0^*$ gilt: (M akzeptiert w *genau dann wenn* $w \in L$).
L heißt **akzeptierbar** (oder auch **semi-entscheidbar**), falls es eine DTM gibt, die L akzeptiert.

Definition 4.35 (Rekursiv Aufzählbar (r.e.))

Sei wieder L eine Sprache über Σ_0, wie oben, und $\mathrm{M} = (\ K, \Sigma, \delta, s\ )$ eine DTM.
M zählt L auf, falls es einen Zustand $q_B \in K$ gibt (den Blinkzustand), so daß

$$L = \{w \in \Sigma_0^* \mid \exists u \in \Sigma^* : s, \underline{\#} \vdash_{\mathrm{M}}^* q_B, \#w\underline{\#}u\}$$

gilt. L heißt rekursiv aufzählbar (kurz r.e.: "recursively enumerable"), falls es eine DTM gibt, die L aufzählt. Oft sagen wir auch nur aufzählbar.

Achtung: aufzählbar versus abzählbar.

Satz 4.36 (Akzeptierbar = Aufzählbar)

Eine Sprache L ist genau dann aufzählbar*, wenn L* akzeptierbar *ist.*

Beweis

„$\Rightarrow$“ Zu zeigen: L ist aufzählbar $\Rightarrow$ L ist akzeptierbar.

- Ohne Einschränkung sei $L \subseteq \Sigma^*$ und $\# \notin \Sigma$.
- Sei M_L eine DTM, die L aufzählt.

Beweis (Fortsetzung)

Wir konstruieren aus M_L eine DTM M, die L akzeptiert:

- M ist eine 2-DTM.
- M wird gestartet mit

$$\begin{array}{ll} s_0, & \#w\underline{\#} \\ & \underline{\#} \end{array}$$

- M simuliert auf Band 2 die Maschine M_L.
- Wenn M_L den Blinkzustand q_B erreicht, dann enthält Band 2 von M ein Wort

$$\#w'\underline{\#}u$$

mit $w' \in L$.

Beweis (Fortsetzung)

- M vergleicht w und w'.
  - Falls $w = w'$, dann hält M: $w \in L$.
  - Ansonsten simuliert M auf Band 2 weiter die Arbeit von M_L.
- Wenn M_L hält, ohne das Wort w auf Band 2 erzeugt zu haben, gerät M in eine Endlosschleife.

Beweis (Fortsetzung)

„$\Leftarrow$“ Zu zeigen: L ist akzeptierbar $\Rightarrow$ L ist aufzählbar.

- Sei L eine Sprache über dem Alphabet Σ.
- Sei M_L eine DTM, die L akzeptiert.
- daraus konstruieren wir eine DTM M, die L aufzählt.

- Grundidee:
  - die Wörter aus Σ^* der Reihe nach aufzählen
  - jedes Wort der Maschine M_L vorlegen
  - wenn M_L das Wort akzeptiert, in den Blinkzustand q_B gehen

Beweis (Fortsetzung)

- Problem:
  M_L akzeptiert, sie entscheidet nicht.
  Wenn M_L ein Wort nicht akzeptiert, rechnet sie unendlich.

- Lösung:
  Wir betrachten die Rechnung von M_L zu allen Wörtern aus Σ^* gleichzeitig.

Beweis (Fortsetzung)

- Sei $\Sigma^* = \{w_1, w_2, w_3, \ldots\}$, wenn man die Wörter in lexikalischer Reihenfolge aufzählt.
- Dann rechnet M so:
  - Im ersten Durchlauf berechnen wir einen (den ersten) Rechenschritt von M_L für w_1.
  - Im zweiten Durchlauf berechnen wir
    - die ersten zwei Rechenschritte von M_L für w_1 - wir fangen also für w_1 noch einmal mit der Startkonfiguration an – und
    - einen Rechenschritt für w_2.
  - Im dritten Durchlauf berechnen wir drei Rechenschritte für w_1, zwei für w_2 und einen für w_3 und so weiter.

Immer wieder bei der Startkonfiguration anfangen: So müssen wir uns den Bandinhalt der Rechnung von M_L zu w_i nach j Schritten nicht merken.

Beweis (Fortsetzung)

Nach dem n-ten Durchlauf haben wir n Schritte der Rechnung zum ersten Wort und einen Schritt der Rechnung zum n-ten Wort aus Σ^* simuliert.

- Wenn M so rechnet, dann gilt:
  - M fängt für jedes $w_i \in \Sigma^*$ in endlicher Zeit (nämlich im i-ten Durchlauf) an, die Arbeit von M_L zu w_i zu simulieren, und
  - falls $w_i \in L$ und falls M_L, gestartet mit $s, \#w_i\underline{\#}$, in j Schritten einen Haltezustand erreicht, dann erreicht M nach endlicher Zeit (nämlich im $i+j$-ten Durchlauf) den Haltezustand von M_L in der Rechnung zu w_i.

Beweis (Ende)

- Wenn M bei Simulation von M_L zur Eingaben w_i auf eine Haltekonfiguration trifft, dann ist $w_i \in L$. M nimmt dann eine Konfiguration

$$q_B, \#w_i\underline{\#}u_i$$

ein – q_B ist der Blinkzustand. In der Nebenrechnung u_i steht, welche Teilrechnung von M_L als nächste zu simulieren ist.

- Also zählt M die $w \in \Sigma^*$ auf, für die M_L hält, und das sind gerade die $w \in L$. □

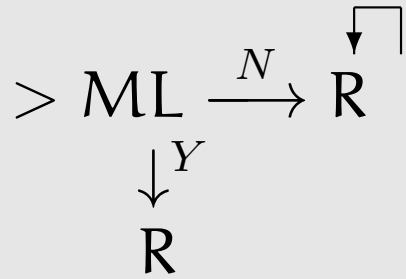
Die Begriffe entscheidbar, akzeptierbar und aufzählbar verwendet man auch für Teilmengen von $\mathbb{N}$, dargestellt durch Sprachen über $\{|\}$.

Satz 4.37 (Entscheidbar vs. Akzeptierbar)

- *Jede entscheidbare Sprache ist akzeptierbar.*
- *Das Komplement einer entscheidbaren Sprache ist entscheidbar.*

Beweis

- Sei L eine entscheidbare Sprache und M eine DTM, die L entscheidet. Dann wird L akzeptiert von der DTM M', die zunächst M simuliert und danach in eine Endlosschleife geht, falls M mit $h, \#N\underline{\#}$ endet:



Also: $w \in L$ M hält bei Input w mit der "Antwort" Y M' hält bei Input w. D.h. M' akzeptiert L.

Beweis (Fortsetzung)

- Sei L eine entscheidbare Sprache und M eine DTM, die L entscheidet. Dann wird $\overline{L}$ entschieden von einer DTM M', die genau wie M rechnet und nur am Schluß die Antworten Y und N vertauscht. □

Satz 4.38 (Charakterisierung von Entscheidbarkeit)

Eine Sprache L ist genau dann entscheidbar, *wenn* sie selbst und ihr Komplement akzeptierbar *sind.*

Beweis

"$\Rightarrow$" Zu zeigen: L ist entscheidbar $\Rightarrow$ L und $\overline{L}$ sind akzeptierbar.

- L ist entscheidbar, also ist L akzeptierbar (Satz 4.37).
- L ist entscheidbar, also ist auch $\overline{L}$ entscheidbar (Satz 4.37).
- $\overline{L}$ ist entscheidbar, also ist $\overline{L}$ akzeptierbar (Satz 4.37).

Beweis (Fortsetzung)

"$\Leftarrow$" Zu zeigen: L und $\overline{L}$ sind akzeptierbar $\Rightarrow$ L ist entscheidbar.

- Sei M_1 eine DTM, die L akzeptiert.
- Sei M_2 eine DTM, die $\overline{L}$ akzeptiert.

Daraus konstruieren wir eine 2-DTM M, die L entscheidet:

- M wird gestartet mit

$$s_0, \#w\underline{\#}$$
$$\underline{\#}$$

- M kopiert w auf Band 2.

Beweis (Ende)

- M simuliert abwechselnd
  - einen Schritt von M_1 auf Band 1 und
  - einen Schritt von M_2 auf Band 2.
- Das tut M, bis entweder M_1 oder M_2 hält.
- Eine von beiden muss halten: w gehört entweder zu L oder zu $\overline{L}$.
- Wenn M_1 hält, dann hält M mit
  - $\#Y\underline{\#}$ auf Band 1 und
  - $\underline{\#}$ auf Band 2.
- Wenn M_2 hält, dann hält M mit
  - $\#N\underline{\#}$ auf Band 1 und
  - $\underline{\#}$ auf Band 2.

□

4.7 DTM = Typ 0

Erinnern Sie sich, daß wir im ersten Kapitel **Sprachen vom Typ 0** eingeführt hatten: das waren Sprachen, die durch beliebige Grammatiken (keinerlei Einschränkungen) erzeugt werden können. Wir zeigen nun:

Satz 4.39 (Aufzählbar = Typ 0)

Die aufzählbaren Sprachen (also die durch DTMn akzeptierbaren Sprachen) sind genau die durch beliebige Grammatiken erzeugten Sprachen (also die vom Typ 0).

Lemma 4.40 (DTM akzeptieren Sprachen vom Typ 0)

Sei L eine Sprache vom Typ 0. Dann gibt es eine DTM, die L akzeptiert.

Beweis

Es genügt eine NTM zu konstruieren, da diese durch eine DTM simuliert werden kann (nach Theorem 4.31).

- Sei L eine Sprache über T vom Typ 0.
- Dann gibt es eine Grammatik $G_0 = (V, T, R, S)$, die L erzeugt.
- Wir konstruieren eine 2-NTM M_0, die L akzeptiert: Sie rät eine Ableitung von G_0.
- M_0 bekommt auf Band 1 ein Eingabewort w.
- Sie erzeugt auf Band 2 das Startsymbol S der Grammatik.

$$s_0, \begin{array}{l}\#w\underline{\#}\\ \underline{\#}\end{array} \vdash^*_{\mathrm{M}_0} s_0, \begin{array}{l}\#w\underline{\#}\\ \#\underline{S}\end{array}$$

Beweis (Fortsetzung)

- Dann rät M_0 auf Band 2 eine Ableitung in G_0.
  Zu $S \Longrightarrow^* u \Longrightarrow^* \ldots$ rechnet sie

$$s_0, \begin{matrix} \#w\underline{\#} \\ \#\underline{S} \end{matrix} \vdash^* \begin{matrix} \#w\underline{\#} \\ \#u\underline{\#} \end{matrix}$$

- M_0 wählt indeterminiert eine Regel $P \to Q \in R$ aus.
- Sie sucht im Wort auf Band 2 indeterminiert nach P.
- Sie ersetzt das ausgewählte P durch Q.
- Wenn Q länger oder kürzer ist als P, benutzt M_0 die Shift-Maschinen S_R und S_L, um den Rest des Wortes rechts neben dem ausgewählten P zu verschieben.
- M_0 wählt die nächste Regel aus.

Beweis (Fortsetzung)

- Irgendwann beschließt M_0 indeterminiert, das Wort auf Band 2 mit w zu vergleichen.
  Falls $u = w$, hält M_0, sonst hält M_0 nie. Damit gilt:

$$M_0 \text{ hält bei Input } w$$
$$\underline{\text{gdw}}$$
$$\text{Es gibt eine Ableitung } S \Longrightarrow_{G_0}^{*} w$$
$$\text{(von } M_0 \text{ auf Band 2 nachzuvollziehen)}$$
$$\underline{\text{gdw}}$$
$$w \in L(G_0).$$

Nun zeigen wir die Umkehrung des letzten Lemmas.

