Berechenbarkeit und Formale Sprachen

gelesen von Prof. Wanka
an der FAU Erlangen

in LaTeX umgesetzt von Alexander Raß
mit Unterstützung von Bernd Bassimir

4. Februar 2025

Version 1.3.2

Vorwort

Die Inhalte dieses Dokuments können an verschiedenen Stellen über die Vorlesungsinhalte hinausgehen und repräsentieren im wesentlichen eine Obermenge der in der Vorlesung behandelten Inhalte.

Im wesentlichen werden sich die Inhalte dieses Dokuments kaum von der Vorlesung unterscheiden. Natürlich ist es möglich, dass die Reihenfolge, in der die Inhalte präsentiert werden, angepasst wird. Sollten neue Inhalte oder neue Beispiele behandelt werden, so können Sie gerne zur Verbesserung des Skripts beitragen indem Sie den Teil als tex-Datei an Matthias Kergaßner (**matthias.kergassner@fau.de**) weitergeben oder zumindest einen entsprechenden Hinweis absetzen. Auch erkannte Fehler können Sie uns so mitteilen.

Im Changelog finden Sie Informationen zu allen Änderungen.

Changelog seit Version vom 11.4.2017

- 24.11.2017
  Kapitel 1.5 „Universelle Turingmaschinen“
  Fehlenden Betragsstrich unmittelbar vor Platzbedarf hinzugefügt:
  „Da $|\langle M\rangle| = \mathcal{O}(1)$ gelten …“

- 18.1.2018
  Kapitel 3.1 „Grammatiken“ Beweis zu Satz 3.3 im Abschnitt „⇒“.
  Grammatiktupel bei Überführung von Turingmaschine M in Grammatik G_M mit $L(M) = L(G_M)$ sowohl in der Grammatik mit Rückwärts- als auch Vorwärtsausführung der Turingmaschine hinzugefügt:
  „Ziel: Grammatik $G_M = (V, \Sigma, \mathcal{P}, \mathcal{S})$ angeben, die diese Rechnung *rückwärts* ausführt.“
  „$G_M = (V, \Sigma, \mathcal{P}, \mathcal{S})$, $V = \left(\{\mathcal{S}, A_1, A_2\} \cup Q \cup \left\{ \begin{bmatrix} a \\ b \end{bmatrix} \;\middle|\; a \in (\Sigma \cup \{\varepsilon\}), b \in \Gamma \right\}\right)$, Σ, Γ wie bei Turingmaschine,“

- 24.1.2018
  Kapitel 3.7 „Kontextfreie Grammatiken und Sprachen“ Beispiel 3.32 zur kontextfreien Pumpeigenschaft Punkt (d). An Stelle von „n_L^2“ wurde an mehreren Stellen fälschlicherweise nur „n_L“ benutzt:
  „Wähle $i = 2$, dann gilt $uv^iwx^iy = uv^2wx^2y = a^{n_L^2+|vx|}$.
  Damit $a^{n_L^2+|vx|} \in L_4$ muss $n_L^2 + |vx|$ eine Quadratzahl sein.
  Es gilt aber $n_L^2 \lneq n_L^2 + \overbrace{|vx|}^{\geq 1} \leq n_L^2 + n_L \lneq n_L^2 + n_L + n_L + 1 = (n_L + 1)^2$“

- 1.2.2018
  Kapitel 3.5 „Das Pumping-Lemma für reguläre Sprachen und nichtreguläre Sprachen“
  Beweis von Satz 3.20: Alternative Version mit regulärer Grammatik hinzugefügt.
  Definition 3.19, Beweis von Satz 3.20, Beispiele danach: Variablennamen verändert (meist x zu z).

- 7.2.2018
  Kapitel 2.0.2 „Das P-NP-Problem“
  Bei den relevanten Sprachen wurden IS, Col, 3Col und BP hinzugefügt.

- 12.4.2018
  Kapitel 3.6 „Reguläre Ausdrücke und Abschlusseigenschaften regulärer Sprachen“
  Ausführliche Erklärung zur Gauss-Elimination um aus einem Automaten einen regulären Ausdruck zu erzeugen.

- 9.7.2018
  Kapitel 3.7 „Kontextfreie Grammatiken und Sprachen“, Unterkapitel „Kellerautomaten“
  Alternative Darstellung der Konfiguration eines Kellerautomaten.

- 15.10.2018
  Kapitel 1 „Einführung in die Berechenbarkeit“, Unterkapitel „Die Registermaschine (RAM)“
  „-“ bei Registermaschinenbefehlen eingefügt.

- 7.1.2019
  Kapitel 3.2 „Endliche Automaten“
  In Abbildung 12 Selfloops bei q_1 und q_3 hinzugefügt (wie in G_4 gefordert); q_5 aus V entfernt (bei G_4).

- 22.1.2019
  Kapitel 3.6 „Reguläre Ausdrücke und Abschlusseigenschaften regulärer Sprachen“
  Es wurde zur Gauss-Elimination, um aus einem Automaten einen regulären Ausdruck zu erzeugen, ein weiteres Beispiel hinzugefügt und die Schreibweise (für beide Beispiele) aus Vorlesung übernommen (Anstelle von R_S, R_A, R_B wird direkt S, A, B verwendet).

- 7.2.2019
  - Vorwort überarbeitet.
  - Kapitel 3.7 „Kontextfreie Grammatiken und Sprachen“
    Im Beweis zu Lemma 3.27 ist ein gerichteter kreisfreier Graph ein DAG (kein GAG).

- 1.4.2019
  Kapitel 2.0.2 „Das P-NP-Problem“
  Bei der Definition von VC wurde „=:“ durch „:=“ ersetzt.

- 4.11.2019
  Kapitel 1.5 „Universelle Turingmaschinen“
  Fehlendes #-Symbol in Definition von $\langle M \rangle$ hinzugefügt. Leerzeichen innerhalb $\langle M \rangle$ entfernt, welche man fälschlicherweise als Blank-Zeichen interpretieren könnte.

- 25.11.2019
  Kapitel 2 „Nichtdeterministische Turingmaschinen und das P-NP-Problem“
  Unmittelbar nach Definition 2.1 „Zustandsübergang“ zu „Konfigurationsübergang“ geändert.
  In Definition 2.2 wurden die Fälle $x \notin L(M)$ ausgeschlossen (bisher wurden sie mit 0 bewertet).
  In Satz 2.3 wurde explizite Platzbeschränkung mit $s(n)$ durch implizite Platzbeschränkung durch die Laufzeit $t(n)$ ersetzt.

- 01.02.2021
  Unterabschnitt 1.8:
  „[...] und für jedes $x \in L$ existiert ein $y \in \{0,1\}^*$ sodass M_g gestartet mit Eingabe y die Ausgabe x produziert, da g surjektiv ist.“

- 01.02.2021
  Unterabschnitt 2.2

$$\text{BP} := \left\{ \langle A, b \rangle \mid A \in \mathcal{M}_{m,n}(\mathbb{Z}) = \mathbb{Z}^{m \times n},\ b \in \mathbb{Z}^m,\ \exists y \in \{0,1\}^n : Ay \leq b \right\}$$

- 01.02.2021
  Unterabschnitt 3.6
  $R = (R_1^*) \quad L(R) = L(R_1)^*$

- 01.02.2021
  Unterabschnitt 3.7
  $E_i = \{A \mid (A \to B_1 \dots B_k) \in \mathcal{P},\ \text{mit } B_1, \dots, B_k \in E_{i-1}\} \cup E_{i-1}$

- 01.02.2021
  Unterabschnitt 3.7
  „ein Wort $w = a_1 \dots a_n \in \Sigma^*$.“

- 15.02.2021 (Version 0.2.0)
  Unterabschnitt 3.6
  Rijk-Verfahren überarbeitet:
  - $R_{i,j}^k$ ist ein regulärer Ausdruck und $L(R_{i,j}^k)$ die Sprache, die von diesem Ausdruck erzeugt wird.
  - Fazit dieses Beweises hinzugefügt

- 15.02.2021 (Version 0.2.1)
  Unterabschnitt 3.4
  p, q markiert $\Rightarrow$ es gibt ein markiertes $\{p', q'\}$ mit $\{\delta(p,a), \delta(q,a)\} = \{p', q'\}$ und

- 25.03.2021 (Version 0.3.0)
  Unterabschnitt 1.7:
  - Leichte Umstrukturierung
  - Beispiele beziffert
  - Im Beispiel die $\mathit{FESTE\_MASCHINE}_{\langle M \rangle w}$ definiert
  - $\mathit{FESTE\_MASCHINE}_{\langle M \rangle w}$ wird nun korrekterweise mit leerer statt mit beliebiger Eingabe gestartet

- 20.10.2021 (Version 1.0.0)
  Das Skript ist absofort offiziell.

- 26.01.2022 (Version 1.1.0)
  Notation der Beispiele in 3.32 korrigiert.

- 07.02.2022 (Version 1.2.0)
  Argumentation der Beispiele in 3.5

- 02.08.2023 (Version 1.3.0)
  Alterative Definition CLIQUE in 2.0.2 ergänzt.

- 18.10.2023 (Version 1.3.1)
  $\mathbb{N}$ durch $\mathbb{N}_0$ ersetzt in Abb. 1

- 04.02.2025 (Version 1.3.2)
  Im Abschnitt der Kellerautomaten waren die 7 Komponenten des Tupels M in verdrehter Reihenfolge. Außerdem fehlte in einem Beispiel zu den Abschlusseigenschaften kontextfreier Sprachen ein L um einen regulären Ausdruck um die daraus resultierende Sprache zu bezeichnen.

Inhaltsverzeichnis

0 Was behandelt diese Vorlesung? 6

1 Einführung in die Berechenbarkeit 7
1.0 Die Registermaschine (RAM) . . . 7
1.1 Die Turingmaschine (TM) . . . 8
1.2 Programmiertechniken und einfache Simulationen . . . 11
1.3 Simulationen zwischen RAMs und TMs . . . 13
1.4 Die Church-Turing-These . . . 14
1.5 Universelle Turingmaschinen . . . 15
1.6 Unentscheidbare Probleme . . . 16
1.7 Reduktionen und der Satz von Rice . . . 19
1.8 Rekursive Aufzählbarkeit . . . 24

2 Nichtdeterministische Turingmaschinen und das P-NP-Problem 27
NTM und P-NP Problem . . . 27
2.0.1 Die Nichtdeterministische Turingmaschine (NTM) . . . 27
2.0.2 Das P-NP-Problem . . . 29
2.1 NP-Vollständigkeit . . . 33
2.2 Satz von Cook . . . 34
2.3 Ein exakter mildexponentieller Algorithmus für das Vertex Cover Problem . . . 43

3 Formale Sprachen 44
3.1 Grammatiken . . . 44
3.2 Endliche Automaten . . . 48
3.3 Nichtdeterministische endliche Automaten (NFA) . . . 51
3.4 Minimierung endlicher Automaten . . . 54
3.5 Das Pumping-Lemma für reguläre Sprachen und nichtreguläre Sprachen . . . 59
3.6 Reguläre Ausdrücke und Abschlusseigenschaften regulärer Sprachen . . . 63
3.7 Kontextfreie Grammatiken und Sprachen . . . 69

4 Primitive Rekursion und μ-Rekursion 82
4.1 LOOP-, WHILE- und GOTO-Berechenbarkeit . . . 82
4.2 Definition primitiv rekursiver und μ-rekursiver Funktionen . . . 85
4.3 Die Ackermann Funktion . . . 89

0 Was behandelt diese Vorlesung?

- Was geht und was geht nicht, ggf. mit welchen Ressourcen?
- Erkennung und Erzeugen von Objekten aus unendlich großen Mengen mit endlichen Mitteln.

Literatur

[1] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] U. Schöning. *Theoretische Informatik - kurzgefaßt (5. Aufl.)*. Hochschultaschenbuch. Spektrum Akademischer Verlag, 2008.

[3] I. Wegener. *Theoretische Informatik - eine algorithmenorientierte Einführung (3. Auflage)*. Teubner, 2005.

1 Einführung in die Berechenbarkeit

1.0 Die Registermaschine (RAM)

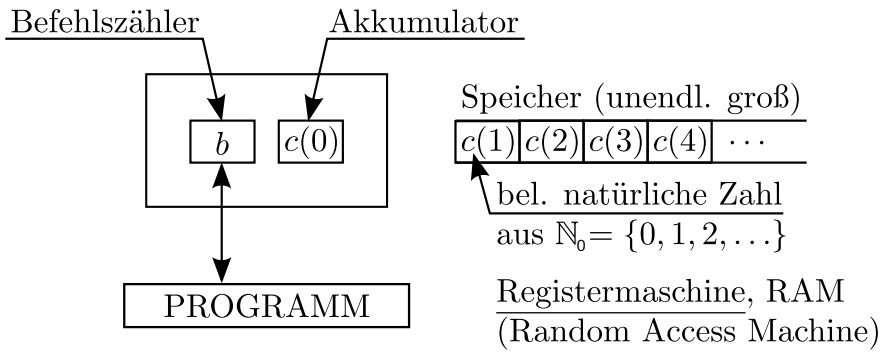


Abbildung 1: Aufbau einer Registermaschine.

Befehlssatz:

LOAD i	$c(0) := c(i);$	$b := b + 1;$	
STORE i	$c(i) := c(0);$	$b := b + 1;$	
ADD i	$c(0) := c(0) + c(i);$	$b := b + 1;$	
SUB i	$c(0) := c(0) \dot{-} c(i);$	$b := b + 1;$	dabei gilt $a \dot{-} b =$„a monus b“$= \max\{0, a - b\}$
MULT i	$c(0) := c(0) \cdot c(i);$	$b := b + 1;$	
DIV i	$c(0) := \lfloor c(0)/c(i) \rfloor;$	$b := b + 1;$	
GOTO j		$b := j;$	
IF $c(0)?l$ GOTO j	$? \in \{=, \leq, \geq, <, >\},$	$l \in \mathbb{N}$	
END			

Befehle mit Konstanten (const):

C-LOAD l	$c(0) := l;$	$b := b + 1;$
C-ADD l	$c(0) := c(0) + l;$	$b := b + 1;$
C-SUB l	$c(0) := c(0) \dot{-} l;$	$b := b + 1;$
C-MULT l	$c(0) := c(0) \cdot l;$	$b := b + 1;$
C-DIV l	$c(0) := \lfloor c(0)/l \rfloor;$	$b := b + 1;$

Befehle mit indirekten Zugriffen:

IND-LOAD i	$c(0) := c(c(i));$	$b := b + 1;$
IND-STORE i	$c(c(i)) := c(0);$	$b := b + 1;$
IND-ADD i	$c(0) := c(0) + c(c(i));$	$b := b + 1;$
IND-SUB i	$c(0) := c(0) \dot{-} c(c(i));$	$b := b + 1;$
IND-MULT i	$c(0) := c(0) \cdot c(c(i));$	$b := b + 1;$
IND-DIV i	$c(0) := \lfloor c(0)/c(c(i)) \rfloor;$	$b := b + 1;$

- Ein Programm besteht aus endlicher Folge von Einzelbefehlen des Befehlssatzes.
- Programmzeilen sind durchnummeriert: 1, 2, 3, ...
- Eingabe steht in den Registern 1,..., k. Rest ist mit 0 initialisiert.
- Ausgabe steht in den Registern 1,..., k'.

„Alle Algorithmen lassen sich mit solchen RAM-Programmen programmieren.“

1.1 Die Turingmaschine (TM)

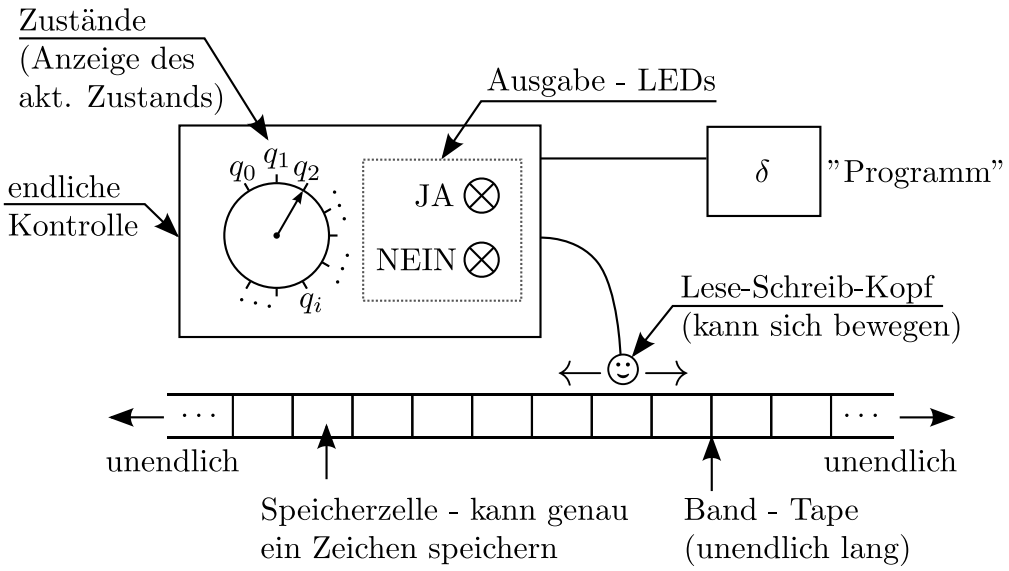


Abbildung 2: Aufbau einer Turingmaschine.

Definition 1.1 (Deterministische 1-Band-Turingmaschine)
Eine deterministische 1-Band-Turingmaschine $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *ist charakterisiert durch:*

- Q*: endliche Zustandsmenge* $\{q_0, \ldots, q_k\}$
- Σ*: endliches Eingabealphabet*
- Γ*: endliches Bandalphabet mit* $\Sigma \subsetneq \Gamma$
- B*: Blank,* $B \in \Gamma$, $B \notin \Sigma$ *(Leerzeichen)*
- q_0*:* $q_0 \in Q$ *Startzustand*
- F*: akzeptierende Endzustände,* $F \subseteq Q$
- *das Programm* δ*:* $Q \times \Gamma \to Q \times \Gamma \times \underbrace{\{R, L, N\}}_{\text{Richtungen für Kopfbewegung (rechts, links, nichts)}}$
  eine partielle Funktion, wobei es für Endzustände keine Übergänge geben soll
- *Zu Beginn steht der Lese-/Schreibkopf auf dem ersten Zeichen der Eingabe*
- *Eingabe:* $\underbrace{w}_{\textbf{Wort aus } n \text{ Zeichen aus } \Sigma} = w_1 w_2 \ldots w_n \in \Sigma^*$
- *bis auf die Eingabe ist das Band „leer“ (mit Blank-Zeichen befüllt)*
- *Ist die Eingabe* $w = \varepsilon$ *(leeres Wort) so steht der Lese-/Schreibkopf auf einem Blank*

Sei $L \subseteq \Sigma^*$, dann heißt L ***Sprache*** über das Alphabet Σ. L ist eine Sammlung von Wörtern aus Σ^*.

Beispiel 1.2 $L = \{0^n 1^n | n \geq 1\} \subseteq \{0,1\}^*$
Gegeben: $w \in \Sigma^*$, *Frage:* $w \in L$? (***Wortproblem***)
Wie löst man das Wortproblem für L *mit einer TM?*

```
1 Teste zuerst, ob die Eingabe von der Form 00...01...1 ist;
2 solange noch Zeichen vorhanden sind tue
3 |  Lies die letzte 1, lösche sie, „merke die gelesene 1 im Zustand“;
4 |  Lies die erste 0, lösche sie;
5 Ende
```

$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ *mit*

$$Q = \{q_0, \ldots, q_7\},\ \Sigma = \{0,1\},\ \Gamma = \{0,1,B\},\ F = \{q_7\}$$

δ	0	1	B
q_0	$(q_0, 0, R)$	$(q_1, 1, R)$	–
q_1	–	$(q_1, 1, R)$	(q_2, B, L)
q_2	–	(q_3, B, L)	–
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	(q_4, B, R)
q_4	(q_5, B, R)	–	–
q_5	$(q_6, 0, R)$	$(q_6, 1, R)$	(q_7, B, N)
q_6	$(q_6, 0, R)$	$(q_6, 1, R)$	(q_2, B, L)
q_7	–	–	–

Eine ***Konfiguration*** K ist ein Element aus $\Gamma^* \times Q \times \Gamma^*$.

„TM M ist in Konfiguration $K = \alpha q \beta$“
$\Leftrightarrow$ Auf dem Band steht $\alpha\beta$ und der Kopf steht unter dem ersten Zeichen von β und der Zustand ist q.

z.B. $00q_20111$ ist eine Konfiguration, welche in Abbildung 3 visualisiert ist.

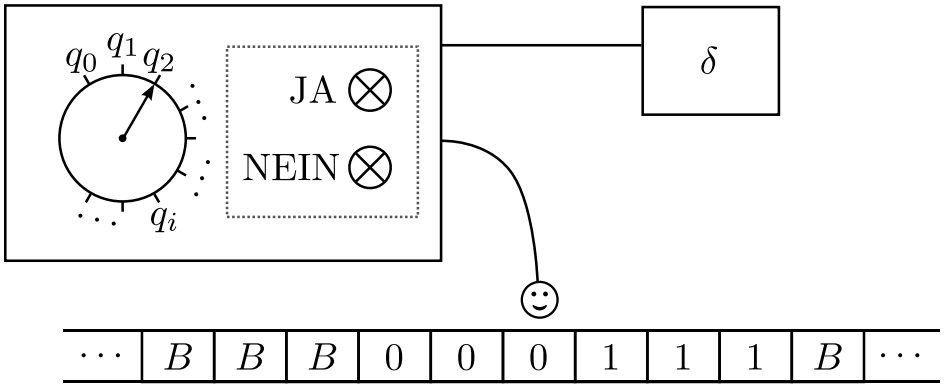


Abbildung 3: Veranschaulichung der Konfiguration $00q_20111$.

Sei $\beta = b\overline{\beta}$, $\alpha = \overline{\alpha}a$.
„$\alpha' q' \beta'$ ist direkte Nachfolgekonfiguration von $\alpha q \beta$“, falls gilt $\delta(q, b) = (q', b', s)$, $s \in \{R, L, N\}$ und

- falls $s = N$: $\alpha' = \alpha$, $\beta' = b'\overline{\beta}$
- falls $s = L$: $\alpha' = \overline{\alpha}$, $\beta' = ab'\overline{\beta}$
- falls $s = R$: $\alpha' = \alpha b'$, $\beta' = \overline{\beta}$

Schreibweise dass $\alpha' q' \beta'$ eine direkte Nachfolgekonfiguration von $\alpha q \beta$ ist: $\alpha q \beta \vdash \alpha' q' \beta'$

$\alpha' q' \beta'$ ist die i-te Nachfolgekonfiguration von $\alpha q \beta$, falls gilt:
$i = 0$: $\alpha q \beta = \alpha' q' \beta'$
$i \geq 1$: es gibt Konfigurationen $K_1, K_2, \ldots, K_{i-1}$, so dass
$\alpha q \beta \vdash K_1 \vdash K_2 \vdash \ldots \vdash K_{i-1} \vdash \alpha' q' \beta'$
Schreibweise dass $\alpha' q' \beta'$ die i-te Nachfolgekonfiguration von $\alpha q \beta$ ist: $\alpha q \beta \vdash^i \alpha' q' \beta'$

$\alpha' q' \beta'$ ist eine Nachfolgekonfiguration von $\alpha q \beta$, falls es ein i gibt, so dass $\alpha q \beta \vdash^i \alpha' q' \beta'$.
Schreibweise dass $\alpha' q' \beta'$ eine Nachfolgekonfiguration von $\alpha q \beta$ ist: $\alpha q \beta \vdash^* \alpha' q' \beta'$

Definition 1.3
M sei deterministische 1-Band-TM. $x \in \Sigma^*$ sei die Eingabe von M.

- *M* **akzeptiert** *$x \in \Sigma^*$, falls es $\alpha, \beta \in \Gamma^*$ und $q \in F$ gibt mit $q_0 x \vdash^* \alpha q \beta$.*
  (im Beispiel 1.2: $q_0 000111 \vdash^* BBBBq_7BBBB$)
  O. B. d. A. wenn eine Konfiguration mit $q \in F$ erreicht wird, gibt es keine Nachfolgekonfiguration mehr.
- *Die Sprache* **L(M)** *von M ist die Menge aller von M akzeptierten $x \in \Sigma^*$. Hier wird keine Aussage getroffen über das Verhalten von M, wenn M die Eingabe x nicht akzeptiert. M* **akzeptiert** *die Sprache $L(M)$.*
- *Ein stärkerer Begriff: Falls M die Sprache L akzeptiert und für alle Eingaben $x \in \Sigma^*$ nach endlich vielen Schritten anhält, so* **entscheidet** *M die Sprache L.*
- *$L \subseteq \Sigma^*$ heißt* **rekursiv aufzählbar** *(r.a., engl. r.e. (recursively enumerable)) genau dann, wenn es eine deterministische 1-Band-TM M gibt mit $L(M) = L$.*
- *$L \subseteq \Sigma^*$ heißt* **entscheidbar** *oder* **rekursiv** *genau dann, wenn es eine deterministische 1-Band-TM M gibt, die L entscheidet.*

Was kann passieren, wenn eine TM M eine Eingabe x nicht akzeptiert?

- „M, gestartet mit Eingabe x, erreicht Konfiguration, für die es keine Nachfolgekonfiguration gibt, aber der aktuelle Zustand ist nicht in F enthalten“ kann bezeichnet werden als
  M, gestartet mit Eingabe x, hält „nicht akzeptierend“ bzw.
  M, gestartet mit Eingabe x, hält verwerfend.
- M, gestartet mit Eingabe x, *hängt sich in Endlosschleife auf* bzw.
  M, gestartet mit Eingabe x, hält nie.

Anmerkung: Wir werden rekursiv aufzählbare Sprachen untersuchen, die nicht entscheidbar sind.

Definition 1.4
Eine deterministische 1-Band-TM M **berechnet die partielle Funktion** *$f : \Sigma^* \to (\Gamma \setminus \{B\})^*$ falls gilt:*

- *$f(x) = y$, falls $q_0 x \vdash^* qy$ und M gestartet mit Eingabe x hält in Konfiguration qy für ein $q \in Q$*
- *$f(x)$ ist nicht definiert, falls M gestartet mit Eingabe x nicht hält.*

Eine deterministische 1-Band-TM M **berechnet die totale Funktion** *$f : \mathbb{N}^r \to \mathbb{N}$, falls $\Sigma = \{0, 1, \#\}$ und für jedes $a_1, \ldots, a_r \in \mathbb{N}$ gilt:*

$$q_0 bin(a_1) \# bin(a_2) \# \ldots \# bin(a_r) \vdash^* q\, bin(f(a_1, \ldots, a_r)), \text{ für ein } q \in Q \text{ und } M \text{hält.}$$

($bin(x)$ ist die Binärdarstellung der Zahl $x \in \mathbb{N}$)

1.2 Programmiertechniken und einfache Simulationen

(1.) Endlicher Speicher(„im Zustand merken“)
$x_1 \dots x_n \in \Sigma^+$ Eingabe.
Frage: Ist x_1 in $x_2 \dots x_n$ enthalten?
$\Gamma = \Sigma \cup \{B\}$, $Q = (\{q_0\} \times \Sigma) \cup \{q_0, q_1\}$, Startzustand q_0, $F = \{q_1\}$.
δ:
für alle $a \in \Sigma$: $\delta(q_0, a) = ((q_0, a), a, R)$
für alle $a \in \Sigma$: $\delta((q_0, a), a) = (q_1, a, N)$
für alle $a, b \in \Sigma$, $a \neq b$: $\delta((q_0, a), b) = ((q_0, a), b, R)$

(2.) Unterprogramme
Wenn wir eine TM „programmieren“, können wir sagen: Wir benutzen ein Unterprogramm, das eine bestimmte Aufgabe löst.

(3.) Spurtechnik

	U	N	I	V	E	R	S	I	T	Ä	T	
	E	R	L	A	N	G	E	N	”B”	”B”	”B”	
	N	Ü	R	N	B	E	R	G	”B”	”B”	”B”	

} ein(!) Band

Beispielsweise liest ☺ das erste Zeichen $\begin{pmatrix} \text{U} \\ \text{E} \\ \text{N} \end{pmatrix} \in \Gamma$,

hier 3 Spuren (allg. k Spuren), aber noch immer nur 1 Kopf [”B” entspricht Blank]

Definition 1.5 *Eine* **deterministische k-Band-TM** *hat k Bänder und k Köpfe.*
$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{R, L, N\}^k$
Die Eingabe steht auf Band 1. *Die Arbeitsweise ist analog zur 1-Band-TM.*

2-Band-TM ist beispielsweise gut für $L = \{\omega \# \omega^R | \omega \in \{0,1\}^*\}$ geeignet.

Definition 1.6
M sei deterministische k-Band-TM, die für jede Eingabe hält.

Zeitkomplexität (Laufzeit): $\mathbf{T_M(x)}$:= *Anzahl der Schritte (Konfigurationsübergänge), die M gestartet mit Eingabe $x \in \Sigma^*$ ausführt, bis gehalten wird (individuell für x)*

Platzkomplexität (Platzbedarf): $\mathbf{S_M(x)}$:= *Anzahl verschiedener Zellen, die M gestartet mit Eingabe $x \in \Sigma^*$ besucht.*

$$\mathbf{T_M(n)} := \max\{T_M(x) | x \in \Sigma^*, |x| \leq n\}$$

$$\mathbf{S_M(n)} := \max\{S_M(x) | x \in \Sigma^*, |x| \leq n\}$$

$t, s : \mathbb{N} \to \mathbb{N}$: *M ist* **t(n)-zeitbeschränkt** *und* **s(n)-platzbeschränkt**, *falls für alle $n \in \mathbb{N}$: $T_M(n) \leq t(n)$ und $S_M(n) \leq s(n)$.*

$T_M(n) \geq S_M(n)$, da in $T_M(n)$ Schritten nicht mehr als $T_M(n)$ Zellen besucht werden können, auf einer 1-Band-TM.

Satz 1.7 *Jede $t(n)$-zeitbeschränkte und $s(n)$-platzbeschränkte deterministische k-Band-TM M' kann durch eine $\mathcal{O}(t(n) \cdot s(n))$-zeitbeschränkte und $\mathcal{O}(s(n))$-platzbeschränkte deterministische 1-Band-TM M simuliert werden.*

BEWEIS: $M' = (Q', \Sigma', \Gamma', \delta', q_0', F')$ k-Band-TM

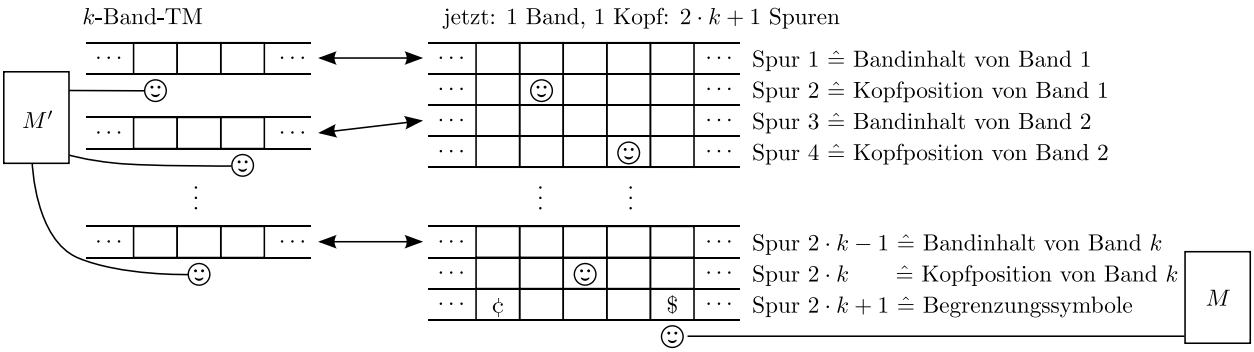


Abbildung 4: Kodierung einer k-Band-TM als 1-Band-TM.

Jedes Band von M' wird jeweils mittels 2 Spuren in M abgebildet. In der ersten der beiden Spuren ist der Bandinhalt abgelegt und in der zweiten Spur wird die Kopfposition vermerkt. Weiterhin wird ein zusätzliches Band benötigt, um den Bereich zu markieren, in dem sich die Köpfe von M' befinden. Die entsprechenden Ränder werden mit ¢ und $ markiert.
Simulation:

```
1 Bereite das Band für die Simulation vor;
  /* Begrenzungssymbole eintragen, Kopfpositionen auf den Spuren vermerken   */
2 solange M' noch nicht hält tue
3 |  Gehe zum Rand mit den ¢;
4 |  Gehe nach rechts bis $ und wenn in einer Spur ein Kopf ☺ gefunden wurde, merke das
  |   zugehörige Zeichen im Zustand bis alle Kopfinhalte gespeichert sind;
  |  /* Dazu ist auch noch die ganze Zeit über der aktuelle Zustand q' von M' im
  |     Zustand gespeichert                                                   */
5 |  Gehe wieder zu ¢;
6 |  Gehe nach rechts und verrichte die lokale Arbeit an den ☺;
7 |  Wir merken uns im Zustand von M den Nachfolgezustand q' von M';
  |  /* der alte Zustand von M' und die im zuletzt simulierten Schritt gelesenen
  |     Zeichen müssen (und können auch) nicht mehr im Zustand gespeichert werden */
8 Ende
9 Räume das Band auf und halte wie M';
```

Laufzeit: $\mathcal{O}(s(n) \cdot t(n))$ ← einen Schritt zu simulieren dauert $4\,s(n)$ Schritte auf M.
Platz: $\mathcal{O}(s(n))$ ← Platz läuft in der Simulation nicht „auseinander“. □

Korollar 1.8 *1-Band-TM und k-Band-TM liefern die gleichen Mengen entscheidbarer Sprachen (Klassen), rekursiv aufzählbarer Sprachen und berechenbarer Funktionen.*

1.3 Simulationen zwischen RAMs und TMs

Definition 1.9 *Für eine RAM M und eine Eingabe x ist*
$\mathbf{T_M(x)} :=$ *Anzahl Schritte von M gestartet mit Eingabe x.*
$\mathbf{S_M(x)} :=$ *Größte benutzte Speicherzelle bei der Rechnung von M gestartet mit Eingabe x.*
*„**t(n)-zeitbeschränkt**“ und „**s(n)-platzbeschränkt**“ ist analog zu TMs. (n kann die Anzahl der Zahlen oder die Anzahl der Bits sein)*

Satz 1.10 *Jede RAM kann durch eine deterministische TM simuliert werden.*
Ist die RAM $t(n)$-zeitbeschränkt, ist die TM $\mathcal{O}(t(n)^3)$-zeitbeschränkt.

BEWEIS:

(a) Wir müssen uns überlegen, wie eine Konfiguration der RAM auf einer Mehrband-TM gespeichert wird.