Lemma 4.41 (Aufzählbar impliziert Typ 0)

Sei L eine aufzählbare Sprache (also eine, die von einer DTM akzeptiert wird). Dann gibt es eine Grammatik die L erzeugt.

Beweis

Gegeben Sprache L und DTM M, die L akzeptiert.

Wir konstruieren Grammatik G, die L erzeugt:

- Sie erzeugt beliebiges Wort $A_1 \dots A_n$,
- und spielt M auf Input $a_1 \dots a_n$ nach.
- Wenn w von M akzeptiert wird, macht G aus $A_1 \dots A_n$ das Wort $a_1 \dots a_n$.
- Wenn $a_1 \dots a_n$ von M nicht akzeptiert wird, macht G aus $A_1 \dots A_n$ keine Terminale, verwirft also das Wort.

Beweis (Fortsetzung)

Turing-Maschinen in einem speziellen Format:

- Bei normalen DTMn ist die Startkonfiguration bei Eingabe va

$$s, \#va\underline{\#}$$

- Wir verwenden DTMn, mit Startkonfiguration bei Eingabe va:

$$\Subset v\underline{a} \Supset$$

Beweis (Fortsetzung)

- Die DTM soll während ihrer ganzen Rechnung
  - das linke Bandende mit $\Subset$ und
  - das rechte "Bandende" mit $\Supset$

  begrenzen.
- Wenn sie nach rechts mehr Platz braucht, soll sie zuerst das $\Supset$ verschieben.
- Zu jeder "normalen" DTM M, die eine Sprache L akzeptiert, gibt es eine DTM M' im neuen Format.

Beispiel 4.42 (TM mit Begrenzern)

Die Turing-Maschine M akzeptiert $\{a^n \mid n \in \mathbb{N}\}$:

- Es ist $\mathrm{M} = (\{q_1, q_2\}, \{a, b\}, \delta, q_1)$
- mit folgendem δ:

$$\begin{aligned} \langle q_1, a\rangle &\mapsto \langle q_1, L\rangle \quad \mapsto \\ \langle q_1, b\rangle &\mapsto \langle q_2, b\rangle \\ \langle q_1, \Subset\rangle &\mapsto \langle h, \Subset\rangle \end{aligned}$$

Die fehlenden δ-Übergänge sind beliebig.

Die Grammatik G soll zu M so ableiten:

- Am Anfang rät G ein Wort. Außerdem muss G noch den Zustand und die Kopfposition von M codieren.
- G fängt so an zu arbeiten:

$$S \Longrightarrow_G^* \Subset \begin{matrix} w \\ w \end{matrix} \left\langle \begin{matrix} a \\ a \\ s_0 \end{matrix} \right\rangle \Supset .$$

Wie kann man das in Regeln ausdrücken?

- Mit folgenden Regeln:

$$S \to \Subset A_1 \mid \Subset \left\langle \begin{matrix} \# \\ \# \\ s_0 \end{matrix} \right\rangle \Supset \quad \text{(letztere wg. Input } \epsilon\text{)}$$

$$A_1 \to {}^{c}_{c} A_1 \mid \left\langle \begin{matrix} c \\ c \\ s_0 \end{matrix} \right\rangle \Supset \qquad \forall c \in T$$

- Die zweistöckigen und dreistöckigen Symbole sind Variablen von G!
  Zustand und Kopfposition zusammen sind in der dritten Spur kodiert.
- Zum Beispiel:

$$S \Longrightarrow_G^* \Subset \begin{matrix} a\,a \\ a\,a \end{matrix} \left\langle \begin{matrix} a \\ a \\ s_0 \end{matrix} \right\rangle \Supset .$$

Weitere Rechnung zu diesem Beispiel an der Tafel.

Falls $(q, a)\Delta(q', a')$: $\forall b \in T \cup \{\#\}$:

Falls $(q, a)\Delta(q', R)$: $\forall b, c, d \in T \cup \{\#\}$:

Falls $(q, a)\Delta(q', L)$: $\forall b, c, d \in T \cup \{\#\}$:

Falls $(q, \Subset)\Delta(q', R)$: $\forall c, d \in T \cup \{\#\}$:

- Wenn M hält, dann ist das Eingabewort in V^*. Wir gehen dann so vor:
  - Wir löschen die untere Spur.
  - Wir behalten nur die obere Spur, das anfangs geratene Eingabewort.

Die Grammatik ist nicht beschränkt:

- Die simulierte DTM darf den rechten Randbegrenzer $\ni$ nach links und rechts verschieben.
- Allgemein haben wir zuerst eine Ableitung

$$S \Longrightarrow_G^* \in \begin{matrix} u \\ u \end{matrix} \left\langle \begin{matrix} a \\ a \\ s_0 \end{matrix} \right\rangle \ni \Longrightarrow_G^* \in \begin{matrix} u_1 \\ x_1 \end{matrix} \left\langle \begin{matrix} c \\ b \\ h \end{matrix} \right\rangle \begin{matrix} u_2 \# \ldots \# \\ x_2 \# \ldots \# \end{matrix} \ni$$

- Danach löscht G die überschüssigen Blanks, das $\ni$ rechts und den $\in$ links (Wort wird kürzer)!

Insgesamt erzeugt G ein Wort w genau dann, wenn die DTM M das Wort w akzeptiert.

4.8 Unentscheidbar

Wie bereits erwähnt ist es nicht einfach, sich eine Funktion vorzustellen, die nicht berechenbar ist, ohne auf allzu künstliche Beispiele zurückzugreifen. Hier nun eine, die auf [Lin and Rado1965] zurückgeht.

Definition 4.43 (Busy Beaver)

Die Funktion $BB : \mathbb{N} \to \mathbb{N}$ sei wie folgt definiert:

$$n \mapsto BB(n) := \text{die maximale Anzahl an Einsen, die eine DTM (Definition 4.1) definiert über } \Sigma = \{\#, 1, c\}) \text{ mit maximal } n \text{ Zuständen auf einem leeren Band erzeugen kann und dann hält.}$$

Wir lassen also ein weiteres Hilfszeichen "c" zu. Dies erlaubt es höhere Werte zu erhalten, ist aber für die Unentscheidbarkeit irrelevant.

Die ersten Werte von BB

Es ist einfach, BB für $0, 1$ zu berechnen. Mit welchem Ergebnis?

Busy Beaver ist in der klassischen Version definiert für zw-DTM'en und Zielzustände (Definition 4.22). Die einzigen bekannten Werte (in unserer Terminologie, Stand 2016) sind: $BB(0) = 1$, $BB(1) = 4$, $BB(2) = 13$. Bereits $BB(4)$ ist nicht bekannt. Und $BB(5)$ ist größer als $1.29 * 10^{865}$.

Theorem 4.44 (BB ist nicht berechenbar)

BB wächst zu stark um berechenbar zu sein: Es gibt keine DTM, die BB berechnet.

Beweis (erster Teil)

Man kann immer mindestens ein | mehr erzeugen, wenn man einen weiteren Zustand zur Verfügung hat:

- man benennt den Haltezustand um in q_{neu} und geht in den richtigen Haltezustand h nur, wenn man in q_{neu} ein Blank # gelesen hat. Zusätzlich ersetzt man das Blank durch |).
- Wenn man ein | liest, geht man nach rechts und bleibt in q_{neu}.

Damit haben wir bewiesen:

BB wächst streng monoton.

Beweis (zweiter Teil)

Angenommen, es gäbe eine DTM BB, die BB berechnet. Sie habe n_0 Zustände.
Wir betrachten die zusammengesetzten Maschinen

$$\mathrm{Comp_m} :> \mathrm{Print_m}\,\mathrm{BB}\,,$$

d.h. zuerst m Einsen auf das leere Band mit $\mathrm{Print_m}$ (aus Beispiel 4.6) und dann BB. Diese Maschinen haben $n_0 + \lfloor \frac{m}{2} \rfloor + 10$ Zustände. Nun wählen wir m_0 so, daß gilt

$$m_0 > n_0 + \lfloor \frac{m_0}{2} \rfloor + 10.$$

Dann schreibt $\mathrm{Comp_m}$ für alle $m \geq m_0$ jeweils $BB(m)$ Einsen auf das Band und arbeitet mit streng weniger als m Zuständen: Widerspruch. □

Einige Probleme für DTMn, die unentscheidbar sind. Besonders berühmt: Das allgemeine und das spezielle Halteproblem.

Definition 4.45 (Probleme über DTMn als Sprachen)

DTMn werden als Gödelzahlen kodiert:

- DTMn sind Gödelwörter über dem Alphabet $\{Z, S, \rho, \lambda, a, |, 0\}$.
- Wir kodieren Buchstaben der Gödelwörter in Ziffern um Gödelnummern zu bekommen.
- Wir schreiben $\hat{g}(\mathrm{M})$ für die Gödelnummer der DTM M.

Wie macht man aus einem Wort eine Gödelnummer?
Das haben wir auf den Folien 486 ff. beschrieben.

In den folgenden Beweisen brauchen wir DTMn, die Zahlen in Unärdarstellung als Eingabe nehmen.
Deshalb soll ihr Alphabet Σ im Folgenden mindestens zwei Elemente enthalten: das Blank $\#$ und den Strich $|$.

Definition 4.46 (Jede Zahl ist Gödelnummer)

Jede natürliche Zahl n soll Gödelnummer einer DTM M_n sein. Wir definieren

$$M_n := \begin{cases} N, & \text{falls } \hat{g}(N) = n, \\ > \#, & \text{falls es kein N gibt mit } \hat{g}(N) = n. \end{cases}$$

Die Maschine "$> \#$" druckt nur ein # und hält.
Man kann von einer Zahl n leicht entscheiden, ob $n = \hat{g}(N)$ ist für irgendein N!

Im Folgenden benutzen wir die Aussage **„die DTM mit Gödelnummer n“** synonym mit **„M_n“**.

Definition 4.47 (Allgemeines Halteproblem)

Das allgemeine Halteproblem ist die Frage, ob die DTM mit Gödelnummer n bei Eingabe i hält. Es entspricht der Sprache

$$\mathcal{H}_{allg} := \{\langle n, i\rangle \mid \mathrm{M}_n \text{ hält bei Eingabe } i\}.$$

Definition 4.48 (Spezielles Halteproblem)

Das spezielle Halteproblem ist die Frage, ob die DTM mit Gödelnummer n bei Eingabe n hält. Es entspricht der Sprache

$$\mathcal{H} := \{n \mid \mathrm{M}_n \text{ hält bei Eingabe } n\}.$$

Definition 4.49 (Null-Halteproblem)

Das Null-Halteproblem ist die Frage, ob die DTM mit Gödelnummer n bei leerer Eingabe hält. Es entspricht der Sprache

$$\mathcal{H}_0 := \{n \mid \mathrm{M}_n \text{ hält bei leerer Eingabe}\}$$

Manchmal sagt man auch „bei Eingabe 0“ anstatt „bei leerer Eingabe“. Dies deshalb, weil die 0 (unäre Schreibweise für natürliche Zahlen) als leeres Wort dargestellt wird.