(b) Wir müssen die Konfigurationsübergänge der RAM realisieren.

zu (a): $c(1)$, $c(2)$,..., $c(k) \in \mathbb{N}_0$ Eingabe der RAM
Konfiguration einer RAM:

- Speicherinhalt: $c(0), c(1), \ldots$ (inklusive Adresse und $c(0)$)
- Programmzähler b

Turing Maschine mit 3 Bändern:

#0#bin($c(0)$)#1#bin($c(1)$)#...#bin(i)#bin($c(i)$)#...	Speicherband mit Paaren aus Adresse, Inhalt
#bin(b)#	Befehlszählerband
	Arbeitsband

Abstand zwischen den # auf Befehlszählerband ist konstant, da das RAM-Programm konstante Länge hat. Daher kann die TM auch mit der Technik „im Zustand merken“ arbeiten.

Nun müssen wir für jeden möglichen RAM-Befehl ein TM-Unterprogramm schreiben, das diesen Befehl realisiert.
z.B. ADD i: $c(0) := c(0) + c(i)$; $b := b + 1$;

- Suche die Speicheradresse bin(i) auf dem Speicherband.
  Ist Adresse bin(i) nicht zu finden, so ist $c(i) = 0$.
- Kopiere bin($c(i)$) aufs Arbeitsband.
- Falls $c(i)$ noch gar nicht auf dem Speicherband ist, füge #bin(i)#0# an.
- „Addiere bin($c(0)$) und bin($c(i)$)“ (Schulmethode).
- Kopiere das Ergebnis aufs Speicherband in den Platz für $c(0)$.
  Gegebenenfalls muss der Inhalt des Speicherbandes verschoben werden.
- Befehlszähler um 1 erhöhen.

Dies funktioniert für jeden unserer Registermaschinenbefehle. Damit können wir auch das Registermaschinenprogramm in eine δ-Tabelle einer deterministischen Turingmaschine übersetzen.
Die Laufzeit zu überprüfen ist länglich, aber elementar. Die Laufzeit ergibt sich im wesentlichen durch die Speicher-Schieberei. □

Satz 1.11 *Jede $t(n)$-zeitbeschränkte deterministische 1-Band-TM kann durch eine $\mathcal{O}(t(n))$-zeitbeschränkte RAM simuliert werden.*

BEWEIS: siehe Übung. □

RAM, deterministische 1-Band-TM, deterministische k-Band-TM, Halbband-TM, 2D-Band-TM, k-Kopf-TM beschreiben alle dieselbe Klasse von „Berechenbarkeit“. μ-rekursive Funktionen (Berechenbarkeitsmodell ohne Maschinen) sind ebenfalls genau gleich den maschinellen Berechnungsmodellen.
Ebenso resultieren viele weitere Modelle für „Berechenbarkeit“ (Vektoradditionssysteme, Chomsky-Typ-0-Grammatiken) auf dieselbe Klasse von „Berechenbarkeit“.

1.4 Die Church-Turing-These

1900	David Hilbert:	Benennung von 10 Problemen der Mathematik, deren Lösung diese revolutionieren würden.
1931	Kurt Gödel:	Zeigt, dass Beweise nicht automatisch generiert werden können.
1936	Alan Turing:	Turingmaschine und Beweis der Unentscheidbarkeit des Halteproblems.

Die Church-Turing-These:

„Die im intuitiven Sinn berechenbaren Funktionen sind genau die, die durch Turingmaschinen berechenbar sind.“

Damit:
Berechenbar zu sein ist eine Eigenschaft einer Funktion.

1.5 Universelle Turingmaschinen

Bislang: Unsere TM können genau eine Aufgabe lösen (special purpose).
Darstellung von (1-Band-) TM M: o.B.d.A. $\Gamma = \{0, 1, B\}$, $Q = \{q_1, \ldots, q_m\}$, Startzustand: q_1, $F = \{q_m\}$.
Diese TM (und deren δ-Übergänge $\delta(q_i, X) = (q_j, Y, D)$) soll nun durch $\{0, 1, \#\}$ kodiert werden.
$X, Y \in \Gamma$ wird kodiert durch $0 \mapsto 00$, $1 \mapsto 01$, $B \mapsto 11$.
$D \in \{R, L, N\}$ wird kodiert durch $R \mapsto 00$, $N \mapsto 01$, $L \mapsto 11$.
q_i wird kodiert durch $1^i = \underbrace{11\ldots1}_{i \text{ viele}}$
Beispiel: $\delta(q_3, 1) = (q_2, B, L) \mapsto 111\#01\#11\#11\#11 =:\text{code}(\delta(q_3, 1) = (q_2, B, L))$
Kodierung der TM mittels Endzustand und der gesamten δ-Tabelle:
$\#\#\#\overbrace{1^m}^{\text{akz. Endzustand}}\#\#\text{code(erster Eintrag der } \delta\text{-Tabelle)}\#\#\text{code(zweiter Eintrag der } \delta\text{-Tabelle)}\#\#\ldots$
$\ldots\#\#\text{code(letzter Eintrag der } \delta\text{-Tabelle)}\#\#\#$
Kodierung der TM M wird auch als „Gödelnummer“ von M bezeichnet.
Mit $0 \mapsto 00$, $1 \mapsto 01$, $\# \mapsto 11$ sogar 0-1-Folge pur auch als natürliche Zahl zu interpretieren. Man kann also jedes Turingmaschinenprogramm auf eine natürliche Zahl abbilden. Es gibt abzählbar unendlich viele Turingmaschinenprogramme, also gibt es auch nur abzählbar unendlich viele berechenbare Funktionen.
Schreibweise: $\langle M \rangle$ bezieht sich auf die Gödelnummer der TM M. M ist dabei die tatsächliche Turingmaschine und $\langle M \rangle$ der Bauplan dieser Maschine.

Eine TM $\tilde{M}$ heißt universell, wenn sie sich bei Eingabe $\langle M \rangle x$, $x \in \{0, 1\}^*$ so verhält, wie M gestartet mit Eingabe x.

Satz 1.12 *Es gibt eine universelle deterministische 2-Band-TM $\tilde{M}$, die jede $t(n)$-zeitbeschränkte und $s(n)$-platzbeschränkte deterministische 1-Band-TM M auf Platz $\mathcal{O}(s(n))$ in Zeit $\mathcal{O}(t(n))$ simuliert, falls M hält.*

BEWEIS: M habe Bandalphabet $\Gamma = \{0, 1, B\}$
$\tilde{M}$: zu Beginn steht auf Band 1 $\langle M \rangle x$
kopiere $\langle M \rangle$ auf Band 2, lösche $\langle M \rangle$ auf Band 1.
Band 1: $\overset{\overset{\text{☺}}{\downarrow}}{\ldots BBx_1x_2\ldots x_nBB\ldots}$ Band 2: $\#\#\#\overbrace{\qquad}^{\text{Kodierung von aktuellem Zustand}}\overbrace{\#\#\#\ldots\#\#\#}^{\langle M \rangle}$
Der Zustand der simulierten Turingmaschine muss explizit auf dem Band stehen, da die Anzahl der Zustände nicht a priori bekannt ist und entsprechend nicht im Zustand der simulierenden TM gemerkt werden kann.

Festsetzung: $|\langle M \rangle| = \mathcal{O}(1)$

- Finde zum aktuellen Zustand q_i und Zeichen X auf Band 1 an der dortigen Kopfposition auf Band 2 die Stelle $\#1^i\#\text{code}(X)\#1^j\#\text{code}(Y)\#\text{code}(D)$.
  (viel Arbeit, aber da $|\langle M \rangle| = \mathcal{O}(1)$, ist es auch nur konstant.)
- aktualisiere den aktuellen Zustand zu 1^j, ersetze auf Band 1 das X durch Y und gehe auf Band 1 in Richtung D.

Da $|\langle M \rangle| = \mathcal{O}(1)$ gelten die im Satz postulierten Zeit- und Platzschranken.
Platzbedarf:
Band 1 benötigt genau so viele Zellen, wie M gestartet mit Eingabe x, Band 2 benötigt nur $\mathcal{O}(1)$ viele Zellen.
Zeitbedarf:
$\underbrace{(\#\text{ Schritte von } M \text{ gestartet mit Eingabe } x)}_{t(|x|)} \cdot \underbrace{(\text{Zeitaufwand zur Verwaltung auf Band 2})}_{\mathcal{O}(1)} = \mathcal{O}(t(|x|))$ □
Beobachtung: Sei $x \in \{0, 1, \#\}^*$.
Es ist entscheidbar, ob $x = \langle M \rangle$ für eine deterministische 1-Band-TM M ist (Syntaxanalyse).
Code $= \{x \mid \exists \text{ deterministische 1-Band-TM } M \text{ mit } x = \langle M \rangle\}$ ist entscheidbar. $|\text{Code}| = |\mathbb{N}|$.

1.6 Unentscheidbare Probleme

Gibt es Grenzen für das, was Turingmaschinen berechnen können? Wir werden sehen, dass es solche Grenzen gibt, die sehr wichtige Probleme betreffen. Genauer gesagt: Wir werden zeigen, dass Probleme, von denen wir uns wünschten, dass sie lösbar wären, algorithmisch nicht lösbar sind!

Die Gödelnummer einer Turingmaschine M besteht nur aus Buchstaben aus $\{0, 1, \#\}$. Durch die Abbildung

$$0 \mapsto 00$$
$$1 \mapsto 01$$
$$\# \mapsto 11$$

dieser Zeichen kommen wir sogar nur mit dem Alphabet $\{0, 1\}$ aus. Wir können also jede 1-Band-Turingmaschine M eindeutig durch eine Zeichenkette $\langle M \rangle \in \{0, 1\}^*$ kodieren. Da der gleiche Kodierungstrick für beliebige Bandalphabete Σ funktioniert, gehen wir im folgenden davon aus, dass für alle Turingmaschinen, die uns begegnen, $\Sigma = \{0, 1\}$ ist.

$L \subseteq \Sigma^*$, L heißt Sprache. Man nennt L Problem, wenn das sogenannte Wortproblem gemeint ist, das heißt die Frage „$x \in L$?“. „Probleme“ sind Mengen.
Sei χ_L die charakteristische Funktion von L, das heißt
$\chi_L(x) := \begin{cases} 0 & \text{falls } x \notin L \\ 1 & \text{falls } x \in L \end{cases}$
auch auffassbar als $f_L : \mathbb{N} \to \{0, 1\}$.
Wie viele solcher Funktionen gibt es? $2^{|\mathbb{N}|}$ = überabzählbar unendlich viele.
Wie viele TM gibt es? Nur $|\mathbb{N}|$ viele, also nur abzählbar unendlich viele.
$\Rightarrow$ Es gibt Funktionen, für die es keine TM gibt, die also „nicht berechenbar“ sind.

Definition 1.13 (Halteproblem) *Die Sprache*

$$H = \{\langle M \rangle w \mid M \textit{ ist deterministische 1-Band-TM, die, gestartet mit Eingabe } w\textit{, hält}\}$$

heißt **(allgemeines) Halteproblem**.

H ist rekursiv aufzählbar. Eine universelle TM „zählt H auf“.

Satz 1.14 (Turing, 1936) *H ist unentscheidbar. (oder alternativ: H ist nicht rekursiv)*

BEWEIS: Wir nehmen an, dass H entscheidbar ist, das heißt es gibt eine TM M_H, die H entscheidet. M_H hält auf jede Eingabe und man kann am Endzustand ablesen, ob $\langle M \rangle w \in H$ gilt oder nicht.
Wir schreiben das Programm (die TM) M_{schlau}:

```
1 Die Eingabe sei y;
  /* y = ⟨M⟩ kann durch Syntaxanalyse bestimmt werden                    */
2 wenn y = ⟨M⟩ dann
3 |   Entscheide mittels M_H, ob ⟨M⟩⟨M⟩ ∈ H;
4 |   wenn ⟨M⟩⟨M⟩ ∈ H dann
5 |   |   Schreibe nach rechts unendlich viele 1en auf das Band;
6 |   Ende
7 Ende
8 bleibe stehen;
```

M_{schlau} wird gestartet mit Eingabe $\langle M_{schlau}\rangle$

Fall a) M_{schlau} gestartet mit Eingabe $\langle M_{schlau}\rangle$ hält
$\Rightarrow \langle M_{schlau}\rangle\langle M_{schlau}\rangle \in H$
$\Rightarrow$ wir gehen in den ja-Zweig und M_{schlau} geht in Endlosschleife
$\Rightarrow M_{schlau}$ gestartet mit Eingabe $\langle M_{schlau}\rangle$ hält nicht $\lightning$

Fall b) M_{schlau} gestartet mit Eingabe $\langle M_{schlau}\rangle$ hält nicht
$\Rightarrow \langle M_{schlau}\rangle\langle M_{schlau}\rangle \notin H$
$\Rightarrow$ wir gehen in den nein-Zweig und M_{schlau} hält
$\Rightarrow M_{schlau}$ gestartet mit Eingabe $\langle M_{schlau}\rangle$ hält $\lightning$

$\Rightarrow M_H$ gibt es nicht $\Rightarrow H$ ist unentscheidbar. □

Satz 1.15

(a) H ist rekursiv aufzählbar.

(b) $\overline{H} = \{0,1\}^* \setminus H$ ist nicht rekursiv aufzählbar.

BEWEIS:

(a) Die universelle TM aus Satz 1.12 ist der rekursive Aufzähler von H.
$\langle M\rangle w \in H \Leftrightarrow$ die universelle TM gestartet mit Eingabe $\langle M\rangle w$ hält.

(b) Annahme: $\overline{H}$ ist rekursiv aufzählbar durch die TM $\tilde{M}$, dann betrachte folgende TM:
Entscheider für H:

```
   /* Ziel: Entscheide ob y = ⟨M⟩w ∈ H ist                              */
 1 Die Eingabe sei y;
   /* y = ⟨M⟩w kann durch Syntaxanalyse entschieden werden              */
 2 wenn y = ⟨M⟩w dann
 3 |  Führe gleichzeitig/parallel aus und stoppe beide Simulationen, wenn eine der
   |   simulierten TM's stoppt:
 4 |  • Starte auf Band 1 die universelle TM mit der Eingabe ⟨M⟩w;
   |  /* Hält für ⟨M⟩w ∈ H                                             */
 5 |  • Starte auf Band 2 die TM M̃ mit Eingabe ⟨M⟩w;
   |  /* Hält für ⟨M⟩w ∉ H                                             */
 6 |  wenn Simulation auf Band 1 gestoppt hat dann
 7 |  |  Halte akzeptierend;
 8 |  Ende
 9 Ende
10 Halte verwerfend;
```

$\Rightarrow$ entweder TM auf Band 1 oder TM auf Band 2 hält
$\Rightarrow$ diese TM entscheidet H
$\Rightarrow H$ ist entscheidbar $\lightning$
$\Rightarrow \tilde{M}$ gibt es nicht

□

Definition 1.16 (Initiales Halteproblem) *Die Sprache*

$$H_\varepsilon = \{\langle M\rangle \mid M \text{ ist deterministische 1-Band-TM, die, gestartet mit Eingabe } \varepsilon \text{ (dem leeren Wort), hält}\}$$

heißt **initiales Halteproblem**.

Satz 1.17 *H_ε ist unentscheidbar.*

BEWEIS:
Wir zeigen: Wäre H_ε entscheidbar, dann könnte man einen Entscheider für H angeben.
Wir nehmen an: H_ε ist entscheidbar.
Wir wollen zeigen: Dann ist H auch entscheidbar.
TM $\tilde{M}$ entscheide H_ε.
Gegeben: $\langle M \rangle w$ und die Frage „$\langle M \rangle w \in H$?“
Ziel: Programm zur Beantwortung der Frage angeben, das $\tilde{M}$ benutzen darf.

FESTE_MASCHINE$_{\langle M \rangle w}$:

```
1 Die Eingabe sei x;
2 Starte M mit Eingabe w;
3 wenn x = ε dann
4 |   halte;
5 sonst
  |   /* Es ist egal was hier passiert.                                  */
6 |   halte;
7 Ende
```

$\langle M \rangle w \in H \Rightarrow$ wir erreichen (3) $\Rightarrow$ *FESTE_MASCHINE*$_{\langle M \rangle w} \in H_\varepsilon$
$\langle M \rangle w \notin H \Rightarrow$ wir erreichen (3) nie $\Rightarrow$ *FESTE_MASCHINE*$_{\langle M \rangle w} \notin H_\varepsilon$

Es gilt damit $\langle$*FESTE_MASCHINE*$_{\langle M \rangle w}\rangle \in H_\varepsilon \Leftrightarrow \langle M \rangle w \in H$ (*)

Konstruiere nun TM die H entscheidet:
Wenn $\tilde{M}$ mit Eingabe $\langle$*FESTE_MASCHINE*$_{\langle M \rangle w}\rangle$ gestartet wird, können wir ihre Antwort für die Frage „$\langle M \rangle w \in H$?“ direkt übernehmen.

M_H:

```
1 Die Eingabe sei x;
2 wenn x ≠ ⟨M⟩w dann
3 |   halte „nicht akzeptierend“;
  |   /* Test ob Eingabe eine Gödelnummer enthält ist in endlicher Zeit möglich  */
4 Ende
5 Sei nun x = ⟨M⟩w;
6 Konstruiere ⟨FESTE_MASCHINE_⟨M⟩w⟩;
  /* Berechnet eine Fkt. f : {0,1}* → {0,1}*                              */
7 Entscheide mittels M̃, ob ⟨FESTE_MASCHINE_⟨M⟩w⟩ ∈ H_ε ist und gib die Antwort aus;
```

Dieses Programm entscheidet wegen (*) das Halteproblem H. ↯
$\Rightarrow \tilde{M}$ gibt es nicht. □

1.7 Reduktionen und der Satz von Rice

Definition 1.18 *Eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ heißt* **berechenbar**, *wenn es eine (1-Band-)TM M_f gibt, für die mit $x \in \{0,1\}^*$ gilt:*

- *Ist $f(x)$ definiert, so hält M_f gestartet mit Eingabe x und $f(x)$ steht am Ende auf dem Band als Ausgabe.*
- *Ist $f(x)$ undefiniert, so hält M_f gestartet mit Eingabe x nicht.*

Ist f **total**, *das heißt für alle $x \in \{0,1\}^*$ definiert und berechenbar, so heißt f* **total berechenbar** *(oder auch:* **total rekursiv***).*
Eine Sprache L ist **entscheidbar** *genau dann wenn die sogenannte charakteristische Funktion*

$$\chi_L : \{0,1\}^* \to \{0,1\} \text{ mit } \chi_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{falls } x \notin L \end{cases}$$

total berechenbar ist.
Eine Sprache L ist **rekursiv aufzählbar** *genau dann wenn die partielle Funktion*

$$\chi'_L : \{0,1\}^* \to \{0,1\} \text{ mit } \chi_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ \textit{undefiniert} & \text{falls } x \notin L \end{cases}$$

berechenbar ist.

Beispiel:

$$f(x) = \begin{cases} 1 & \text{falls am 12.3.2007 um 8:24 Uhr } x \text{ viele Fahrräder vor dem blauen Hochhaus geparkt waren} \\ 0 & \text{sonst.} \end{cases}$$

Die Funktion f ist total berechenbar, denn es gibt ein $x_0 \in \mathbb{N}_0$ sodass

$$f(x) = f_{x_0}(x) = \begin{cases} 1 & x = x_0 \\ 0 & x \neq x_0. \end{cases}$$

Wir wissen zwar nicht welche Turingmaschine die Funktion f berechnet, aber für jedes x_0 gibt es eine Turingmaschine die f_{x_0} berechnet. Eine dieser Turingmaschinen berechnet f und somit ist f berechenbar. Dabei wäre sogar irrelevant, ob sich der Messzeitpunkt in der Vergangenheit oder in der Zukunft befindet.

Sehen wir uns nun den Beweis von Satz 1.17 noch einmal an. Wir können ihn in mehrere Teile gliedern. Was wir dort gemacht haben, ist, dass wir mit Hilfe eines (nicht existierenden) Entscheiders für H_ε einen Entscheider für H programmiert haben. Dabei bekommt der H_ε-Entscheider eine Turingmaschine in Form ihrer Gödelnummer (ihres Programms) als Eingabe, so dass die Antwort des H_ε-Entscheiders direkt (ohne Modifikationen) die Antwort auf die Frage „$\langle M \rangle w \in H$?“ ist.
Das wollen wir im folgenden abstrahieren/verallgemeinern.

Definition 1.19 *Seien L_1 und L_2 Sprachen über dem Alphabet $\{0,1\}$.*
Eine **Reduktion** *ist eine total berechenbare Funktion $f : \{0,1\}^* \to \{0,1\}^*$, für die gilt:*

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

Wir schreiben „$L_1 \leq L_2$“und sagen „L_1 wird (mittels f) auf L_2 reduziert“.

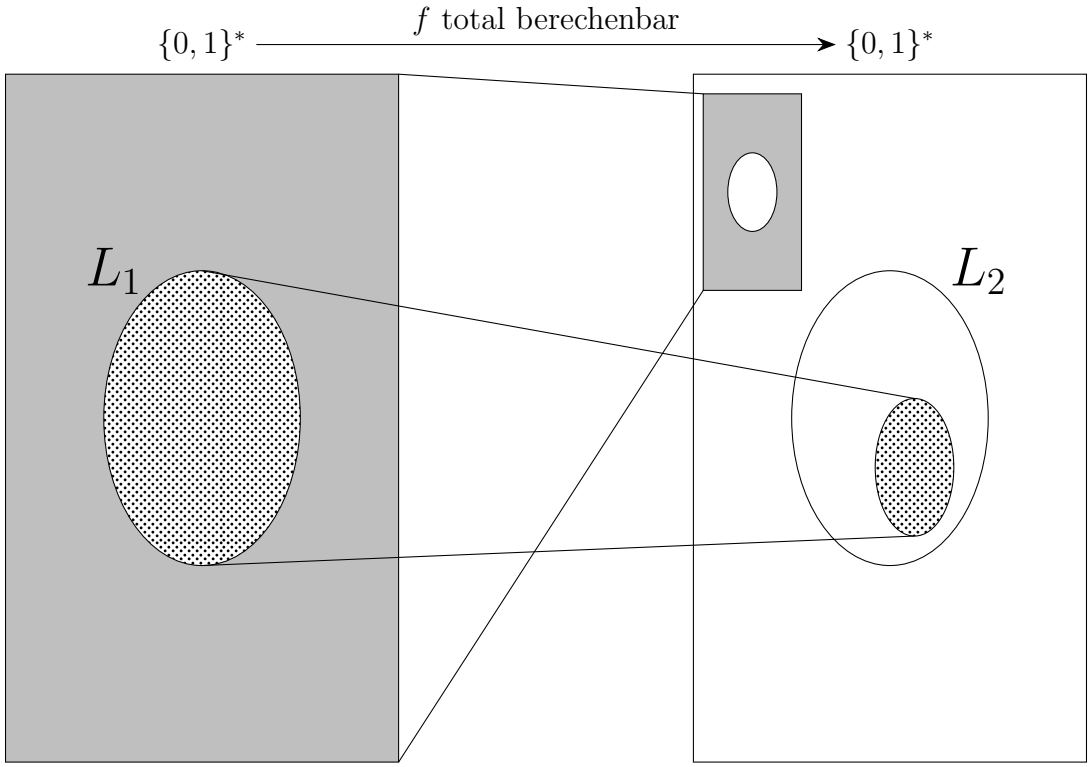


Abbildung 5: Wirkungsweise einer Reduktion (das Spiegelei-Bild)

Beispiel 1.20 *Die folgende total berechenbare Funktion f ist eine solche* **Reduktionsfunktion**, *die zeigt, dass $H \leq H_\varepsilon$ ist:*

$$f(x) = \begin{cases} \langle \text{FESTE\_MASCHINE}_{\langle M\rangle w}\rangle & \textit{falls } x \textit{ von der Form } \langle M\rangle w \textit{ ist (entscheidbar mit Syntaxanalyse)} \\ 0 & \textit{sonst} \end{cases}.$$

Ganz wichtig ist hierbei, dass die Eigenschaft „x von der Form $\langle M\rangle w$“ entscheidbar ist! Dadurch kann nämlich f durch die folgende Turingmaschine berechnet werden:
$H\_auf\_H_\varepsilon\_Reduzierer$:

```
1 Die Eingabe sei x;
2 wenn x von der Form ⟨M⟩w dann
3 |  gib ⟨FESTE_MASCHINE_⟨M⟩w⟩ aus;
  |  /* Test ob Eingabe eine Gödelnummer und eine Eingabe enthält ist in endlicher
  |     Zeit möglich.                                                          */
4 sonst
5 |  gib 0 aus;
6 Ende
```

f ist $\underbrace{\textit{total}}_{\textit{Muss vermerkt sein.}}$ $\underbrace{\textit{berechenbar}}_{\textit{Muss man nachprüfen.}}$.

Sei FESTE_MASCHINE$_{\langle M\rangle w}$*:*

```
1 Die Eingabe sei y;
2 Starte mit universeller Turingmaschine die TM M mit Eingabe w;
3 Halte;
```

Dann gilt:
$x \in H$
$\Rightarrow x = \langle M \rangle w$ *und* M *gestartet mit Eingabe* w *hält*
$\Rightarrow f(x) = \langle \text{FESTE\_MASCHINE}_{\langle M \rangle w} \rangle$ *und* $\text{FESTE\_MASCHINE}_{\langle M \rangle w}$ *gestartet mit Eingabe* ε *hält*
$\Rightarrow f(x) \in H_\varepsilon$

$$x \notin H \Rightarrow \begin{cases} x \neq \langle M \rangle w \Rightarrow f(x) = 0 \notin H_\varepsilon. \\ x = \langle M \rangle w \text{ und } M \text{ gestartet mit Eingabe } w \text{ hält nicht} \\ \quad \Rightarrow f(x) = \langle \text{FESTE\_MASCHINE}_{\langle M \rangle w} \rangle \\ \qquad \text{und } \text{FESTE\_MASCHINE}_{\langle M \rangle w} \text{ gestartet mit Eingabe } \varepsilon \text{ hält nicht} \\ \quad \Rightarrow f(x) \notin H_\varepsilon. \end{cases}$$

$\Rightarrow \Big(x \in H \iff f(x) \in H_\varepsilon \Big)$
Damit gilt insgesamt $H \leq H_\varepsilon$.

Beispiel 1.21 *Wir untersuchen nun die folgende Sprache:*

$$HW := \{\langle C \rangle \,|\langle C \rangle \text{ ist ein C-Programm, das, wenn man es als „Maschine“ laufen lässt, „Hallo Welt“ ausgibt und hält}\}.$$

Beispielprogramm:
C_1*:*

```
1 main()
2 {
3     printf("Hallo Welt");
4 }
```

$\langle C_1 \rangle \in HW$.
HW *ist unentscheidbar.*
Wir zeigen $H \leq HW$ *um die Unentscheidbarkeit von* HW *zu beweisen.*
Gegeben: Frage „Ist $\langle M \rangle w \in H$*“?*

$\text{FESTES\_PROGRAMM}_{\langle M \rangle w}$*:*

```
1 main()
2 {
3     printf("Hallo Welt");
4     extern(M,w);
5 }
```

$$f(x) = \begin{cases} \langle \text{FESTES\_PROGRAMM}_{\langle M \rangle w} \rangle & x \text{ von der Form } \langle M \rangle w \text{ (mit Syntaxanalyse entscheidbar)} \\ 0 & \text{sonst} \end{cases}.$$

f *ist total berechenbar.*

$x \in H \Rightarrow x = \langle M \rangle w$ *und* M *gestartet mit Eingabe* w *hält*
$\Rightarrow f(x) = \langle \text{FESTES\_PROGRAMM}_{\langle M \rangle w} \rangle$ *und nach Start dieses Programms erreichen wir „}“*
$\Rightarrow f(x) \in HW$.

$$x \notin H \Rightarrow \begin{cases} x \neq \langle M \rangle w \Rightarrow f(x) = 0 \notin HW \\ x = \langle M \rangle w \text{ und } M \text{ gestartet mit Eingabe } w \text{ hält nicht} \\ \quad \Rightarrow f(x) = \langle \text{FESTES\_PROGRAMM}_{\langle M \rangle w} \rangle \text{ und} \\ \qquad \text{nach Start dieses Programms erreichen wir „}\}\text{“ nicht} \\ \quad \Rightarrow f(x) \notin HW. \end{cases}$$

Es gilt somit insgesamt $H \leq HW$.

Satz 1.22 *Seien L_1 und L_2 Sprachen, und sei $L_1 \leq L_2$ mittels total berechenbarer Funktion f.*

(a) L_2 entscheidbar $\Rightarrow$ L_1 entscheidbar.

(b) L_2 rekursiv aufzählbar $\Rightarrow$ L_1 rekursiv aufzählbar.

BEWEIS: Der Beweis ist jetzt noch eine einfache Programmieraufgabe:

(a) Gegeben ist die Frage „$x \in L_2$?“und eine Entscheidungsturingmaschine M_{L_2} für L_2. Der Entscheidungsalgorithmus für L_1 besteht darin, $f(x)$ zu berechnen und als Eingabe für M_{L_2} zu benutzen. Die Antwort von M_{L_2} auf die Frage „$f(x) \in L_2$?“ist die Antwort auf die ursprüngliche Frage.

(b) Analog: nur dass wir jetzt die Antwort „nein“gar nicht mehr benötigen, sondern uns nur noch für Halten und Nicht-Halten interessieren. □

Die folgende Konsequenz bekommen wir direkt aus Satz 1.22, indem man die Kontraposition der Aussagen bildet ($(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$).

Korollar 1.23 *Seien L_1 und L_2 Sprachen, und sei $L_1 \leq L_2$ mittels total berechenbarer Funktion f.*

(a) L_1 unentscheidbar $\Rightarrow$ L_2 unentscheidbar.

(b) L_1 nicht rekursiv aufzählbar $\Rightarrow$ L_2 nicht rekursiv aufzählbar.

Was wir als „Drum-Herum-Programmieren“ bezeichnet hatten, lässt sich derart verallgemeinern, dass man einen sehr mächtigen Satz beweisen kann, der, etwas ungenau formuliert, besagt, dass all nicht-trivialen semantischen Eigenschaften von Turingmaschinen-Programmen und damit Computer-Programmen unentscheidbar sind (Die Semantik eines Programms ist seine Bedeutung; für jemanden ohne Wissen über Turingmaschinen ist die Gödelnummer einer Turingmaschine lediglich eine nichtssagende 0-1-Folge).
Dazu bezeichnen wir im Folgenden mit

$$\mathcal{R} = \{f \mid f : \{0,1\}^* \to \{0,1\}^* \text{ ist berechenbar}\}$$

die Menge der berechenbaren Funktionen.
Zu einer Teilmenge S, $S \subseteq \mathcal{R}$, bezeichne

$$Prog(S) = \{\langle M \rangle \mid M \text{ ist deterministische 1-Band TM, die eine Funktion } f \in S \text{ berechnet}\}$$

die Menge *aller* (!ganz wichtig) Programme, das heißt Gödelnummern von Turingmaschinen, welche die Funktionen aus S berechnen.

Beispiel:
$S = \{f : n \mapsto n^2\}$, $|S| = 1$, also $Prog(S) = \{\langle M \rangle \mid M$ berechnet Quadratfunktion$\}$ und $|Prog(S)| = \infty$.

Die Mengen $Prog(\emptyset)$ und $Prog(\mathcal{R})$ sind entscheidbar durch einfache Syntaxanalyse, wie wir sie schon kennengelernt hatten. Die Eigenschaften der Wörter dieser beiden Mengen werden als *trivial* bezeichnet. Wir zeigen nun, dass alle anderen Mengen $Prog(.)$ unentscheidbar sind!

Satz 1.24 (Rice, 1953) *Sei S, $S \subseteq \mathcal{R}$, mit $\emptyset \neq S \neq \mathcal{R}$. Dann ist $Prog(S)$ nicht entscheidbar.*

BEWEIS: Sei u die für keine Eingabe definierte Funktion. u ist berechenbar im Sinne der Definition 1.18, zum Beispiel durch die (ganz konkrete) Turingmaschine M_u, die sofort in eine Endlosschleife geht. Mit anderen Worten: $u \in \mathcal{R}$ und $\langle M_u \rangle \in Prog(\mathcal{R})$.
Beachten Sie, dass schon $Prog(\{u\})$ unendlich viele Programme enthält.

Entweder ist $u \notin S$ („Fall (a)“) oder $u \in S$ („Fall (b)“).

(a) $u \notin S$ und damit auch $\langle M_u \rangle \notin Prog(S)$. Wir zeigen: $H_\varepsilon \leq Prog(S)$.

Da $S \neq \emptyset$, gibt es eine Funktion $s \in S$, die durch eine (wieder ganz konkrete) (1-Band-)Turingmaschine M_s mit $\langle M_s \rangle \in Prog(S)$ berechnet werden kann. (An dieser Stelle wird der Beweis *nicht-konstruktiv*. Niemand berechnet für uns dieses s. Es fällt beinahe vom Himmel.) Nun schreiben wir ein neues Programm:

$DRUMHERUM_{\langle M \rangle}$:

```
1 Die Eingabe sei y;
2 Starte M mit leerem Band auf einem Hilfsband;
3 Starte M_s mit y;
```

Als Reduktionsfunktion nehmen wir

$$f(x) = \begin{cases} \langle DRUMHERUM_{\langle M \rangle} \rangle & \text{falls } x = \langle M \rangle \\ \langle M_u \rangle & \text{falls } x \neq \langle M \rangle \end{cases}$$

f ist total berechenbar via Syntaxanalyse.

$$\begin{aligned} x \in H_\varepsilon &\Rightarrow x = \langle M \rangle \text{ und } M \text{ gestartet mit Eingabe } \varepsilon \text{ (dem leeren Wort) hält} \\ &\Rightarrow f(x) = \langle DRUMHERUM_{\langle M \rangle} \rangle \text{ und } DRUMHERUM \text{ berechnet } s \\ &\Rightarrow f(x) \in Prog(S) \end{aligned}$$

$$x \notin H_\varepsilon \Rightarrow \begin{cases} x = \langle M \rangle \text{ und } M \text{ gestartet mit Eingabe } \varepsilon \text{ (dem leeren Wort) hält nicht} \\ \qquad \Rightarrow f(x) = \langle DRUMHERUM_{\langle M \rangle} \rangle \text{ und } DRUMHERUM \text{ berechnet } u \\ \qquad \Rightarrow f(x) \notin Prog(S) \\ x \text{ ist nicht von der Form } \langle M \rangle \Rightarrow f(x) = \langle M_u \rangle \notin Prog(S) \end{cases}$$

(b) $u \in S$ und damit auch $\langle M_u \rangle \in Prog(S)$. Wir zeigen: $\overline{H_\varepsilon} \leq Prog(S)$.

Da $S \neq \mathcal{R}$, gibt es eine Funktion $s \notin S$, die durch eine (wieder ganz konkrete) (1-Band-)Turingmaschine M_s mit $\langle M_s \rangle \notin Prog(S)$ berechnet werden kann. Wir benutzen $DRUMHERUM_{\langle M \rangle}$ und f aus Fall (a). f ist auch hier total berechenbar via Syntaxanalyse.

$$x \in \overline{H_\varepsilon} \Rightarrow \begin{cases} x = \langle M \rangle \text{ und } M \text{ gestartet mit Eingabe } \varepsilon \text{ (dem leeren Wort) hält nicht} \\ \qquad \Rightarrow f(x) = \langle DRUMHERUM_{\langle M \rangle} \rangle \text{ und } DRUMHERUM \text{ berechnet } u \\ \qquad \Rightarrow f(x) \in Prog(S) \\ x \text{ ist nicht von der Form } \langle M \rangle \Rightarrow f(x) = \langle M_u \rangle \in Prog(S) \end{cases}$$

$$\begin{aligned} x \notin \overline{H_\varepsilon} &\Rightarrow x = \langle M \rangle \text{ und } M \text{ gestartet mit Eingabe } \varepsilon \text{ (dem leeren Wort) hält} \\ &\Rightarrow f(x) = \langle DRUMHERUM_{\langle M \rangle} \rangle \text{ und } DRUMHERUM \text{ berechnet } s \\ &\Rightarrow f(x) \notin Prog(S) \end{aligned}$$

□

1.8 Rekursive Aufzählbarkeit

Bislang wurde die Mengeneigenschaft *rekursiv aufzählbar* einfach nur als abstrakte Bezeichnung benutzt. Im folgenden werden wir wenigstens den Bestandteil *aufzählbar* erklären.

Satz 1.25 *Sei L eine unendliche Sprache.*
L ist rekursiv aufzählbar $\iff$ es gibt eine total berechenbare *surjektive Funktion $g : \{0,1\}^* \to L$.*

BEWEIS:
„$\Leftarrow$“:
Sei $g : \{0,1\}^* \to L$ eine *total berechenbare* surjektive Funktion, die von der 1-Band-Turingmaschine M_g berechnet wird. M_g hält also für jede beliebige Eingabe, produziert nur Elemente aus L und für jedes $x \in L$ existiert ein $y \in \{0,1\}^*$ sodass M_g gestartet mit Eingabe y die Ausgabe x produziert, da g surjektiv ist. Wir wollen nun eine Turingmaschine M programmieren, die genau für die Elemente aus L hält und dabei M_g aufrufen darf.
M (M bekommt als Eingabe x und sucht nach y mit $g(y) = x$):

```
1 Sei x die Eingabe;
2 für alle y ∈ {0,1}* tue
  |  /* Durchlaufe {0,1}* systematisch                                  */
  |  /* Beispielsweise mit wachsender Länge ε,0,1,00,01,10,11,000,001,... */
3 |  Starte M_g mit Eingabe y;
4 |  wenn Ausgabe von M_g mit x übereinstimmt dann
5 |  |  akzeptiere und halte;
6 |  Ende
7 Ende
```

Wenn $x \in L$, dann gibt es ja ein $y \in \{0,1\}^*$ mit $g(y) = x$ und dieses y wird gefunden, da M_g immer hält.
Wenn $x \notin L$, dann haben wir hier eine Endlosschleife.
Insgesamt gilt: M, gestartet mit Eingabe x, hält $\iff x \in L$.
$\Rightarrow L$ ist rekursiv aufzählbar.$\surd$

„$\Rightarrow$“:
Sei L eine rekursiv aufzählbare Sprache.
Somit gibt es eine Turingmaschine M, sodass M gestartet mit Eingabe $x \in L$ akzeptierend hält und M gestartet mit Eingabe $x \notin L$ entweder nicht hält oder nicht-akzeptierend hält.
Ziel: M_g programmieren, sodass

(1) M_g für jede Eingabe hält,

(2) M_g nur Wörter aus L ausgibt und

(3) M_g jedes Wort aus L ausgibt.

„M gestartet mit Eingabe x hält“ heißt auch „Es gibt ein $t \in \mathbb{N}$, sodass M gestartet mit Eingabe x nach t Schritten stoppt“. Als Eingabe y für M_g werden Paare (x,t) betrachtet. M_g führt dabei t Schritte von M gestartet mit Eingabe x aus. Wenn dabei eine akzeptierende Endkonfiguration erreicht wurde, wird x ausgegeben, ansonsten ein festes $x_{fix} \in L$. Da M_g ja ein $y \in \{0,1\}^*$ als Eingabe bekommt, müssen wir uns nun überlegen, wie wir daraus ein Paar $(x,t) \in \{0,1\}^* \times \mathbb{N}$ machen. Das geht beispielsweise wie folgt:

$$aus\_eins\_mach\_zwei(y) = \begin{cases} (a_1 \dots a_n, t) & \text{falls } y = a_1 \dots a_n 0 \underbrace{1 \dots 1}_{t \text{ viele}} \\ (0,1) & \text{sonst} \end{cases}$$

aus_eins_mach_zwei ist total berechenbar und surjektiv.

Instruktive Beispiele sind:

$$\begin{aligned}
&\mathit{aus_eins_mach_zwei}(011010111) = (01101, 3),\\
&\mathit{aus_eins_mach_zwei}(011) = (\varepsilon, 2),\\
&\mathit{aus_eins_mach_zwei}(1111) = (0, 1),\\
&\mathit{aus_eins_mach_zwei}(0110) = (011, 0).
\end{aligned}$$

Man sieht, wir können sagen, dass wir in y die rechteste 0 als Komma interpretieren (und den Fall abfangen, dass y gar keine 0 enthält).
M_g:

```
1 Die Eingabe sei y ∈ {0,1}*;
2 (x,t):=aus_eins_mach_zwei(y);
3 Simuliere t Schritte von M gestartet mit Eingabe x;
4 wenn M eine akzeptierende Endkonfiguration erreicht hat dann
5 |  Gib x aus;
6 sonst
7 |  Gib x_fix aus;
8 Ende
```

M_g hält für jede Eingabe und gibt nur Wörter aus L aus. Außerdem gibt es für jedes beliebige Wort $x \in L$ ein Urbild y. Sei t die Anzahl Schritte bis M gestartet mit Eingabe x hält. Dann ist die Ausgabe von M_g gestartet mit Eingabe $y = x01^t$ wieder x. Damit erfüllt M_g alle geforderten Eigenschaften. □

Satz 1.25 kann man noch verschärfen indem man surjektiv durch bijektiv ersetzt.

Man kann dazu $x \in \{0,1\}^*$ als Zahl zur Basis 2 interpretieren: $(1x)_2$.

Nachfolgende Sprechweisen werden ab hier benutzt:

- Eine Menge $\mathcal{L}$ von Sprachen bezeichnet man als ***Sprachklasse*** oder auch als ***Sprachfamilie***.
  Beispiele sind $\mathcal{E} = \{L \mid L \text{ ist entscheidbar}\}$, die Klasse der entscheidbaren Sprachen, und $\mathcal{L}_0 = \{L \mid L \text{ ist rekursiv aufzählbar}\}$, die Klasse der rekursiv aufzählbaren Sprachen.
- Wenn wir eine k-stellige Operation $op(.,\ldots,.)$ auf Sprachen haben, also eine Operation, die aus k Sprachen eine neue Sprache macht, dann ist eine Sprachklasse $\mathcal{L}$ genau dann ***gegen*** *op* ***abgeschlossen*** (alternative Sprechweise: *unter op abgeschlossen*), wenn gilt:

  $$L_1, \ldots, L_k \in \mathcal{L} \Rightarrow op(L_1, \ldots, L_k) \in \mathcal{L}.$$

  Beispielsweise sind die entscheidbaren Sprachen gegen die Vereinigung abgeschlossen:

  $$L_1, L_2 \in \mathcal{E} \Rightarrow L_1 \cup L_2 \in \mathcal{E}.$$

  Auch wissen wir, dass die rekursiv aufzählbaren Sprachen nicht gegen Komplementbildung abgeschlossen sind, da $H \in \mathcal{L}_0$ und $\overline{H} \notin \mathcal{L}_0$.

Eine Schwierigkeit beim Beweis von Abschlusseigenschaften für rekursiv aufzählbare Sprachen besteht darin, dass man dieses Nacheinander nicht machen kann. Stellen Sie sich vor, $x \notin L_1$ und $x \in L_2$. Würde zuerst überprüft, ob x in L_1 ist, würde das Verfahren in eine Endlosschleife laufen obwohl $x \in L_1 \cup L_2$ ist.

Zum Ziel führen hier zwei mögliche Programmiertechniken:

- Zum einen könnte man beide Maschinen gleichzeitig (bzw. parallel) auf zwei Bändern mit jeweils der Eingabe x laufen lassen. Sobald eine der beiden hält, hält die Maschine für die Vereinigung.

- Zum anderen könnte man wie ein Ein-Prozessor-Computer im Multitasking-Betrieb arbeiten: Jede Maschine bekommt ein Zeitfenster, während dessen sie rechnen darf, danach wird die Konfiguration gespeichert, und nun ist die nächste Maschine dran, deren abgespeicherte Konfiguration geladen wird, so dass deren Rechnung für die Dauer des Zeitfensters weitergeführt werden kann. Die Kombination der Maschinen lässt man also kontrolliert laufen, damit nicht eine Maschine endlos läuft, lässt man jede „ein bisschen“ laufen. Implizit ist dieser Trick auch im Beweis der Richtung „$\Rightarrow$“ von Satz 1.25 zu finden, nämlich in Form der Zeitschranke t.

Satz 1.26 *Seien L_1 und L_2 rekursiv aufzählbare Sprachen. Dann gilt:*

(1) $L_1 \cup L_2$ ist rekursiv aufzählbar.

(2) $L_1 \cap L_2$ ist rekursiv aufzählbar.

BEWEIS:
„Vereinigungsmaschine“ muss Verhalten von M_1 für L_1 und M_2 für L_2 „widerspiegeln“.
Problem:
Ist $x \notin L_1$, aber in L_2, kann nicht „erst M_1 mit x“ gestartet werden, da dies zu einer Endlosschleife führen kann.
Nach der Vorstellung des Parallellaufens und des Zeitscheibentricks ist nun klar, wie die Maschinen arbeiten müssen. Für die Vereinigung reicht es, wenn eine der beiden Maschinen hält, beim Durchschnitt müssen beide halten. □

Satz 1.27 *L ist entscheidbar $\iff$ L und $\overline{L}$ sind rekursiv aufzählbar.*

Um für rekursiv aufzählbare Sprachen zu zeigen, dass sie gegen komplizierte Operationen abgeschlossen sind, ist der Programmieraufwand zwar größer, aber die beiden genannten Tricks funktionieren noch immer. Eine Beispieloperation ist die Konkatenation „$\circ$“ (auch Produkt genannt).

$$L_1 \circ L_2 := \{w \mid \exists u \in L_1 \exists v \in L_2 : w = uv\}.$$

Merke: $L \circ \{\varepsilon\} = L$ und $L \circ \emptyset = \emptyset$!

2 Nichtdeterministische Turingmaschinen und das P-NP-Problem

2.0.1 Die Nichtdeterministische Turingmaschine (NTM)

Definition 2.1 *Eine* **nichtdeterministische 1-Band-Turingmaschine (1-Band-NTM) M** *wird beschrieben durch 6 Komponenten $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, deren Bedeutungen, bis auf δ, gleich denen der deterministischen Turingmaschine sind. (Analog für k-Band-(N)TM).*
Nun ist $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R, N\})$, wobei $\mathcal{P}(A)$ die Potenzmenge von A bezeichnet.
Startkonfiguration*: q_0x, $x \in \Sigma^*$ und x ist die Eingabe.*
Eine nichtdeterministische Turingmaschine M **akzeptiert** *x genau dann wenn es eine Rechnung $q_0x \vdash^* \alpha q \beta$ mit $q \in F$ gibt. M* **akzeptiert die Sprache** *$L(M) = \{x \mid M \text{ akzeptiert } x\}$*

Sei beispielsweise $(q', b, R) \in \delta(q, a)$, dann ist $\alpha q a \beta \vdash \alpha b q' \beta$ ein möglicher Konfigurationsübergang.

Beispiel: (Palindromerkennung)
2-Band-NTM: $M = (\{q_0, q_1, q_2\}, \Sigma, \Gamma, \delta, q_0, F)$ mit $\Gamma = \{0, 1, B\}, \Sigma = \{0, 1\}, F = \{q_2\}$
δ: $\forall a \in \{0, 1\}$:

$$\begin{aligned}\delta(q_0, (a, B)) &= \{(q_0, (a, a), (R, R)), (q_1, (a, a), (R, N))\}\\ \delta(q_1, (a, a)) &= \{(q_1, (a, a), (R, L))\}\\ \delta(q_1, (B, B)) &= \{(q_2, (B, B), (N, N))\}\end{aligned}$$

$L(M) = \{ww^R \mid w \in \{0, 1\}^* \setminus \{\varepsilon\}\}$, wobei w^R das gespiegelte Wort von w bezeichnet

Beispiel: (Zufallszahlengenerator)
1-Band-NTM: $M = (\{q_0, q_1\}, \Sigma, \Gamma, \delta, q_0, F)$ mit $\Gamma = \{0, 1, B\}, \Sigma = \{0, 1\}, F = \{q_1\}$
δ:

$$\delta(q_0, B) = \{(q_0, 0, R), (q_0, 1, R), (q_1, B, N)\}$$

TM M schreibt eine beliebige 0-1-Folge aufs Band. Wir sagen: M rät eine 0-1-Folge.

Rucksackproblem:

$$K = \{\underbrace{\langle a_1, a_2, \ldots, a_n, b\rangle}_{\text{Kodierung von } a_1,\ldots,a_n,b} \mid a_1, \ldots, a_n, b \in \mathbb{N}, \exists \alpha_1, \ldots, \alpha_n \in \{0, 1\} : \sum_{i=1}^{n} \alpha_i a_i = b\}$$

Eine NTM, die das Rucksackproblem verifiziert geht folgendermaßen vor („generate and test“):
Rate eine 0-1-Folge $\alpha_1, \ldots, \alpha_n$ und verifiziere $\sum_{i=1}^{n} \alpha_i a_i = b$.

Definition 2.2
M sei nichtdeterministische (k-Band-)TM.
Für $x \in L(M)$ gilt:

Zeitkomplexität (Laufzeit): $\mathbf{T_M(x)}$:= *Länge einer kürzesten akzeptierenden Rechnung von M gestartet mit Eingabe x.*

Platzkomplexität (Platzbedarf): $\mathbf{S_M(x)}$:= *geringster Platzbedarf einer akzeptierenden Rechnung von M gestartet mit Eingabe x.*

$T_M(n)$-Zeit- und $S_M(n)$-Platzbeschränktheit gilt analog zu den deterministischen Turingmaschinen:

$$\mathbf{T_M(n)} := \max\{T_M(x) | x \in L(M), |x| \leq n\}$$

$$\mathbf{S_M(n)} := \max\{S_M(x) | x \in L(M), |x| \leq n\}$$

$t, s : \mathbb{N} \to \mathbb{N}$: M ist **t(n)-zeitbeschränkt** *und* **s(n)-platzbeschränkt**, *falls für alle $n \in \mathbb{N}$: $T_M(n) \leq t(n)$ und $S_M(n) \leq s(n)$*

Satz 2.3 *Jede NTM M kann durch eine deterministische TM M' simuliert werden. Falls M $t(n)$-zeitbeschränkt ist, so ist M' $2^{\mathcal{O}(t(n))}$-zeitbeschränkt und $\mathcal{O}(t(n)^2)$-platzbeschränkt.*

BEWEIS: M sei gegeben. r sei die maximale Zahl an unmittelbaren Nachfolgekonfigurationen, die in einer Rechnung von M zur Auswahl stehen:

$$r = \max\{|\delta(q,a)| \mid q \in Q, a \in \Gamma\}.$$

Startkonfiguration von M: q_0x ($x \in \Sigma^*$, $|x| = n$)
Interpretation der möglichen Rechnungen als r-ärer Baum von Konfigurationen
(**Berechnungsbaum** der nichtdeterministischen Turingmaschine M, gestartet mit Eingabe x):

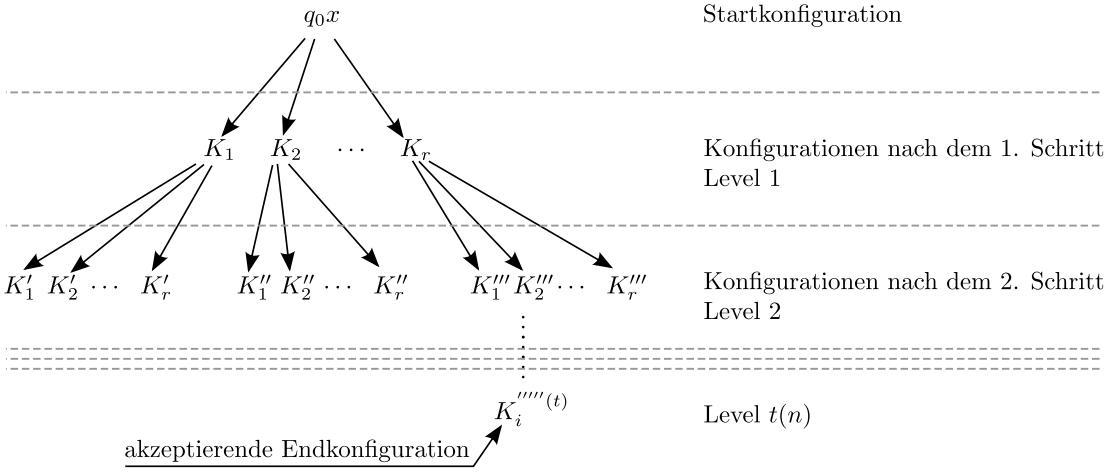


Abbildung 6: Berechnungsbaum von M gestartet mit Eingabe x.

Anmerkung: $t(n)$ ist auch eine Beschränkung für den Platzbedarf bei der kürzesten akzeptierenden Rechnung.

Optionen / Ideen:

(1) Tiefensuche (Problem: unendliche Zweige im Baum)

(2) Breitensuche auf dem Berechnungsbaum
Laufzeit: $\mathcal{O}(r^{t(n)}) = 2^{\mathcal{O}(t(n))}$
Platzbedarf: „Beschränkung Platzbedarf“·„maximale Anzahl Konfiguration in einem Level“
=„Beschränkung Platzbedarf“·$2^{\mathcal{O}(\text{„maximale Anzahl an Schritten“})} = t(n) \cdot 2^{\mathcal{O}(t(n))}$ (viel zu groß)

(3) kontrollierte Tiefensuche
Vorgehen bei kontrollierter Tiefensuche:

```
1 T:=2;
2 solange noch keine akzeptierende Rechnung gefunden und Baum noch nicht abgearbeitet tue
3 |  Führe Tiefensuche bis zur Tiefe T aus;
4 |  T:=T+1;
5 Ende
```

Platzbedarf: „Anzahl Level“·„Anzahl benutzter Speicherzellen“= $\mathcal{O}(t(n)^2)$
(Je Rekursionsabstieg wird der Speicher (maximal $t(n)$ Zellen) kopiert)
Laufzeit: $\mathcal{O}(\sum_{T=2}^{t(n)} 2^{\mathcal{O}(T)}) = \mathcal{O}(2^{\mathcal{O}(t(n))}) = 2^{\mathcal{O}(t(n))}$ □

Korollar 2.4 *NTMs akzeptieren genau die rekursiv aufzählbaren Sprachen.*

2.0.2 Das P-NP-Problem

Wir haben 5 Probleme P_1, P_2, P_3, P_4, P_5 mit unterschiedlichen Laufzeitschranken.

Tabelle 1: Annahme: Ein Rechner führt 1 000 Operationen pro Sekunde aus. Angegeben sind jeweils maximale n sodass die vorgegebene Rechenzeit nicht überschritten wird.

	$t(n)$	$0.01s$	$1s$	$1min$	$1h$
P_1	n	10	1 000	60 000	3 600 000
P_2	$n\log(n)$	4	140	4 893	204 094
P_3	n^2	3	31	244	1 897
P_4	n^3	2	10	39	153
P_5	2^n	3	9	15	21

Tabelle 2: Annahme: Ein neuer Rechner erledigt 10 mal so viele Operationen also 10 000 Operationen pro Sekunde. Angegeben ist, wie sich die maximal berechenbaren Werte verändern.

	$t(n)$	vorher (1 000)	nachher (10 000)
P_1	n	m	$10 \cdot m$
P_2	$n\log(n)$	m	(fast)$10 \cdot m$
P_3	n^2	m	$3.16 \cdot m$
P_4	n^3	m	$2.15 \cdot m$
P_5	2^n	m	$m + 3.3$

Definition 2.5

$$\texttt{DTIME}(t(n)) := \{L \mid \textit{Es gibt eine deterministische } \mathcal{O}(t(n))\textit{zeitbeschränkte (Mehrband-)TM, die } L \textbf{\textit{ entscheidet}}\}$$

$$\texttt{NTIME}(t(n)) := \{L \mid \textit{Es gibt eine } \textbf{\textit{nicht}}\textit{deterministische } \mathcal{O}(t(n))\textit{zeitbeschränkte (Mehrband-)TM, die } L \textbf{\textit{ akzeptiert}}\}$$

$$\mathbf{P} := \bigcup_{k \in \mathbb{N}} \mathit{DTIME}\left(n^k\right)$$

$$\mathbf{NP} := \bigcup_{k \in \mathbb{N}} \mathit{NTIME}\left(n^k\right)$$

Offensichtlich gilt $P \subseteq NP$.
Die große Frage lautet: Gilt $P \subsetneq NP$ oder $P = NP$?

Vorteile von P:

- P ist robust
  - Hängt nicht von der Anzahl der Bänder ab
  - Hängt nicht davon ab, ob Turingmaschinen oder Registermaschinen betrachtet werden, weil ein $t(n)$-zeitbeschränktes Registermaschinenprogramm in Zeit $\mathcal{O}(t(n)^3)$ auf einer Turingmaschine simuliert werden kann.
  - Polynome sind unter Hintereinanderausführung abgeschlossen
- P enthält die Probleme, die in der Praxis als handhabbar erkannt wurden
- P ermöglicht eine reichhaltige Theorie, da man wegen der Robustheit meist in P bleibt.

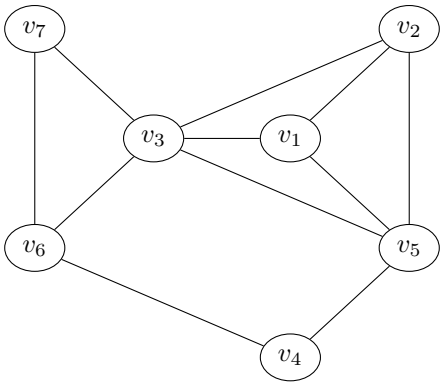


Abbildung 7: Beispielgraph $G_0 = (V_0, E_0)$, $V_0 = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$,
$E_0 = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_5\}, \{v_3, v_5\}, \{v_3, v_6\}, \{v_3, v_7\}, \{v_4, v_5\}, \{v_4, v_6\}, \{v_6, v_7\}\}$.

Nachfolgend werden einige relevante Sprachen/Probleme eingeführt:

Cliquen-Problem (CLIQUE):

$$\text{CLIQUE} := \{\langle G, k\rangle \mid k \in \mathbb{N},\ G \text{ ist ungerichteter Graph,}\\ \text{der einen vollständigen Teilgraphen der Größe } k \text{ enthält}\}\ .$$

Alternative Definition:

$$\text{CLIQUE} := \{\langle G, k\rangle \mid k \in \mathbb{N},\ G = (V, E) \text{ ist ungerichteter Graph,}\\ \exists U \subseteq V, |U| = k : \forall u, v \in U, u \neq v : \{u, v\} \in E\}\ .$$

In Abbildung 7 induziert die Knotenteilmenge $\{v_1, v_2, v_3, v_5\}$ einen vollständigen Teilgraphen und es gibt keine Teilmenge der Mächtigkeit 5, die einen vollständigen Teilgraphen induziert.
Entsprechend gilt: $\langle G_0, 4\rangle \in \text{CLIQUE}$ und $\langle G_0, 5\rangle \notin \text{CLIQUE}$.

Independent-Set-Problem (IS):

$$\text{IS} := \{\langle G, k\rangle \mid k \in \mathbb{N},\ G = (V, E) \text{ ist ungerichteter Graph,}\\ \exists U \subseteq V, |U| = k : \forall u, v \in U : \{u, v\} \notin E\}\ .$$

Sofern $G = (V, E)$ ein ungerichteter Graph ist, so ist $U \subseteq V$ eine ***unabhängige Menge***, wenn es keine Kante zwischen Knoten aus U gibt, also $\forall u, v \in U : \{u, v\} \notin E$. In Abbildung 7 bildet die Knotenteilmenge $\{v_2, v_4, v_7\}$ eine unabhängige Menge und es gibt keine Teilmenge der Mächtigkeit 4, die eine unabhängige Menge darstellt. Entsprechend gilt: $\langle G_0, 3\rangle \in \text{IS}$ und $\langle G_0, 4\rangle \notin \text{IS}$.

Färbungs-Problem (COLORING - COL):
Sei $k \in \mathbb{N}$. Ein ungerichteter Graph $G = (V, E)$ heißt **k-*färbbar***, wenn es eine Abbildung $c : V \to \{1, \ldots, k\}$ gibt mit der Eigenschaft, dass $\forall\{u, v\} \in E$ gilt, dass $c(u) \neq c(v)$.

$$\text{COL} := \{\langle G, k\rangle \mid G \text{ ist ein ungerichteter Graph und } G \text{ ist } k\text{-färbbar}\}\ .$$

$$3\text{COL} := \{\langle G\rangle \mid G \text{ ist ein ungerichteter Graph und } G \text{ ist 3-färbbar}\}\ .$$

Der Graph G_0 aus Abbildung 7 kann mit 4 oder mehr Farben gefärbt werden, aber G_0 ist nicht 3-färbbar.
Entsprechend gilt: $\langle G_0, 3\rangle \notin \text{COL}$, $\langle G_0, 4\rangle \in \text{COL}$ und $\langle G_0\rangle \notin 3\text{COL}$.

Hamiltonkreisproblem (HAMILTONIANCYCLE - HC):
Ein ***Hamiltonkreis*** ist ein Kreis in G, der jeden Knoten genau einmal enthält.

$$\text{HC} := \{\langle G\rangle \mid G \text{ ist eine ungerichteter Graph und enthält einen Hamilton-Kreis}\}\ .$$

In G_0 (siehe Abbildung 7) gibt es den Hamiltonkreis $(v_3, v_2, v_1, v_5, v_4, v_6, v_7[, v_3])$.
Entsprechend gilt: $\langle G_0 \rangle \in \text{HC}$.

Traveling-Salesman-Problem (TSP):

$$\text{TSP} := \{\langle G, c, k\rangle \mid \text{der Graph } G \text{ mit Kantengewichten } c : E \to \mathbb{R} \text{ (meist } c : E \to \mathbb{N}) \text{ enthält eine Rundreise (Hamiltonkreis) mit Gewicht } \leq k\} \ .$$

Vertex-Cover-Problem (VC**):**
Gegeben ist ein Graph $G = (V, E)$. Eine Knotenmenge $U \subseteq V$ ist eine ***Knotenüberdeckung***, wenn jede Kante des Graphen mindestens einen Knoten von U berührt, also $\forall\{u, v\} \in E : u \in U \vee v \in U$.

$$\text{VC} := \{\langle G, k\rangle \mid k \in \mathbb{N},\ G \text{ ist ein ungerichteter Graph und hat eine Knotenüberdeckung der Größe } k\} \ .$$

In G_0 (Abbildung 7) gibt es eine Knotenüberdeckung der Größe 4 aber keine der Größe 3.
Entsprechend gilt: $\langle G_0, 3\rangle \notin \text{VC}$ und $\langle G_0, 4\rangle \in \text{VC}$.

Binary-Programming-Problem (BP**):**

$$\text{BP} := \{\langle A, \vec{b}\rangle \mid A \text{ ist eine } m \times n\text{-Matrix mit ganzzahligen Einträgen, } \vec{b} \text{ ist ein Vektor mit } m \text{ ganzzahligen Einträgen, und es gibt einen 0-1-Vektor } \vec{x} \in \{0,1\}^n \text{ mit } A \cdot \vec{x} \leq \vec{b}\} \ .$$

Dies sind wichtige Sprachen mit der Eigenschaft: Lösung finden ist (bis heute jedenfalls) schwierig, angebliche Lösungen nachprüfen ist jedoch einfach. Falls $P \neq NP$ ist, gibt es keine schnellen (polynomiellen) Algorithmen, die diese Probleme lösen können.

Definition 2.6 *Sei L eine Sprache über $\{0,1\}$.*
Eine deterministische *Turingmaschine V_L heißt $t(n)$-beschränkter* **Verifizierer** *für L, wenn gilt:*

(i) Die Eingaben von V_L sind von der Form $x\#w$, $w, x \in \{0,1\}^*$

(ii) Die Laufzeit ist in $\mathcal{O}(t(|x|))$.

(iii) Für alle $x \in \{0,1\}^*$:
$$x \in L \Leftrightarrow \exists w : |w| \leq t(|x|) \text{ und } V_L \text{ akzeptiert } x\#w.$$
Dieses w heißt **Zertifikat** *von x.*

Satz 2.7 *Sei L eine Sprache. Dann gilt:*
$L \in \texttt{NTIME}(t(n)) \Leftrightarrow$ es gibt einen $t(n)$-beschränkten Verifizierer V_L für L.

BEWEIS:
„⇐“: Nichtdeterministische Turingmaschine M:

```
1 Die Eingabe sei x;
2 Rate w (in |w| Schritten);
3 wenn V_L, gestartet mit Eingabe x#w, akzeptierend hält dann
4 |  Akzeptiere x;
5 sonst
6 |  Verwerfe x;
7 Ende
```

Offensichtlich akzeptiert M nur Eingaben $x \in L$. Aus Punkt (iii) in Definition 2.6 folgt, dass es ein w mit $|w| \leq t(|x|)$ gibt, sodass V_L die Eingabe $x\#w$ in Zeit $\mathcal{O}(t(|x|))$ akzeptiert. M rät dieses w in Zeit $\mathcal{O}(t(|x|))$.
$\Rightarrow M$ ist $\mathcal{O}(t(n))$-zeitbeschränkt und somit $L \in \texttt{NTIME}(t(n))$.

„⇒“: $L \in \texttt{NTIME}(t(n))$. M sei eine $t(n)$-zeitbeschränkte, nichtdeterministische Turingmaschine für L.
O. B. d. A.: Jede Konfiguration einer Rechnung von M hat keine oder genau 2 Nachfolgekonfigurationen.

Bei Eingabe $x\#w$ arbeitet V_L wie folgt:
Falls M im i-ten Schritt die Wahl zwischen zwei Konfigurationen K_0, K_1 hat, so wählt V_L die Konfiguration K_{w_i} mit $w = w_1 \dots w_{t(|x|)}$.

$x \in L \Leftrightarrow$ es gibt eine akzeptierende Rechnung von M, gestartet mit Eingabe x, mit der Laufzeit $\mathcal{O}(t(|x|))$

$\Leftrightarrow$ es gibt ein $w \in \{0,1\}^{t(|x|)}$, sodass V_L die Eingabe $x\#w$ akzeptiert,
da jedes w eine Rechnung von M, gestartet mit Eingabe x, beschreibt.

□

Korollar 2.8

$$NP = \{L \mid \textit{es gibt einen polynomiellen Verifizierer für } L\}$$

2.1 NP-Vollständigkeit

Definition 2.9 *Seien $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$.*
*L_1 ist genau dann **polynomiell reduzierbar** auf L_2 ($L_1 \leq_p L_2$), wenn gilt:*

(i) $L_1 \leq L_2$ *mittels Reduktionsfunktion* f.

(ii) Die Laufzeit zur Berechnung von $f(x)$ *ist* $\mathcal{O}\left(|x|^k\right)$ *für* $k \in \mathbb{N}$.

Lemma 2.10 *„$\leq_p$" ist transitiv.*

BEWEIS: Siehe Übung □

Definition 2.11

- *L heißt* **NP-schwer** *(engl.* **NP-hard**, *auch* **NP-schwierig** - *Vorsicht vor dem false friend „NP-hart", der in der Literatur auch auftritt), wenn gilt:*

$$\forall L' \in NP : L' \leq_p L$$

- *L heißt* **NP-vollständig** *(engl.* **NP-complete**, **NPC**), *wenn gilt:*

  (i) $L \in NP$

  (ii) L *ist* NP*-schwer.*

Lemma 2.12

(i) L sei NP-schwer. Dann gilt:

  (a) $L \in P \Rightarrow P = NP$

  (b) $P \neq NP \Rightarrow L \notin P$

(ii) L sei NP-vollständig. Dann gilt:

  (a) $L \in P \Leftrightarrow P = NP$

  (b) $L \notin P \Leftrightarrow P \neq NP$

  (c) $L' \in NP$ *und* $L \leq_p L' \Rightarrow L'$ *ist* NP*-vollständig.*

2.2 Satz von Cook

Definition 2.13

- *$V = \{x_1, \ldots, x_n\}$ sei eine* **Menge Boolescher Variablen**.
  Ein **Literal** *ist eine Variable oder eine negierte Variable (x oder $\overline{x}$)*
  Eine **Klausel** *K der Länge s ist eine Veroderung von (ein Boolescher Ausdruck aus) s Literalen:*

$$K = y_1 \vee y_2 \vee \ldots \vee y_s \text{ mit } y_i \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$$

- *Ein Ausdruck in* **konjunktiver Normalform (KNF)** *ist ein Boolescher Ausdruck Φ der Form*

$$\Phi = K_1 \wedge K_2 \wedge \ldots \wedge K_t \text{ mit Klauseln } K_i.$$

  Die Anzahl der in Φ vorkommenden $\wedge$- und $\vee$-Operationen ist die ***Größe*** *von Φ, oder kurz* **size**(Φ).

- *Eine totale Abbildung $c : V \to \{\texttt{TRUE}, \texttt{FALSE}\}$, die jeder Variablen einen Wahrheitswert zuweist, heißt* **Belegung** *der Variablen. c wird kanonisch auf die Literale, Klauseln K und die KNF Φ fortgesetzt. $c(K)$ und $c(\Phi)$ ist das Ergebnis der Auswertung von K bzw. Φ unter Anwendung von c.*

- *Eine KNF Φ heißt* **erfüllbar**, *wenn es eine Belegung c gibt, sodass $c(\Phi) =$* `TRUE` *ist.*

Beispiel:

$$\Phi = (x_1 \vee \overline{x_3} \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x_4})$$

$size(\Phi) = 5$, $c(x_1) = c(x_2) =$ `TRUE` und Rest beliebig ist eine erfüllende Belegung ($c(\Phi) =$ `TRUE`).