Definition 4.50 (Leerheitsproblem)

Das Leerheitsproblem ist die Frage, ob die DTM mit Gödelnummer n bei keiner Eingabe aus Σ^* hält. Es entspricht der Sprache

$$\mathcal{E} := \{n \mid \mathrm{M}_n \text{ hält bei keiner Eingabe aus } \Sigma^*\}.$$

Definition 4.51 (Totalitätsproblem)

Das Totalitätsproblem ist die Frage, ob die DTM mit Gödelnummer n bei jeder Eingabe aus Σ^* hält. Es entspricht der Sprache

$$\mathcal{T} := \{n \mid \mathrm{M}_n \text{ hält bei jeder Eingabe aus } \Sigma^*\}.$$

Definition 4.52 (Gleichheitsproblem)

Das Gleichheitsproblem ist die Frage, ob die DTM mit Gödelnummer n die gleiche Sprache über Σ akzeptiert wie die DTM mit Gödelnummer m. Es entspricht der Sprache

$$\mathcal{E}q := \{\langle n, m\rangle \mid \mathrm{M}_n \text{ akzeptiert die gleiche Sprache über } \Sigma \text{ wie } \mathrm{M}_m \}.$$

Definition 4.53 (Entscheidungsproblem)

Das Entscheidungsproblem ist die Frage, ob die DTM mit Gödelnummer n eine entscheidbare Sprache über Σ akzeptiert. Es entspricht der Sprache

$$\mathcal{E}nt := \{n \mid \mathrm{M}_n \text{ akzeptiert eine entscheidbare Sprache über } \Sigma\}.$$

Satz 4.54 (Spezielles Halteproblem ist unentscheidbar)

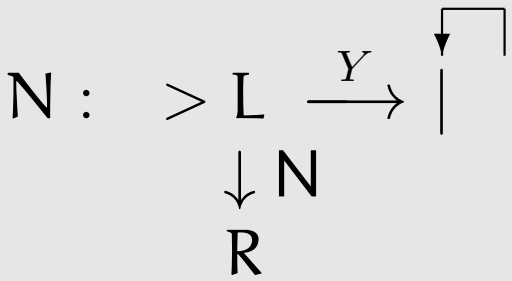
Das spezielle Halteproblem $\mathcal{H}$ ist unentscheidbar.

Beweis

Wir beweisen den Satz durch Widerspruch, und zwar mit einem Diagonalisierungsargument.
Angenommen, die DTM H entscheidet $\mathcal{H}$.
Das heißt, **H hält bei Eingabe einer Zahl n auf jeden Fall**, und zwar in einer Haltekonfiguration

$$\begin{array}{ll} h, \#Y\underline{\#} & \text{falls } n \in \mathcal{H}, \text{ bzw.} \\ h, \#N\underline{\#} & \text{falls } n \notin \mathcal{H} \end{array}$$

Sei N die folgende DTM (hält nicht bei Input Y):



Beweis (Fortsetzung)

Sei $\mathrm{M} := \mathrm{HN}$. Sie arbeitet so:

- Sie erwartet als Eingabe eine Zahl n.
- Auf diesem n arbeitet zunächst H; sie entscheidet, ob die DTM mit Gödelnummer n bei Eingabe n hält.
- Danach steht entweder „Y“ oder „N“ auf dem Band.
- Jetzt arbeitet N:
  - Falls H mit „Y“ geantwortet hat, so schreibt N endlos dasselbe Zeichen.
  - Falls H mit „N“ geantwortet hat, hält N und damit M.

Sei t die Gödelnummer von M.

Beweis (Fortsetzung)

Wie reagiert M bei Eingabe ihrer Gödelnummer?

- Wenn $t \in \mathcal{H}$ ist:
  - H liefert bei Input t das Ergebnis $\#Y\underline{\#}$.
  - Daraufhin druckt N endlos dasselbe Zeichen.
  - **M hält also nicht bei Eingabe t.**
  - Das heißt, $t \notin \mathcal{H}$ nach der Definition von $\mathcal{H}$ – ein Widerspruch.
- Wenn $t \notin \mathcal{H}$ ist:
  - H liefert bei Input t das Ergebnis $\#N\underline{\#}$.
  - N hält.
  - **M hält also bei Eingabe t.**
  - Das heißt, $t \in \mathcal{H}$ – ein Widerspruch.

Die Annahme, $\mathcal{H}$ sei entscheidbar, führt also zu einem Widerspruch. □

Satz 4.55 (Akzeptierbarkeit von $\mathcal{H}$)

Das spezielle Halteproblem $\mathcal{H}$ ist aufzählbar. D.h. die Sprache

$$\mathcal{H} = \{n \mid \mathrm{M}_n \text{ hält bei Eingabe } n\}$$

ist akzeptierbar.

Beweis

Man kann es auf jeden Fall in endlicher Zeit feststellen, wenn die DTM mit Gödelnummer n bei Eingabe n hält:
Man muss nur ihre Arbeit simulieren.
Das tut zum Beispiel eine universelle DTM.
Man kann aber auch wie folgt vorgehen:

1. Man lässt M_1 auf Eingabe 1 **einen Schritt** laufen.
2. Man lässt M_1, M_2 **zwei Schritte** laufen (auf die entsprechenden Eingaben).
3. Im n-ten Durchgang lässt man die ersten n Maschinen jeweils n Schritte laufen
4. Nur wenn eine der Maschinen hält, geht man in den Blinkzustand. Dann arbeitet man weiter.

□

Satz 4.56 ($\overline{\mathcal{H}}$ ist nicht aufzählbar)

Die folgende Menge ist nicht akzeptierbar *(d.h. nicht (rekursiv) aufzählbar).*

$$\overline{\mathcal{H}} = \{n \mid \mathrm{M}_n \textit{ hält } \textbf{nicht} \textit{ bei Eingabe } n\}.$$

Beweis.

Angenommen, $\overline{\mathcal{H}}$ ist akzeptierbar.
$\mathcal{H}$ ist akzeptierbar nach Satz 4.55.
Wenn sowohl ein Problem als auch sein Komplement akzeptierbar sind, so ist nach Satz 4.38 das Problem entscheidbar.
Das steht im Widerspruch zu Satz 4.54, nach dem $\mathcal{H}$ unentscheidbar ist. □

Wie zeigt man, daß ein Problem unentscheidbar ist?

Klassische Reduktion:

- Gegeben ein neues Problem, das gelöst werden soll.
- Man sucht sich ein altes Problem,
- und baut das neue Problem so um, daß das alte Problem ein Spezialfall des neuen Problems ist.

Nichts anderes tun Sie, wenn Sie ein neues Problem mit Standard-Algorithmen und Standard-Datenstrukturen lösen!

Hier: Wir verwenden ein altes Problem, von dem wir wissen, daß es unentscheidbar ist: etwa $\mathcal{H}$.

Gelingt die Reduktion des „alten" Problems auf das „neue" (etwa $\mathcal{H}_0$), so heißt diess, daß das „neue" Problem auch unentscheidbar ist.

Wir müssen eine totale, berechenbare Funktion f angeben, die

- eine Instanz p_1 von P_1
- in eine Instanz p_2 von P_2 umwandelt,
- und zwar so, daß die Antwort zu p_1 „ja“ lautet gdw die Antwort zu p_2 „ja“ ist.

Definition 4.57 (Reduktion)

Seien L_1, L_2 Sprachen über $\mathbb{N}$.
Wir sagen, L_1 wird auf L_2 reduziert, $L_1 \preceq L_2$,
gdw
es gibt eine TM-berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$,
so daß gilt:

$$\forall n \in \mathbb{N} \quad \big(n \in L_1 \text{ gdw } f(n) \in L_2\big).$$

Lemma 4.58

Ist $L_1 \preceq L_2$, und ist L_1 unentscheidbar, so ist auch L_2 unentscheidbar.

Beweis.

- Angenommen, L_2 ist entscheidbar.
- Sei M_2 eine Turing-Maschine, die L_2 entscheidet.
- Wegen $L_1 \preceq L_2$ gibt es eine Funktion $f : \mathbb{N} \to \mathbb{N} \in TM$ mit $n \in L_1$ gdw $f(n) \in L_2$.
- Sei M_f eine DTM, die f berechnet.
- Dann kann man daraus die Maschine $\mathrm{M}_1 := \mathrm{M}_f\mathrm{M}_2$ konstruieren, für die gilt:
  - M_1, gestartet mit Input n, hält mit $h, \#Y\underline{\#}$, falls $f(n) \in L_2$, d.h. wenn $n \in L_1$ ist.
  - M_1, gestartet mit Input n, hält mit $h, \#N\underline{\#}$, falls $f(n) \notin L_2$, d.h. wenn $n \notin L_1$ ist.
- Die Maschine M_1 entscheidet also L_1, ein Widerspruch.

Satz 4.59 (Unentscheidbarkeit von $\mathcal{H}_0$)

Das Null- Halteproblem

$$\mathcal{H}_0 = \{n \mid \mathrm{M}_n \text{ hält bei Eingabe } 0\}$$

ist unentscheidbar.

Beweis

Wir reduzieren $\mathcal{H}$ auf $\mathcal{H}_0$:

Wir konstruieren eine TM-berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}$, so dass

$$i \in \mathcal{H} \text{ gdw } f(i) \in \mathcal{H}_0.$$

Gegeben eine Zahl i, es soll $f(i) = j$ sein für die Gödelnummer j einer DTM, so daß gilt:

M_i hält bei Eingabe i gdw
M_j hält bei leerer Eingabe.

Beweis (Fortsetzung)

Wie erreichen wir das?

Wir betrachten die folgende Maschine

$$M_j := \mathbf{Print}_i M_i$$

Beweis (Fortsetzung)

Also:

- M_j mit leerem Band gestartet
- $\mathtt{Print}_i$ erzeugt die Zahl i auf dem Band.
- M_i wird gestartet mit Eingabe i und hält gdw $i \in \mathcal{H}$ ist.

Wir definieren: Die Funktion f soll dem Wert $i \in \mathbb{N}$ die Gödelnummer j der DTM M_j zuweisen.

Beweis (Fortsetzung)

Dann gilt:

1. f ist berechenbar: Der einzig variable Teil ist Print_i, und deren Beitrag zur Gödelnummer ist leicht zu berechnen.
2. $f(i) \in \mathcal{H}_0$ genau dann, wenn $i \in \mathcal{H}$:

$$
\begin{aligned}
& f(i) = j \in \mathcal{H}_0 \\
\iff\ & \mathrm{M}_j = \mathrm{Print}_i \mathrm{M}_i \text{ hält bei Input } 0\ (\#\underline{\#}) \\
\iff\ & \mathrm{M}_i \text{ hält bei Input } i\ (\#|^i\underline{\#}) \\
\iff\ & i \in \mathcal{H}
\end{aligned}
$$

Damit ist $\mathcal{H}$ auf $\mathcal{H}_0$ reduziert. □

Hier noch einmal eine nicht so formale Darstellung. Um $\mathcal{H}$ auf $\mathcal{H}_0$ zu reduzieren, muss man **zu jeder Maschine M_i eine Maschine M_j so konstruieren**, dass

$$(1) \quad M_i \text{ hält bei Eingabe } i$$

$$\underline{\text{gdw}}$$

$$(2) \quad M_j \text{ hält bei Eingabe } 0$$

M_j tut folgendes: zunächst wird i auf das Band geschrieben, dann wird M_i gestartet ($M_j := > \text{Print}_i\ M_i$).

(1) ist äquivalent zu $i \in \mathcal{H}$ und

(2) ist äquivalent zu $j \in \mathcal{H}_0$

Ähnlich kann man die Unentscheidbarkeit der folgenden Probleme per Reduktion zeigen:

- $\mathcal{E}$, das Leerheitsproblem.
- $\mathcal{T}$, das Totalitätsproblem.
- $\mathcal{E}q$, das Gleichheitsproblem.
- $\mathcal{E}nt$, das Entscheidbarkeitsproblem.