Φ wird kodiert durch $\langle\Phi\rangle$:

- $\langle x_i \rangle = 1\#\text{bin}(i)$
- $\langle \overline{x_i} \rangle = 0\#\text{bin}(i)$
- $\langle K \rangle = \langle y_1 \vee \ldots \vee y_s \rangle = \langle y_1 \rangle \#\# \langle y_2 \rangle \#\# \ldots \#\# \langle y_s \rangle$
- $\langle \Phi \rangle = \langle K_1 \wedge \ldots \wedge K_t \rangle = \langle K_1 \rangle \#\#\# \ldots \#\#\# \langle K_t \rangle$
- $|\langle \Phi \rangle| = \mathcal{O}\left(size(\Phi) \cdot \log(|V|)\right)$

Das **Erfüllbarkeitsproblem** (engl. Satisfyability Problem, SAT):

$$SAT := \{\langle \Phi \rangle \mid \Phi \text{ ist eine erfüllbare KNF}\}.$$

Satz 2.14 (Satz von Cook, 1971) *SAT ist NP-vollständig.*

BEWEIS: Wir zeigen, dass die Definition von NP-Vollständigkeit erfüllt ist.

(i) Zu zeigen: $SAT \in NP$
Rate eine Belegung c der Variablen $x_1, \ldots, x_n$ der Eingabe-KNF Φ und überprüfe, ob $c(\Phi) =$ `TRUE`.
Die Laufzeit des Ratens ist in $\mathcal{O}(n)$ und die Laufzeit der Auswertung ist in $\mathcal{O}(|\langle\Phi\rangle|)$.
Also ist die Gesamtlaufzeit polynomiell in $|\langle\Phi\rangle|$.✓

(ii) Zu zeigen: SAT ist NP-schwer, das heißt $\forall L \in NP : L \leq_p SAT$.
Sei $L \in NP$ beliebig, aber fest. Sei $M_L = (Q, \Sigma, \Gamma, \delta, q_0, F)$ eine nichtdeterministische Einband-Turingmaschine, mit der Laufzeit $T_{M_L}(n) = c \cdot n^k$, mit Konstanten c, k, die L akzeptiert.
Ziel: Für $w \in \Sigma^*$ muss eine KNF Φ_w konstruiert werden, die zwar „sehr viele“ Variablen enthält, aber nur polynomielle Größe verglichen mit $|w|$ hat, sodass gilt:

$$w \in L \Leftrightarrow \Phi_w \text{ ist erfüllbar } \Leftrightarrow \langle \Phi_w \rangle \in SAT.$$

Dabei soll Φ_w eine erlaubte Rechnung von M_L, gestartet mit Eingabe w, beschreiben und $\langle \Phi_w \rangle$ soll in Polynomzeit in $|w|$ berechnet werden können.

O. B. d. A.: $F = \{q_F\}, \delta(q_F, a) = \{(q_F, a, N)\}$ für alle $a \in \Gamma$.

$$
\begin{aligned}
& w \in L \\
\Leftrightarrow\ & \text{es gibt eine Berechnung von } M_L, \text{ gestartet mit } w, \\
& \text{der Länge } T := T_{M_L}(|w|) = c \cdot |w|^k, \text{ die } w \text{ akzeptiert.} \\
\Leftrightarrow\ & q_0 w = K_0 \vdash K_1 \vdash \ldots \vdash K_T \text{ und Zustand in } K_T \text{ ist } q_F.
\end{aligned}
$$

Wir benutzen die folgende Menge an Variablen:

$$
\begin{aligned}
V_{M_L} = \quad & \{\texttt{zelle}_{t,i,a} \mid 0 \leq t \leq T,\ -T \leq i \leq T,\ a \in \Gamma\} \\
& \cup \left\{\texttt{kopf}_{t,i} \mid 0 \leq t \leq T,\ -T \leq i \leq T\right\} \cup \{\texttt{zustand}_{t,q} \mid 0 \leq t \leq T,\ q \in Q\}
\end{aligned}
$$

Mit diesen Variablen lässt sich jede Konfiguration beschreiben.

$$|V_{M_L}| = (T+1) \cdot (2T+1) \cdot |\Gamma| + (T+1) \cdot (2T+1) + (T+1) \cdot |Q| = \mathcal{O}(T^2) = \mathcal{O}(|w|^{2 \cdot k})$$

Die Variablen haben die folgende Bedeutung:

$$\texttt{zelle}_{t,i,a} = \texttt{TRUE}$$
$$\Leftrightarrow$$
In der Berechnung von M_L, gestartet mit Eingabe w,
steht in der Konfiguration K_t in der Zelle i das Zeichen a.

$$\texttt{kopf}_{t,i} = \texttt{TRUE}$$
$$\Leftrightarrow$$
In der Berechnung von M_L, gestartet mit Eingabe w,
steht in der Konfiguration K_t der Kopf unter der Zelle i.

$$\texttt{zustand}_{t,q} = \texttt{TRUE}$$
$$\Leftrightarrow$$
In der Berechnung von M_L, gestartet mit Eingabe w,
ist in der Konfiguration K_t die TM M_L im Zustand q.

Hilfsmittel:

$$\texttt{GenauEineVariableIstTrue}(x_1, \ldots, x_r) := (x_1 \vee \ldots \vee x_r) \wedge \bigwedge_{1 \leq i < j \leq r} (\overline{x_i} \vee \overline{x_j})$$

Größe des Ausdrucks: $\mathcal{O}(r^2)$.

$$\texttt{IstGleich}(x, y) := (\overline{x} \vee y) \wedge (x \vee \overline{y})$$

Größe des Ausdrucks: $\mathcal{O}(1)$.

$$V_t := \left\{ \texttt{zelle}_{t,i,a}, \texttt{kopf}_{t,i}, \texttt{zustand}_{t,q} \mid -T \leq i \leq T,\, a \in \Gamma,\, q \in Q \right\} \subseteq V_{M_L}$$

$$\texttt{Konf}(V_t) = \texttt{TRUE}$$
$$\Leftrightarrow$$
Die Belegung der Variablen in V_t beschreibt exakt eine Konfiguration.

$$\begin{aligned}
\texttt{Konf}(V_t) := \quad & \texttt{GenauEineVariableIstTrue}(\texttt{zustand}_{t,q_0}, \ldots, \texttt{zustand}_{t,q_{|Q|-1}}) \\
& \wedge \texttt{GenauEineVariableIstTrue}(\texttt{kopf}_{t,-T}, \ldots, \texttt{kopf}_{t,T}) \\
& \wedge \bigwedge_{-T \leq i \leq T} \texttt{GenauEineVariableIstTrue}(\texttt{zelle}_{t,i,a_1} \ldots, \texttt{zelle}_{t,i,a_{|\Gamma|}})
\end{aligned}$$

Größe des Ausdrucks: $\mathcal{O}(T^2)$.

$$\begin{aligned}
\Phi_w &:= \texttt{Rechnung}(V_0, V_1, \ldots, V_T) \\
&:= \texttt{Startkonf}(V_0) \wedge \bigwedge_{0 \leq t \leq T} \texttt{Konf}(V_t) \wedge \bigwedge_{0 \leq t < T} \texttt{EinSchritt}(V_t, V_{t+1}) \wedge \texttt{zustand}_{T,q_F}
\end{aligned}$$

Zu beachten:
In `Startkonf` geht w in die Konstruktion ein, in `EinSchritt` geht δ in die Konstruktion ein.

$$\texttt{Startkonf}(V_0) := \bigwedge_{0 \leq i < |w|} \texttt{zelle}_{0,i,w_i} \wedge \bigwedge_{-T \leq i < 0,\, |w| \leq i \leq T} \texttt{zelle}_{0,i,B} \wedge \texttt{kopf}_{0,0} \wedge \texttt{zustand}_{0,q_0}$$

$$\texttt{EinSchritt}(V_t, V_{t+1}) := \bigvee_{\gamma \in \delta} \texttt{SchrittDurch}\gamma(V_t, V_{t+1}) \text{ mit } \gamma = [(q', a', D') \in \delta(q, a)]$$

$$\begin{aligned}
\texttt{SchrittDurch}\gamma(V_t, V_{t+1}) := \quad & \texttt{zustand}_{t,q} \\
& \wedge \texttt{zustand}_{t+1,q'} \\
& \wedge \text{„Unter Kopf in } K_t \text{ steht Zeichen } a\text{“} \\
& \wedge \text{„An derselben Position steht in } K_{t+1} \text{ das Zeichen } a'\text{“} \\
& \wedge \text{„Kopf wird in Richtung } D \text{ bewegt“} \\
& \wedge \text{„Rest vom Bandinhalt ist unverändert} \\
& \quad \text{beim Übergang von } K_t \text{ nach } K_{t+1}\text{“}
\end{aligned}$$

Anmerkung:
`EinSchritt` wird mittels Veroderung von Klauseln umgesetzt, was in KNFs nicht erlaubt ist. Da es sich jedoch nur um $\mathcal{O}(1)$ viele Veroderungen handelt, kann diese Formel wieder in eine KNF umgewandelt werden ohne das sich die Größe der resultierenden Formel im $\mathcal{O}$-Kalkül verändert.
Außerdem geht an dieser Stelle auch der Nichtdeterminismus ein. Jeder δ-Übergang ist eine Option.

Ausformuliert:

$$\text{„Unter Kopf in } K_t \text{ steht Zeichen } a\text{“} := \bigwedge_{-T \le i \le T} \left(\overline{\texttt{kopf}_{t,i}} \vee \texttt{zelle}_{t,i,a}\right)$$

$$\text{„An derselben Position steht in } K_{t+1} \text{ das Zeichen } a'\text{“} := \bigwedge_{-T \le i \le T} \left(\overline{\texttt{kopf}_{t,i}} \vee \texttt{zelle}_{t+1,i,a'}\right)$$

Fall $D = L$:

$$\text{„Kopf wird in Richtung } D \text{ bewegt“} := \bigwedge_{-T < i < T} \texttt{IstGleich}(\texttt{kopf}_{t,i}, \texttt{kopf}_{t+1,i-1})$$

Fälle $D = N$ und $D = R$ analog.

$$\text{„Rest vom Bandinhalt ist unverändert beim Übergang von } K_t \text{ nach } K_{t+1}\text{“} :=$$

$$:= \bigwedge_{\substack{-T \le i \le T \\ a \in \Gamma}} (\texttt{kopf}_{t,i} \vee \texttt{IstGleich}(\texttt{zelle}_{t,i,a}, \texttt{zelle}_{t+1,i,a})$$

Größe von Φ_w ist $\mathcal{O}(T^3) = \mathcal{O}(|w|^{3k})$, das heißt $\langle \Phi_w \rangle$ kann in polynomieller Zeit hingeschrieben werden.

$w \in L$
$\Rightarrow$ Es gibt eine akzeptierende Rechnung von M_L, gestartet mit Eingabe w
$\Rightarrow$ Die Variablen in V_{M_L} können so belegt werden, dass Φ_w erfüllt wird
(abzulesen an der akzeptierenden Rechnung)
$\Rightarrow$ $\langle \Phi_w \rangle \in SAT$.

$\langle \Phi_w \rangle \in SAT$
$\Rightarrow$ Es gibt eine erfüllende Belegung der Variablen V_{M_L} sodass Φ_w erfüllt ist
$\Rightarrow$ Es kann eine akzeptierende Rechnung von M_L, gestartet mit Eingabe w,
daraus konstruiert werden
$\Rightarrow$ $w \in L$.

□

*Der Beweis des Satzes von Cook enthält die sogenannte **Masterreduktion**.*

Analogie:
Erst haben wir aufwändig die Unentscheidbarkeit des Halteproblems H bewiesen.
Für nachfolgende Probleme L hat es dann ausgereicht, H auf L zu reduzieren, um die Unentscheidbarkeit von L zu beweisen.

Jetzt haben wir initial die NP-Vollständigkeit von SAT bewiesen.
Für nachfolgende Probleme L reicht es jetzt aus, SAT auf L in Polynomzeit zu reduzieren, um die NP-Vollständigkeit von L zu beweisen.

kSAT:

$kSAT := \{\langle \Phi \rangle \mid \Phi$ sei eine erfüllbare KNF,
in der jede Klausel aus genau k Literalen über k verschiedenen Variablen besteht$\}$.

Offensichtlich: $kSAT \subseteq SAT$.
Es gilt $2SAT \in P$!

Satz 2.15 *3SAT ist NP-vollständig.*

BEWEIS:

(i) 3SAT $\in NP$ ✓

(ii) Wir zeigen: $3SAT$ ist NP-schwer indem wir zeigen, dass $SAT \leq_p 3SAT$

$$\langle\Phi\rangle \in SAT \Leftrightarrow f(\langle\Phi\rangle) \in 3SAT,\ f \text{ in Polynomzeit berechenbar.}$$

$$\Phi = K_1 \wedge K_2 \wedge \ldots \wedge K_T$$

Fall $K_i = l$ ist **ein** Literal:
Benutze zwei zusätzliche Variablen $y_{i,1}$ und $y_{i,2}$.

$$f(K_i) = (l \vee y_{i,1} \vee y_{i,2}) \wedge (l \vee y_{i,1} \vee \overline{y_{i,2}}) \wedge (l \vee \overline{y_{i,1}} \vee y_{i,2}) \wedge (l \vee \overline{y_{i,1}} \vee \overline{y_{i,2}})$$

$size(K_i) = 0$, $size(f(K_i)) = 11$.

Fall $K_i = l_1 \vee l_2$ besteht aus **zwei** Literalen:
Benutze eine zusätzliche Variable y_i.

$$f(K_i) = (l_1 \vee l_2 \vee y_i) \wedge (l_1 \vee l_2 \vee \overline{y_i})$$

$size(K_i) = 1$, $size(f(K_i)) = 5$

Fall $K_i = l_1^{(i)} \vee l_2^{(i)} \vee \ldots \vee l_k^{(i)}$, mit $k > 3$:
Benutze zusätzliche Variablen $y_{i,1}, \ldots, y_{i,k-3}$, um die Klausel in mehrere Klauseln aufzuteilen.

$$f(K_i) = (l_1^{(i)} \vee l_2^{(i)} \vee y_{i,1}) \wedge (\overline{y_{i,1}} \vee l_3^{(i)} \vee y_{i,2}) \wedge (\overline{y_{i,2}} \vee l_4^{(i)} \vee y_{i,3}) \wedge \ldots \wedge (\overline{y_{i,k-3}} \vee l_{k-1}^{(i)} \vee l_k^{(i)})$$

$size(K_i) = k - 1$, $size(f(K_i)) = 3k - 7$

$$f(\Phi) = f(K_1) \wedge f(K_2) \wedge \ldots \wedge f(K_t)$$

$$\text{und } f(\langle\Phi\rangle) = \langle f(\Phi)\rangle$$

Φ erfüllbar $\Rightarrow f(\Phi)$ erfüllbar

Belege die Variablen in $f(\Phi)$, die bereits in Φ vorkommen, analog zu einer erfüllenden Belegung für Φ. In den ersten beiden Fällen sind die Zusatzvariablen beliebig wählbar. Im letzten Fall gibt es mindestens ein Literal $l_j^{(i)}$ mit Belegung `TRUE`. Von dieser Teilklausel ausgehend können die $y_{i,l}$ belegt werden, so dass die übrigen Teilklauseln zu den entsprechenden Klauseln in Φ erfüllt werden.

$f(\Phi)$ erfüllbar $\Rightarrow$ je Klausel aus Φ muss mindestens ein ursprüngliches Literal erfüllt sein $\Rightarrow \Phi$ ist erfüllbar

$$\langle\Phi\rangle \in SAT \Leftrightarrow f(\Phi) \in 3SAT$$

f ist wegen der *size*-Werte in Polynomzeit berechenbar. □

Satz 2.16 Clique *ist NP-vollständig.*

BEWEIS:

(i) Clique $\in NP$ ✓

(ii) Wir zeigen: Clique ist NP-schwer indem wir zeigen, dass $SAT \leq_p$ Clique

$$\Phi = \bigwedge_{i=1}^{T} K_i \text{ über Variablen } \{x_1, \ldots, x_n\}$$

$$K_i = y_1^{(i)} \vee \ldots \vee y_{s_i}^{(i)}, \quad y_j^{(i)} \text{ sind Literale.}$$

Ziel: Konstruiere zur KNF Φ eine Eingabe $\langle G_\Phi, T\rangle$ für Clique mit

$$\langle\Phi\rangle \in SAT \Leftrightarrow f(\langle\Phi\rangle) = \langle G_\Phi, T\rangle \in \text{Clique}, \; G_\Phi = (V, E) \text{ ist ein Graph, } T = \text{Anzahl Klauseln in } \Phi.$$

$$V := \{(i,j) \mid i \in \{1,\ldots,T\}, j \in \{1,\ldots,s_i\}\} \qquad \text{mit } s_i = \text{Anzahl Literale in Klausel } K_i$$

$$E := \left\{\{(i,j),(i',j')\} \mid i \neq i' \wedge y_j^{(i)} \neq \overline{y_{j'}^{(i')}}\right\} \qquad \text{mit } K_i = y_1^{(i)} \vee \ldots \vee y_{s_i}^{(i)}$$

Zu zeigen: Φ ist erfüllbar $\Leftrightarrow$ G_Φ enthält eine Clique der Größe T.
„$\Rightarrow$“:
$c = (c_1, \ldots, c_n)$ sei eine erfüllende Belegung der Variablen $\{x_1, \ldots, x_n\}$, das heißt $c_i \in \{\texttt{TRUE}, \texttt{FALSE}\}$. Es wird in jedem K_i mindestens ein Literal $y_j^{(i)}$ erfüllt.
O. B. d. A. seien diese $y_1^{(1)}, \ldots, y_1^{(T)}$. Dann ist **NIE** $\overline{y_1^{(i)}} = y_1^{(i')}$, mit $i \neq i'$.
Also ist immer $\{(i,1),(i',1)\} \in E$.
Das heißt $(1,1), \ldots, (T,1)$ bilden eine Clique der Größe T in G_Φ. Also ist immer $\{(i,1),(i',1)\} \in E$, das heißt $(1,1),(2,1),\ldots,(T,1)$ bilden eine T-Clique in G_Φ.

$$\Rightarrow \langle G_\Phi, T\rangle \in \text{Clique } \checkmark$$

„$\Leftarrow$“:
Sei $l = \{(i_1, j_1), \ldots, (i_T, j_T)\}$ eine Clique der Größe T in G_Φ.
Dann ist wegen $i \neq i'$ in der Definition von E für jede Klausel aus Φ genau ein zugehöriger Knoten in der Clique vorhanden. Diese Knoten werden durchnummeriert zu

$$C = \{(1, l_1), (2, l_2), \ldots, (T, l_T)\}$$

Seien $y_{l_i}^{(i)} = x_{k_i}^{\alpha_i}$ mit $\alpha_i \in \{\text{negiert}, \text{ nicht negiert}\}$.
Falls $k_i = k_{i'}$ so gilt auch $\alpha_i = \alpha_{i'}$, da sonst keine Kante zwischen den Literalknoten vorhanden wäre. Entsprechend generiert man eine erfüllende Belegung indem alle Variablen x_{k_i} so gesetzt werden, dass $x_{k_i}^{\alpha_i} =$ `TRUE`, denn damit gibt es je Klausel mindestens ein Literal, welches mit dem Wert `TRUE` belegt ist. Alle noch unbelegten Variablen können auf einen beliebigen Wahrheitswert gesetzt werden. Durch diese Belegung wird Φ erfüllt, das heißt $\langle\Phi\rangle \in SAT$.
Konstruktion von G_Φ geht in Polynomzeit in $|\langle\Phi\rangle|$. □

Beispiel:

$$\Phi = (x_1 \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$$

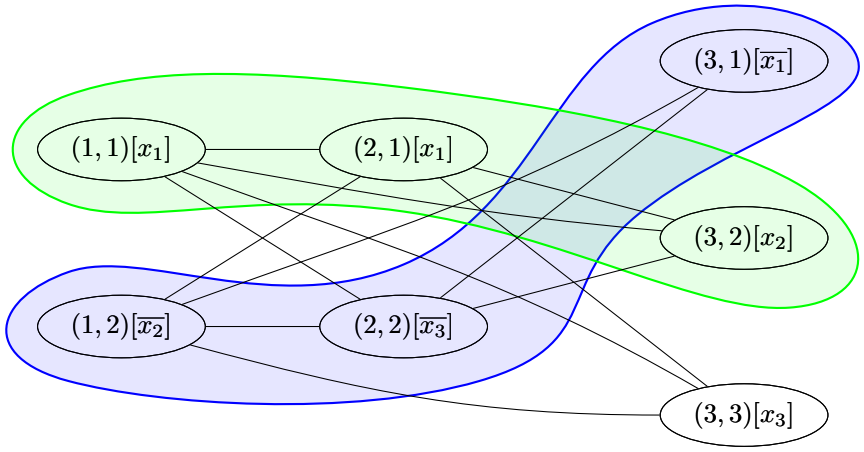


Abbildung 8: Graph G_Φ, wobei bei jedem Knoten das zugehörige Literal in eckigen Klammern vermerkt ist. In blau bzw. grün sind zwei Cliquen der Größe 3 hinterlegt.

Der Graph G_Φ ist in Abbildung 8 visualisiert. Die grüne markierte Clique $\{(1,1),(2,1),(3,2)\}$ führt zur Belegung der Variablen $x_1 = \texttt{TRUE}$, $x_2 = \texttt{TRUE}$ und x_3 beliebig. Die blaue markierte Clique $\{(1,2),(2,2),(3,1)\}$ führt zur Belegung der Variablen $x_1 = \texttt{FALSE}$, $x_2 = \texttt{FALSE}$ und $x_3 = \texttt{FALSE}$. Beide Cliquen führen zu erfüllenden Belegungen der Variablen, sowie auch jede weitere Clique.

Anders als bei der *Unentscheidbarkeit* gibt es nicht ***das*** *Kochrezept*, um auf eine Polynomzeitreduktion zu kommen. Man braucht *ein NP-vollständiges Problem, das man auf das neue Problem reduzieren will,* und muss dann die *Bastelei erfinden*. Deswegen braucht man einen Vorrat an NP-vollständigen Problemen.

Satz 2.17 VertexCover *(*VC*) ist NP-vollständig.*

BEWEIS:

(i) VC $\in NP$✓

(ii) Wir zeigen: VC ist NP-schwer indem wir zeigen, dass Clique $\leq_p$ VC

$K(G)$ sei der Komplementgraph von G.
$K(G) = K((V,E)) := (V, \{\{u,v\} \mid u,v \in V,\, u \neq v\} \setminus E)$

$$f(x) = \begin{cases} \langle K(G), |V| - k\rangle & \text{falls } x = \langle G,k\rangle = \langle (V,E),k\rangle \\ 0 & \text{sonst.} \end{cases}$$

Behauptung: $\langle G, k\rangle \in$ Clique $\Leftrightarrow \langle K(G), |V| - k\rangle \in$ VC
Das heißt: G enthält eine k-Clique $\Leftrightarrow$ $K(G)$ enthält eine Knotenüberdeckung der Größe $|V| - k$.

Sei C eine Clique in G
$\Rightarrow$ in $K(G)$ gibt es keine Kante zwischen 2 Knoten in C
$\Rightarrow$ je Kante in $K(G)$ liegt mindestens ein beteiligter Knoten in $V \setminus C$
$\Rightarrow$ $V \setminus C$ ist eine Knotenüberdeckung in $K(G)$.

Sei U eine Knotenüberdeckung in $K(G)$
$\Rightarrow$ in $K(G)$ gibt es keine Kante zwischen 2 Knoten in $V \setminus U$
$\Rightarrow$ $V \setminus U$ ist eine Clique in G.

Also kann jeder Clique C der Größe k im Graphen G die Knotenüberdeckung $V \setminus C$ der Größe $|V| - k$ im Graphen $K(G)$ zugeordnet werden. Das gleiche gilt auch umgekehrt. Entsprechend gilt die behauptete Äquivalenz.

Somit gilt auch

$$x \in \text{Clique} \Rightarrow f(x) \in \text{VC}$$

$$x \notin \text{Clique} \Rightarrow \begin{cases} x = \langle G, k\rangle \wedge x \notin \text{Clique} \Rightarrow f(x) \notin \text{VC} \\ x \neq \langle G, k\rangle \Rightarrow f(x) = 0 \notin \text{VC} \end{cases}$$

und insgesamt

$$x \in \text{Clique} \Leftrightarrow f(x) \in \text{VC}.$$

Außerdem ist die Berechnung von f in Polynomzeit möglich.
$\Rightarrow$Clique $\leq_p$ VC. □

Problem *Binary Programming* (BP):

$$\text{BP} := \left\{\langle A, b\rangle \mid A \in \mathcal{M}_{m,n}(\mathbb{Z}) = \mathbb{Z}^{m\times n},\, b \in \mathbb{Z}^m,\, \exists y \in \{0,1\}^n : Ay \leq b\right\}$$

$\mathcal{M}_{n,m}(\mathbb{Z})$ ist dabei die Menge aller $n \times m$-Matrizen mit ganzzahligen Einträgen.

Satz 2.18 BP *ist NP-vollständig.*

BEWEIS:

(i) BP $\in NP$ ✓

(ii) Wir zeigen: BP ist NP-schwer indem wir zeigen, dass $SAT \leq_p$ BP

Sei Φ eine KNF.

$$\Phi = K_1 \wedge \ldots \wedge K_T,\ K_i = y_1^{(i)} \vee \ldots \vee y_{s_i}^{(i)} \text{ mit } y_j^{(i)} \in \{x_1, \ldots, x_n\} \cup \{\overline{x_1}, \ldots, \overline{x_n}\}$$

Wir erzeugen ein Ungleichungssystem über den Variablen $z_1, \ldots, z_n, z_1', \ldots, z_n'$, wobei $z_i = 1$ falls $x_i =$ `TRUE` und $z_i' = 1$ falls $x_i =$ `FALSE` sein soll.
Für jedes x_i benutze die Ungleichungen

$$z_i + z_i' \leq 1 \text{ und } -z_i - z_i' \leq -1$$

um folgende Gleichung zu gewährleisten

$$z_i + z_i' = 1.$$

Für jedes K_i benutze die Ungleichung

$$\sum_{j=1}^{s_i} \tilde{y}_j^{(i)} \geq 1, \text{ mit } \tilde{y}_j^{(i)} = \begin{cases} z_l & y_j^{(i)} = x_l \\ z_l' & y_j^{(i)} = \overline{x}_l. \end{cases}$$

Die Reduktionsfunktion soll nun, sofern die Eingabe die Kodierung einer KNF ist, die Kodierung des angegebenen Ungleichungssystems ausgeben. Ansonsten kann ein beliebiges festes Wort $w \notin \text{BP}$ ausgegeben werden. Die Reduktionsfunktion ist in Polynomzeit berechenbar. Das Gleichungssystem ist natürlich genau dann lösbar, wenn die KNF erfüllbar ist.
$\Rightarrow SAT \leq_p \text{BP}$. □

Aufbauend auf unseren Betrachtungen zur NP-Vollständigkeit:

- Komplexitätstheorie
  - Platzhierarchie
  - Zeithierarchie
  - Nichtdeterminismus
    - `PSPACE` = `NPSPACE`
    - `Co-NSPACE`$(s(n))$ = `NSPACE`$(s(n))$
  - Spiele
    Meistens `PSPACE`-vollständig.
- Approximationsalgorithmen
  NP-vollständige Probleme tauchen praktisch überall auf.
  Wie geht man mit diesen um?
- Mildexponentielle Algorithmen für NP-vollständige Probleme.

2.3 Ein exakter mildexponentieller Algorithmus für das Vertex Cover Problem

```
1  Algorithmus VC_OPT(G = (V, E), Kandidat)
2  wenn |E| = 0 dann
3  |  wenn |VC_OPT| > |Kandidat| dann
4  |  |  VC_OPT := Kandidat;
5  |  Ende
6  sonst
7  |  solange es Kante {u, v} ∈ E gibt mit Grad von u ist 1 tue
8  |  |  Kandidat:=Kandidat ∪{v};
9  |  |  Lösche u, v und alle Kanten an denen v beteiligt ist;
10 |  Ende
   |  /* Es gibt keine Knoten mehr mit Knotengrad 1.                    */
11 |  wenn alle Knoten jetzt Grad 0 oder 2 haben dann
   |  |  /* Es gibt nur noch Kreise und isolierte Knoten.                */
   |  |  /* Von jedem Kreis muss jeder zweite Knoten in Kandidat aufgenommen werden.
   |  |     */
12 |  |  Berechne optimale Überdeckung und füge sie zu Kandidat hinzu;
13 |  |  E:=∅;
14 |  |  VC_OPT(G, Kandidat);
15 |  sonst
   |  |  /* Es gibt mindestens einen Knoten mit Grad mindestens 3.       */
16 |  |  Wähle v ∈ V mit Grad mindestens 3;
17 |  |  Bestimme dessen Nachbarn v_1, ..., v_k mit k ≥ 3;
   |  |  /* Wenn v nicht in einer optimalen Überdeckung liegt müssen alle seine
   |  |     Nachbarn in der optimalen Lösung liegen.                     */
   |  |  /* G \ C bezeichnet den Graphen in dem alle Knoten aus C und alle Kanten, an
   |  |     denen diese Knoten beteiligt sind, entfernt wurden.          */
18 |  |  VC_OPT(G \ {v}, Kandidat ∪ {v});
19 |  |  VC_OPT(G \ {v, v_1, ..., v_k}, Kandidat ∪ {v1, ..., v_k});
20 |  Ende
21 Ende
```

Der Algorithmus ist *correct by construction*✓

Sei $t(n)$ die Laufzeit bei n Knoten, dann gilt

$$\begin{aligned} t(n) &\leq t(n-1) + t(n-4) + poly(n) \\ t(n) &= const \text{ für } n \leq 4. \end{aligned}$$

Bemerkung: $\mathcal{O}^*(f(n))$ ist eine Abkürzung für $\mathcal{O}(f(n) \cdot poly(n))$.
Für das Rechnen mit $\mathcal{O}^*$-Notation kann die folgende vereinfachte Rekursion benutzt werden:

$$\begin{aligned} t(n) &\leq t(n-1) + t(n-4) \\ t(n) &= const \text{ für } n \leq 4. \end{aligned}$$

Annahme: $t(n) = \lambda^n$
Damit vereinfacht sich die Rekursion weiter zu

$$\begin{aligned} \lambda^n &= \lambda^{n-1} + \lambda^{n-4} \\ \lambda^4 &= \lambda^3 + 1. \end{aligned}$$

Die größte Nullstelle liegt bei $\lambda \approx 1.3803$. Damit liegt die Laufzeit von VC_{OPT} bei $\mathcal{O}^*(1.3803^n)$.

3 Formale Sprachen

Bisher: **Erkennen** (Automaten, Maschinen),
Syntaxanalyse
Roter Faden der Vorlesung: „Unendliche Objekte/Sprachen mit endlichen Mitteln schreiben".
Nun: **Erzeugen:**
Regeln, die mächtig genug sind, aufwendige Konstrukte zu beschreiben.
Regeln, die einfach genug sind, um eine effiziente Analyse zu erlauben.

3.1 Grammatiken

Definition 3.1 *Eine **Grammatik** G vom Typ **Chomsky-0** ist beschrieben durch ein 4-Tupel $(V, \Sigma, \mathcal{P}, \mathcal{S})$ mit:*

- *V: Endliche Menge von **Variablen***
- *Σ: Endliche Menge der **Terminalsymbole** bzw. **Terminale***
- *$\mathcal{S}$: $\mathcal{S} \in V$: **Startsymbol***
- *$\mathcal{P} \subseteq ((V \cup \Sigma)^+ \setminus \Sigma^*) \times (V \cup \Sigma)^*$: Endliche Menge von **Produktionen**, oder auch **Ersetzungsregeln**, oder **Ableitungsregeln**, (oder ganz kurz: Regeln)*
  Statt $(u, v) \in \mathcal{P}$ schreiben wir auch: $u \to v$.

Wir sagen:

- *$w' \in (V \cup \Sigma)^*$ ist aus $w \in (V \cup \Sigma)^+$ **direkt ableitbar** (herleitbar) $(w \to w')$, falls es $(u \to v) \in \mathcal{P}$ und $\alpha, \beta \in (V \cup \Sigma)^*$ gibt mit*
  $$w = \alpha u \beta \text{ und } w' = \alpha v \beta.$$
- *w' ist aus w **indirekt ableitbar** (herleitbar) $(w \overset{*}{\to} w')$, falls w' durch endlich viele Ableitungsschritte aus w erzeugbar ist.*

*Die **von G erzeugte Sprache** $\mathbf{L(G)}$ ist*

$$L(G) := \left\{ w \,\middle|\, \mathcal{S} \overset{*}{\to} w,\ w \in \Sigma^* \right\}.$$

*Anmerkung: Um zu zeigen, dass $L = L(G)$ ist für ein L ist zu zeigen, dass $L \subseteq L(G)$ **und** $L \supseteq L(G)$.*
Für eine Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ gilt:

- *G heißt **kontextsensitiv** oder vom Typ **Chomsky-1**, falls für jede Produktion $u \to v \in \mathcal{P}$ gilt:*
  $$|u| \leq |v|.$$
  Ausnahme: $\mathcal{S} \to \varepsilon$ ist erlaubt, wenn $\mathcal{S}$ nie in der rechten Seite einer Produktion enthalten ist.
- *G heißt **kontextfrei** oder vom Typ **Chomsky-2**, falls für alle $u \to v \in \mathcal{P}$ gilt:*
  $$u \in V \text{ (und damit } |u| = 1).$$
- *G heißt **regulär** oder vom Typ **Chomsky-3**, falls für alle $u \to v \in \mathcal{P}$ gilt:*
  $$u \in V \text{ und } v \in \{\varepsilon\} \cup \Sigma \text{ oder}$$
  $$u \in V \text{ und } v \in \Sigma \circ V \text{ (das heißt } v = aw \text{ mit } a \in \Sigma \text{ und } w \in V).$$

Beispiel:
$G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ mit $V = \{\mathcal{S}, B, C\}$, $\Sigma = \{a, b, c\}$ und
$\mathcal{P} = \{$ (1) $\mathcal{S} \to a\mathcal{S}BC$, (2) $\mathcal{S} \to aBC$, (3) $CB \to BC$, (4) $aB \to ab$, (5) $bB \to bb$, (6) $bC \to bc$, (7) $cC \to cc\}$.

Mögliche Ableitung:
$\mathcal{S} \overset{(1)}{\to} a\mathcal{S}BC \overset{(2)}{\to} aaBCBC \overset{(3)}{\to} aaBBCC \overset{(4)}{\to} aabBCC \overset{(5)}{\to} aabbCC \overset{(6)}{\to} aabbcC \overset{(7)}{\to} aabbcc \in L(G)$

Behauptung 3.2

$$L(G) = \{a^n b^n c^n \mid n \geq 1\}$$

BEWEIS:
„$L(G) \supseteq \{a^n b^n c^n \mid n \geq 1\}$“: Sei $a^n b^n c^n$, $n \geq 1$ gegeben.
Gesucht ist eine Folge von Ableitungsregeln, die dieses Wort erzeugt.

1. Wende $n - 1$ mal die Produktion (1) an und einmal die Produktion (2).

$$\mathcal{S} \overset{*}{\to} \underbrace{a^n (BC)^n}_{Satzform}$$

2. Wende $1 + 2 + 3 + \ldots + (n - 1) = \frac{n(n-1)}{2}$ mal Produktion (3) an, also:

$$\mathcal{S} \overset{*}{\to} a^n B^n C^n$$

3. Wende einmal Produktion (4) und $n - 1$ mal Produktion (5) an:

$$\mathcal{S} \overset{*}{\to} a^n b^n C^n$$

4. Wende einmal Produktion (6) und $n - 1$ mal Produktion (7) an:

$$\mathcal{S} \overset{*}{\to} a^n b^n c^n \ \checkmark$$

„$L(G) \subseteq \{a^n b^n c^n \mid n \geq 1\}$“:(„Mit G kann nichts anderes erzeugt werden“)
Initial ist die Anzahl der Symbole a gleich der Anzahl der Symbole b und B und ist gleich der Anzahl der Symbole c und C. Alle Regeln in P erhalten diese Gleichheit.
$\Rightarrow$ In jeder Satzform ist die Anzahl aller Symbole a gleich der Anzahl aller Symbole b und B gleich der Anzahl aller Symbole c und C.
a-Symbole können nur durch die Regeln (1) und (2) erzeugt werden, und daher stehen sie immer ganz vorne. Das heißt in jedem $w \in L(G)$ stehen vorne immer a-Symbole.
b-Symbole werden nur durch die Regeln (4) und (5) erzeugt, dadurch folgen b-Symbole immer den a-Symbolen.
Gleiches gilt für c-Symbole.
$\Rightarrow$ In jedem $w \in L(G)$ stehen die a-Symbole stets vor den b-Symbolen und diese vor den c-Symbolen.
Zusammen mit der *Anzahl-Invarianten* gilt: Nur Wörter der Form $a^n b^n c^n$ können erzeugt für $n \geq 1$. □

Anmerkungen:
G ist kontextsensitiv.
Die Argumentation „Die c-Symbole können ohnehin nirgendwo anders sein“ reicht nicht aus, weil man dabei nicht auf weitere mögliche Regeln eingeht.

$\mathcal{L}_\mathbf{i}$ sei die Menge der Sprachen, die durch eine CHOMSKY-i-Grammatik erzeugt werden kann.

Tabelle 3: Diese Zusammenhänge zwischen Erzeugern und Erkennern von Sprachen bestehen, welche wir teilweise noch beweisen werden.

Sprachtyp	Menge	Erzeuger	Erkenner
rekursiv aufzählbar	$\mathcal{L}_0$	CHOMSKY-0	Turingmaschinen
kontextsensitiv	$\mathcal{L}_1$	CHOMSKY-1	Linear beschränkte Turingmaschinen
kontextfrei	$\mathcal{L}_2$	CHOMSKY-2	Kellerautomaten
regulär	$\mathcal{L}_3$	CHOMSKY-3	Automaten

Satz 3.3 *Eine Sprache L ist genau dann rekursiv aufzählbar, wenn es eine* CHOMSKY*-0-Grammatik G mit $L = L(G)$ gibt ($\mathcal{L}_0$ ist die Menge der rekursiv aufzählbaren Sprachen).*

BEWEIS:
„⇐“: CHOMSKY-0-Grammatik G sei gegeben.
Ziel: Eine Turingmaschine M angeben, die genau die Sprache $L(G)$ akzeptiert.
Nichtdeterministische Turingmaschine M:

```
1 Die Eingabe sei w;
2 Rate eine Ableitung in G und vergleiche das so erzeugte Wort mit w;
3 wenn Gleichheit besteht dann
4 |  Halte akzeptierend;
5 else
6 |  Endlosschleife;
7 end
```

Es gibt dabei genau dann eine akzeptierende Rechnung, wenn w in $L(G)$ enthalten ist.
Also ist $L(G)$ rekursiv aufzählbar.
„⇒“: Eine 1-Band-Turingmaschine M sei gegeben.
Ziel: Eine Grammatik G angeben mit $L(G) = \{w | M$, gestartet mit Eingabe w, hält.$\}$.
O. B. d. A.:

- M hat nur einen akzeptierenden Endzustand q_{acc}
  (Bei mehreren Endzuständen q_i: $\delta(q_i, x) = (q_{acc}, x, N)$ hinzufügen).
- Das Band ist leer, wenn der Endzustand erreicht ist.
- M arbeitet nur auf den Zellen $i, i \geq 0$, und die Zelle -1 enthält das Sonderzeichen „►“.
- M ist zu Beginn im Zustand q_0.

$$\blacktriangleright\, q_0 w \vdash K_1 \vdash K_2 \vdash \ldots \vdash \blacktriangleright\, q_{acc} BBB \ldots \text{ (akzeptierende Rechnung für } w)$$

Ziel: Grammatik $G_M = (V, \Sigma, \mathcal{P}, \mathcal{S})$ angeben, die diese Rechnung *rückwärts* ausführt.

Variablen/Nichtterminale: $V = Q \cup \Gamma \setminus \Sigma \cup \{\mathcal{S}\} \cup \{\blacktriangleright, \blacktriangleleft\}$
Terminale Σ werden aus der Turingmaschine übernommen.
Startsymbol: $\mathcal{S}$
Produktionen: $\mathcal{P}$:

- $\mathcal{S} \rightarrow \blacktriangleright\, q_{acc}\, \blacktriangleleft$
- $q_{acc} \rightarrow q_{acc} B$
  Um alle Blanksymbole zu generieren, die im Laufe der Rechnung von M irgendwann mal gelesen wurden.

- $a'q' \to qa$
  Zur inversen Umsetzung aller δ-Übergänge der Form $\delta(q,a) = (q', a', R)$, das heißt von Konfiguration $K = \alpha qa\beta$ kann in die Konfiguration $K' = \alpha a'q'\beta$ übergegangen werden.
- $q'a' \to qa$
  Zur inversen Umsetzung aller δ-Übergänge der Form $\delta(q,a) = (q', a', N)$, das heißt von Konfiguration $K = \alpha qa\beta$ kann in die Konfiguration $K' = \alpha q'a'\beta$ übergegangen werden.
- $\forall b \in \Gamma : q'ba' \to bqa$
  Zur inversen Umsetzung aller δ-Übergänge der Form $\delta(q,a) = (q', a', L)$, das heißt von Konfiguration $K = \alpha bqa\beta$ kann in die Konfiguration $K' = \alpha q'ba'\beta$ übergegangen werden.
- $B \blacktriangleleft\!\!\to\!\!\blacktriangleleft$
  Um am Ende alle Blanksymbole zu entfernen.
  $\blacktriangleleft\!\!\to \varepsilon$
  Um anschließend das $\blacktriangleleft$-Symbol zu entfernen.
- $\blacktriangleright\, q_0 \to \varepsilon$
  Um am Ende das Zustandssymbol und das $\blacktriangleright$-Symbol zu entfernen.

$$\mathcal{S} \underbrace{\overset{*}{\to}}_{\text{Phase 1}} \blacktriangleright\, q_{acc} B^* \blacktriangleleft \underbrace{\overset{*}{\to}}_{\text{Phase 2}} \blacktriangleright\, q_0 w_1 \dots w_n B^* \blacktriangleleft \underbrace{\overset{*}{\to}}_{\text{Phase 3}} w_1 \dots w_n = w \in L(M).$$

1. Endkonfiguration inklusive Blanksymbole erzeugen.
2. Rechnung von M rückwärts ausführen.
3. Aufräumen (Blanksymbole, q_0, $\blacktriangleright$ und $\blacktriangleleft$ löschen).

Alternativ kann auch in Vorwärtsrichtung vorgegangen werden:

Idee: $\mathcal{S} \overset{*}{\to} \begin{bmatrix}\varepsilon\\B\end{bmatrix}\begin{bmatrix}\varepsilon\\B\end{bmatrix}\dots\begin{bmatrix}\varepsilon\\B\end{bmatrix} q_0 \begin{bmatrix}w_1\\w_1\end{bmatrix}\begin{bmatrix}w_2\\w_2\end{bmatrix}\dots\begin{bmatrix}w_n\\w_n\end{bmatrix}\begin{bmatrix}\varepsilon\\B\end{bmatrix}\begin{bmatrix}\varepsilon\\B\end{bmatrix}\dots\begin{bmatrix}\varepsilon\\B\end{bmatrix}$ Wunsch: $w_1 w_2 \dots w_n \in L(M)$

$G_M = (V, \Sigma, \mathcal{P}, \mathcal{S})$, $V = \left(\{\mathcal{S}, A_1, A_2\} \cup Q \cup \left\{ \begin{bmatrix}a\\b\end{bmatrix} \,\middle|\, a \in (\Sigma \cup \{\varepsilon\}), b \in \Gamma \right\}\right)$, Σ, Γ wie bei Turingmaschine,

Produktionen $\mathcal{P}$:

$$\begin{aligned}
\mathcal{S} &\to \begin{bmatrix}\varepsilon\\B\end{bmatrix}\mathcal{S} \mid q_0 A_1 && \\
A_1 &\to \begin{bmatrix}a\\a\end{bmatrix} A_1 \mid A_2 && \forall a \in \Sigma \\
A_2 &\to \begin{bmatrix}\varepsilon\\B\end{bmatrix} A_2 \mid \varepsilon && \\
q\begin{bmatrix}a\\x\end{bmatrix} &\to \begin{bmatrix}a\\y\end{bmatrix} q' && \forall q \in Q\, \forall x, y \in \Gamma, \delta(q,x) = (q', y, R)\, \forall a \in \Sigma \cup \{\varepsilon\} \\
q\begin{bmatrix}a\\x\end{bmatrix} &\to q'\begin{bmatrix}a\\y\end{bmatrix} && \forall q \in Q\, \forall x, y \in \Gamma, \delta(q,x) = (q', y, N)\, \forall a \in \Sigma \cup \{\varepsilon\} \\
\begin{bmatrix}b\\z\end{bmatrix} q\begin{bmatrix}a\\x\end{bmatrix} &\to q'\begin{bmatrix}b\\z\end{bmatrix}\begin{bmatrix}a\\y\end{bmatrix} && \forall q \in Q\, \forall x, y \in \Gamma, \delta(q,x) = (q', y, L)\, \forall a, b \in \Sigma \cup \{\varepsilon\}\, \forall z \in \Gamma \\
\begin{bmatrix}a\\x\end{bmatrix} q &\to qaq && \forall q \in F\, \forall a \in \Sigma \cup \{\varepsilon\}\, \forall x \in \Gamma \\
q\begin{bmatrix}a\\x\end{bmatrix} &\to qaq && \forall q \in F\, \forall a \in \Sigma \cup \{\varepsilon\}\, \forall x \in \Gamma \\
q &\to \varepsilon && \forall q \in F
\end{aligned}$$

Korrekt: Induktion über die Schritte von M gestartet mit w. □

3.2 Endliche Automaten

Definition 3.4 *Ein* **deterministischer endlicher Automat** ***(DFA, deterministic finite automaton)*** *A ist beschrieben durch 5 Komponenten $A = (Q, \Sigma, \delta, q_0, F)$ mit:*

- *Q: Endliche Menge der Zustände*
- *Σ : Endliches Alphabet, $Q \cap \Sigma = \emptyset$*
- *$\delta : Q \times \Sigma \to Q$ Übergangsfunktion*
- *q_0 : Startzustand*
- *$F : F \subseteq Q$, akzeptierende Endzustände*

Man kann sich einen DFA als eine *Turingmaschine, die nicht schreibt und den Kopf nur nach rechts bewegt,* vorstellen.

Definition 3.5 *Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat.*

- *Die* **kanonische Fortsetzung** *von δ ist definiert als $\delta : Q \times \Sigma^* \to Q$,*
  Für $q \in Q, w = w_1 \dots w_n \in \Sigma^n$ gilt:

$$\delta(q, w) = q' \text{ falls es } q_1, \dots, q_n \in Q \text{ gibt mit}$$

$$\begin{gathered} q_n = q' \\ \delta(q, w_1) = q_1 \\ \delta(q_i, w_{i+1}) = q_{i+1} \text{ für } i = 1, \dots, n-1 \end{gathered}$$

  Alternativ:

$$\begin{gathered} \delta(q, \varepsilon) = q \\ \delta(q, w_1 \dots w_n) = \delta\left(\delta(q, w_1 \dots w_{n-1}), w_n\right) = \delta\left(\delta(q, w_1), w_2 \dots w_n\right) \text{ für alle } q \in Q, w_i \in \Sigma \end{gathered}$$

- *Der DFA A* **akzeptiert** *$w \in \Sigma^*$, falls $\delta(q_0, w) \in F$. Das bedeutet: A akzeptiert $w \in \Sigma^*$, wenn er, gestartet mit Eingabe w im Zustand q_0, in einem Zustand aus F landet, nachdem* ***alle*** *Zeichen von w gelesen und verarbeitet worden sind. Die Zustände aus $Q \setminus F$ sind verwerfende Zustände.*
- *Die von A* **akzeptierte Sprache** *ist*

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

Die Laufzeit eines deterministischen endlichen Automaten ist immer n, wobei n die Länge der Eingabe ist.

Beispiel:

$$\text{Sei } A_1 = (Q, \Sigma, \delta, q_0, F)$$

$$\text{mit } Q = \{q_0, q_1\},\ \Sigma = \{0, 1\},\ F = \{q_1\}$$

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_1

$$\Rightarrow \delta(q_0, 001) = \delta(q_0, 01) = \delta(q_0, 1) = q_1 \in F$$

$$\Rightarrow 001 \in L(A_1)$$

$$L(A_1) = \{w \in \{0,1\}^* \mid w \text{ endet mit einer } 1\}.$$

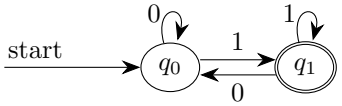


Abbildung 9: Automat A_1.

Beispiel:

$$\text{Sei } A_2 = (Q, \Sigma, \delta, q_0, F)$$

$$\text{mit } Q = \{q_0, q_1, q_2\},\ \Sigma = \{0, 1, 2, z\},\ F = \{q_0\}$$

δ	0	1	2	z
q_0	q_0	q_1	q_2	q_0
q_1	q_1	q_2	q_0	q_0
q_2	q_2	q_0	q_1	q_0

$L(A_2) = \{w \in \{0,1,2,z\}^* \mid$ die Summe der Zahlen in w nach dem letzten z ist durch 3 teilbar$\}$.

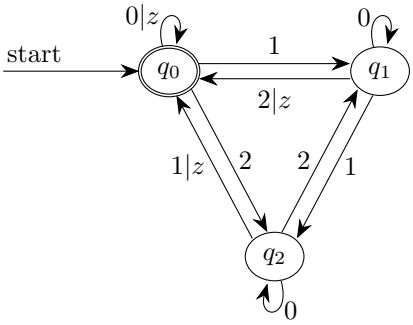


Abbildung 10: Automat A_2.

Beispiel:

$$L(A_3) = \left\{w \in \{0,1\}^* \,\middle|\, w \text{ enthält } 010\right\}$$

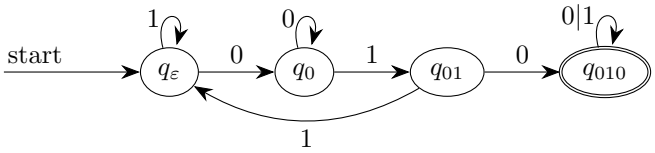


Abbildung 11: Automat A_3.

Beispiel:

$$L(A_4) = \{w \in \{a,b\}^* \mid w = w_1 w_2 \dots w_n \text{ mit } w_1 = w_n\}$$

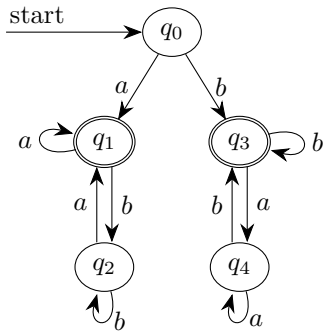


Abbildung 12: Automat A_4.

Die zugehörige reguläre Grammatik G_4 mit $L(A_4) = L(G_4)$ lautet:

$$G_4 = (V, \Sigma, \mathcal{P}, \mathcal{S} = q_0) \text{ mit}$$

$$V = \{q_0, q_1, q_2, q_3, q_4\},\ \Sigma = \{a, b\},$$

$\mathcal{P}$:

$$q_0 \to aq_1 \mid bq_3$$
$$q_1 \to aq_1 \mid bq_2 \mid \varepsilon$$
$$q_2 \to aq_1 \mid bq_2$$
$$q_3 \to bq_3 \mid aq_4 \mid \varepsilon$$
$$q_4 \to bq_3 \mid aq_4$$

Satz 3.6 *L werde von einem deterministischen endlichen Automaten A akzeptiert. Dann gibt es eine* Chomsky-*3-Grammatik (bzw.: reguläre Grammatik) G, die L erzeugt, also $L = L(A) = L(G)$.*

BEWEIS: Sei $L \subseteq \Sigma^*$ und A gegeben durch $A = (Q, \Sigma, \delta, q_0, F)$ mit $L(A) = L$.
Gesucht ist Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ vom Typ Chomsky-3 mit $L(G) = L$.
Dieser Beweis ist in zwei verschiedenen Versionen möglich. Entweder **(*1)** sind beliebige ε-Produktionen erlaubt, oder **(*2)** es sind keine ε-Produktionen außer vom Startsymbol aus erlaubt. Definiere G wie folgt:

- $V = Q$
- $\mathcal{S} = q_0$
- $\mathcal{P}$:
  Für alle $\delta(q, a) = q'$ nehme Produktion $q \to aq'$ in $\mathcal{P}$ auf.
  (*1): Für alle $q \in F$ nehme Produktionen $q \to \varepsilon$ in $\mathcal{P}$ auf.
  (*2): Für alle $\delta(q, a) = q'$ mit $q' \in F$ nehme Produktionen $q \to a$ in $\mathcal{P}$ auf sowie $q_0 \to \varepsilon$ falls $\varepsilon \in L$.

Zu zeigen:

$$w_1 \dots w_n \in L(A) \Leftrightarrow w_1 \dots w_n \in L(G).$$

$w_1 \dots w_n \in L(A)$
$\Leftrightarrow$ Es gibt Zustände $q_1, \dots, q_n \in Q$ mit $\delta(q_i, w_{i+1}) = q_{i+1}$ für $i = 0 \dots n-1$, $q_n \in F$
(*1) $\Leftrightarrow$ es gibt Variablen $q_1, \dots, q_n \in V$ mit $S \to w_1 q_1 \to w_1 w_2 q_2 \overset{*}{\to} w_1 \dots w_n q_n \to w_1 \dots w_n$
(*2) $\Leftrightarrow$ es gibt Variablen $q_1, \dots, q_{n-1} \in V$ mit $S \to w_1 q_1 \to w_1 w_2 q_2 \overset{*}{\to} w_1 \dots w_{n-1} q_{n-1} \to w_1 \dots w_n$
$\Leftrightarrow$ $w_1 \dots w_n \in L(G)$.

□

Damit ist gezeigt, dass Typ-3-Grammatiken mindestens so viel erzeugen können, wie deterministische endliche Automaten erkennen können.

3.3 Nichtdeterministische endliche Automaten (NFA)

$$L_1 = \{w \in \{0,1\}^* \mid w \text{ hat als drittletztes Symbol } 1\}$$

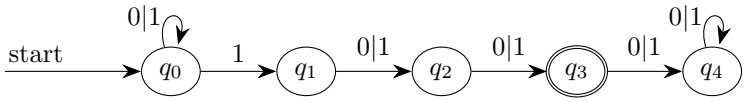


Abbildung 13: Automat N_1 mit $L(N_1) = L_1$.

Auch ein ε-Übergang, also das „spontane“ Zustandwechseln, gehört zum Nichtdeterminismus:

$$L_2 = \{w \in \{0,1\}^* \mid w \text{ enthält die Teilfolge 101 oder 11}\}$$

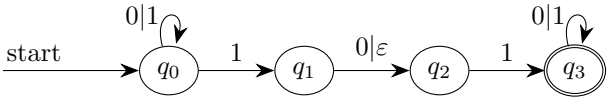


Abbildung 14: Automat N_2 mit $L(N_2) = L_2$

Die Schleife in q_3 ist übrigens notwendig, damit jedes Zeichen abgearbeitet wird - Eine Grundvoraussetzung für endliche Automaten.

Definition 3.7 *Ein* **nichtdeterministischer endlicher Automat** ***(NFA, non-deterministic finite automaton)*** *N ist beschrieben durch die 5 Komponenten $N = (Q, \Sigma, \delta, q_0, F)$ mit:*

- *Q: Endliche Menge der Zustände*
- *Σ : Endliches Alphabet, $Q \cap \Sigma = \emptyset$*
- *$\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$, wobei $\mathcal{P}$ die Potenzmenge ist und $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.*
- *q_0 : Startzustand*
- *F : $F \subseteq Q$, akzeptierende Endzustände*

N **akzeptiert** *$w \in \Sigma^*$, falls*

- *$w = w_1 \dots w_n \in \Sigma_\varepsilon^*$*
- *$\delta(q_0, w_1 \dots w_n) \cap F \neq \emptyset$*

Die **kanonische Fortsetzung** *von δ ist definiert als $\delta : Q \times \Sigma_\varepsilon^* \to \mathcal{P}(Q)$,*

$$\begin{aligned}
\delta(q, w) &= \delta(q, w_1 \dots w_n) \\
&= \{r \in Q \mid \exists q' \in \delta(q, w_1 \dots w_{n-1}) : r \in \delta(q', w_n)\} \text{ mit } w_i \in \Sigma_\varepsilon \\
&= \bigcup_{q' \in \delta(q, w_1 \dots w_{n-1})} \delta(q', w_n)
\end{aligned}$$

Satz 3.8 *Sei L eine reguläre Sprache. Dann gibt es einen nichtdeterministischen, endlichen Automaten, der L akzeptiert.*

BEWEIS: Sei $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ eine Typ-3-Grammatik, die L erzeugt. Ziel: Konstruktion von N.

- $Q = V \cup \{q_f\}$ mit $q_f \notin V$
- Σ bleibt unverändert
- $q_0 = \mathcal{S}$
- Aus Regelmenge P:
  1. Falls $(u \to av) \in P$, $u, v \in V$, $a \in \Sigma$:
     $\Rightarrow v \in \delta(u, a)$ („Lege v in die Menge $\delta(u, a)$“)
  2. Falls $(u \to a) \in P$, $u \in V$, $a \in \Sigma$:
     $\Rightarrow q_f \in \delta(u, a)$
  3. Falls $(u \to \varepsilon) \in P$, $u \in V$:
     $\Rightarrow u \in F$
- $q_f \in F$

$w \in L \Leftrightarrow w \in L(N)$ (Übung) □

Satz 3.9 *Sei N ein nichtdeterministischer, endlicher Automat. Dann gibt es einen deterministischen, endlichen Automaten A mit $L(N) = L(A)$.*

BEWEIS: Sei $N = (Q, \Sigma, \delta, q_0, F)$.
1. Schritt: Es gibt keine ε-Übergänge.
Potenzmengenkonstruktion
(Erzeugung des deterministischen, endlichen Automaten $A = (Q', \Sigma, \delta', q_0', F')$):

- $Q' = \mathcal{P}(Q)$
- $q_0' = \{q_0\}$
- $F' = \{R \in Q' \mid R \cap F \neq \emptyset\}$
- $\delta'(R, a) = \bigcup_{r \in R} \delta(r, a) = \{q \in Q \mid q \in \delta(r, a), r \in R\}$ $(\in Q')$

$L(N) = L(A)$, da
$w \in L(N) \Leftrightarrow \delta(q_0, w) \cap F \neq \emptyset \Leftrightarrow \delta'(q_0', w) \in F' \Leftrightarrow w \in L(A)$

2. Schritt: Auch ε-Übergänge vorhanden.

$$E(R) = \{q \in Q \mid \exists r_1, \ldots, r_k \in Q,\, r_1 \in R, r_k = q \,:\, \delta(r_i, \varepsilon) \ni r_{i+1},\, i = 1, \ldots, k-1\}$$

Da auch 0 ε-Übergänge vorkommen können, gilt: $R \subseteq E(R)$.

- $q_0' = E(\{q_0\})$
- $F' = \{R \in Q' \mid E(R) \cap F \neq \emptyset\}$
- $\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)) = \{q \in Q \mid q \in E(\delta(r, a)), r \in R\}$

Korrektheit: Übung. □

Beispiel:

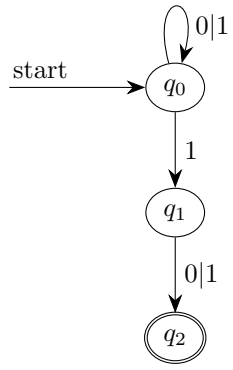
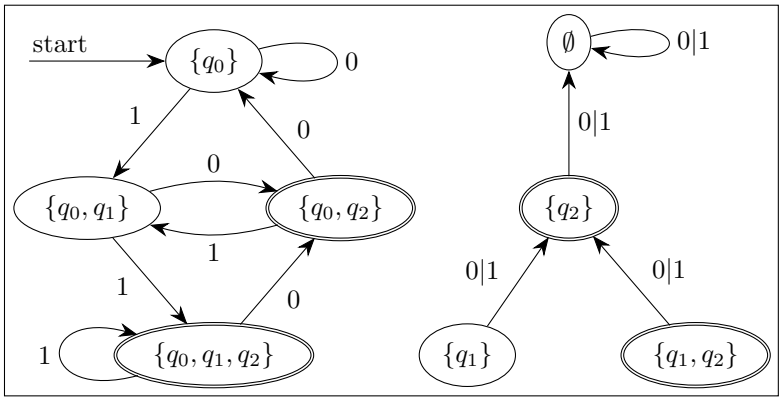


Abbildung 15: NFA N_1



$F' = \{R \in Q' \mid E(R) \cap F \neq \emptyset\} = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$

Abbildung 16: DFA A_1 nach Potenzmengenkonstruktion von N_1

Beispiel:

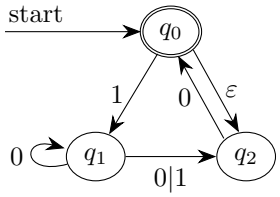


Abbildung 17: Nichtdeterministischer Automat N_2

$q_0' = E(\{q_0\}) = \{q_0, q_2\}$ $\quad F' = \{R \in Q' \mid E(R) \cap F \neq \emptyset\} = \{\{q_0\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_0, q_1, q_2\}\}$

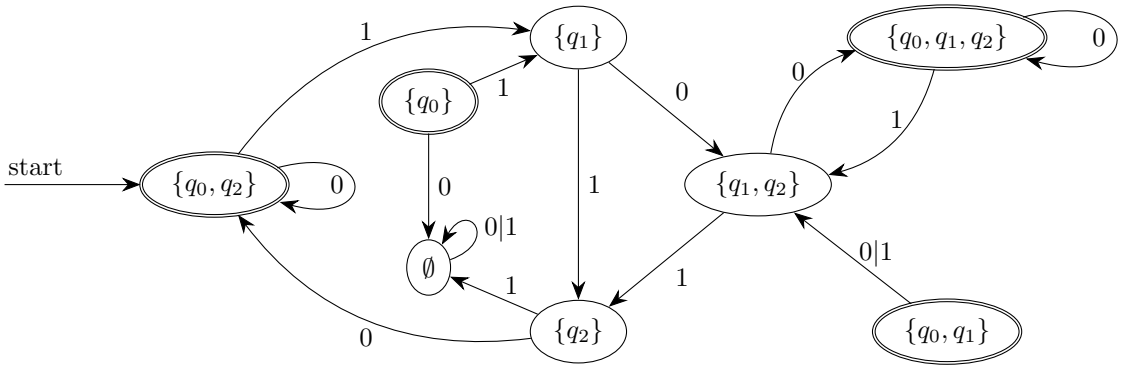


Abbildung 18: Deterministischer Automat A_2 nach Potenzmengenkonstruktion von N_2

Korollar 3.10 *Sei L eine reguläre Sprache. Dann gibt es einen DFA A mit $L(A) = L$.*

Korollar 3.11 *L ist genau dann regulär, wenn es einen DFA A gibt mit $L = L(A)$.*

3.4 Minimierung endlicher Automaten

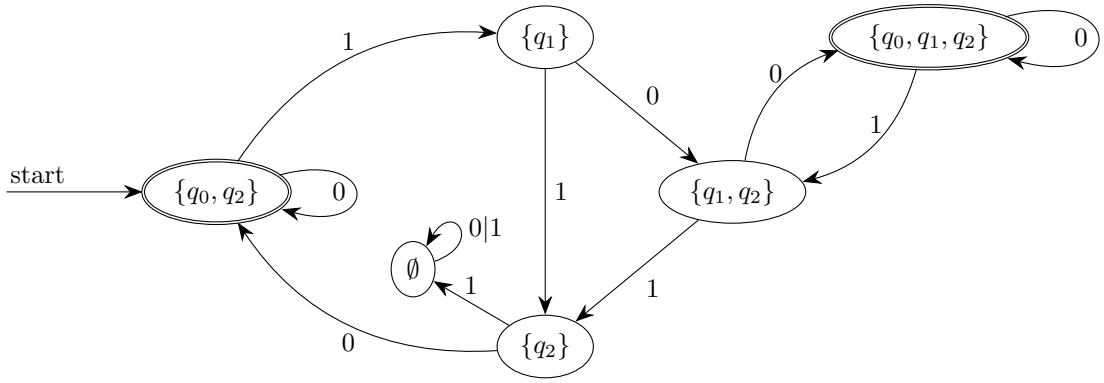


Abbildung 19: Deterministischer Automat A_2 aus Abbildung 18 ohne Zustände, die vom Startzustand nicht erreicht werden können.

Definition 3.12 *Sei L eine reguläre Sprache.*
Der deterministische, endliche Automat A heißt **Minimalautomat** *für L, falls gilt:*

(i) A akzeptiert L.

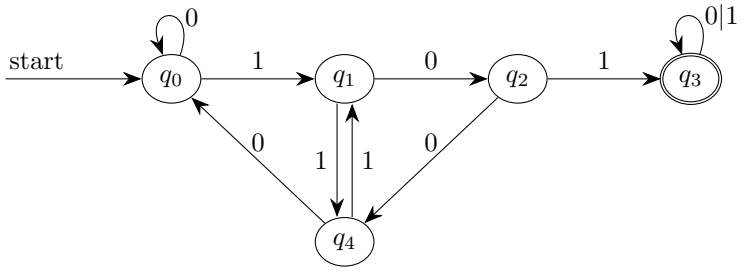
(ii) Jeder andere deterministische, endliche Automat A', der L akzeptiert ($L(A') = L$), hat mindestens so viele Zustände wie A.

Ein Zustand q in A ist *unerreichbar*, falls es kein Wort $w \in \Sigma^*$ gibt mit $\delta(q_0, w) = q$.
Unerreichbare Zustände können mittels Breiten- oder Tiefensuche identifiziert und gelöscht werden. Der Automat, der nach dem Löschen herauskommt, braucht aber nicht unbedingt ein Minimalautomat zu sein.

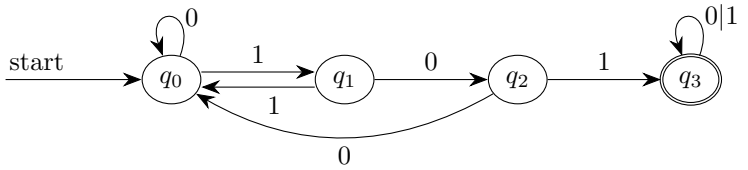
Definition 3.13 *Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer, endlicher Automat.*
Zwei Zustände $q, q' \in Q$ heißen **äquivalent**, *wenn für jedes $w \in \Sigma^*$ gilt:*

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

Wir schreiben: $q \equiv q'$.
Da „$\equiv$“ eine Äquivalenzrelation ist, wird Q durch „$\equiv$“ in **Äquivalenzklassen** *partitioniert. Die Klasse, zu dem q gehört, bezeichnen wir mit $[q]$.*



(a) Automat A_1. Es gilt: $\forall w \in \Sigma^* : \delta(q_0, w) \in F \Leftrightarrow \delta(q_4, w) \in F$



(b) Automat A_2. Zustände q_0 und q_4 aus Automat A_1 wurden zu einem Zustand vereinigt.

Abbildung 20: Zwei deterministische Automaten, welche die gleiche Sprache akzeptieren.

Bezüglich des Automaten A_1 in Abbildung 20 gilt demnach:

$$[q_0] = [q_4] = \{q_0, q_4\}, \quad [q_1] = \{q_1\}, \quad [q_2] = \{q_2\}, \quad [q_3] = \{q_3\}$$

Warum ist $q_1 \not\equiv q_2$?
Man muss ein Teilwort finden, welches einmal von q_i aus in F ist, und einmal nicht.
Man nehme nun das Wort 01. Hier gilt: $\delta(q_1, 01) \in F$, aber auch $\delta(q_2, 01) \notin F$.
Was **nicht** ausreicht, ist, zu zeigen, dass $w = 0$ in verschiedenen Zuständen landet!

Definition 3.14 *Sei $A = (Q = \{q_0, \dots q_m\}, \Sigma, \delta, q_0, F)$ ein deterministischer, endlicher Automat. Der zu A gehörende **Äquivalenzautomat** $A' = (Q', \Sigma, \delta', q_0', F')$ ist definiert durch:*

- $Q' = \{[q_0], [q_1], \dots, [q_m]\}$ *(Duplikate $[q_i]$ bzw. $[q_j]$ fallen in Mengen automatisch weg)*
- $q_0' = [q_0]$
- $\forall q \in Q,\ a \in \Sigma : \delta'([q], a) = [\delta(q, a)]$
- $F' = \{[q] \mid q \in F\}$

Satz 3.15 *Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer, endlicher Automat und $A' = (Q', \Sigma, \delta', q_0', F')$ der zu A gehörende Äquivalenzautomat, dann ist $L(A) = L(A')$.*

BEWEIS: *correct by construction.* □

Nun ein Algorithmus zum finden äquivalenter Zustände:
Algorithmus **NEQ** (***N**ot-**Eq**uivalent*):

```
1 Eingabe: ein deterministischer, endlicher Automat A = (Q, Σ, δ, q0, F);
2 Markiere alle Paare von Zuständen {p, q}, von denen genau einer ein akzeptierender Zustand ist;
3 solange es noch ein Paar {p, q} von Zuständen und ein Zeichen a ∈ Σ gibt,
4 sodass {δ(p, a), δ(q, a)} bereits markiert ist tue
5 |  markiere {p, q};
6 Ende
7 Ausgabe: Die nicht-markierten Zustandspaare bilden Paare äquivalenter Zustände.
```

Beispiel:

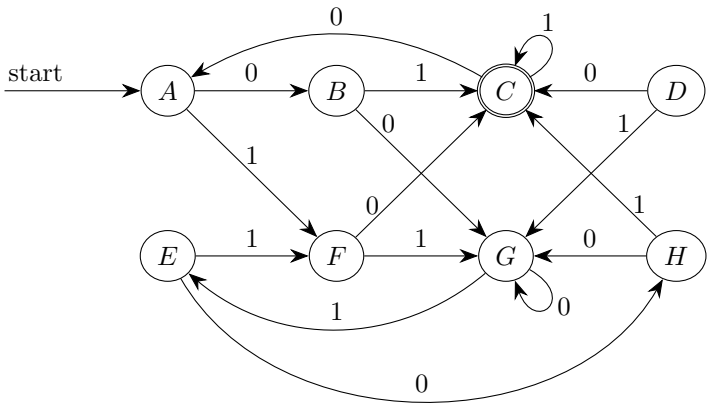


Abbildung 21: Deterministischer Automat A.