Wir benutzen auch die Schreibweisen

- $\mathrm{M}_i(w)$: DTM M_i angesetzt auf w,
- $\mathrm{M}_i(w)\downarrow$: die DTM M_i angesetzt auf w hält,
- $\mathrm{M}_i(w)\uparrow$: die DTM M_i angesetzt auf w hält nicht.

Reduktion von $\mathcal{H}$ auf $\mathcal{T}$: $\mathcal{H} \preceq \mathcal{T}$

Zu einer Maschine M_i konstruieren wir M_j:

1. zunächst wird der Input gelöscht, dann i auf das Band geschrieben,
2. schließlich wird M_i gestartet.

Dann ist $i \in \mathcal{H}$ (also $M_i(i)$ hält) gdw $j \in \mathcal{T}$ (also M_j hält immer).

Man kann dies auch so ausdrücken: Es gibt kein Programm zur Erkennung von unendlichen Schleifen.

Reduktion von $\mathcal{H}$ auf $\mathcal{E}nt$: $\mathcal{H} \preceq \mathcal{E}nt$

Zunächst wählen wir eine DTM M, die eine nicht-entscheidbare Sprache akzeptiert, etwa die, die $\mathcal{H}$ akzeptiert.

Zu einer Maschine M_i konstruieren wir M_j:

1. wir lassen M auf dem gegebenen Input laufen, **und gleichzeitig (abwechselnd mit M),** $\mathrm{M}_i(i)$.
2. Falls $\mathrm{M}_i(i)$ hält, lassen wir M_j ebenfalls halten.
3. Sonst hält M_j genau dann, wenn auch M hält.

Dann ist $i \in \mathcal{H}$ (also $\mathrm{M}_i(i)$ hält) gdw $j \in \mathcal{E}nt$ (also M_j akzeptiert eine entscheidbare Sprache, nämlich Σ^*).

Reduktion von $\mathcal{H}$ auf $\overline{\mathcal{E}}$: $\mathcal{H} \preceq \overline{\mathcal{E}}$

Zu einer Maschine M_i konstruieren wir M_j:

1. zunächst wird der Input gelöscht, dann i auf das Band geschrieben,
2. schließlich wird M_i gestartet.

Dann ist $i \in \mathcal{H}$ (also $M_i(i)$ hält) gdw $j \in \overline{\mathcal{E}}$ (also M_j hält auf mindestens einem Input).

Damit ist $\overline{\mathcal{E}}$ unentscheidbar.

Da $\overline{\mathcal{E}}$ aber offensichtlich aufzählbar ist, kann $\mathcal{E}$ nicht aufzählbar sein, also erst recht nicht entscheidbar.

Reduktion von $\mathcal{T}$ auf $\mathcal{E}q$: $\mathcal{T} \preceq \mathcal{E}q$

Zu M_i konstruieren wir zwei Maschinen M_j, M_k:

1. $\mathrm{M}_j(w)$ startet $\mathrm{M}_i(w)$ und gibt Y aus, wenn $\mathrm{M}_i(w)$ anhält,
2. $\mathrm{M}_k(w)$ druckt Y und hält.

Dann ist $i \in \mathcal{T}$ gdw $\langle j, k\rangle \in \mathcal{E}q$.

Man kann dies auch so ausdrücken: In einem Compiler kann es kein optimales Verfahren zur Codeoptimierung geben.

Entscheidbar oder unentscheidbar?

Wir betrachten die folgenden Probleme

$$Move_1 := \{ n : \textit{ es gibt } w\textit{, so daß } \mathrm{M}_n(w) \textit{ mindestens einmal nach rechts geht} \}$$

$$Move_2 := \{ n : \textit{ es gibt } w\textit{, so daß } \mathrm{M}_n(w) \textit{ ein Feld rechts des } \textbf{Anfangsfeldes} \textit{ besucht} \}$$

Dabei ist in $Move_2$ mit **Anfangsfeld** dasjenige Feld gemeint, von dem die DTM startet (also das erste Blank rechts vom Input w).

5. P versus NP

Motivation

In diesem Abschnitt definieren wir die berühmten Klassen P und NP.

Wir illustrieren und motivieren außerdem:

1. Den Begriff der Reduktion bezüglich der Komplexität – ein Problem (eine Sprache) wird auf ein zweites reduziert. Das erste Problem ist dann höchstens so kompliziert wie das zweite.
2. Den Begriff eines NP-schweren Problems: es ist mindestens so schwer wie **alle Probleme aus NP**.
3. Den Begriff NP-vollständig: das sind die schwersten Probleme in NP.
4. Wir definieren die wichtigsten Komplexitätsklassen und betrachten ihre Struktur.

5.1 Die Struktur von PSPACE

1. Sortieralgorithmen: Bubble Sort, Quicksort, ...
2. Erfüllbarkeitsproblem: siehe Folie 54. **Gibt es eine erfüllende Belegung für die Variablen $\{x_1, x_2, \ldots x_n\}$?**
3. Graphenprobleme: Gibt es einen hamiltonschen Kreis in einem Graphen? Sind zwei Knoten in einem Graphen voneinander erreichbar?

Welche Arten von Komplexität gibt es?

- Zeit,
- Speicher.

Definition 5.1 ($\mathbf{NTIME}(T(n))$, $\mathbf{DTIME}(T(n))$)

Basismodell: k-DTM M (ein Band für die Eingabe).
Wenn M mit jedem Eingabewort der Länge n höchstens $T(n)$ Schritte macht, dann wird sie $T(n)$-zeitbeschränkt genannt.
Die von M akzeptierte Sprache hat Zeitkomplexität $T(n)$ (tatsächlich meinen wir $\max(n+1, \lceil T(n) \rceil)$).

- $\mathbf{DTIME}(T(n))$ ist die Klasse der Sprachen, die von $T(n)$-zeitbeschränkten, DTMn akzeptiert werden.
- $\mathbf{NTIME}(T(n))$ ist die Klasse der Sprachen, die von $T(n)$-zeitbeschränkten, NTMn akzeptiert werden.

Wir zählen nur die Kopfbewegungen, nicht das Schreiben.

Definition 5.2 ($\mathbf{NSPACE}(S(n))$, $\mathbf{DSPACE}(S(n))$)

Basismodell: k-DTM M davon ein spezielles Eingabeband (**offline DTM**).
Wenn M für jedes Eingabewort der Länge n maximal $S(n)$ Zellen auf den Ablagebändern benutzt, dann heißt M $S(n)$-speicherbeschränkt.
Die von M akzeptierte Sprache hat Speicherkomplexität $S(n)$ (tatsächlich meinen wir $\max(1, \lceil S(n) \rceil)$)

- $\mathbf{DSPACE}(S(n))$ ist die Klasse der Sprachen, die von $S(n)$-speicherbeschränkten, DTMn akzeptiert werden.
- $\mathbf{NSPACE}(S(n))$ ist die Klasse der Sprachen, die von $S(n)$-speicherbeschränkten, NTMn akzeptiert werden.

Band 1 dient nur zum Lesen der Eingabe. Alle anderen sind Arbeitsbänder.

Frage:

Wieso eine *offline*-Turing-Maschine?
Bandbeschränkung von weniger als linearem Wachstum.

Frage:

Zu welcher Zeit-/Speicherkomplexitätsklasse gehört

$$\mathcal{L}_{\text{mirror}} := \{wcw^R : \ w \in (0+1)^*\},$$

also die Menge aller Wörter die um den mittleren Buchstaben c gespiegelt werden können?

Zeit: $\mathbf{DTIME}(2n+1)$. Die Eingabe wird einfach rechts vom c in umgekehrter Reihenfolge kopiert. Wenn das c gefunden wird, wird einfach der übrige Teil (das w) mit der Kopie von w auf dem Band verglichen.

Speicher: Die gerade beschriebene Maschine liefert eine Schranke von $\mathbf{DSPACE}(\frac{n-1}{2})$.

Geht es noch besser?

Ja: $\mathbf{DSPACE}(\lg n)$. Wir benutzen zwei Bänder als Binär-Zähler. Die Eingabe auf das Auftreten von genau einem c benötigt keinen Speicher (kann mit einigen Zuständen getan werden). Als zweites prüfen wir Zeichen für Zeichen auf der linken und auf der rechten Seite: dazu müssen die zu prüfenden Positionen gespeichert werden (sie werden auf den beiden Bändern kodiert).
Man kommt auch mit einem einzigen Zähler aus (und einem Band).

Wichtige Fragen (1):

Zeit: Wird jede Sprache aus $\mathbf{DTIME}(f(n))$ von einer DTM entschieden?

Speicher: Wird jede Sprache aus $\mathbf{DSPACE}(f(n))$ von einer DTM entschieden?

Die Funktionen f sind immer sehr einfache Funktionen, insbesondere sind sie alle berechenbar. In dieser Vorlesung betrachten wir nur Potenzen $f(n) = n^i$, $f(n) = \lg n$ oder sehr einfache Varianten davon.

Wichtige Fragen (2):

Zeit/Speicher: Wie sieht es mit $\mathbf{NTIME}(f(n))$, $\mathbf{NSPACE}(f(n))$ aus?

Zeit vs. Speicher: Welche Beziehungen gelten zwischen $\mathbf{DTIME}(f(n))$, $\mathbf{DSPACE}(f(n))$, $\mathbf{NTIME}(f(n))$, $\mathbf{NSPACE}(f(n))$?

Halten, hängen nach n Schritten.

Zeitbeschränkt: Was bedeutet es, daß eine DTM höchstens n Schritte macht?

Halten?: Streng genommen, müßte sie dann entweder halten oder hängen. Das bedeutet, im ersten Fall, daß die Eingabe akzeptiert wird.

Hängen?: DTM'n auf beidseitig unendlichem Band können aber nicht hängen.

Stoppen nach n Schritten.

Stoppen: Wir wollen unter eine DTM macht höchstens n Schritte folgendes verstehen:

- Sie hält (und akzeptiert die Eingabe) innerhalb von n Schritten.
- Sie hängt (und akzeptiert die Eingabe nicht) innerhalb von n Schritten.
- Sie stoppt innerhalb von n Schritten ohne in den Haltezustand überzugehen. Auch hier wird die Eingabe nicht akzeptiert.

Antworten (informell):

Zeit: **Jede Sprache aus $\mathrm{DTIME}(f(n))$ ist entscheidbar**: Man warte einfach solange wie $f(n)$ angibt. Falls bis dahin kein „Ja“ gekommen ist, ist die Antwort eben „Nein“.

Antworten (informell):

Speicher: **Jede Sprache aus $\mathrm{DSPACE}(f(n))$ ist entscheidbar.** Denn es gibt nur endlich viele verschiedene Konfigurationen. Falls also die DTM nicht terminiert (was passiert), macht man folgendes: man schreibt alle Konfigurationen auf (die komplette Rechnung). Falls die DTM nicht terminiert, muss sie in eine Schleife laufen (eine Konfiguration erreichen, die sie schon einmal hatte). Das lässt sich feststellen.

Antworten (informell):

NTM vs. DTM: Trivial sind

$$\mathbf{DTIME}(f(n)) \subseteq \mathbf{NTIME}(f(n)) \text{ und}$$
$$\mathbf{DSPACE}(f(n)) \subseteq \mathbf{NSPACE}(f(n)).$$

Versucht man aber eine NTM durch eine DTM zu simulieren, braucht man (vermutlich) exponentiell mehr Zeit.

Für die Speicherkomplexität kann man zeigen (Theorem von Savitch):

$$\mathbf{NSPACE}(f(n)) \subseteq \mathbf{DSPACE}(f^2(n)).$$

Antworten (informell):

Zeit vs. Speicher: Trivial sind

$\mathbf{DTIME}(f(n)) \subseteq \mathbf{DSPACE}(f(n))$ und
$\mathbf{NTIME}(f(n)) \subseteq \mathbf{NSPACE}(f(n))$.

Aber $\mathbf{DSPACE}(f(n))$, $\mathbf{NSPACE}(f(n))$ sind viel größer.

Nur **die funktionale Wachstumsrate einer Funktion in Komplexitätsklassen zählt**: Konstante Faktoren werden ignoriert.

Theorem 5.3 (Bandkompression)

Für jedes $c \in \mathbb{R}^+$ und jede Speicherfunktion $S(n)$ gilt:

$$\mathbf{DSPACE}(S(n)) = \mathbf{DSPACE}(cS(n))$$
$$\mathbf{NSPACE}(S(n)) = \mathbf{NSPACE}(cS(n))$$

Beweis

Eine Richtung ist trivial. Die andere beweist man, indem man eine feste Anzahl r ($> \frac{2}{c}$) von **benachbarten Zellen** auf dem Band als **ein neues Symbol** darstellt. **Die Zustände der neuen Maschine simulieren die alten Kopfbewegungen als Zustandsübergänge** (innerhalb des neuen Symbols). D.h. für r Zellen der alten Maschine werden nun nur maximal 2 benutzt: im ungünstigsten Fall, wenn man von einem Block in den benachbarten geht.

Theorem 5.4 (Zeitbeschleunigung)

Für jedes $c \in \mathbb{R}^+$ und jede Zeitfunktion $T(n)$ mit $\lim_{n\to\infty} \frac{T(n)}{n} = \infty$ gilt:

$$\mathbf{DTIME}(T(n)) = \mathbf{DTIME}(cT(n))$$
$$\mathbf{NTIME}(T(n)) = \mathbf{NTIME}(cT(n))$$

Beweis

Eine Richtung ist trivial. Der Beweis der anderen läuft wieder über die Repräsentation einer festen Anzahl r ($> \frac{2}{c}$) benachbarter Bandzellen durch neue Symbole. Im Zustand der neuen Maschine wird wieder kodiert, welches Zeichen und welche Kopfposition (der simulierten, alten Maschine) aktuell ist.

Wenn die alte Maschine simuliert wird, muss die neue nur 4 statt r Schritte machen: 2 um die neuen Felder zu beschreiben und weitere 2 um den Kopf auf das neue und wieder zurück auf das alte (im schlimmsten Fall) zu bewegen. Da wir das schreiben der Felder nicht zählen, sind es nur 2 Kopfbewegungen.

Lemma 5.5 (Wachstumsrate von DTIME, DSPACE)

Es sei $T(n)$ eine Zeitfunktion mit $\lim_{n\to\infty} \frac{T(n)}{n} = \infty$, und es sei $S(n)$ eine Speicherfunktion.

(a) *Es sei $f(n) = \mathbf{O}(T(n))$. Dann gilt:*
$\mathbf{DTIME}(f(n)) \subsetneqq \mathbf{DTIME}(T(n))$.

(b) *Es sei $g(n) = \mathbf{O}(S(n))$. Dann gilt:*
$\mathbf{DSPACE}(g(n)) \subsetneqq \mathbf{DSPACE}(S(n))$.

(In einer Übungsaufgabe sollen Sie einen Beweis dafür skizzieren.)

Definition 5.6 (P, NP, PSPACE)

$$\begin{aligned} \mathbf{P} &:= \bigcup_{i \geqslant 1} \mathbf{DTIME}(n^i) \\ \mathbf{NP} &:= \bigcup_{i \geqslant 1} \mathbf{NTIME}(n^i) \\ \mathbf{PSPACE} &:= \bigcup_{i \geqslant 1} \mathbf{DSPACE}(n^i) \end{aligned}$$

Intuitiv:

- Probleme in $\mathbf{P}$ sind **effizient lösbar**, jene aus $\mathbf{NP}$ können (nur?) in exponentieller Zeit gelöst werden.
- $\mathbf{PSPACE}$ ist eine sehr große Klasse, weit größer als $\mathbf{P}$ oder $\mathbf{NP}$.

Etwas genauer: Man betrachte ein Sudoku Puzzle

2		3		9				6
	1							4
	9				5			
		7						
6			9				7	
8	4							
9	3		1		7	5		
7		2			9	4		
1			2				6	

Abbildung 27: Ein sehr schweres Sudoku

- **Wie schwer ist es, eine Belegung zu finden, die das Puzzle löst?**
- Wie schwer ist es, eine gegebene Belegung als richtig/falsch nachzuweisen?

Noch genauer: Man betrachte die Formel ϕ

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

und betrachte das **Erfüllbarkeitsproblem**:

Gibt es eine Belegung von x_1, x_2, x_3 mit **wahr (1), falsch (0)** *so, daß die obige Formel wahr wird?*

- **Wie findet man eine Belegung?** Alle ausprobieren! Oder weiß jemand etwas besseres? **Welche Komplexität ergibt dies?**
- Bei n Variablen höchstens 2^n mal die ganze Formel überprüfen. Also exponentiell im Input?

Verallgemeinertes Sudoku

Man zeigt leicht (siehe meine *Logik und Verifikation* Vorlesung), daß das finden einer erfüllenden Belegung einer Formel äquivalent zum lösen eines Sudoku ist.

- Klassisches Sudoku wird auf einem 9×9 Raster gespielt. Um sinnvolle Komplexitätsaussagen zu machen, muss es verallgemeinert werden.
- Eine **feste** Formel für das Erfüllbarkeitsproblem zu betrachten, ist ja auch nicht sinnvoll.
- Verallgemeinertes Sudoku der Ordnung n wird auf einem Raster der Größe $n^2 \times n^2$ gespielt. Es besteht aus einzelnen $n \times n$ Rastern, die mit Zahlen $1, 2, \ldots, n$ gefüllt werden müssen
- Die Regeln ergeben sich direkt aus der klassischen Variante, diese hat Ordnung 3.

Theorem 5.7 (Charakterisierung von NP)

Eine Sprache L is in $\mathbf{NP}$ genau dann wenn es eine Sprache L' in $\mathbf{P}$ und ein $k \geq 0$ gibt, so das für alle $w \in \Sigma$ gilt:

$$w \in L \text{ gdw. es gibt ein } c: \langle w, c \rangle \in L' \text{ und } |c| < |w|^k.$$

c wird Zeuge (witness oder Zertifikat/certificate) von w in L genannt. Eine DTM, die die Sprache L' akzeptiert, wird Prüfer (verifier) von L genannt.

Wichtig:

Ein Entscheidungsproblem ist in $\mathbf{NP}$ genau dann wenn jede Ja-Instanz ein kurzes Zertifikat hat (d.h. seine Länge polynomial in der Länge der Eingabe ist), welche in polynomial-Zeit verifiziert werden kann.

Wir haben die Komplexitätsklassen nur für Sprachen definiert. Man könnte das aber auch direkt für Funktionen f tun:

Komplexitätsklassen für Funktionen

Eine Funktion $f : \mathbb{N} \to \mathbb{N}$ ist in $\mathbf{P}$, falls es eine DTM M und ein Polynom $p(n)$ gibt, so daß für jedes n der Funktionswert $f(n)$ in höchstens $p(\mathit{länge}(n))$ Schritten von M berechnet wird. Dabei gilt $\mathit{länge}(n) = \lg n$, denn man braucht $\lg n$ Zeichen um die Zahl n binär darzustellen.

Analog funktioniert dies für alle anderen Komplexitätsklassen.

Frage:

Was sind die genauen Beziehungen zwischen den Komplexitätsklassen aus Definition 5.6?

Offensichtlich:

$$\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$$

und mehr weiß man bis heute nicht!

Betrachte $\mathcal{L}_{\text{mirror}}$ von Folie 581.
$\mathcal{L}_{\text{mirror}} \in \mathbf{DTIME}(n)$, $\mathcal{L}_{\text{mirror}} \in \mathbf{DSPACE}(\lg n)$.

Intuitiv sollte $\mathrm{DSPACE}(n)$ auch Sprachen enthalten, die nur in exponentieller Zeit gelöst werden können.

Frage

Wie zeigen wir, daß ein gegebenes Problem in einer bestimmten Klasse ist?

Antwort

Reduktion auf ein bekanntes!

Wir brauchen eines, mit dem wir anfangen können: SAT (Seite 55).

Denken Sie an unsere Technik zu zeigen, daß eine Sprache L nicht rekursiv ist (Folie 559): dazu mussten wir eine unentscheidbare Sprache H auf L reduzieren.

Frage

Können wir in NP Probleme finden, die die schwierigsten in NP sind?

Antwort

Es gibt mehrere Wege, ein schwerstes Problem zu definieren. Sie hängen davon ab, welchen **Begriff von Reduzierbarkeit** wir benutzen.

Für die von uns betrachteten Reduktionen ist die Antwort: Ja. Solche Probleme werden vollständig in der gegebenen Klasse bezüglich der Reduktion genannt.

Theorem 5.8 (Existenz von schwersten Problemen)

Es gibt in $\mathbf{NP}$ *Sprachen, auf die man alle anderen Sprachen aus* $\mathbf{NP}$ *reduzieren kann.*

Beweisidee

Dies liegt an der Mächtigkeit indeterminierter TMn. Wir hatten schon gesehen, daß es universelle DTMn gibt. Wir betrachten die folgende Sprache L_U die in $\mathbf{NP}$ ist (da die universelle DTM jede DTM $\mathcal{M}_n$ für t Schritte emulieren kann)

$$L_U := \{\langle w, n, t\rangle : \text{ es gibt } c \text{ so, daß } \mathcal{M}_n \text{ das Paar } \langle w, c\rangle \text{ innerhalb von } t \text{ Schritten akzeptiert}\}.$$

Jede Sprache in $\mathbf{NP}$ kann darauf reduziert werden.

Definition 5.9 (Polynomial-Zeit-Reduzibilität)

Seien L_1, L_2 Sprachen.

Polynomial-Zeit: L_2 ist polynomial-Zeit-reduzibel auf L_1 (poly-Zeit reduzibel), bezeichnet mit $L_2 \preceq_{\text{pol}} L_1$, wenn es eine polynomial-Zeit beschränkte DTM gibt, die für jede Eingabe w eine Ausgabe $f(w)$ erzeugt, so daß

$$w \in L_2 \text{ gdw } f(w) \in L_1$$

Beweis von Theorem 5.8.

Sei $L \in \mathbf{NP}$, also von der Form

$$L = \{w : \text{ es gibt } c \text{ mit } \langle w, c\rangle \in L',\ |c| \leq |w|^k\}$$

für eine Sprache $L' \in \mathbf{P}$ und $k \in \mathbb{N}$. Wir reduzieren L auf L_U: $L \preceq_{\text{pol}} L_U$ (Folie 606).

Da $L' \in \mathbf{P}$ gibt es eine DTM $\mathcal{M}_{n_0}$ und $k' \in \mathbb{N}$ so, daß "$\langle w, c\rangle \in L'$" in höchstens $(|w| + |c|)^{k'}$ Schritten von $\mathcal{M}_{n_0}$ entschieden wird. **Wie definieren wir nun unsere polynomiale Transformation f?**
Wir ordnen jedem $w \in L$ die folgende Instanz von w' aus L_U zu (warum klappt das?):

$$\langle w, n_0, (|w|^{k+1})^{k'}\rangle.$$

Lemma 5.10 (Polynomial-Zeit-Reduktionen)

1. *Sei L_2 polynomial-Zeit-reduzibel auf L_1. Dann gilt:*

   (1) L_2 ist in $\mathbf{NP}$ wenn L_1 in $\mathbf{NP}$ ist;
   (2) L_2 ist in $\mathbf{P}$ wenn L_1 in $\mathbf{P}$ ist.