Durchlauf von NEQ angewendet auf deterministischen Automaten A aus Abbildung 21:

Tabelle 4: Start von NEQ auf A - nur Paare mit genau einem Endzustand werden als nicht äquivalent markiert

A			✓					
B			✓					
C	✓	✓		✓	✓	✓	✓	✓
D			✓					
E			✓					
F			✓					
G			✓					
H			✓					
	A	B	C	D	E	F	G	H

Die Tabelle ist natürlich symmetrisch. Es reicht den Teil oberhalb oder unterhalb der Diagonalen darzustellen.

Tabelle 5: Nach und nach werden diese Paare von NEQ als nicht äquivalent erkannt (n. ä.).

n. ä.	weil
$\{A, B\}$	$\{\delta(A,1), \delta(B,1)\} = \{F, C\}$
$\{A, D\}$	$\{\delta(A,0), \delta(D,0)\} = \{B, C\}$
$\{A, F\}$	$\{\delta(A,0), \delta(F,0)\} = \{B, C\}$
$\{A, H\}$	$\{\delta(A,1), \delta(H,1)\} = \{F, C\}$
$\{B, D\}$	$\{\delta(B,0), \delta(D,0)\} = \{G, C\}$
$\{B, E\}$	$\{\delta(B,1), \delta(E,1)\} = \{C, F\}$
$\{B, F\}$	$\{\delta(B,0), \delta(F,0)\} = \{G, C\}$
$\{B, G\}$	$\{\delta(B,1), \delta(G,1)\} = \{C, E\}$
$\{D, E\}$	$\{\delta(D,0), \delta(E,0)\} = \{C, H\}$
$\{D, G\}$	$\{\delta(D,0), \delta(G,0)\} = \{C, G\}$
$\{D, H\}$	$\{\delta(D,0), \delta(H,0)\} = \{C, G\}$
$\{E, F\}$	$\{\delta(E,0), \delta(F,0)\} = \{H, C\}$
$\{E, H\}$	$\{\delta(E,1), \delta(H,1)\} = \{F, C\}$
$\{F, G\}$	$\{\delta(F,0), \delta(G,0)\} = \{C, G\}$
$\{F, H\}$	$\{\delta(F,1), \delta(H,1)\} = \{G, C\}$
$\{G, H\}$	$\{\delta(G,1), \delta(H,1)\} = \{E, C\}$
$\{A, G\}$	$\{\delta(A,0), \delta(G,0)\} = \{B, G\}$
$\{E, G\}$	$\{\delta(E,0), \delta(G,0)\} = \{H, G\}$

Tabelle 6: Finale Tabelle aller nicht äquivalenten Paare nach dem Durchlauf von NEQ auf A.

A		✓	✓	✓		✓	✓	✓
B	✓		✓	✓	✓	✓	✓	
C	✓	✓		✓	✓	✓	✓	✓
D	✓	✓	✓		✓		✓	✓
E		✓	✓	✓		✓	✓	✓
F	✓	✓	✓		✓		✓	✓
G	✓	✓	✓	✓	✓	✓		✓
H	✓		✓	✓	✓	✓	✓	
	A	B	C	D	E	F	G	H

Die Tabelle ist natürlich symmetrisch. Es reicht den Teil oberhalb oder unterhalb der Diagonalen darzustellen.

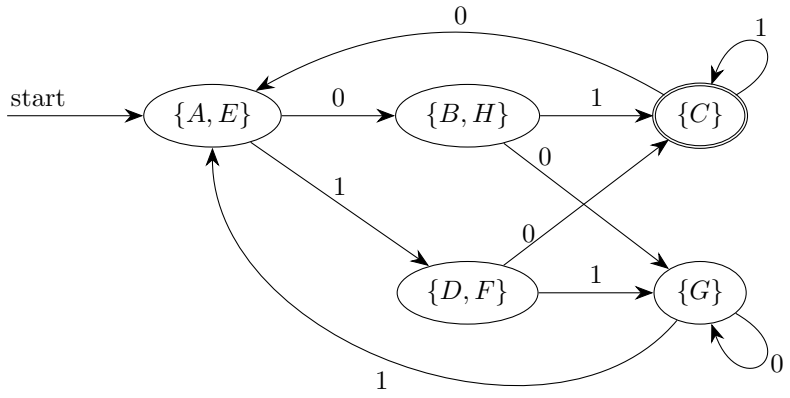


Abbildung 22: Äquivalenzautomat A' zum Automaten A aus Abbildung 21.

(i) NEQ markiert nur Paare nichtäquivalenter Zustände.

(ii) Jedes Paar nichtäquivalenter Zustände wird von NEQ markiert.

Zu (i): (Beweis mittels Induktion)
Induktionsanfang:
Im ersten Schritt von NEQ werden in der Tat nichtäquivalente Paare markiert.
(Beleg / "Beweis" ist das Wort $w = \varepsilon$.)
Induktionsschritt $(i - 1 \to i)$:
Es werde nun $\{p, q\}$ im i-ten Durchlauf der Schleife markiert.
Zu zeigen: $p \not\equiv q$.
p, q markiert $\Rightarrow$ es gibt ein markiertes $\{p', q'\}$ mit $\{\delta(p, a), \delta(q, a)\} = \{p', q'\}$ und $\{p', q'\}$ sind per Induktionsannahme nicht äquivalent $\Rightarrow p \not\equiv q$, denn da $p' \not\equiv q'$ gibt es ein Wort w sodass o.B.d.A. $\delta(p', w) \in F$ und $\delta(q', w) \notin F$ und damit ist $\delta(p, aw) \in F$ und $\delta(q, aw) \notin F$.

Zu (ii): (indirekter Beweis)
Annahme: Es gibt ein Paar $\{p, q\}$, welches nicht markiert ist, aber nicht äquivalent ist.
Wenn $p \not\equiv q$, dann gibt es ein $w \in \Sigma^*$ mit $\delta(p, w) \in F$ und $\delta(q, w) \notin F$ (oder umgekehrt). Unter allen nicht markierten Paaren, die markiert werden sollten, nehme ein Paar $\{p, q\}$, welches ein kürzestes w hat mit der Eigenschaft dass $\delta(p, w) \in F$ und $\delta(q, w) \notin F$. Sei $w = w_1 \dots w_n$. $\{\delta(p, w_1), \delta(q, w_1)\}$ ist ein Paar nicht

äquivalenter Zustände, da p, q nach Annahme nicht äquivalent sind. Also beweist $w_2 \dots w_n$, dass $\delta(p, w_1)$ und $\delta(q, w_1)$ nicht äquivalent sind. Aber $\{\delta(p, w_1), \delta(q, w_1)\}$ darf nicht markiert sein, da sonst $\{p, q\}$ markiert wäre. Dies widerspricht aber der Annahme, dass $\{p, q\}$ ein Paar mit minimal langem w ist. ↯

Da es nur $\binom{|Q|}{2} = |Q| \cdot (|Q| - 1)/2$ Paare gibt, wird die Schleife maximal $\mathcal{O}(|Q|^2 \cdot |\Sigma|)$ mal durchlaufen.

Satz 3.16 *Gegeben sei ein deterministischer, endlicher Automat $A = (Q, \Sigma, \delta, q_0, F)$. Dann kann der zu A gehörende Äquivalenzautomat in Zeit $\mathcal{O}\left(|Q|^2 \cdot |\Sigma|\right)$ auf einer Registermaschine berechnet werden.*

Satz 3.17 *Sei D ein deterministischer, endlicher Automat. Dann ist der Äquivalenzautomat A zu D ein Minimalautomat für $L(D)$.*

BEWEIS: Seien $A = (Q, \Sigma, \delta, q_0, F)$ und $A' = (Q', \Sigma, \delta', q_0', F')$ zwei deterministische, endliche Automaten, $Q \cap Q' = \emptyset$, $q, q' \in Q \cup Q'$.
$q \equiv q'$ mit $q, q' \in Q \cup Q'$, falls eine der folgenden Bedingungen gilt:

(i) $q, q' \in Q$: für alle $w \in \Sigma^* : \delta(q, w) \in F \Leftrightarrow \delta(q', w) \in F$

(ii) $q, q' \in Q'$: für alle $w \in \Sigma^* : \delta'(q, w) \in F' \Leftrightarrow \delta'(q', w) \in F'$

(iii) $q \in Q\,, q' \in Q'$: für alle $w \in \Sigma^* : \delta(q, w) \in F \Leftrightarrow \delta'(q', w) \in F'$.

Akzeptieren A und A' dieselbe Sprache, so gilt: $q_0 \equiv q_0'$.
Sei M ein Minimalautomat für $L(D)$ (D soll keine von q_0 nicht erreichbaren Zustände enthalten).
Wir zeigen: Jeder Zustand q von A ist zu einem Zustand q' von M äquivalent.
Wir wissen: Die beiden Startzustände sind äquivalent.

Sei q ein beliebiger Zustand von A. Es gibt ein $w \in \Sigma^*$ mit $\delta(q_0, w) = q$ für den Startzustand q_0 von A und der δ-Funktion δ von A.
Betrachte q' von M mit: $\delta_M(q_0', w) = q'$, wobei q_0' der Startzustand von M ist.

Wir behaupten: $q \equiv q'$, also $\delta(q, u) \in F \Leftrightarrow \delta_M(q', u) \in F'$ für alle $u \in \Sigma^*$

Annahme:
Es gibt ein $u \in \Sigma^*$ mit $\delta(q, u) \in F$ und $\delta_M(q', u) \notin F'$ (oder umgekehrt), dann ist $\delta(q_0, wu) \in F$ und $\delta_M(q_0', wu) \notin F'$
$\Rightarrow L(A) \neq L(M)$ ↯
$\Rightarrow q \equiv q'$

Zu jedem q von A gibt es in M einen äquivalenten Zustand q'. Nun müssen unterschiedliche Zustände von A äquivalent zu unterschiedlichen Zuständen von M sein. Dies gilt, da $\equiv$ eine Äquivalenzrelation und somit transitiv ist.
$\Rightarrow A$ und M haben die gleiche Anzahl an Zuständen. □

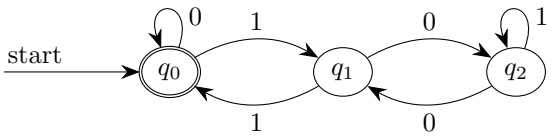
Satz 3.18 *Der Minimalautomat ist eindeutig.*

BEWEIS: Wie bei Satz 3.17. □

3.5 Das Pumping-Lemma für reguläre Sprachen und nichtreguläre Sprachen

Beispiel:

$$L = \{w \mid w \in \{0,1\}^*, (w)_2 \mod 3 = 0\}$$



Der Automat ist im Zustand q_i genau dann, wenn für das bisher gelesene Wort w gilt $(w)_2 \mod 3 = i$. Dies ist möglich, denn für $x \in \{0,1\}$ und $w \in \{0,1\}^*$ gilt

$$(wx)_2 \mod 3 = ((w)_2 \cdot 2 + x) \mod 3 = (((w_2) \mod 3) \cdot 2 + x) \mod 3.$$

Damit ergibt sich der angegebene Automat. Auch $L = L(A)$ ist klar, da sich der Automat genau dann im Zustand q_0 befindet, wenn für das gelesene Wort w gilt $(w)_2 \mod 3 = 0$.
Betrachte das Wort $w = 1001$. Dieses Wort wird von dem Automaten A über die Folge der Zustände q_0, q_1, q_2, q_1, q_0 akzeptiert. Da mehr als $|Q|$ Zustände in der akzeptierenden Rechnung vorkommen muss es einen Zustand geben, der mehrfach auftaucht. Beispielsweise taucht q_1 mehrfach auf. Der Teil der zwischen den beiden Vorkommen von q_1 gelesen wird (00), kann nun sowohl aus dem Wort entfernt (11) als auch beliebig vervielfältigt (100001, 10000001, ...) werden. Für all diese Wörter gibt es eine akzeptierende Rechnung von A, da in der akzeptierenden Rechnung die Teilfolge der Zustände zwischen den beiden Vorkommen von q_1 inklusive eines der q_1 entweder entfernt oder beliebig oft wiederholt werden kann.

Diese Idee wird in folgender Definition verallgemeinert.

Definition 3.19 *Sei L eine Sprache über Σ.*
L hat die **reguläre Pump-Eigenschaft***, falls gilt:*

$$\exists n_L \in \mathbb{N}\, \forall z \in L,\, |z| \geq n_L\, \exists u, v, w \in \Sigma^* : uvw = z \text{ und}$$

(i) $|uv| \leq n_L$

(ii) $v \neq \varepsilon$

(iii) $\forall i \geq 0 : uv^i w \in L$

Beispiel:

$$L = \{a^i b^j \mid i, j \geq 0\}$$

Setze $n_L = 5$. Sei $z \in L$ beliebig, aber fest mit $|z| \geq 5$. Setze $u = \varepsilon$. v ist dann das erste Zeichen von z, w dessen Rest.

(i) $uv = v =$ erstes Zeichen von z, dessen Betrag ist $1 \leq 5$ ✓.

(ii) $v \neq \varepsilon$ ✓

(iii) $uv^i w =$ (erstes Zeichen von $z)^i w \in L$ ✓

$\Rightarrow L$ hat die reguläre Pump-Eigenschaft.

Satz 3.20 (Pumping-Lemma für reguläre Sprachen)
Wenn eine Sprache regulär ist, dann hat sie die reguläre Pump-Eigenschaft.

BEWEIS: L sei regulär. Sei $G = (V, \Sigma, P, S)$ eine reguläre Grammatik (Typ Chomsky-3) mit $L = L(G)$.
Setze $n_L := |V| + 1$.
Sei $z = a_1 \dots a_{|z|} \in L$ beliebig, aber fest, mit $|z| \geq n_L$.
Da $z \in L$ ist, gibt es eine Herleitung mittels G für z.
Sei $V_0, V_1, \dots, V_m$ die Folge der Variablen, die bei der Herleitung von z produziert werden (Es gibt bei regulären Grammatiken in jeder vom Startsymbol erreichbaren Satzform maximal eine Variable).
Falls die letzte Ableitung von der Form $V_m \to \varepsilon$ so gilt $m = |z|$.
Falls die letzte Ableitung von der Form $V_m \to a$ mit $a \in \Sigma$ so gilt $m = |z| - 1$.
Also $m \geq |z| - 1 \geq n_L - 1 \geq |V|$.
$\Rightarrow$ Es muss innerhalb von $V_0, \dots, V_{|V|}$ eine doppelt vorkommende Variable geben.
Sei also V_k die erste doppelt vorkommende Variable und $V_k = V_j$, $0 \leq k < j \leq |V| < n_L$. Im Syntaxbaum ist V_k die erste von oben kommend doppelte Variable.
Setze:

$$\begin{array}{lcl} u := a_1 \dots a_k & & V_0 \overset{*}{\to} uV_k = uV_j \\ v := a_{k+1} \dots a_j & \Rightarrow & V_k \overset{*}{\to} vV_j = vV_k \\ w := a_{j+1} \dots a_m & & V_j \overset{*}{\to} w \end{array}$$

- (i): $|uv| = j \leq n_L$ ✓
- (ii): $|v| = j - k \geq 1$, also $v \neq \varepsilon$ ✓
- (iii):
  - $i = 0$:
    $V_0 \overset{*}{\to} uV_k = uV_j \overset{*}{\to} uw$
    $\Rightarrow uw \in L$ ✓
  - $i > 0$:
    $V_0 \overset{*}{\to} uV_k \overset{*}{\to} uvV_j = uvV_k \overset{*}{\to} \dots \overset{*}{\to} uv^iV_j \overset{*}{\to} uv^iw$
    $\Rightarrow uv^iw \in L$ ✓

□

Alternativer Beweis mit Automaten:
BEWEIS: L sei regulär. Sei $A = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer, endlicher Automat mit $L = L(A)$.
Setze $n_L := |Q|$.
Sei $z = a_1 \dots a_m \in L$ beliebig, aber fest, mit $m \geq n_L$.
Da $z \in L$ ist, gibt es eine akzeptierende Rechnung von A für z.
Sei $r_0, r_1, \dots, r_m$ die Folge der Zustände, die bei der akzeptierenden Rechnung durchlaufen werden mit $r_0 = q_0$ und $r_m \in F$.
Dies sind $m + 1 \geq n_L + 1 > |Q|$ Zustände.
$\Rightarrow$ Es muss innerhalb von $r_0, \dots, r_{n_L}$ einen doppelt vorkommenden Zustand geben.
Sei also $r_k = r_j$, $0 \leq k < j \leq n_L$.
Setze:

$$\begin{array}{lll} u := a_1 \dots a_k & & \delta(q_0, u) = r_k = r_j \\ v := a_{k+1} \dots a_j & \Rightarrow & \delta(r_k, v) = r_j = r_k \\ w := a_{j+1} \dots a_m & & \delta(r_j, w) = r_m \in F \end{array}$$

- (i): $|uv| = j \leq n_L$ ✓
- (ii): $|v| = j - k \geq 1$, also $v \neq \varepsilon$ ✓
- (iii):
  - $i = 0$:

$$\delta(q_0, uw) = \delta(\underbrace{\delta(q_0, u)}_{=r_k=r_j}, w) = r_m$$

    $\Rightarrow uw \in L$ ✓
  - $i > 0$:

$$\delta(q_0, uv^i w) = \delta(\underbrace{\delta(q_0, u)}_{=r_k=r_j}, v^i w) = \delta(r_k, v^i w) = \delta(\delta(r_k, v), v^{i-1} w)$$
$$= \delta(r_k, v^{i-1} w) = \dots = \delta(r_k, w) = \delta(r_j, w) = r_m$$

    $\Rightarrow uv^i w \in L$ ✓ □

Anmerkung:
Die Rückrichtung des Pumping-Lemmas gilt nicht!
Es gibt sogar unentscheidbare Sprachen, die die reguläre Pump-Eigenschaft besitzen!
Äquivalente Formulierung des Pumping-Lemmas:
Wenn L die reguläre Pump-Eigenschaft nicht besitzt, ist sie nicht regulär.

Die Pump-Eigenschaft ist notwendig, aber nicht hinreichend für reguläre Sprachen. Das heißt, wenn L die reguläre Pump-Eigenschaft nicht hat $\Rightarrow L$ ist nicht regulär.

Beispiel: (Endliche Sprachen)
Endliche Sprachen sind regulär.
Sei L eine beliebige aber feste endliche Sprache. Demnach muss L die reguläre Pump-Eigenschaft haben.
Wie geht das mit $uv^i w \in L$ aus dem dritten Punkt einher?
Ganz einfach: Es gibt ein längstes Wort w mit $|w| = c$.
Wir wählen $n_L = c + 1$.
Dann gibt es keine $z \in L$ mit $|z| \geq n_L$. Der Allquantor bezieht sich also auf die leere Menge. Für alle Objekte in der leeren Menge gelten beliebige Eigenschaften.
$\Rightarrow$ jede endliche Sprache besitzt die reguläre Pump-Eigenschaft.

Die **Negation der regulären Pump-Eigenschaft** kann folgendermaßen formuliert werden:
Sei L eine Sprache.

$$\forall n_L \in \mathbb{N}\, \exists x \in L,\, |x| \geq n_L\, \forall u,v,w \in \Sigma^* : uvw = x :$$
$$(\underbrace{|uv| \leq n_L}_{(i)} \wedge \underbrace{v \neq \varepsilon}_{(ii)}) \Rightarrow \underbrace{\exists i \geq 0 : uv^i w \notin L}_{\neg(iii)}$$

Beispiel:
$L = \{0^n 1^n \mid n \geq 1\}$ hat nicht die reguläre Pump-Eigenschaft.
BEWEIS:
Sei $n_L \in \mathbb{N}$ beliebig, aber fest.
Setze $z = 0^{n_L} 1^{n_L} \in L$ und $|z| \geq n_L$.
Sei u, v, w mit $z = uvw$ beliebig, aber fest, wobei $|uv| \leq n_L$ und $|v| \geq 1$ gilt.
Aus $|uv| \leq n_L$ folgt: v besteht nur aus Nullen, und da $|v| \geq 1$ besteht v aus mindestens einer Null.
Setze $i := 0 \Rightarrow uw = 0^{n_L - |v|} 1^{n_L} \notin L$, da $n_L - |v| < n_L$ ist. □

Beispiel:
$L = \{aa \mid a \in \{0,1\}^*\}$ hat nicht die reguläre Pump-Eigenschaft.
BEWEIS:
Sei $n_L \in \mathbb{N}$ beliebig, aber fest.
Setze $z = 0^{n_L} 1^{n_L} 0^{n_L} 1^{n_L} \in L$ und $|z| \geq n_L$.
Sei u, v, w mit $z = uvw$ beliebig, aber fest, wobei $|uv| \leq n_L$ und $|v| \geq 1$ gilt.
Aus $|uv| \leq n_L$ folgt: v besteht nur aus Nullen, die vor der ersten Eins stehen. Da $|v| \geq 1$ gilt, besteht v aus mindestens einer dieser Nullen.
Setze $i := 0 \Rightarrow uw = 0^{n_L - |v|} 1^{n_L} 0^{n_L} 1^{n_L} \notin L$, da:

- Fall $|uw|$ ungerade: Somit kann es keine Aufteilung von uw in zwei gleichlange Teilworte geben.
- Fall $|uw|$ gerade: Somit gilt $|uw| = 4n_L - |v|$. Teilt man nun uw in zwei gleichlange Teilworte $uw = ab$ auf, so gilt, dass $2n_L - |v| \overset{|v| \geq 1}{<} 2n_L - \frac{1}{2}|v| = |a| \overset{1 \leq |v| \leq n_L}{<} 3n_L - |v|$. Somit gilt, dass a mit einer Null endet, b jedoch mit einer Eins $\Rightarrow a \neq b$

□

Beispiel:
$L = \{1^p \mid p \text{ ist Primzahl}\}$ hat nicht die reguläre Pump-Eigenschaft.
BEWEIS:
Sei $n_L \in \mathbb{N}$ beliebig, aber fest.
Sei q eine Primzahl mit $q \geq n_L + 2$.
Setze $z = 1^q \in L$ und $|z| = q \geq n_L + 2 \geq n_L$.
Sei u, v, w mit $z = uvw$ beliebig, aber fest, wobei $|uv| \leq n_L$ und $|v| \geq 1$ gilt.
Daraus folgt $|w| = |uvw| - |uv| \geq n_L + 2 - n_L \geq 2$.
Setze $i := |uw|$.
Es gilt $|uw| \geq |w| \geq 2$.
$\Rightarrow uv^i w = 1^{|uw| + |uw| \cdot |v|} = 1^{|uw| \cdot (1 + |v|)} \notin L$,
denn wegen $|uw| \geq 2$ und $|v| \geq 1$ ist $|uw| \cdot (1 + |v|)$ nicht prim. □

3.6 Reguläre Ausdrücke und Abschlusseigenschaften regulärer Sprachen

Definition 3.21 *Sei Σ ein endliches Alphabet. R ist ein* **regulärer Ausdruck** *über Σ, wenn R einer der folgenden Ausdrücke ist:*

1. *a wobei $a \in \Sigma$*
2. *ε*
3. *$\emptyset$*
4. *$(R_1 \cup R_2)$ wobei R_1, R_2 reguläre Ausdrücke sind*
5. *$(R_1 \cdot R_2)$ wobei R_1, R_2 reguläre Ausdrücke sind*
6. *(R_1^*) wobei R_1 regulärer Ausdruck ist*

Semantik
Sei R ein regulärer Ausdruck. Dann bezeichnet $L(R)$ folgende Sprache:

1. $R = a \quad L(R) = \{a\}$
2. $R = \varepsilon \quad L(R) = \{\varepsilon\}$
3. $R = \emptyset \quad L(R) = \emptyset$
4. $R = (R_1 \cup R_2) \quad L(R) = L(R_1) \cup L(R_2)$
5. $R = (R_1 \cdot R_2) \quad L(R) = L(R_1) \circ L(R_2) = \{w_1 w_2 \mid w_1 \in L(R_1),\ w_2 \in L(R_2)\}$
6. $R = (R_1^*) \quad L(R) = L(R_1)^*$

Anmerkungen:
ε ist das neutrale Element der Konkatenation, $\emptyset$ nicht! $R \cdot \emptyset = \emptyset$!
Konkatenation bindet stärker als Vereinigung.

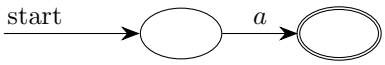
Beispiele:

$L(0^*10^*)$	$= \{w \in \{0,1\}^* \mid w$ enthält genau eine $1\}$
$L\left((0 \cup 1)^*1(0 \cup 1)^*\right)$	$= \{w \in \{0,1\}^* \mid w$ enthält mindestens eine $1\}$
$L\left((0 \cup 1)^*001(0 \cup 1)^*\right)$	$= \{w \in \{0,1\}^* \mid w$ enthält die Zeichenfolge $001\}$
$L\left(((0 \cup 1)(0 \cup 1))^*\right)$	$= \{w \in \{0,1\}^* \mid w$ hat gerade Länge$\}$
$L\left(0(0 \cup 1)^*0 \cup 1(0 \cup 1)^*1 \cup 0 \cup 1\right)$	$= \{w \in \{0,1\}^* \mid w$ beginnt und endet mit dem gleichen Zeichen$\}$

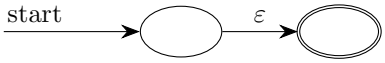
Satz 3.22 *Sei $L \subseteq \Sigma^*$ eine Sprache.*
L ist genau dann regulär, wenn es einen regulären Ausdruck R über Σ gibt mit $L = L(R)$.

BEWEIS:
„L regulär $\Leftarrow$ es gibt R“:
Vorgehen: Entlang der rekursiven Definition der regulären Ausdrücke werden NFAs angegeben.

1. a wobei $a \in \Sigma$:

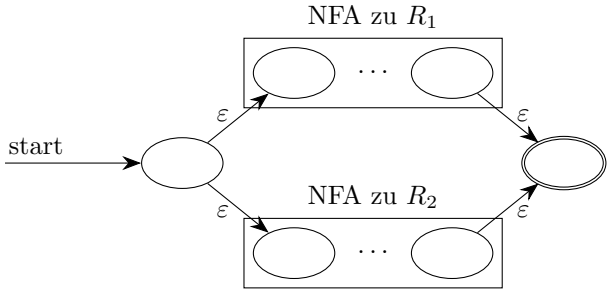


2. ε

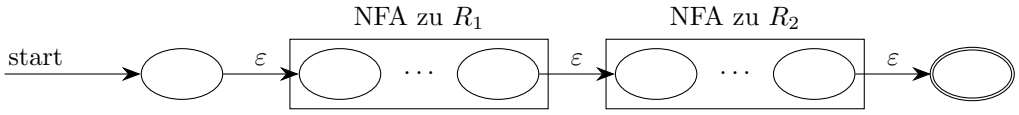


3. $\emptyset$

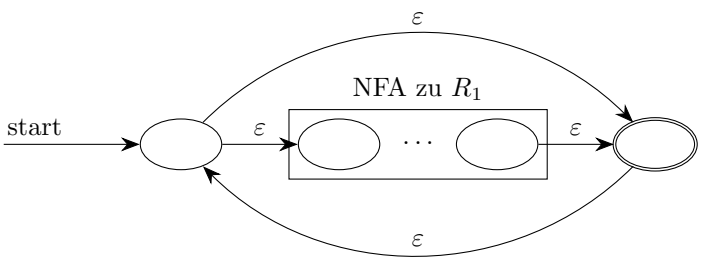
4. $(R_1 \cup R_2)$ wobei R_1, R_2 reguläre Ausdrücke sind



5. $(R_1 \cdot R_2)$ wobei R_1, R_2 reguläre Ausdrücke sind



6. (R_1^*) wobei R_1 regulärer Ausdruck ist



„L regulär $\Rightarrow$ es gibt R“:
L regulär $\Rightarrow$ es gibt einen DFA $A = (Q = \{q_1, \ldots, q_n\}, \Sigma, \delta, q_1, F)$ mit $L(A) = L$.
Ohne Beschränkung der Allgemeinheit sei $\delta(q, a)$ definiert für alle $q \in Q, a \in \Sigma$.

Variante 1 (dynamische Programmierung):
Verwende folgende Hilfskonstruktion:
Für $i, j \in \{1, \ldots, n\}$ und $k \in \{0, \ldots, n\}$ definieren wir die regulären Ausdrücke $R^k_{i,j}$ mit folgender Eigenschaft:

$$L(R^k_{i,j}) = \{x \in \Sigma^* \mid \delta(q_i, x) = q_j, \text{ kein echter Zwischenzustand hat Index} > k\}$$

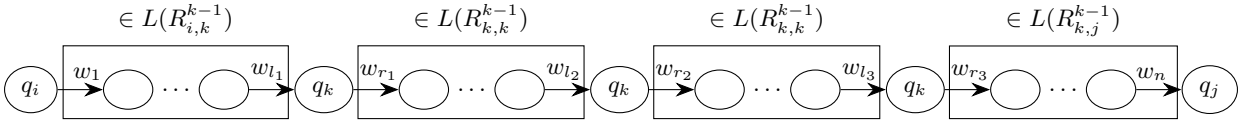
Diese regulären Ausdrücke lassen sich folgendermaßen rekursiv konstruieren:

$$k = 0 \text{ und } i \neq j\text{: } R^0_{i,j} = \bigcup_{\substack{a \in \Sigma: \\ \delta(q_i, a) = q_j}} a$$

$$k = 0 \text{ und } i = j\text{: } R^0_{i,i} = \left(\bigcup_{\substack{a \in \Sigma: \\ \delta(q_i, a) = q_i}} a \right) \cup \varepsilon$$

$$k > 0 \qquad \text{: } R^k_{i,j} = R^{k-1}_{i,j} \cup (R^{k-1}_{i,k} \cdot (R^{k-1}_{k,k})^* \cdot R^{k-1}_{k,j})$$

Beispielwort $w \in L(R^{k-1}_{i,k} \cdot (R^{k-1}_{k,k})^2 \cdot R^{k-1}_{k,j}) \subseteq L(R^k_{i,j})$ für das der Automat zwischen q_i und q_j genau dreimal den Zustand q_k durchläuft. Sonst werden zwischen q_i und q_j nur Zustände q_m mit $m < k$ durchlaufen:



Da alle $R^0_{i,j}$ reguläre Ausdrücke sind, sind auch alle $R^k_{i,j}$ für beliebiges k reguläre Ausdrücke.
Wir setzen $R = \bigcup_{q_j \in F} R^n_{1,j}$.
Für $q_j \in F$ gilt $L(R^n_{1,j}) = \{x \in \Sigma^* \mid \delta(q_1, x) = q_j\}$. Somit terminert der DFA A für jede Eingabe $x \in L(R^n_{1,j})$ in dem akzeptierenden Endzustand q_j. Entsprechend gilt für jede Eingabe $x \in L(R)$, dass der DFA A in einem akzeptierenden Endzustand hält. Per Konstruktion der regulären Ausdrücke $R^k_{i,j}$ enthält $L(R)$ ausschließlich Wörter der Sprache L. Folglich existiert ein regulärer Ausdruck R mit $L = L(R)$. □

Variante 2 (Gauss-Elimination):
Verwende R_q als Bezeichnung für den regulären Ausdruck, der die Sprache repräsentiert, die akzeptiert wird, wenn q der Startzustand wäre.
Es gilt

$$R_q = \begin{cases} \bigcup_{z \in \Sigma} z \cdot R_{\delta(q,z)} & \text{falls } q \notin F \\ \left(\bigcup_{z \in \Sigma} z \cdot R_{\delta(q,z)} \right) \cup \varepsilon & \text{falls } q \in F \end{cases} .$$

Jedes R_q lässt sich also schreiben als

$$R_q = \left(\bigcup_{q' \in Q} \alpha_{q'} \cdot R_{q'} \right) \cup \alpha \qquad \text{wobei } \alpha_{q'}, \alpha \text{ reguläre Ausdrücke für alle } q' \in Q \text{ sind.}$$

Anmerkung: Ein $R_{q'}$ auf einer rechten Seite kann als Platzhalter betrachtet werden.

Außerdem gilt folgende Selbstersetzungsregel

$$
\begin{aligned}
R_q =& \alpha_q R_q \cup \left(\bigcup_{q' \in Q \setminus \{q\}} \alpha_{q'} \cdot R_{q'} \right) \cup \alpha \\
=& (\alpha_q)^* \cdot \left(\left(\bigcup_{q' \in Q \setminus \{q\}} \alpha_{q'} \cdot R_{q'} \right) \cup \alpha \right) && \text{nachdem } \alpha_q \text{ benutzt wurde, kann erneut der gleiche reguläre Ausdruck } (R_q) \text{ verwendet werden, um ein Wort aus der durch } R_q \text{ repräsentierten Sprache zu erzeugen} \\
=& \left(\bigcup_{q' \in Q \setminus \{q\}} ((\alpha_q)^* \cdot \alpha_{q'}) \cdot R_{q'} \right) \cup ((\alpha_q)^* \cdot \alpha) && \text{wobei } \alpha_{q'}, \alpha \text{ reguläre Ausdrücke für alle } q' \in Q \text{ sind.}
\end{aligned}
$$

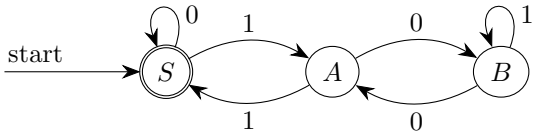
„Gauss-Elimination:“
Definiere nun eine beliebige Reihenfolge/Ordnung der Zustände ($q < q'$ wenn q vor q').
Iteriere nun über die Zustände in umgekehrter Reihenfolge. Sei q der aktuelle Zustand über den iteriert wird.
In der Iteration führe erst die Selbstersetzungsregel auf R_q aus und ersetze danach alle Vorkommen von R_q auf einer rechten Seite von $R_{q'}$ durch den regulären Ausdruck von R_q, der ohne R_q selbst auskommt, aber gegebenenfalls weitere $R_{\tilde{q}}$ mit $\tilde{q} < q$ auf der rechten Seite nutzt.
Nach jeder Iteration über einen Zustand q können für jedes $\tilde{q}$ die $R_{\tilde{q}}$ damit folgendermaßen dargestellt werden

$$
R_{\tilde{q}} = \left(\bigcup_{\substack{q' \in Q \\ \mathbf{q' < q}}} \alpha_{q'} \cdot R_{q'} \right) \cup \alpha \qquad \text{wobei } \alpha_{q'}, \alpha \text{ reguläre Ausdrücke für alle } q' \in Q \text{ sind.}
$$

Nachdem über alle Zustände iteriert wurde bleibt je R_q nur ein gewöhnlicher regulärer Ausdruck übrig, der ohne weitere $R_{q'}$ auf der rechten Seite auskommt.
$\Rightarrow L(R_{q_0}) = L(A)$ und R_{q_0} ist regulärer Ausdruck. □

Beispiel: Gauss-Elimination

$$L_1 = \{w \mid w \in \{0,1\}^*, (w)_2 \bmod 3 = 0\}$$



Sei $L_A = \{u \mid u \in \{0,1\}^*, u \text{ wird akzeptiert falls } A \text{ Startzustand wäre}\}$, analog L_B, L_S.
Gesucht sind reguläre Ausdrücke R_A, R_B und R_S für L_A, L_B und L_S. Vereinfachend/Abkürzend verwenden wir S, A, B stellvertretend für R_S, R_A, R_B.

„Gauss-Elimination“:

- initiales Gleichungssystem:
  $S = 0S \cup 1A \cup \varepsilon$, denn im Zustand S gibt es den Übergang mit gelesenem Zeichen 0 in den Zustand S zurück, mit gelesenem Zeichen 1 in den Zustand A und zusätzlich ist ε enthalten, da S ein Endzustand ist.
  $A = 0B \cup 1S$, denn im Zustand A gibt es den Übergang mit gelesenem Zeichen 0 in den Zustand B und mit gelesenem Zeichen 1 in den Zustand S.
  $B = 1B \cup 0A$, denn im Zustand B gibt es den Übergang mit gelesenem Zeichen 0 in den Zustand A und mit gelesenem Zeichen 1 in den Zustand B zurück.
  Hinweis: Die Aufstellung und das Lösen des Gleichungssystems ist analog auch bei nichtdeterministischen Automaten möglich.

- Elimination von A auf der rechten Seite:
  Zunächst muss A ohne A auf der rechten Seite dargestellt werden. Das ist jedoch bereits der Fall und es muss demnach nichts angepasst werden. Anschließend wird A durch die entsprechende rechte Seite $0B \cup 1S$ in den Ausdrücken für S und B ersetzt. Damit ergeben sich für S und B folgende Ausdrücke:
  $S = 0S \cup 1(0B \cup 1S) \cup \varepsilon = 0S \cup 10B \cup 11S \cup \varepsilon = (0 \cup 11)S \cup 10B \cup \varepsilon$
  $B = 1B \cup 0(0B \cup 1S) = 1B \cup 00B \cup 01S = (1 \cup 00)B \cup 01S$

- Elimination von B auf der rechten Seite:
  Zunächst muss B ohne B auf der rechten Seite dargestellt werden. Das geschieht durch die Selbstersetzungsregel. Dabei ist $\alpha_B = 1 \cup 00$ und $\left(\bigcup_{q' \in Q \setminus \{B\}} \alpha_{q'} q'\right) = 01S$ und $\alpha = \emptyset$. Das Ergebnis der Selbstersetzungsregel lautet dann
  $B = (1 \cup 00)^*(01S \cup \emptyset) = (1 \cup 00)^*01S$.
  Anschließend wird B durch die entsprechende rechte Seite $(1 \cup 00)^*01S$ in den Ausdrücken für S und A ersetzt. Damit ergeben sich für S und A folgende Ausdrücke:
  $A = 0((1 \cup 00)^*01S) \cup 1S = (0(1 \cup 00)^*01 \cup 1)S$
  $S = (0 \cup 11)S \cup 10((1 \cup 00)^*01S) \cup \varepsilon = (0 \cup 11 \cup 10(1 \cup 00)^*01)S \cup \varepsilon$
  Hinweis: Wäre sowohl $\alpha = \emptyset$ als auch $\left(\bigcup_{q' \in Q \setminus \{B\}} \alpha_{q'} q'\right) = \emptyset$ so wäre das Ergebnis der Selbstanwendung $\emptyset$ unabhängig davon, wie α_B aussieht.

- Elimination von S auf der rechten Seite:
  Zunächst muss S ohne S auf der rechten Seite dargestellt werden. Das geschieht durch die Selbstersetzungsregel. Das Ergebnis der Selbstersetzungsregel lautet dann
  $S = (0 \cup 11 \cup 10(1 \cup 00)^*01)^*(\emptyset \cup \varepsilon) = (0 \cup 11 \cup 10(1 \cup 00)^*01)^*\varepsilon = (0 \cup 11 \cup 10(1 \cup 00)^*01)^*$
  Anschließend wird S durch die entsprechende rechte Seite $(0 \cup 11 \cup 10(1 \cup 00)^*01)^*$ in den Ausdrücken für A und B ersetzt. Damit ergeben sich für A und B folgende Ausdrücke:
  $A = (0(1 \cup 00)^*01 \cup 1)(0 \cup 11 \cup 10(1 \cup 00)^*01)^*$
  $B = (1 \cup 00)^*01(0 \cup 11 \cup 10(1 \cup 00)^*01)^*$

- Da S Startzustand ist, ist S ein regulärer Ausdruck mit $L_S = L(S) = L((0 \cup 11 \cup 10(1 \cup 00)^*01)^*) = L_1$.

Kurz mit alternativer Eliminationsreihenfolge.

	initiales Gleichungssystem	Elimination von B	Elimination von A
S	$= 0S \cup 1A \cup \varepsilon$	$= 0S \cup 1A \cup \varepsilon$	$= 0S \cup 1(01^*0)^*1S \cup \varepsilon$
A	$= 1S \cup 0B$	$= 1S \cup 01^*0A$	$= (01^*0)^*1S$
B	$= 1B \cup 0A$	$= 1^*0A$	$= 1^*0(01^*0)^*1S$

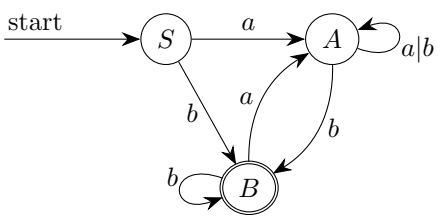
	Ausklammern von S	Elimination von S	
S	$= (0 \cup 1(01^*0)^*1)S \cup \varepsilon$	$= (0 \cup 1(01^*0)^*1)^*\varepsilon$	$= (0 \cup 1(01^*0)^*1)^*$
A	$= (01^*0)^*1S$	$= (01^*0)^*1(0 \cup 1(01^*0)^*1)^*$	
B	$= 1^*0(01^*0)^*1S$	$= 1^*0(01^*0)^*1(0 \cup 1(01^*0)^*1)^*$	

Damit gilt $L_S = L(S) = L((0 \cup 1(01^*0)^*1)^*) = L_1$.

Hinweis: Es ergeben sich dadurch andere reguläre Ausdrücke. Das kann jedoch sein, da die Darstellung mittels regulären Ausdrücken nicht eindeutig ist.

Beispiel:

$$L_2 = \{w \mid w = w_1 w_2 \dots w_{|w|} \in \{a, b\}^*, w_{|w|} = b\}$$



	initiales Gleichungssystem	Elimination von B	Ausklammern von A
S	$= aA \cup bB$	$= aA \cup bb^*(aA \cup \varepsilon)$	$= (a \cup bb^*a)A \cup bb^*$
A	$= (a \cup b)A \cup bB$	$= (a \cup b)A \cup bb^*(aA \cup \varepsilon)$	$= (a \cup b \cup bb^*a)A \cup bb^*$
B	$= aA \cup bB \cup \varepsilon$	$= b^*(aA \cup \varepsilon)$	$= b^*aA \cup b^*$

	Elimination von A
S	$= (a \cup bb^*a)(a \cup b \cup bb^*a)^*bb^* \cup bb^*$
A	$= (a \cup b \cup bb^*a)^*bb^*$
B	$= b^*a(a \cup b \cup bb^*a)^*bb^* \cup b^*$

Damit gilt $L_S = L(S) = L((a \cup bb^*a)(a \cup b \cup bb^*a)^*bb^* \cup bb^*) = L_2$.

Satz 3.23 *Die regulären Sprachen sind abgeschlossen unter den Operationen*

- *Vereinigung*
- *Konkatenation*
- *Durchschnitt*
- *Sternabschluss (Kleene-Abschluss)*
- *Komplementbildung*
- *Spiegelung*

BEWEIS:

- Vereinigung nach Definition regulärer Ausdrücke ✓
- Konkatenation nach Definition regulärer Ausdrücke ✓
- Sternabschluss nach Definition regulärer Ausdrücke ✓
- Komplementbildung:
  Sei L regulär und $A = (Q, \Sigma, \delta, q_0, F)$ ein DFA mit $L = L(A)$ und $\delta(q, a)$ ist definiert $\forall q \in Q, a \in \Sigma$, dann akzeptiert der DFA $A' = (Q, \Sigma, \delta, q_0, F')$ mit $F' = Q \setminus F$ das Komplement $\overline{L}$ ✓
- Durchschnitt ist mit vorherigen Operationen umsetzbar ($L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$) ✓
- Spiegelung durch inverse Reihenfolge bei jeder Konkatenation in regulären Ausdrücken ✓ □

3.7 Kontextfreie Grammatiken und Sprachen

Eine Sprache heißt ***kontextfrei***, wenn es eine kontextfreie Grammatik G mit $L(G) = L$ gibt (siehe Definition 3.1).

Beispiel:
$G_1 = (V, \Sigma, \mathcal{P}, \mathcal{S})$ mit
$V = \{\mathcal{S}\}$, $\Sigma = \{a, b\}$,
$\mathcal{P} : \mathcal{S} \to a\mathcal{S}b \mid \varepsilon$
$L(G_1) = \{a^n b^n \mid n \geq 0\}$
Eigentlich noch zu zeigen:

(i) Jedes derartige Wort kann erzeugt werden.

(ii) Kein anderes Wort ist erzeugbar.

Beispiel:
$G_2 = (V, \Sigma, \mathcal{P}, \langle Ausdruck\rangle)$ mit
$V = \{\langle Ausdruck\rangle, \langle Term\rangle, \langle Faktor\rangle\}$, $\Sigma = \{+, *, (,), a\}$,
$\mathcal{P}$:

$$\begin{aligned}
\langle Ausdruck\rangle &\to \langle Term\rangle \mid \langle Ausdruck\rangle + \langle Term\rangle \\
\langle Term\rangle &\to \langle Faktor\rangle \mid \langle Term\rangle * \langle Faktor\rangle \\
\langle Faktor\rangle &\to a \mid (\langle Ausdruck\rangle)
\end{aligned}$$

Beispielableitung:

$$\begin{aligned}
\langle Ausdruck\rangle &\to \langle Ausdruck\rangle + \langle Term\rangle \\
&\to \langle Term\rangle + \langle Term\rangle \\
&\to \langle Faktor\rangle + \langle Term\rangle \\
&\to (\langle Ausdruck\rangle) + \langle Term\rangle \\
&\to (\langle Ausdruck\rangle + \langle Term\rangle) + \langle Term\rangle \\
&\to (\langle Term\rangle + \langle Term\rangle) + \langle Term\rangle \\
&\to (\langle Faktor\rangle + \langle Term\rangle) + \langle Term\rangle \\
&\to (a + \langle Term\rangle) + \langle Term\rangle \\
&\to (a + \langle Faktor\rangle) + \langle Term\rangle \\
&\to (a + a) + \langle Term\rangle \\
&\to (a + a) + \langle Faktor\rangle \\
&\to (a + a) + a
\end{aligned}$$

$\Rightarrow (a + a) + a \in L(G_2)$
Dies war eine Linksableitung, da immer die linkeste Variable abgeleitet wurde.

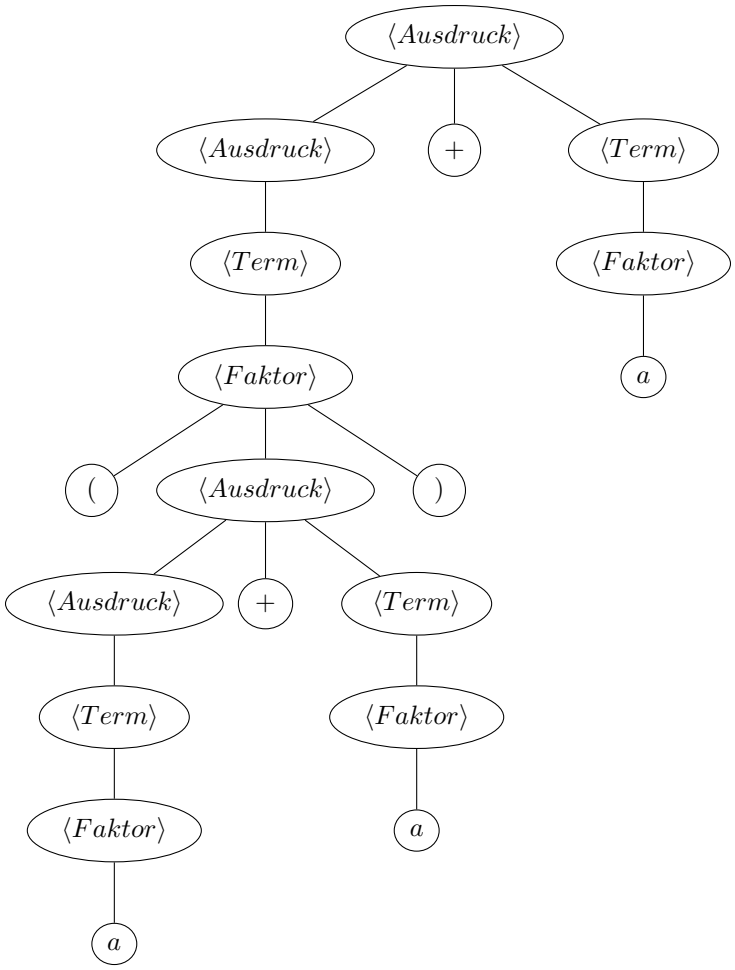


Abbildung 23: Syntaxbaum (Herleitungsbaum) zur Ableitung von $(a + a) + a$ mittels G_2.

Die Herstellung des Syntaxbaums ist die Syntaxanalyse. Die Blätter des Syntaxbaums (von links nach rechts) beschreiben das hergeleitete Wort. Der Syntaxbaum muss im Allgemeinen nicht eindeutig sein.

Definition 3.24 *Eine kontextfreie Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ ist in* **Chomsky-Normalform (ChNF)**, *wenn jede Regel in G von folgender Form ist:*

- $A \to BC$ *mit* $A \in V$, $B, C \in V \setminus \{\mathcal{S}\}$ *oder*
- $A \to a$ *mit* $A \in V$, $a \in \Sigma$ *oder*
- $\mathcal{S} \to \varepsilon$.

Satz 3.25 *Zu jeder kontextfreien Grammatik G kann eine kontextfreie Grammatik G' in ChNF konstruiert werden mit $L(G) = L(G')$.*

Der Beweis dieses Satzes ist auf die folgenden Lemmata aufgeteilt.

Lemma 3.26 *Zu jeder kontextfreien Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ gibt es eine kontextfreie Grammatik $G_{\varepsilon\text{-}frei} = (V', \Sigma, \mathcal{P}', \mathcal{S}')$ mit $L(G) = L(G_{\varepsilon\text{-}frei})$ und $G_{\varepsilon\text{-}frei}$ enthält keine ε-Regeln bis auf ggf. $\mathcal{S}' \to \varepsilon$ und das Startsymbol taucht auf keiner rechten Seite einer Produktion auf.*

BEWEIS: Wir benutzen folgende Hilfskonstruktion:

$$E_0 = \{A \mid (A \to \varepsilon) \in \mathcal{P}\}$$
$$E_i = \{A \mid (A \to B_1 \dots B_k) \in \mathcal{P}, \text{ mit } B_1, \dots, B_k \in E_{i-1}\} \cup E_{i-1} \ .$$

Da es nur endliche viele Variablen gibt, gibt es ein i_0 mit $E_{i_0} = E_{i_0+1}$.

- Lösche nun alle ε-Regeln aus $\mathcal{P}$.
- Für alle Regeln $(A \to w) \in \mathcal{P}$: Wenn w k Variablen aus E_{i_0} enthält, so füge alle $2^k - 1$ möglichen Regeln, die man durch Weglassen von Variablen aus E_{i_0} in w erhalten kann, hinzu, außer $(A \to \varepsilon)$.
- Führe neues Startsymbol $\mathcal{S}'$ ein und füge die Regel $\mathcal{S}' \to \mathcal{S}$ hinzu.
- Falls $\mathcal{S} \in E_{i_0}$ so füge die Regel $\mathcal{S}' \to \varepsilon$ hinzu. □

Lemma 3.27 *Zu jeder kontextfreien Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ gibt es eine kontextfreie Grammatik $G' = (V', \Sigma, \mathcal{P}', \mathcal{S})$ ohne Kettenregeln (das heißt Regeln der Form $A \to B$, wobei A, B jeweils* eine *Variable ist) mit $L(G) = L(G')$.*

BEWEIS:

- Konstruiere einen gerichteten Graphen $G_{Kette} = (V, E_{Kette})$, in dem die Variablen die Knoten sind und die Kanten die Kettenregeln.
- Ersetze in den starken Zusammenhangskomponenten (Äquivalenzklassen) die Variablen darin durch einen Repräsentanten davon. Falls $\mathcal{S}$ in einer der starken Zusammenhangskomponente enthalten ist, so muss $\mathcal{S}$ als Repräsentant benutzt werden.
- Ersetze in allen Produktionen die ersetzten Variablen durch den entsprechenden Repräsentanten. Führe die Ersetzung ebenfalls in G_{Kette} durch.
- Ersetze nun im dem übrig gebliebenen kreisfreien Graphen (directed acyclic graph - DAG) in umgekehrter topologischer Reihenfolge („von unten nach oben“) die Kettenregeln $A \to B$ durch die Regeln $A \to$ „alle rechten Seiten von B“ und entferne jeweils im Anschluss die verarbeitete Kettenregel. □

kontextfreie Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$
$V = \{\mathcal{S}, A, B, C, D, E, F\}$
Sei $\mathcal{P}'$ die Menge der Kettenregeln aus $\mathcal{P}$

$$\mathcal{P}' = \{\mathcal{S} \to A,\ A \to B,\ B \to \mathcal{S},\ A \to E,\ E \to F,\ F \to A,\ A \to C,\ C \to D\}$$

Äquivalenzklassen (rot markiert) definiert durch gegenseitige Erreichbarkeit

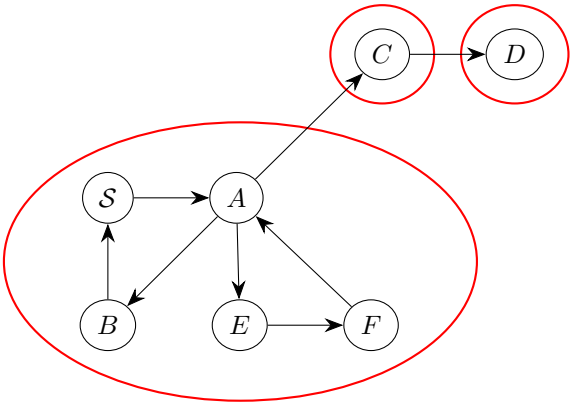


Abbildung 24: Beispiel: Durch Kettenregeln definierte Äquivalenzklassen

BEWEIS: (Satz 3.25)
Konstruktion von G' in Chomsky-Normalform aus der kontextfreien Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ mit $L(G') = L(G)$:

- Konstruiere gemäß Lemma 3.26 eine kontextfreie Grammatik $G_{\varepsilon\text{-frei}} = (V', \Sigma, \mathcal{P}', \mathcal{S}')$ mit $L(G) = L(G_{\varepsilon\text{-frei}})$ und $G_{\varepsilon\text{-frei}}$ enthält keine ε-Regeln bis auf ggf. $\mathcal{S}' \to \varepsilon$ und das Startsymbol taucht auf keiner rechten Seite einer Produktion auf.
- Konstruiere gemäß Lemma 3.27 aus $G_{\varepsilon\text{-frei}}$ eine kontextfreie Grammatik $G' = (V'', \Sigma, \mathcal{P}'', \mathcal{S}')$, welche zusätzlich keine Kettenregeln enthält.
- Für alle $a \in \Sigma$ erzeuge zusätzliche Variablen A_a. Füge die Regeln $A_a \to a$ hinzu. Ersetze alle Vorkommen von a durch die Variable A_a außer wenn dadurch Kettenregeln entstehen würden.
- Teile Regeln mit mehr als zwei Variablen auf der rechten Seite auf:
  Sei die i-te Regel $A \to B_1 \dots B_k$ mit $k \geq 3$.
  Führe neue Variablen $C_1^{(i)}, \dots, C_{k-2}^{(i)}$ ein und ersetze die i-te Regel durch die folgenden Regeln:
  $A \to B_1 C_1^{(i)} \quad C_1^{(i)} \to B_2 C_2^{(i)} \quad C_2^{(i)} \to B_3 C_3^{(i)} \quad \dots \quad C_{k-3}^{(i)} \to B_{k-2} C_{k-2}^{(i)} \quad C_{k-2}^{(i)} \to B_{k-1} B_k$

Wird G entsprechend dieser Konstruktion modifiziert, so ergibt sich eine Grammatik G' in Chomsky-Normalform. □

Die Reihenfolge der Modifikationen ist relevant, denn nicht jede Reihenfolge liefert notwendigerweise eine Chomsky-Normalform. Beispielsweise kann es passieren, wenn man das Entfernen von Kettenregeln vor das Entfernen der ε-Regeln zieht, dass bei der Entfernung der ε-Regeln erneut Kettenregeln generiert werden.

Eine Variable A heißt ***nutzlos***, wenn es kein Wort $w \in \Sigma^*$ gibt mit $A \overset{*}{\to} w$. Sonst heißt diese Variable ***nützlich***.

Satz 3.28 *Sei G eine kontextfreie Grammatik in Chomsky-Normalform.*
Man kann die nutzlosen Variablen bestimmen und alle Produktionen, in denen sie vorkommen, löschen, ohne die Sprache zu verändern.

BEWEIS: siehe Übung. □

Ab jetzt: Kontextfreie Grammatiken haben Chomsky-Normalform und alle Variablen sind nützlich.

Der CYK-Algorithmus (Cocke–Younger–Kasami Algorithmus)

Gegeben ist

- eine kontextfreie Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ in Chomsky-Normalform, wobei alle Variablen nützlich sind, und
- ein Wort $w = a_1 \dots a_n \in \Sigma^*$.

Frage: Ist $w \in L(G)$?
Benutze dazu folgende Hilfskonstruktion:

$$V(i,j) = \{A \mid A \in V,\ A \overset{*}{\to} a_i \dots a_j\} \text{ mit } i \leq j \ .$$

Die Berechnung kann mittels folgender rekursiver Beziehung umgesetzt werden (Dynamische Programmierung):

$$V(i,i) = \{A \mid (A \to a_i) \in \mathcal{P}\} \ \forall i \in \{1, \dots, n\}$$
$$V(i,j) = \{A \mid (A \to BC) \in \mathcal{P},\ B \in V(i,k),\ C \in V(k+1,j),\ k \in \{i, \dots, j-1\}\}$$

Die $V(i,j)$ können damit in der Reihenfolge aufsteigender Länge $l = (j - i + 1)$ konstruiert werden.

Tabelle 7: Berechnungsreihenfolge der $V(i,j)$ beim CYK-Algorithmus.

l	w_1	w_2	w_3	w_4	$\dots$	w_{n-1}	w_n
1	$V(1,1)$	$V(2,2)$	$V(3,3)$	$V(4,4)$	$\dots$	$V(n-1,n-1)$	$V(n,n)$
2	$V(1,2)$	$V(2,3)$	$V(3,4)$	$V(4,5)$	$\dots$	$V(n-1,n)$	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		
$n-3$	$V(1,n-3)$	$V(2,n-2)$	$V(3,n-1)$	$V(4,n)$			
$n-2$	$V(1,n-2)$	$V(2,n-1)$	$V(3,n)$				
$n-1$	$V(1,n-1)$	$V(2,n)$					
n	$V(1,n)$						

Es gilt $w \in L(G) \Leftrightarrow \mathcal{S} \in V(1,n)$.
Mittels dieser Tabelle kann ein Syntaxbaum hergestellt werden.

Satz 3.29 *Der CYK-Algorithmus entscheidet in Zeit $\mathcal{O}(|\mathcal{P}| \cdot |w|^3)$, ob $w \in L(G)$ ist.*

BEWEIS: Der Algorithmus ist *correct by construction*.
Je $V(i,j)$ läuft die Laufvariable der rekursiven Definition von i bis $j-1$ nimmt also $\mathcal{O}(|w|)$ viele Werte an. Insgesamt gibt es $\mathcal{O}(|w|^2)$ viele verschiedene $V(i,j)$, die zu berechnen sind. Jeweils werden die $|\mathcal{P}|$ Produktionen in $\mathcal{P}$ durchsucht.
$\Rightarrow$ Laufzeit $\mathcal{O}(|\mathcal{P}| \cdot |w|^3)$. □

Damit ist das Wortproblem für kontextfreie Sprachen in P (also in Polynomzeit lösbar).

Korollar 3.30

$$L = \{\langle G \rangle w \mid G \text{ ist kontextfreie Grammatik in Chomsky-Normalform, } w \in L(G)\} \in P$$

Anmerkung:
Weitere Einschränkung des Grammatik-Typs für Programmiersprachen (Look-Ahead-Grammatiken)
$\Rightarrow$ Ziel: Das Wortproblem soll in Linearzeit entschieden werden.

Das Pumping-Lemma für kontextfreie Sprachen

Definition 3.31 *Sei L eine Sprache über Σ.*
L hat die **kontextfreie Pump-Eigenschaft**, *falls gilt:*

$$\exists n_L \in \mathbb{N}\, \forall z \in L,\ |z| \geq n_L\ \exists u,v,w,x,y \in \Sigma^* : uvwxy = z \text{ und}$$

(i) $|vwx| \leq n_L$

(ii) $vx \neq \varepsilon$

(iii) $\forall i \geq 0 : uv^iwx^iy \in L$

Die **Negation der kontextfreien Pump-Eigenschaft** kann folgendermaßen formuliert werden:
Sei L eine Sprache.

$$\forall n_L \in \mathbb{N}\, \exists z \in L,\ |z| \geq n_L\, \forall u,v,w,x,y \in \Sigma^* : uvwxy = z :$$
$$(\underbrace{|vwx| \leq n_L}_{(i)} \wedge \underbrace{vx \neq \varepsilon}_{(ii)}) \Rightarrow \underbrace{\exists i \geq 0 : uv^iwx^iy \notin L}_{\neg(iii)}$$

Beispiel 3.32

(a) $L_1 = \{a, ba\}$.
Setze $n_L = 4$.
Es gilt: $\{z \in L_1 \mid n_L \leq |z|\} = \emptyset$
Entsprechend sind alle Anforderungen der kontextfreien Pump-Eigenschaft erfüllt.
$\Rightarrow L_1$ *hat die kontextfreie Pump-Eigenschaft.*

(b) $L_2 = \{a^n b^n \mid n \geq 1\}$.
Setze $n_L = 4$.
Sei $z \in L_2$ *mit* $|z| \geq n_L = 4$ *beliebig aber fest.*
Es gilt $z = a^j aabbb^j \in L_2$ *mit* $j \geq 0$.
Setze $u = a^j a$, $v = a$, $w = \varepsilon$, $x = b$, $y = bb^j$.

(i) $|vwx| = 2 \leq n_L = 4$ ✓

(ii) $vx \neq \varepsilon$ ✓

(iii) $uv^i wx^i y = a^{i+j+1} b^{i+j+1} = a^k b^k \in L_2$ *für alle* $i \geq 0$, *da* $k \geq 1$ ✓

$\Rightarrow L_2$ *hat die kontextfreie Pump-Eigenschaft.*

(c) $L_3 = \{a^n b^n c^n \mid n \geq 0\}$
Beweis der Negation der kontextfreien Pump-Eigenschaft:
Sei n_L *beliebig aber fest.*
Setze $z = a^{n_L} b^{n_L} c^{n_L} \in L_3$, *also gilt* $|z| = 3 \cdot n_L \geq n_L$.
Sei u, v, w, x, y *eine beliebige Zerlegung von* z *mit* $z = uvwxy$ *und (i) und (ii).*

- *Wegen (i) gilt:* v *und* x *bestehen aus maximal zwei verschiedenen Sorten von Zeichen.*
- *Wegen (ii) gilt:* v *und* x *bestehen aus mindestens einer Sorte von Zeichen.*

Setze nun in (iii) das $i = 0$.
Damit ist die Kopplung der a's an die b's und an die c's verletzt und $uv^0 wx^0 y = uwy \notin L_3$.
$\Rightarrow L_3$ *hat die kontextfreie Pump-Eigenschaft nicht.*

(d) $L_4 = \{a^{n^2} \mid n \geq 0\}$
Beweis der Negation der kontextfreien Pump-Eigenschaft:
Sei n_L *beliebig aber fest.*
Setze $z = a^{n_L^2}$, $|z| = n_L^2 \geq n_L$.
Sei u, v, w, x, y *eine beliebige Zerlegung von* z *mit* $z = uvwxy$ *und (i) und (ii).*
Wähle $i = 2$, *dann gilt* $uv^i wx^i y = uv^2 wx^2 y = a^{n_L^2 + |vx|}$.
Damit $a^{n_L^2+|vx|} \in L_4$ *muss* $n_L^2 + |vx|$ *eine Quadratzahl sein.*
Es gilt aber $n_L^2 < n_L^2 + \overbrace{|vx|}^{\geq 1} \leq n_L^2 + n_L < n_L^2 + n_L + n_L + 1 = (n_L + 1)^2$
$\Rightarrow n_L^2 + |vx|$ *ist keine Quadratzahl und damit* $uv^2 wx^2 y \notin L_4$.
$\Rightarrow L_4$ *hat die kontextfreie Pump-Eigenschaft nicht.*

(e) $L_5 = \{c^i w \mid i \geq 1, w \in H_\varepsilon\} \cup \{0,1\}^*$
Beweis, dass L_5 die kontextfreie Pumpeigenschaft hat:
Setze $n_L = 5$.
Sei $z \in L_5$ mit $|z| \geq n_L$ beliebig aber fest.

- *Fall $z \in \{0,1\}^*$:*
  Setze $u = \varepsilon, v = z_1, w = \varepsilon, x = \varepsilon, y = z_2 z_3 \dots z_{|z|}$.
  Dann gilt:
  (a) $vx \neq \varepsilon$
  (b) $|vwx| \leq n_L$
  (c) $\forall i \geq 0$: *$uv^i wx^i y \in \{0,1\}^*$. Also folgt $uv^i wx^i y \in L_5$.*
- *Fall $z \in \{c^k y \mid k \geq 1, y \in H_\varepsilon\}$:*
  Setze $u = \varepsilon, v = z_1 = c, w = \varepsilon, x = \varepsilon, y = z_2 z_3 \dots z_{|z|}$.
  (a) $vx \neq \varepsilon$
  (b) $|vwx| \leq n_L$
  (c) *Sei i beliebig aber fest:*
  - *Fall $i = 0$:*
    - *Fall $k = 1$:*
      Es gilt $uv^i wx^i y = uv^0 wx^0 y = uwy \in H_\varepsilon$, da das einzige c entfernt wurde.
      Daraus folgt $uwy \in \{0,1\}^*$, also $uwy \in L_5$.
    - *Fall $k > 1$:*
      Es gilt $uv^i wx^i y = uv^0 wx^0 y = uwy \in L_5$, da nur ein c entfernt wird und mindestens ein weiteres c übrig ist.
  - *Fall $i > 0$: Es gilt $uv^i wx^i y \in L_5$, da sich nur gleich viele oder mehr c am Anfang des Worts befinden.*

$\Rightarrow$ *L_b hat die reguläre Pump-Eigenschaft.*

Satz 3.33 (Pumping-Lemma für kontextfreie Sprachen) *Wenn eine Sprache kontextfrei ist, dann hat sie die kontextfreie Pump-Eigenschaft.*

BEWEIS: Sei L eine kontextfreie Sprache.
$\Rightarrow$ Es gibt eine Grammatik $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ in Chomsky-Normalform mit $L(G) = L$.
Setze $n_L = 2^{|V|+1}$.
Sei $z \in L$ mit $|z| \geq n_L$ beliebig aber fest. Da $|z| \geq n_L = 2^{|V|+1}$, muss es im Syntaxbaum einen Pfad von $\mathcal{S}$ bis zum letzten Vorkommen einer Variablen geben, mit einer Länge von mindestens $|V| + 1$. Betrachte den längsten Pfad von der Wurzel $\mathcal{S}$ zu einem Blatt und wähle die Variable A von unten aus betrachtet, die als erste doppelt vorkommt. Es gibt so eine Variable, da der Pfad mindest Länge $|V| + 1$ hat und es nur $|V|$ viele verschiedene Variablen gibt.
Wähle nun die Aufteilung von z in folgende Abschnitte (vergleiche mit Abbildung 25):

$u =$ „Teilwort welches sich im Syntaxbaum links vom oberen A befindet“
$v =$ „Teilwort welches sich im Syntaxbaum unterhalb des oberen A und links vom unteren A befindet“
$w =$ „Teilwort welches sich im Syntaxbaum unterhalb des unteren A befindet“
$x =$ „Teilwort welches sich im Syntaxbaum unterhalb des oberen A und rechts vom unteren A befindet“
$y =$ „Teilwort welches sich im Syntaxbaum rechts vom oberen A befindet“

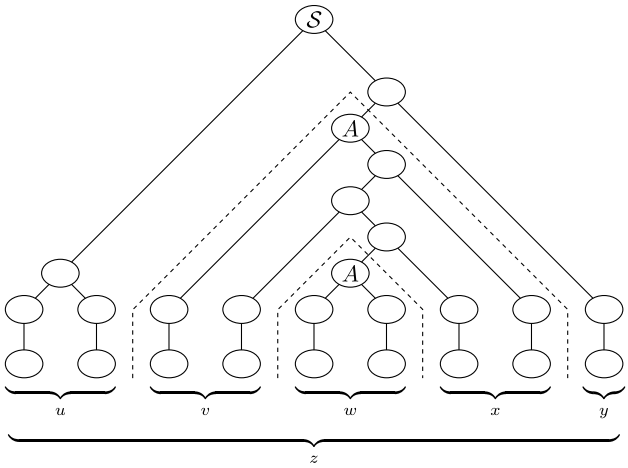


Abbildung 25: Aufteilung anhand eines beispielhaften Syntaxbaums eines Wortes z.