2. *Die Komposition von poly-Zeit-Reduktionen ist wieder eine poly-Zeit-Reduktion.*

Problemstellung:

Man erinnere sich an das Problem L_{ILP} (ganzahlige lineare Programmierung) auf Folie 62. In welcher Beziehung steht es zu L_{sat}?

Die nächste Folie zeigt eine polynomial-Zeit-Reduktion von L_{sat} auf L_{ILP}. Die Umkehrung gilt auch, ist aber komplizierter.

Gegeben also eine Formel $\phi : \psi_1 \wedge \cdots \wedge \psi_l$, wobei jedes ψ_i eine Disjunktion von Literalen (über den Variablen $x_1, \ldots, x_n$) ist.

Wir konstruieren ein ILP-Problem $\langle \mathbf{A}, \mathbf{b} \rangle$ wie folgt:

Matrix $\mathbf{A}$: $2n$ Spalten von $\mathbf{A}$ entsprechen $x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}$. Die ersten n Zeilen enthalten jeweils genau zwei aufeinander folgende 1 (beginnend an der Position i in der Zeile i. Die nächsten n Zeilen enthalten jeweils genau zwei aufeinander folgende -1 an genau den gleichen Positionen. Die nächsten l Zeilen enthalten 0 und 1 an den Positionen, die durch die Klauseln ψ_i ($i = 1, \ldots, l$) bestimmt sind. Schließlich haben wir n Zeilen, die nur eine 1 (auf der Position x_i) enthalten, und weitere n Zeilen, die nur eine 1 (auf der Position $\overline{x_i}$) enthalten.

Vektor $\mathbf{b}$: Die ersten n Einträge sind 1, die nächsten n sind -1, die nächsten l (Anzahl Klauseln) sind 1. Schließlich haben wir $2n$ Einträge, die 0 sind.

Ergebnis:

$$\phi \in L_{\text{sat}} \text{ gdw. } \langle \mathbf{A}, \mathbf{b} \rangle \in L_{\text{ILP}}$$

Warum? Man splitte die Matrix-Multiplikation in ihre vielen Ungleichungen auf und finde heraus, was sie wirklich bedeuten.

Vorsicht: Wir haben nur eine Reduktion gezeigt. Was müssen wir noch nachweisen, um zu sehen, daß tatsächlich $L_{\text{ILP}} \in \mathbf{NP}$ (man beachte Theorem 5.7)?

5.2 Vollständige und schwere Probleme

Definition 5.11 (Vollständig, Schwer (Hart))

- Eine Sprache L heißt **NP-vollständig** wenn $L \in \mathbf{NP}$ gilt und jede Sprache $L' \in \mathbf{NP}$ poly-Zeit-reduzibel auf L ist.
- Eine Sprache L heißt **NP-schwer (NP-hart)** wenn jede Sprache $L' \in \mathbf{NP}$ poly-Zeit-reduzibel auf L ist.
- Eine Sprache L heißt **PSPACE-vollständig** wenn $L \in \mathbf{PSPACE}$ gilt und jede Sprache $L' \in \mathbf{PSPACE}$ poly-Zeit-reduzibel auf L ist.

Bemerkenswert:

- Falls ein $\mathbf{NP}$-schweres Problem in $\mathbf{P}$ liegt, dann ist $\mathbf{P} = \mathbf{NP}$.
- Falls ein $\mathbf{PSPACE}$-schweres Problem in $\mathbf{P}$ liegt, dann ist $\mathbf{P} = \mathbf{PSPACE}$.
- Wenn $\mathbf{P} \neq \mathbf{NP}$ gilt, dann ist kein $\mathbf{NP}$-vollständiges Problem in polynomial-Zeit lösbar.
  Eine Million Euro für den, der das $\mathbf{P} = \mathbf{NP}$ Problem löst! (Millenium Probleme)

Wie zeigt man NP-Vollständigkeit?

- Um von einer Sprache L zu zeigen, daß sie NP-schwer ist, genügt es ein anderes NP-schweres Problem finden, etwa L_{sat} und es auf L zu reduzieren:

$$L_{\text{sat}} \preceq_{\text{pol}} L$$

- Da jede Sprache aus NP auf L_{sat} reduziert werden kann, und $L_{\text{sat}} \preceq_{\text{pol}} L$, ist auch L NP-schwer.
- Wenn nun noch $L \in \mathbf{NP}$ gezeigt werden kann, dann ist L NP-vollständig.

Theorem 5.12 (NP-vollständige Probleme)

L_{sat} *ist* NP-vollständig.

Beweis.

Daß $L_{\text{sat}} \in \mathbf{NP}$ ist klar. Die Reduktion von allen $\mathbf{NP}$ Problemen darauf ist allerdings kompliziert. Dies wurde in den 70'ern von Stephen Cook und Leonid Levin unabhängig voneinander gezeigt. □

Wichtig:

Die Komplexitätsklassen, die auf **deterministischen Turing-Maschinen** definiert sind, sind **abgeschlossen unter Komplement-Bildung**: Wenn eine Sprache L dazu gehört, dann auch ihr Komplement (einfach die alte Maschine ausführen und die Ausgabe invertieren).

Frage:

Gilt das auch für $\mathbf{NP}$?

Antwort:

Keiner weiß es!

Die letzte Komplexitätsklasse, die wir einführen, ist **co-NP**.

Definition 5.13 (co-NP)

co-NP ist die Klasse der Sprachen L deren Komplemente $\overline{L}$ $(= \Sigma^* \setminus L)$ in **NP** liegen.

Prim $\in$ NP $\cap$ co-NP

Wir diskutieren die Sprache **Prim**, aller Primzahlen, binär codiert. Trivial ist, daß $\overline{\textbf{Prim}}$ in NP liegt (siehe Folie 470). Also **Prim** $\in$ co-NP.
Daß Prim in NP liegt, ist schon etwas schwieriger zu zeigen.
Tatsächlich ist Prim sogar in P (Agrawal, Saxena, Kayar (2002)).

Eine offene Vermutung ist folgende Gleichheit:

$$\mathbf{P} = \mathbf{NP} \cap \mathbf{co\text{-}NP}$$

Die folgenden Beziehungen sind momentan noch unbekannt:

1. $\mathbf{P} \neq \mathbf{NP}$,
2. $\mathbf{NP} \neq \mathbf{co\text{-}NP}$,
3. $\mathbf{P} \neq \mathbf{PSPACE}$,
4. $\mathbf{NP} \neq \mathbf{PSPACE}$.

5.3 Beispiele

Beispiel 5.14 (NP vollständige Beispiele)

- Ist ein (un-)gerichteter Graph hamiltonsch? (**Hamiltonian circle**)
- Hamilton mit Kostenfunktion g auf den Kanten: Gibt es einen Hamilton-Kreis mit Kosten kleiner gleich k? (**Hamiltonian circle with costs**)
- Ist eine logische Formel erfüllbar? (**Satisfiability**)
- Gibt es in einem Graphen eine Clique der Größe k? (**Clique of size k**)
- Ist ein Graph mit drei Farben zu färben? (**3-colorability**)
- Gibt es in einer Menge von ganzen Zahlen eine Teilmenge mit der Gesamtsumme x? (**Subset Sum**)

Definition 5.15 (Varianten von Hamilton Circle)

Hier geht es darum, ob es in einem Graphen einen Hamilton-Kreis gibt: also einen geschlossenen Weg entlang der Kanten, der jeden Knoten genau einmal besucht.

$L_{\text{Ham}_{\text{undir}}}$: Sei G ein ungerichteter Graph. $L_{\text{Ham}_{\text{undir}}}$ ist die Sprache die aus allen ungerichteten Graphen besteht, in denen es einen Hamilton-Kreis gibt.

$L_{\text{Ham}_{\text{dir}}}$: Sei G ein gerichteter Graph. $L_{\text{Ham}_{\text{dir}}}$ ist die Sprache die aus allen gerichteten Graphen besteht, in denen es einen Hamilton-Kreis gibt.

Hamilton -weg und -kreis

Man beachte, daß die Existenz eines **Hamiltonweges** (also nicht notwendigerweise **Hamiltonkreises**) von der gleichen Komplexität ist: Zu jedem Graph G betrachtet man den Graphen $G' = G \cup \{e_{neu}\}$, in dem der neue Knoten mit allen anderen verbunden ist. Dann gilt:

es gibt einen Hamilton**weg** in G
genau dann, wenn
es gibt einen Hamilton**kreis** in G'.

Definition 5.15 (Travelling Salesperson)

Die letzte Variante führt eine Kostenfunktion auf den Kanten ein.

$L_{\text{Ham}_{cost \leq k}}$: Sei $G = (V, E)$ ein vollständiger, ungerichteter Graph und g eine Kantengewichtsfunktion $g : E \to \mathbb{N}$.
Für ein $k \in \mathbb{N}$ sei $L_{\text{Ham}_{\text{cost} \leq k}}$ die Sprache, die aus allen vollständigen, ungerichteten Graphen besteht, in denen es einen Hamilton-Kreis mit Gesamtkosten kleiner gleich k gibt:

$$\sum_{i=1}^{n} g(v_i, v_{i+1}) \leq k,$$

wobei $v_1, \ldots, v_n$ ein Hamilton-Kreis in G ist.

Definition 5.16 (Maximale Clique: L_{Clique_k})

Sei $G = (V, E)$ ein ungerichteter Graph. Eine Clique in einem Graphen ist ein vollständiger Teilgraph von G.
Für ein $k \in \mathbb{N}$ sei L_{Clique_k} die Sprache, die aus allen ungerichteten Graphen besteht, die eine Clique der Größe k enthalten.

Definition 5.17 (k-colorability: $L_{\text{Color}_{\leq k}}$)

Sei $G = (V, E)$ ein ungerichteter Graph. G heißt k-färbbar, falls jeder Knoten mit einer von k Farben so gefärbt werden kann, daß benachbarte Knoten verschiedene Farben haben.
Für ein $k \in \mathbb{N}$ sei $L_{\text{Color}_{\leq k}}$ die Sprache, die aus allen ungerichteten, mit höchstens k Farben färbbaren Graphen besteht.

SAT, 3-CNF sind NP-vollständig.

Definition 5.18 (SAT, k-CNF-SAT, k-DNF-SAT)

Wir haben bereits das Erfüllbarkeits-Problem auf Folie 55 vorgestellt. Ein Literal (*literal*) l_i ist ein Variable x_i oder seine Negation $\neg x_i$. Eine Klausel (*clause*) ist eine Disjunktion von Literalen. Wir definieren wie folgt:

DNF: Eine Formel ist in **disjunktiver Normalform**, wenn sie von der Form $(l_{11} \wedge \ldots \wedge l_{1n_1}) \vee \ldots \vee (l_{m1} \wedge \ldots \wedge l_{mn_m})$ ist.

CNF: Eine Formel ist in **konjunktiver Normalform**, wenn sie von der Form $(l_{11} \vee \ldots \vee l_{1n_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mn_m})$ ist.

Definition (Fortsetzung):

k-DNF: Eine Formel ist in ***k-DNF*** wenn sie in DNF ist und jede Disjunktion höchstens k Literale hat.

k-CNF: Eine Formel ist in ***k-CNF*** wenn sie in CNF ist und jede Konjunktion höchstens k Literale hat.

Wir benutzen auch die Notation k-CNF-SAT für k-CNF: Jede Klausel hat maximal k Literale (sie können natürlich von Klausel zu Klausel variieren, so daß die Gesamtformel eine beliebige Anzahl an Literalen hat).

Theorem 5.19 (NP-vollständige Probleme)

$L_{3\text{-CNF-SAT}}$ ist NP-vollständig. Und damit natürlich auch $L_{k\text{-CNF-SAT}}$ für $k \geq 3$.

Fragen:

DNF: Was ist die Zeit-Komplexität von Sat für Formeln in DNF?

2-CNF-SAT: Was ist die Zeit-Komplexität von Sat für Formeln in 2-CNF?

Reduzierbarkeit: Wie können wir Probleme in CNF in DNF umwandeln und umgekehrt? Was sagt das über die wechselseitige poly-Reduzierbarkeit von SAT in Bezug auf DNF bzw. CNF aus?

Definition 5.20 (Vertex Cover: $L_{\text{Vertex}_{\leq k}}$)

Sei $G = (V, E)$ ein ungerichteter Graph. Ein Vertex Cover ist eine Teilmenge $A \subseteq V$ mit: für je 2 Knoten v, v' die durch eine Kante im Graphen verbunden ist, ist mindestens einer in A.

Für gegebenes $k \in \mathbb{N}$ besteht die Sprache $L_{\text{Vertex}_{\leq k}}$ aus allen Graphen die ein Vertex Cover der Größe kleiner oder gleich k besitzen.

Einige Reduktionen:

- $L_{\text{CNF-SAT}} \preceq_{\text{pol}} L_{\text{Clique}_k}$,
- $L_{\text{Ham}_{\text{dir}}} \preceq_{\text{pol}} L_{\text{Ham}_{\text{undir}}}$,
- $L_{\text{Ham}_{\text{undir}}} \preceq_{\text{pol}} L_{\text{Ham}_{cost \leq k}}$,
- $L_{\text{3-CNF-SAT}} \preceq_{\text{pol}} L_{\text{Vertex}_{\leq k}}$,
- $L_{\text{Vertex}_{\leq k}} \preceq_{\text{pol}} L_{\text{Clique}_k}$,
- $L_{\text{CNF-SAT}} \preceq_{\text{pol}} L_{\text{3-CNF-SAT}}$,
- $L_{\text{SAT}} \preceq_{\text{pol}} L_{\text{CNF-SAT}}$,
- $L_{\text{3-CNF-SAT}} \preceq_{\text{pol}} L_{\text{Color}_{\leq k}}$,
- $L_{\text{Color}_{\leq k}} \preceq_{\text{pol}} L_{\text{3-CNF-SAT}}$.

Was ist mit **Ist eine Zahl zusammengesetzt (nonprimeness)?**

Beispiel 5.21 (CNF-SAT $\preceq_{\text{pol}}$ Clique$_k$)

Gegeben eine Instanz ϕ von CNF-SAT, also eine Konjunktion von Klauseln über den (eventuell negierten) Variablen $x, y, \ldots$: $C_1 \wedge C_2 \wedge \ldots \wedge C_k$. Wir konstruieren daraus einen Graphen $G = (V, E)$:

Knoten (V): dies sind Paare (x, i) wobei x eine Variable ist, die in der Klausel i vorkommt.

Kanten (E): Es gibt eine Kante zwischen (x, i) und (y, j) falls: (1) $i \neq j$, und (2) x, y kommen nicht komplementär vor.

Es gilt dann: ϕ ist erfüllbar genau dann, wenn der zugeordnete Graph G eine Clique der Größe k hat.

Beispiel 5.22 ($\text{Ham}_{\text{dir}} \preceq_{\text{pol}} \text{Ham}_{\text{undir}}$)

Gegeben ein gerichteter Graph G. Wir machen aus G einen ungerichteten Graphen G' wie folgt.

- Für jeden Knoten v in G gibt es 3 Knoten in G': v_{in}, v_{out}, v_{mid}.
- Die Kanten in G' werden wie folgt definiert. Falls es eine gerichtete Kante von u nach v in G gibt, dann gibt es folgende ungerichtete Kanten in G': u_{out} ist mit v_{in} verbunden, und u_{in} ist mit u_{out} verbunden. Weiterhin ist v_{in} mit v_{mid} und v_{mid} mit v_{out} verbunden.

Es gilt dann: G hat einen Hamilton-Kreis genau dann, wenn G' einen Hamilton-Kreis hat.

Beispiel 5.23 ($\text{Ham}_{\text{undir}} \preceq_{\text{pol}} \text{Ham}_{cost \leq k}$,)

Gegeben ein ungerichteter Graph $G = (V, E)$. Wir müssen daraus einen vollständigen ungerichteten Graphen G', eine Kantengewichtsfunktion g und eine natürliche Zahl $k \in \mathbb{N}$ konstruieren, so daß gilt: es gibt einen Hamilton-Kreis in G genau dann, wenn es einen Hamilton-Kreis in G' mit Kosten $\leq k$ gibt.

G' wird wie folgt konstruiert:

G': G' sei die Vervollständigung von G,

g: die Kantengewichtsfunktion ist wie folgt definiert

$$g : E \to \mathbb{N}; e \mapsto \begin{cases} 1, & \text{falls } e \in E; \\ 2, & \text{sonst.} \end{cases}$$

k: $k := |V|$.

6. Anwendungen

Motivation

In diesem Kapitel geben wir einige wichtige Anwendungen der Informatik 3 in der Praxis an.

1. Neben dem **Compilerbau**, betrachten wir kurz Syntaxanalyse und den Nutzen regulärer Ausdrücke.
2. Wir stellen den Satz von Myhill/Nerode vor, eine andere Methode, um zu zeigen, ob eine Sprache regulär ist.
3. Schließlich benutzen wir diese Theorie um den Minimalautomaten einer Sprache zu bestimmen.

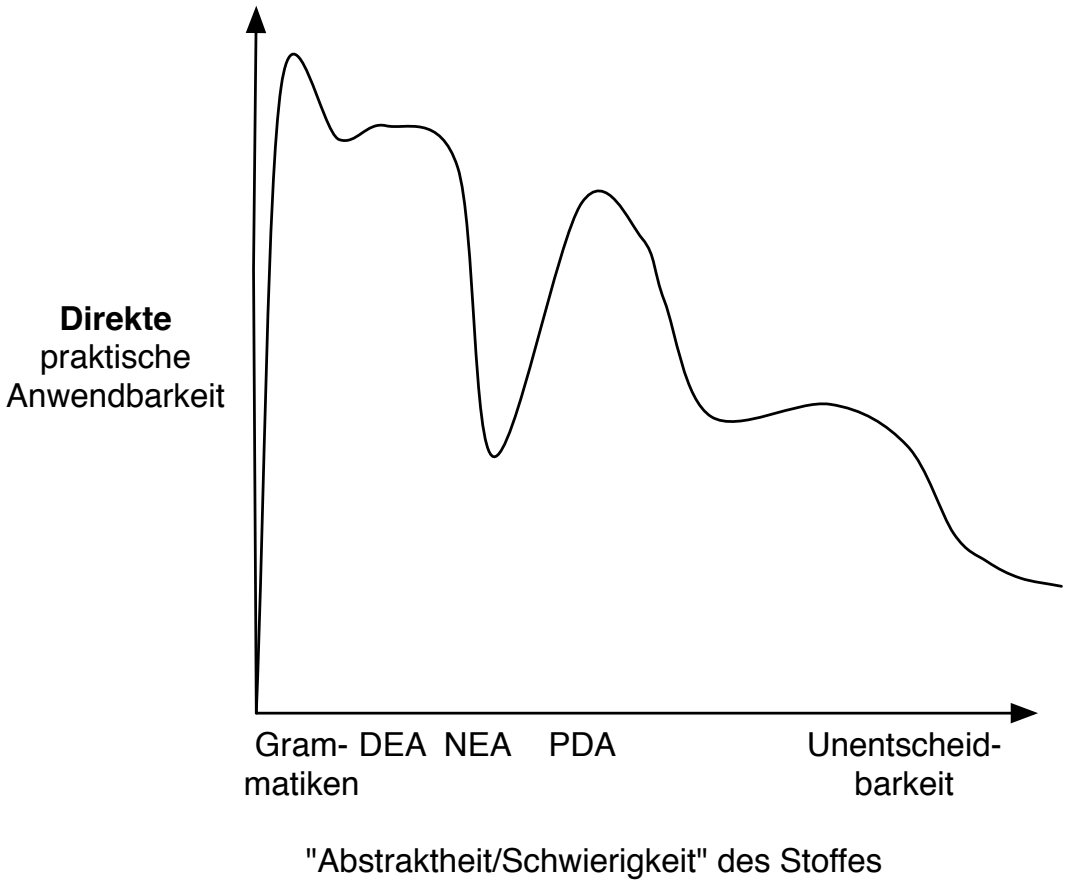
6.1 Praktische Bedeutung

Wozu brauchen wir Informatik III?

- Grammatiken,
- Reguläre Ausdrücke,
- Endliche Automaten,
- Kellerautomaten,
- (Linear) beschränkte Automaten,
- Turing-Maschinen,
- Entscheidbarkeit vs. Unentscheidbarkeit, und
- Komplexität.

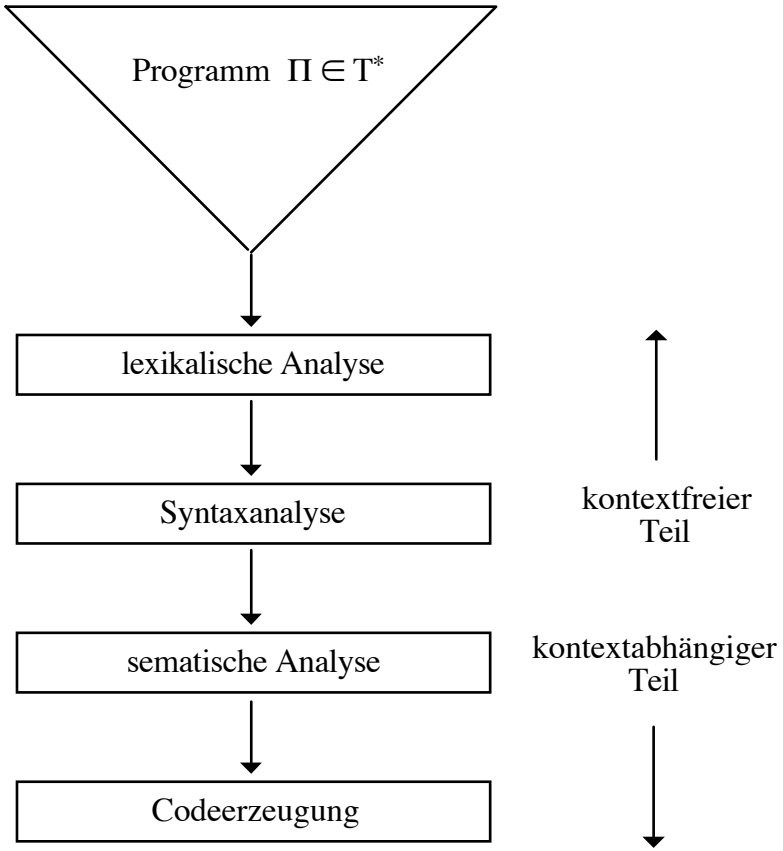
Gibt es auch **Anwendungen in der Praxis**?

Theoretische Komplexität vs. praktische Anwendbarkeit:



Im Folgenden betrachten wir zwei konkrete Anwendungen.

Compilerbau



Compilerbau

Ein Compiler (Übersetzer) übersetzt Quellcode (C-Programm) in Zielcode (Maschinencode).
Das geht in drei Schritten:

1. **lexikalische Analyse** (Zeichenreihe $\mapsto$ Tokenfolge)
   Überprüfung, Erkennung, Entfernung irrelevanter Zeichen
2. **Syntaxanalyse** (strukturellen Korrektheit) Ableitungsbaum des Programms
3. **semantische Analyse** (statischen Semantik)

Wo können Methoden der theoretischen Informatik angewandt werden?

Welche Sprachklassen könnte man den einzelnen Punkten zuordnen?

Lexikalische Analyse

Zur Generierung lexikalischer Analyser kann unter anderem ein **regulärer Ausdruck** herangezogen werden, der die Tokens beschreibt, der dann wiederum in einen ea umgewandelt wird. Endzustände entsprechen dann einem bestimmten Token.

Syntaxanalyse

Indeterministische Analysalgorithmen:

- bottom-up (Rechtsreduktion des Wortes)
- top-down (Finden einer Linksableitung)

Deterministische Analysalgorithmen:

- tabellenorientiert
  (Cocke-Younger-Kasami-Algorithmus)
- ableitungsorientiert (bestimmte Grammatiken)

Reguläre Ausdrücke

Verfahren zur Beschreibung von Zeichenketten (im Unterschied zum Automaten!). Im „Hintergrund“ wird ein regulärer Ausdruck in einen **endlichen Automaten** zum Finden entsprechender Zeichenreihen umgewandelt.
Beispiele:

- ?: Null oder ein Vorkommen von...
- +: Ein oder mehr Vorkommen von...
- ...

In der Praxis gibt es viele Operatoren, die in der Theorie nicht vorkommen, aber (fast) alle geeignet definiert werden können.

	Theorie	ERE	BRE	Wildcard	
				Shell	SQL
leere Menge	Ø	*(fehlt)*	*(fehlt)*	*(fehlt)*	*(fehlt)*
einzelnes Zeichen	*a*	*a*	*a*	*a*	*a*
Konkatenation (Hintereinanderschreiben)	*xy*	*xy*	*xy*	*xy*	*xy*
Alternative, Vereinigung	*x* | *y*	*x* | *y*	*(fehlt)*	*(fehlt)*	*(fehlt)*
Klammer	(. . .)	(. . .)	\(. . . \)	*(fehlt)*	*(fehlt)*
Rückverweis auf Inhalt der *i*-ten Klammer	*(fehlt)*	*(fehlt)*	*i*	*(fehlt)*	*(fehlt)*
einzelnes Zeichen aus endlicher Menge	*a* | *b* | *c*	[*abc*]	[*abc*]	[*abc*]	*(fehlt)*
einzelnes Zeichen nicht aus endlicher Menge	*(fehlt)*	[^*abc*]	[^*abc*]	[!*abc*]	*(fehlt)*
beliebiges einzelnes Zeichen	*(fehlt)*	.	.	?	_
beliebiger Text	*(fehlt)*	.*	.*	*	%
Kleenescher Abschluss (0..?-fache Wiederholung)	*x**	*x**	*x**	*(fehlt)*	*(fehlt)*
optionaler Teilausdruck (0..1-fache Wiederholung)	(| *x*)	*x*?	*(fehlt)*	*(fehlt)*	*(fehlt)*
1..?-fache Wiederholung	*xx**	*x*+	*xx**	*(fehlt)*	*(fehlt)*
m-fache Wiederholung	*(fehlt)*	*x*{*m*}	*x*\{*m*\}	*(fehlt)*	*(fehlt)*
m..*n*-fache Wiederholung	*(fehlt)*	*x*{*m*,*n*}	*x*\{*m*,*n*\}	*(fehlt)*	*(fehlt)*
m..?-fache Wiederholung	*(fehlt)*	*x*{*m*,}	*x*\{*m*,\}	*(fehlt)*	*(fehlt)*

Abbildung 28: Reguläre Ausdrücke in der Praxis (http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/).

Beispiel

ls -l abc:

```
rw-r—— 1 joeuser staff 580 Feb 25 14:19 abc
```

Wie können wir mithilfe eines regulären Ausdrucks den **Dateieigentümer** herausbekommen?

```
expr " `ls -l abc` " : '  *[bcdprswx-]* *[0-9][0-9]*
                          *\([^ ]*\)  '
```

Was bedeutet der Ausdruck?

> *Erst kommen vielleicht ein paar Zwischenräume, dann kommt ein Teil, der nur aus den Buchstaben b, c, d, p, r, s, w und x sowie aus Minuszeichen besteht, dann kommt mindestens ein Zwischenraum, dann mindestens eine Ziffer, dann mindestens ein Zwischenraum, und dann - hier fängt der Teil an, der uns interessiert - kommen Zeichen, die keine Zwischenräume sind und dann - aber das interessiert uns nicht mehr - kommt noch ein Zwischenraum ...*

Quelle:

`http://www.lrz-muenchen.de/services/schulung/unterlagen/regul/`

Reguläre Ausdrücke sind sehr mächtig, wenn sie **richtig** angewandt werden.

Für diverse Anwendungen ist es vorteilhaft, einen möglichst kleinen Automaten zu haben, denn er wird ja oftmals im Hintergrund konstruiert und ggf. viele Male benutzt!

Wie konstruieren wir einen „kleinen Automaten“ und woher wissen wir, daß wir den kleinsten Automaten gefunden haben?

6.2 Satz von Myhill-Nerode

Der Satz von Myhill-Nerode ist eine andere Methode, um festzustellen, daß eine Sprache **nicht** regulär ist. Wie kann man zu gegebener Sprache L feststellen, wie der kleinste ea für L aussieht (falls es ihn gibt)?

Definition 6.1 ($\sim_L$)

Für eine Sprache $L \subseteq \Sigma^*$ definieren wir die Relation $\sim_L \subseteq \Sigma^* \times \Sigma^*$:

$$v \sim_L x \text{ falls } \forall w \in \Sigma^* : (vw \in L \text{ gdw } xw \in L)).$$

Lemma 6.2

- $\sim_L$ *ist eine Äquivalenzrelation.*
- $\sim_L$ *ist eine* **Rechtskongruenz**: *(*$v \sim_L t$ *gdw* $\forall\, w \in \Sigma^* :\ (vw \sim_L tw)$*).*

Beispiel 6.3

$$L = \{w \in \{a,b\}^* : \ \#_a(w) = 1 \mod 3\}.$$

Es gibt genau 3 Äquivalenzklassen:

1. $[\varepsilon] = \{w \in \{a,b\}^* : \ \#_a(w) = 0 \mod 3\}.$
2. $[a] = \{w \in \{a,b\}^* : \ \#_a(w) = 1 \mod 3\}.$
3. $[aa] = \{w \in \{a,b\}^* : \ \#_a(w) = 2 \mod 3\}.$

Theorem 6.4 (Satz von Myhill-Nerode)

Sei $L \subseteq \Sigma^*$ eine Sprache. Dann sind folgende Bedingungen äquivalent:

(1) *L ist rational (vom Typ 3).*

(2) *L ist die Vereinigung einiger Äquivalenzklassen einer Rechtskongruenz über Σ^*, die selbst nur endlich viele Äquivalenzklassen besitzt.*

(3) *$\sim_L$ hat nur endlich viele Äquivalenzklassen.*

Beweis

(1) $\rightarrow$ (2): Sei $A = (K, \Sigma, \delta, s_0, F)$ ein e.a., der L akzeptiert.
Wir definieren folgende Relation R_A:

$$xR_Ay \text{ falls } \delta(s_0, x) = \delta(s_0, y).$$

R_A ist rechtskongruent (da ja e.a.'s deterministisch sind) und hat nur endlich viele Äquivalenzklassen (da ja $|K|$ endlich ist).
Offensichtlich ist L die Vereinigung aller $[x]$ mit $\delta(s_0, x) \in F$.

(2) $\rightarrow$ (3): Wir zeigen: **Sei R' eine Äquivalenzrelation die (2) erfüllt. Dann ist jede Äquivalenzklasse von R' in einer Äquivalenzklasse von $\sim_L$ enthalten.**

Dann hat $\sim_L$ **höchstens** soviele Äquivalenzklassen wie R' (es können ja mehrere von R' vollständig in einer von $\sim_L$ liegen), also nur endlich viele.

Sei also $xR'y$. Aus der Rechtskongruenz folgt (für jedes $z \in \Sigma^*$): $xzR'yz$. Also $yz \in L$ gdw $xz \in L$. Also $x \sim_L y$. Also ist $[x]_{\sim_{R'}}$ enthalten in $[x]_{\sim_L}$.

(3) $\rightarrow$ (1): Offensichtlich ist $\sim_L$ eine Rechtskongruenz.

Wir müssen nun einen e.a. $\hat{M}_L$ definieren, der L akzeptiert:

- $K := \{[x]_{\sim_L} : x \in \Sigma^*\}$,
- $\delta([x]_{\sim_L}, a) = [xa]_{\sim_L}$,
- $s_0 := [\varepsilon]_{\sim_L}$,
- $F := \{[x]_{\sim_L} : x \in L\}$.

Man mache sich klar, daß die Definition von δ wohldefiniert ist (wegen der Rechtskongruenz). Der eben konstruierte e.a. akzeptiert L, da

$$\delta(s_0, x) = [x]_{\sim_L} \text{ also } x \in L(\hat{M}_L) \underline{\text{gdw}} [x]_{\sim_L} \in F$$

6.3 Der Minimalautomat

Wir machen nun die Konstruktion des Automaten im letzten Beweis explizit.

Definition 6.5 ($\hat{M}_L$, der zu L passende Automat)

Zu $L \subseteq \Sigma^*$ definieren wir den folgenden Automaten $\hat{M}_L = (K, \Sigma, \delta, s_0, F)$:

- $K := \{[w]_{\sim_L} : w \in \Sigma^*\}$,
- $s_0 := [\varepsilon]_{\sim_L}$,
- $F := \{[w]_{\sim_L} : w \in L\}$,
- $\delta([w]_{\sim_L}, a) := [wa]_{\sim_L}$.

Ist $\hat{M}_L$ ein endlicher Automat?

Angenommen, ein Automat enthält Zustände q, q' mit

$$\forall w \in \Sigma^* : \ \delta(q, w) \in F \text{ gdw } \delta(q', w) \in F.$$

Was bedeutet das?
Man sagt q und q' sind äquivalent.

Gibt es zu jedem ea A einen Automaten, der genau dieselbe Sprache akzeptiert und eine minimale Anzahl von Zuständen hat?

Theorem 6.6 (Minimalität von $\hat{M}_L$)

Sei A ein endlicher Automat und $L = L(A)$. Dann sind folgende Aussagen äquivalent:

1. *A ist minimal: jeder andere ea A' mit $L(A') = L(A)$ hat mindestens soviele Zustände wie A.*
2. *A ist isomorph zu $\hat{M}_L$.*

Beweis

Wir haben schon gezeigt (Beweis von Theorem 6.4), dass jeder andere ea der L akzeptiert, eine Äquivalenzrelation definiert, die **feiner** ist als $\sim_L$ (also jede dieser Äquivalenzklassen ist in einer von $\sim_L$ enthalten).
Also hat jeder solche e.a. mindestens so viele Zustände wie $\hat{M}_L$.

Sei nun M' ein ea der L akzeptiert und genau soviele Zustände hat wie $\hat{M}_L$. **Wir definieren einen Isomorphismus zwischen M' und $\hat{M}_L$.** Sei q' ein Zustand von M'. Dann gibt es ein Wort w mit $\delta'(s'_0, w) = q'$ (sonst könnte q' weggelassen werden). Wir bilden q' auf $\delta(s_0, w)$ ab (man beachte, daß dies wohldefiniert ist). □