Zu (i):
Der längste Pfad vom oberen der beiden A bis zu einem Blatt ist $|V| + 2$ lang. Im letzten Level wird nur noch je *eine* Variable in *ein* Terminal umgewandelt. Entsprechend kann sich der Baum ab dem oberen A maximal $|V| + 1$ mal verzweigen und damit maximal $2^{|V|+1}$ Blätter erzeugen.
$\Rightarrow |vwx| \leq 2^{|V|+1} = n_L$ ✓
Zu (ii):
$\mathcal{P}$ enthält keine Kettenregeln. Damit muss der Syntaxbaum von z einen Pfad vom oberen A zu einem Blatt enthalten, der nicht durch das untere A verläuft.
$\Rightarrow vx \neq \varepsilon$ ✓
Zu (iii):
Bei $i = 1$ muss der Baum nicht verändert werden.
$\Rightarrow uv^1wx^1y \in L(G) = L$ ✓

Bei $i = 0$ kann aus dem Syntaxbaum der Bereich ab dem oberen A durch den Bereich ab dem unteren A ersetzt werden. Dies ist weiterhin ein gültiger Syntaxbaum und er stellt die Ableitung des Wortes $uwy = uv^0wx^0y$ dar.
$\Rightarrow uv^0wx^0y \in L(G) = L$ ✓

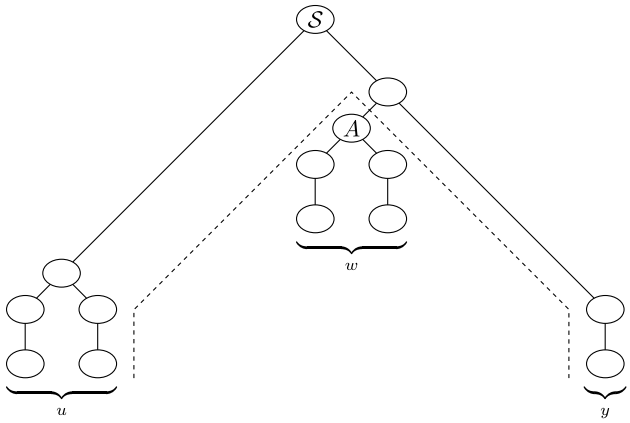


Abbildung 26: Beispielhafter Syntaxbaum für das Wort $uv^0wx^0y = uwy$.

Bei $i > 1$ muss $i-1$ mal der Teil ab dem oberen A beim unteren A wiederholt werden und erst beim *letzten unteren* A wird der Teil unter dem ursprünglichen unteren A angehängt. Das dadurch abgeleitete Wort ist das gewünschte uv^iwx^iy.
$\Rightarrow uv^iwx^iy \in L(G) = L$ ✓ □

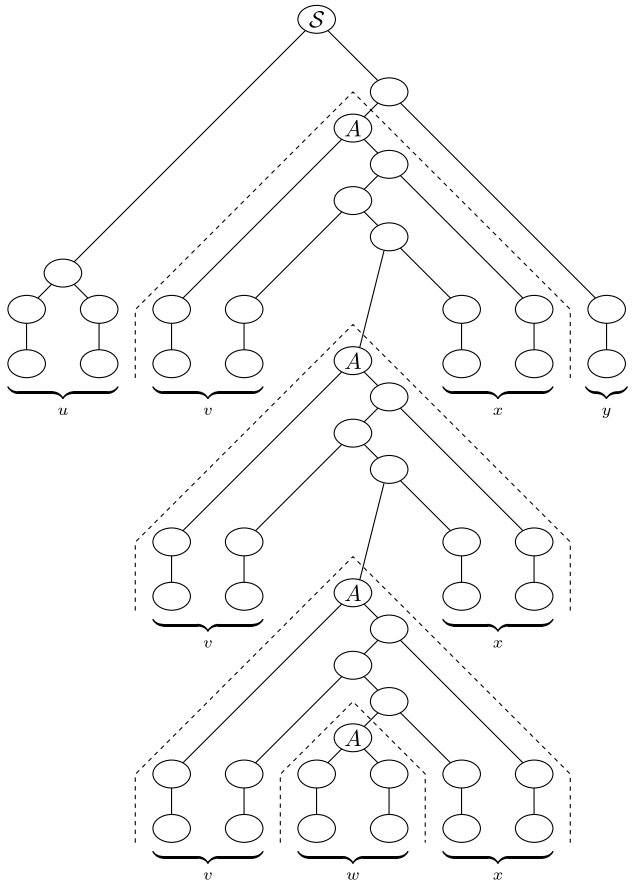


Abbildung 27: Beispielhafter Syntaxbaum für das Wort uv^3wx^3y.

Die Chomsky-Hierarchie

Sei $\mathcal{L}_i$, $i \in \{0, 1, 2, 3\}$ die Klasse der Sprachen, die durch Grammatiken vom Typ CHOMSKY-i erzeugt werden können:

- $\mathcal{L}_3$: Reguläre Sprachen
- $\mathcal{L}_2$: Kontextfreie Sprachen
- $\mathcal{L}_1$: Kontextsensitive Sprachen
- $\mathcal{L}_0$: Rekursiv aufzählbare Sprachen

$\mathcal{E}$: Entscheidbare Sprachen

Satz 3.34

$$\mathcal{L}_3 \subsetneq \mathcal{L}_2 \subsetneq \mathcal{L}_1 \subsetneq \mathcal{E} \subsetneq \mathcal{L}_0 \subsetneq \text{„Alle Sprachen“}$$

BEWEIS:

$$L(a^*b^*) \in \mathcal{L}_3 \not\ni \{a^nb^n \mid n \geq 0\} \in \mathcal{L}_2 \not\ni \{a^nb^nc^n \mid n \geq 0\} \in \mathcal{L}_1 \not\ni H \in \mathcal{L}_0 \not\ni \overline{H}$$

Alle kontextsensitiven Sprachen sind entscheidbar. Umgekehrt gilt dies nicht. Allgemein sind alle Probleme nicht kontextsensitiv, die nicht mit linearem Platz auskommen. Beispielsweise gibt es Instanzen des Problems Äquivalenz regulärer Ausdrücke (EXPSPACE-vollständig) die mindestens exponentiellen Platz benötigen. □

Kellerautomaten

Definition 3.35 *Ein* **nichtdeterministischer Kellerautomat** ***(NPDA, nondeterministic pushdown acceptor / nondeterministic pushdown automaton)*** *M bzw. ein* **deterministischer Kellerautomat** ***(DPDA, deterministic pushdown acceptor / deterministic pushdown automaton)*** *M ist beschrieben durch 7 Komponenten $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ mit*

- *Q: Endliche Menge von Zuständen*
- *Σ: Endliches Eingabealphabet*
- *Γ: Endliches Kelleralphabet*
- *δ: $Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times \Gamma^*)$*
- *q_0: Startzustand*
- *Z_0 mit $Z_0 \in \Gamma$: Kellergrundsymbol*
- *F: Akzeptierende Zustände*

Sobald ein Zustand $q \in F$ erreicht wird hält der Kellerautomat.
Zu Beginn steht der Lesekopf unter dem ersten Zeichen der Eingabe und der Keller ist leer.
Das Kellergrundsymbol liegt immer auf dem Kellerboden.
$x \in L(M) \Leftrightarrow$ Es gibt eine Rechnung, die einen Zustand $q \in F$ erreicht, und alle Zeichen von x wurden gelesen.
M ist deterministischer Kellerautomat, wenn es zu jeder Konfiguration genau einen oder keinen möglichen Zustandsübergang in δ gibt.

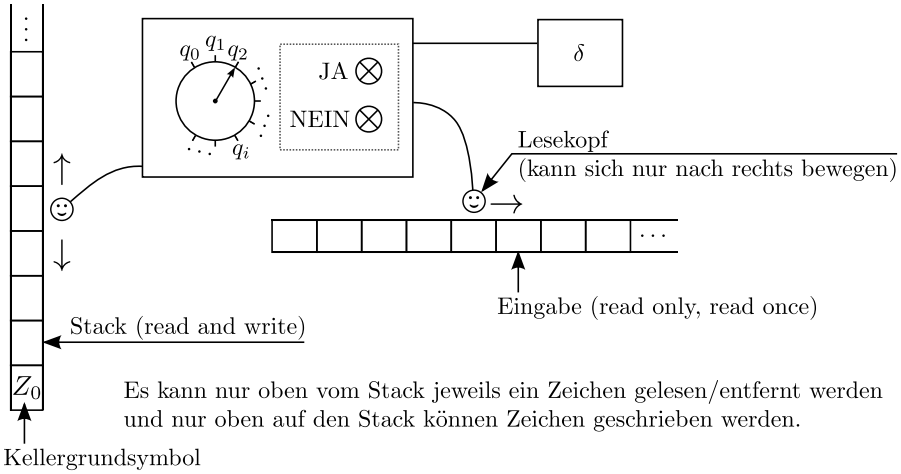


Abbildung 28: Aufbau eines Kellerautomaten.

Beispiel:
Die Sprache „Klammerungen“, welche alle wohlgeformten Klammerausdrücke mit runden und eckigen Klammern enthält ist kontextfrei.
Die Sprache kann mittels folgender kontextfreier Grammatik erzeugt werden:
$G = (V = \{\mathcal{S}\}, \Sigma = \{(, [,],)\}, \mathcal{P}, \mathcal{S})$ mit
$\mathcal{P}$:

$$\mathcal{S} \to (\mathcal{S}) \mid [\mathcal{S}] \mid \mathcal{S}\mathcal{S} \mid \varepsilon$$

Zum Beispiel das folgende Wort liegt in $L(G)$.

$$[\ (\ )\ [\ ]\ ] \in L(G)$$

Satz 3.36 *L ist kontextfrei $\Leftrightarrow$ Es gibt einen* nichtdeterministischen *Kellerautomaten mit $L(M) = L$.*

BEWEIS:
„$\Leftarrow$“:
Gegeben ist NPDA M.
Zu konstruieren ist kontextfreie Grammatik G mit $L(G) = L(M)$.
Extrem schwieriger Beweis, beispielsweise nachzulesen in [1] (HOPCROFT/ULLMAN), benutzt den Nichtdeterminismus auf äußerst komplizierte Art und Weise.

„$\Rightarrow$“:
Gegeben ist kontextfreie Sprache L und kontextfreie Grammatik G mit $L(G) = L$.
Ziel: Einen nichtdeterministischen Kellerautomaten M mit $L = L(M)$ konstruieren.
Idee: M vollzieht eine Linksableitung für das Eingabewort x nach. Auf dem Keller steht die aktuelle Satzform ohne die bereits verarbeiteten (mit Eingabe verglichenen) Terminalsymbole.

Konstruiere aus $G = (V, \Sigma, \mathcal{P}, \mathcal{S})$ einen nichtdeterministischen Kellerautomaten:
NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ mit $\Gamma = \{Z_0\} \dot\cup V \dot\cup \Sigma$, wobei die Mengen $\{Z_0\}$, V und Σ disjunkt sind, $Q = \{q_0, \overline{q}, q_{accept}\}$, $F = \{q_{accept}\}$ und
δ :

$$\delta(q_0, \varepsilon, Z_0) = \left\{ \left(\overline{q}, \begin{pmatrix} \mathcal{S} \\ Z_0 \end{pmatrix} \right) \right\}$$

$$\forall A \in V : \delta(\overline{q}, \varepsilon, A) = \left\{ \left(\overline{q}, \begin{pmatrix} w_1 \\ \vdots \\ w_k \end{pmatrix} \right) \,\middle|\, (A \to w_1 \dots w_k) \in \mathcal{P} \right\}$$

$$\forall a \in \Sigma : \delta(\overline{q}, a, a) = \{(\overline{q}, \varepsilon)\}$$

$$\delta(\overline{q}, \varepsilon, Z_0) = \{(q_{accept}, Z_0)\}$$

Dieser Kellerautomat ist *correct by construction.* □

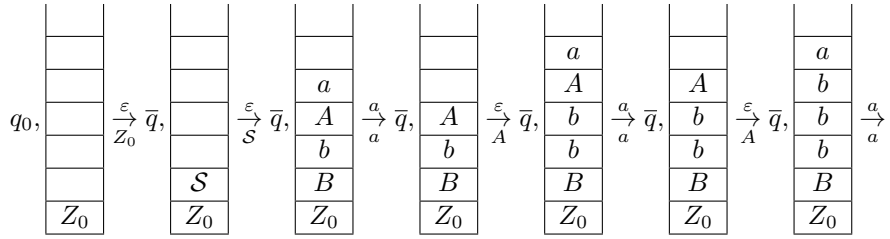
Beispiel: $G = (V = \{\mathcal{S}\}, \Sigma = \{a, b\}, \mathcal{P}, \mathcal{S})$ mit
$\mathcal{P}$:

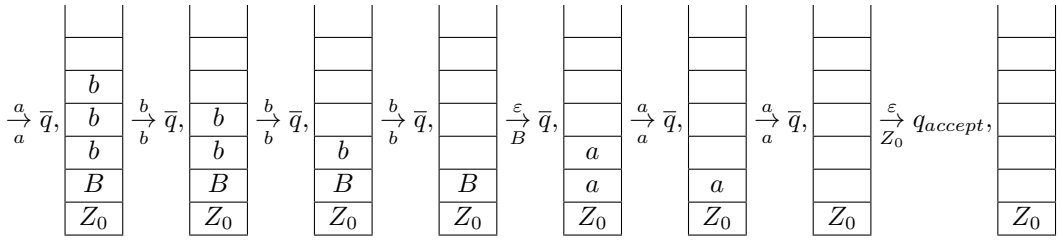
$$\mathcal{S} \to \varepsilon \mid aAbB \quad A \to ab \mid aAb \quad B \to aBb \mid aa$$

Beispiel-(links-)ableitung:

$$\mathcal{S} \to aAbB \to aaAbbB \to aaabbbB \to aaabbbaa \in L(G)$$

Beschreibung des Ablaufs des NPDA der aus der Grammatik erzeugt wird bei der akzeptierenden Rechnung zur Eingabe *aaabbbaa*:
zu lesen als:
„Zustand“, „Kellerinhalt“ $\xrightarrow[\text{„vom Keller gelesenes Zeichen oder } \varepsilon\text{“}]{\text{„von Eingabe gelesenes Zeichen oder } \varepsilon\text{“}}$ „neuer Zustand“, „neuer Kellerinhalt“





Alternative Darstellung der Konfiguration:
„Kellerinhalt (von unten nach oben)“ „Zustand“ „noch zu lesende Zeichen“
Darstellung der selben Rechnung mit dieser Notation:
$Z_0\,\mathbf{q_0}\,aaabbbaa \to Z_0\mathcal{S}\,\overline{\mathbf{q}}\,aaabbbaa \to Z_0BbAa\,\overline{\mathbf{q}}\,aaabbbaa \to Z_0BbA\,\overline{\mathbf{q}}\,aabbbaa \to Z_0BbbAa\,\overline{\mathbf{q}}\,aabbbaa \to$
$\to Z_0BbbA\,\overline{\mathbf{q}}\,abbbaa \to Z_0Bbbba\,\overline{\mathbf{q}}\,abbbaa \to Z_0Bbbb\,\overline{\mathbf{q}}\,bbbaa \to Z_0Bbb\,\overline{\mathbf{q}}\,bbaa \to Z_0Bb\,\overline{\mathbf{q}}\,baa \to$
$\to Z_0B\,\overline{\mathbf{q}}\,aa \to Z_0aa\,\overline{\mathbf{q}}\,aa \to Z_0a\,\overline{\mathbf{q}}\,a \to Z_0\,\overline{\mathbf{q}} \to Z_0\,\mathbf{q_{accept}}$

Fakt:

$$\mathcal{L}_{DPDA} \subsetneq \mathcal{L}_{NPDA}$$

Beispiel: (Beleg für den Fakt)

$$L = \left\{a^ib^ic^j \mid i,j \geq 0\right\} \cup \left\{a^jb^ic^i \mid i,j \geq 0\right\}$$

L ist kontextfrei, aber es gibt keinen DPDA A mit $L(A) = L$. Also $L \in \mathcal{L}_{NPDA}$ und $L \notin \mathcal{L}_{DPDA}$.

Abschlusseigenschaften kontextfreier Sprachen

Satz 3.37 *Die kontextfreien Sprachen sind abgeschlossen unter* $\cup, \cdot, ()^*$.

BEWEIS:
Sei L_i kontextfrei, erzeugt durch eine kontextfreie Grammatik G_i, wobei die Variablenmengen disjunkt sind.
$L_1 \cup L_2$ wird erzeugt durch: $\mathcal{S} \mapsto \mathcal{S}_1 \mid \mathcal{S}_2$.
$L_1 \cdot L_2$ wird erzeugt durch: $\mathcal{S} \mapsto \mathcal{S}_1\mathcal{S}_2$.
L_1^* wird erzeugt durch: $\mathcal{S} \mapsto \mathcal{S}\mathcal{S}_1 \mid \varepsilon$. □

Satz 3.38 *Die kontextfreien Sprachen sind* ***nicht*** *abgeschlossen unter* $\cap, \overline{()}$.

BEWEIS:

$$L_1 = \left\{a^nb^nc^i \mid n,i \geq 0\right\} \in \mathcal{L}_2$$
$$L_2 = \left\{a^ib^nc^n \mid n,i \geq 0\right\} \in \mathcal{L}_2$$
$$L_1 \cap L_2 = \{a^nb^nc^n \mid n \geq 0\} \notin \mathcal{L}_2$$

$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, d.h.: Wäre $\mathcal{L}_2$ unter Komplementbildung abgeschlossen, dann auch unter $\cap$.

□

Anmerkung: Satz 3.38 bedeutet **nicht**, dass für kontextfreie Sprachen L_1 und L_2 der Durchschnitt $L_1 \cap L_2$ grundsätzlich nicht kontextfrei ist.

Satz 3.39 *Sei L eine kontextfreie und R eine reguläre Sprache. Dann ist $L \cap R$ kontextfrei.*

BEWEIS: Übung. □

Beispiel: (Anwendung)

$$L = \{1\}^* \cup \left\{\text{☺}^i 1^p \mid i \geq 1, p \text{ ist prim}\right\}$$

Setze $R = L(☺1^*)$. R ist regulär.
$L \cap R = \{☺1^p \mid p \text{ ist prim}\} = \{☺\} \cdot \{1^p \mid p \text{ prim}\}$
$L \cap R$ hat die kontextfreie Pumpeigenschaft nicht und ist deshalb nicht kontextfrei.
$\Rightarrow L$ kann nicht kontextfrei sein.

„Gefühl für Kontextfreiheit“:
Eine kontextfreie Sprache ist „*sowas wie korrekte Klammerung*“ mit regulären Teilen zwischen den Klammern. **Fakt:** Für jede kontextfreie Sprache L gibt es n, R und h mit $h(D_n \cap R) = L$, wobei D_n die jeweilige Dyck-Sprache bezeichnet, h ein Homomorphismus ist und R eine reguläre Sprache ist. D_n ist die Sprache, welche alle korrekten Klammerungen mit n verschiedenen Klammersymbolen enthält.

Die Sprache

$$L = \left\{ww^R \mid w \in \{a, b\}^*\right\}, \text{ wobei } w^R \text{ die Spiegelung von } w \text{ bezeichnet}$$

ist kontextfrei.

Die Sprache

$$L = \{ww \mid w \in \{a, b\}^*\}$$

ist nicht kontextfrei.

4 Primitive Rekursion und μ-Rekursion

4.1 LOOP-, WHILE- und GOTO-Berechenbarkeit

Definition 4.1
Variablen: $x_1, x_2, \ldots$
Konstanten: $0, 1, 2, \ldots$
Trennsymbole: $\boxed{;}$ $\boxed{:=}$
Operationszeichen: $+, \dot{-}$ *dabei gilt* $a \dot{-} b =$ „*a monus b*“$= \max\{0, a - b\}$
Schlüsselwörter: *LOOP, DO, END*

Bei Eingabe $n_1, \ldots, n_k \in N$ *stehen diese Zahlen in* $x_1, \ldots, x_k$ *und alle anderen Variablen* x_i *sind mit* 0 *initialisiert. Am Ende der Rechnung steht die Ausgabe in* x_1*.*

Induktive Definition von* LOOP*-Programmen:
Wertzuweisungen: $x_i := x_j + c$
$x_i := x_j \dot{-} c$ *wobei* $c \in \mathbb{N}$ *eine beliebige Konstante ist.*
Also sind auch folgende Wertzuweisungen möglich:
$x_i := x_j$ *mit* $c = 0$
$x_i := c$ *mit* $x_i = x_j + c$ *und* $x_j = 0$ *eine unbenutzte Variable.*
Sind P_1 *und* P_2 *LOOP-Programme, so ist auch*

$$P_1; P_2$$

ein LOOP-Programm. Ist P ein LOOP-Programm, so ist auch

$$LOOP\ \ x_i\ \ DO\ \ P\ \ END$$

ein LOOP-Programm.

Die Bedeutung von $+$ und $\dot{-}$ ist klar.
$P_1; P_2$: Führe erst P_1 aus, danach P_2.
$LOOP\ \ x_i\ \ DO\ \ P\ \ END$: Führe P so oft hintereinander aus, wie der Wert von x_i zu Beginn der *LOOP*-Schleife ist.

Definition 4.2 *Eine Funktion* $f : \mathbb{N}^k \to \mathbb{N}$ *heißt* **LOOP*-berechenbar****, falls es ein LOOP-Programm P gibt, das f berechnet, so dass zu Beginn* $n_1, \ldots, n_k$ *in* $x_1, \ldots, x_k$ *stehen und zum Schluss* $f(n_1, \ldots, n_k)$ *in* x_1 *steht.*

LOOP-berechenbare Funktionen sind *total*.
Frage: Sind alle totalen, berechenbaren Funktionen *LOOP*-berechenbar? [Nein! Z.B. Ackermannfunktion]

Mit den vorhandenen Anweisungen können viele andere nützliche Kommandos mittels *LOOP*-Programm simuliert werden:
IF $x_i = 0$ *THEN P END*

```
y := 1;
LOOP x_i DO y := 0 END;
LOOP y DO P END;
```

$x_3 := x_1 + x_2$

```
x_3 := x_1 + 0;
LOOP x_2 DO x_3 := x_3 + 1 END;
```

$x_3 := x_1 \cdot x_2$

```
x_3 := 0;
LOOP x_2 DO x_3 := x_3 + x_1 END;
```

$x_3 = x_1 \mod x_2$ und $x_3 = \lfloor x_1/x_2 \rfloor$ gehen entsprechend.

Definition 4.3 WHILE-*Programme* *sind LOOP-Programme mit dem zusätzlichen Konstrukt*

$$WHILE\ \ x_i \neq 0\ \ DO\ \ P\ \ END;$$

P wird so lange wiederholt, wie x_i nicht 0 ist.

LOOP-Anweisungen können in *WHILE*-Anweisungen überführt werden.

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

Definition 4.4 *Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt* **WHILE-*berechenbar***, *falls es ein WHILE-Programm P gibt, das f berechnet, so dass zu Beginn $n_1, \ldots, n_k$ in $x_1, \ldots, x_k$ stehen und zum Schluss $f(n_1, \ldots, n_k)$ in x_1 steht.*

Satz 4.5 *Jede WHILE-berechenbare Funktion kann von einer deterministischen 1-Band-Turingmaschine berechnet werden.*

Definition 4.6 *Ein* **GOTO-*Programm*** *besteht aus Anweisungen A_i und Marken M_i:*

$$M_1 : A_1;\ \ M_2 : A_2;\ \ M_3 : A_3;\ \ \ldots\ \ M_k : A_k;$$

Es gibt dabei folgende Anweisungen:

Wertzuweisungen:	$x_i := x_j + c;$
	$x_i := x_j \dot{-} c;$
unbedingter Sprung:	$GOTO\ \ M_i;$
bedingter Sprung:	$IF\ \ x_i = c\ \ THEN\ \ GOTO\ \ M_j;$
Ende:	$HALT;$

WHILE-Anweisungen können in *GOTO*-Anweisungen überführt werden.

$$WHILE\ \ x \neq 0\ \ DO\ \ P\ \ END$$
$$\Leftrightarrow$$
$$\begin{array}{ll} M_1 : & IF\ \ x = 0\ \ THEN\ \ GOTO\ \ M_4; \\ M_2 : & P; \\ M_3 : & GOTO\ \ M_1 \\ M_4 : & \end{array}$$

Damit kann jedes *WHILE*-Programm in ein *GOTO*-Programm umgewandelt werden.

Umgekehrt kann auch jedes *GOTO*-Programm in ein *WHILE*-Programm umgewandelt werden.

$$M_1 : A_1;\ \ M_2 : A_2;\ \ M_3 : A_3;\ \ \ldots\ \ M_k : A_k;$$

$$\Leftrightarrow$$

$$\begin{array}{l} count := 1; \\ WHILE\ \ count \neq 0\ \ DO \\ \quad IF\ \ count = 1\ \ THEN\ \ A'_1\ \ END; \\ \quad IF\ \ count = 2\ \ THEN\ \ A'_2\ \ END; \\ \quad IF\ \ count = 3\ \ THEN\ \ A'_3\ \ END; \\ \quad \vdots \\ \quad IF\ \ count = k\ \ THEN\ \ A'_4\ \ END; \\ END \end{array}$$

wobei

$$A'_i = \begin{cases} x_j := x_l + c;\ \ count := count + 1; & \text{falls } A_i = (x_j := x_l + c;) \\ x_j := x_l \dot{-} c;\ \ count := count + 1; & \text{falls } A_i = (x_j := x_l \dot{-} c;) \\ count := j & \text{falls } A_i = (GOTO\ \ M_j) \\ IF\ \ x_j = c\ \ THEN\ \ count := l; & \\ \qquad ELSE\ \ count := count + 1 & \text{falls } A_i = (IF\ \ x_j = c\ \ THEN\ \ GOTO\ \ M_l) \\ count := 0 & \text{falls } A_i = (HALT) \end{cases}$$

Satz 4.7 (Kleenesche Normalform für WHILE-Programme)
Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einem WHILE-Konstrukt berechnet werden.

BEWEIS: Beliebiges *WHILE*-Programm $\rightarrow$ *GOTO*-Programm $\rightarrow$ *WHILE*-Programm. $\square$

Es gilt:

$$TM \Leftrightarrow RAM \Leftrightarrow GOTO \Leftrightarrow WHILE \supset LOOP$$

4.2 Definition primitiv rekursiver und μ-rekursiver Funktionen

Definition 4.8 *Die Klasse der* ***primitiv rekursiven Funktionen*** *ist induktiv definiert:*

1. *Die konstanten Funktionen sind primitiv rekursiv.*
2. *Die Projektionen sind primitiv rekursiv ($\pi_k^n(x_1, \ldots, x_n) = x_k$).*
3. *Die Nachfolger Funktion $s(n) = n + 1$ ist primitiv rekursiv.*
   $1 - 3$ sind die Basisfunktionen.
4. *Jede Funktion, die durch einsetzen von primitiv rekursiven Funktionen entsteht, ist primitiv rekursiv.*
5. *Jede Funktion, die durch primitive Rekursion aus primitiv rekursiven Funktionen entsteht, ist primitiv rekursiv.*

$$\begin{aligned} f(0, y_1, \ldots, y_k) &= g(y_1, \ldots, y_k) \\ f(\underbrace{s(n)}_{=n+1}, y_1, \ldots, y_k) &= h(f(n, y_1, \ldots, y_k), n, y_1, \ldots, y_k) \end{aligned}$$

Beispiele:

$$\begin{aligned} add &: \mathbb{N}^2 \to \mathbb{N} \\ add(x, y) &= „x + y“ \\ add(0, y) &= \pi_1^1(y) \\ add(s(x), y) &= s(add(x, y)) \end{aligned} \qquad \begin{aligned} u &: \mathbb{N} \to \mathbb{N} \\ u(n) &= „\max(n - 1, 0)“ \\ u(0) &= 0 \\ u(s(n)) &= \pi_2^2(u(n), n) = n \end{aligned}$$

$$\begin{aligned} mult &: \mathbb{N}^2 \to \mathbb{N} \\ mult(x, y) &= „x \cdot y“ \\ mult(0, y) &= 0 \\ mult(s(x), y) &= add(mult(x, y), x) \end{aligned} \qquad \begin{aligned} sub &: \mathbb{N}^2 \to \mathbb{N} \\ sub(x, y) &= „x \dot{-} y“ \\ sub(x, 0) &= x \\ sub(x, s(y)) &= u(sub(x, y)) \end{aligned}$$

$$c(x, y) = \binom{x + y + 1}{2} + x = \frac{1}{2}(x + y + 1)(x + y) + x \quad \text{ist primitiv rekursiv}$$

$y \downarrow$ / $x \to$	0	1	2	3	4	$\cdots$
0	0	2	5	9	14	$\ddots$
1	1	4	8	13	19	$\ddots$
2	3	7	12	18	25	$\ddots$
3	6	11	17	24	32	$\ddots$
4	10	16	23	31	40	$\ddots$
$\vdots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$	$\ddots$

$c : \mathbb{N}^2 \to \mathbb{N}$ ist eine bijektive Abbildung.
Seien nun e und f die inverse Abbildung von c, wobei e die erste und f die zweite Komponente des Urbilds berechnet.

$$\begin{aligned} e(c(x, y)) &= x & \qquad f(c(x, y)) &= y \\ e(18) &= 3 & \qquad f(18) &= 2 \end{aligned}$$

Man kann c auch auf mehr Dimensionen erweitern:
$\langle x, y, z\rangle = c(z, c(x, y))$
$\langle n_0, n_1, \ldots, n_k\rangle = c(n_0, c(n_1, \ldots, c(n_k, 0)\ldots))$
Seien nun d_i die inverse Abbildung von c, wobei d_i die i-te Komponente des Urbilds berechnet.

$$\begin{aligned}
d_0(n) &= e(n) && (= n_0)\\
d_1(n) &= e(f(n) && (= n_1)\\
&\vdots\\
d_i(n) &= e(\underbrace{f(f(\ldots f}_{i \text{ mal}}(n)\ldots))) && (= n_i)\\
&\vdots\\
d_k(n) &= e(\underbrace{f(f(\ldots f}_{k \text{ mal}}(n)\ldots))) && (= n_k)
\end{aligned}$$

Wenn e und f primitiv rekursiv sind, so sind auch die d_i primitiv rekursiv.

Ein Prädikat $P(x)$ ist primitiv rekursiv, falls die charakteristische Funktion primitiv rekursiv ist.

Beschränkte Maximumbildung:

$$q(n) = \max\{x \mid x \le n, P(x) = 1\}$$

Existiert das Maximum nicht, so ist $q(n) = 0$.

$$\begin{aligned}
q(0) &= 0\\
q(s(n)) &= \left\{ \begin{array}{ll} s(n+1) & \text{falls } P(n+1) = 1\\ q(n) & \text{sonst} \end{array} \right\} = q(n) + P(n+1)\cdot(n+1-q(n))
\end{aligned}$$

Ist $P(x)$ primitiv rekursiv, so ist auch $q(n)$ primitiv rekursiv.

Beschränkter Existenzquantor:

$$Q(n) = \begin{cases} 1 & \text{falls es ein } x \in \{0,\ldots,n\} \text{ mit } P(x) = 1 \text{ gibt}\\ 0 & \text{sonst .} \end{cases}$$

$$\begin{aligned}
Q(0) &= P(0)\\
Q(s(n)) &= P(s(n)) + Q(n) - P(s(n))\cdot Q(n)
\end{aligned}$$

Ist $P(x)$ primitiv rekursiv, so ist auch $Q(n)$ primitiv rekursiv.

Damit sind die folgenden Funktionen ebenfalls primitiv rekursiv:

$$\begin{aligned}
e'(n,m,k) &= \max\{x \mid x \le n \ \exists\, y \le k : c(x,y) = m\} && \text{ist primitiv rekursiv.}\\
e(n) &= e'(n,n,n) && \text{ist primitiv rekursiv.}\\
f'(n,m,k) &= \max\{y \mid y \le n \ \exists\, x \le k : c(x,y) = m\} && \text{ist primitiv rekursiv.}\\
f(n) &= f'(n,n,n) && \text{ist primitiv rekursiv.}
\end{aligned}$$

Satz 4.9 *Die Klasse der primitiv rekursiven Funktionen ist genau die Klasse der LOOP-berechenbaren Funktionen.*

BEWEIS:
„⇐“:
$f : \mathbb{N}^r \to \mathbb{N}$ sei *LOOP*-berechenbar mittels *LOOP*-Programm P
Ziel: $g_P(\underbrace{\langle a_0, a_1, \ldots, a_r\rangle}_{=n}) = \langle b_0, b_1, \ldots, b_k\rangle$ primitiv rekursiv, wobei die $b_0, \ldots, b_k$ die Inhalte der Register $x_0, \ldots, x_k$ sind nach Ausführung von P.

- Falls P von der Form ist: $x_i = x_j \pm c$:

$$g_P(n) = \langle d_0(n), \ldots, d_{i-1}(n), d_j(n) \pm c, d_{i+1}(n), \ldots, d_k(n)\rangle$$

- Falls P von der Form ist: $P_1; P_2 : g_P(n) = g_{P_2}(g_{P_1}(n))$
- Falls P von der Form ist *LOOP* x_i *DO* Q *END* :

$$\begin{aligned} \text{Hilfsfunktion:} \qquad h(0,x) =& x \\ h(s(n),x) =& g_Q(h(n,x)) \end{aligned}$$

  So ist $g_P = h(d_i(n), n)$

Damit gilt $f(n_1, \ldots, n_r) = d_0(g_P(\langle 0, n_1, \ldots, n_r, \underbrace{0, \ldots, 0}_{k-r \text{ viele}}\rangle))$

„⇒“:
Gegeben sei eine primitiv rekursive Funktion f.
Ziel: Angabe eines *LOOP*-Programms, das f berechnet.
Sei f durch primitive Rekursion gegeben, das heißt

$$\begin{aligned} f(0, y_1, \ldots, y_k) =& g(y_1, \ldots, y_k) \\ f(s(n+1), y_1, \ldots, y_k) =& h(f(n, y_1, \ldots, y_k), n, y_1, \ldots, y_k) \end{aligned}$$

Dies wird umgewandelt zu

$x_1 := n;$
$z := 0;$
$x_0 := g(y_1, \ldots, y_k);$
LOOP n *DO* $z := z + 1;\ x_0 := h(x_0, z, y_1, \ldots, y_k)$ *END*

In x_0 steht am Ende $f(n, y_1, \ldots, y_k)$. □

Definition 4.10
Der μ***-Operator*** *macht aus einer* $(k+1)$*-stelligen Funktion* f *eine* k*-stellige Funktion wie folgt:*

$$\mu f(x_1, \ldots, x_k) = \min\{n \mid f(n, x_1, \ldots, x_k) = 0 \text{ und } \forall m < n \text{ ist } f(m, x_1, \ldots, x_k) \text{ definiert}\} \ .$$

Konvention: $\min(\emptyset)$ *ist undefiniert.*
Die ***Klasse der*** μ***-rekursiven Funktionen****, ist die Klasse von (eventuell partiellen) Funktionen, die die Basisfunktionen enthält, abgeschlossen unter Einsetzung, primitiver Rekursion und Anwendung des* μ*-Operators ist.*

Satz 4.11 *Die Klasse der* μ*-rekursiven Funktionen ist genau die Klasse der berechenbaren Funktionen (das heißt Turing-berechenbaren Funktionen).*

BEWEIS:
„⇐“: Sei f berechenbar durch ein *WHILE*-Programm P.
Falls P von der Form *WHILE* $x_i \neq 0$ *DO* Q *END*

$$h(0,x) = x$$
$$h(s(n),x) = g_Q(h(n,x)) \qquad \text{(gegebenenfalls partiell)}$$

$$f(x) = g_P(x) = h(\underbrace{\mu(d_i \circ h)(x)}_{\text{besagt, wie oft die } \textit{WHILE}\text{-Schleife durchlaufen wird, bis } x_i = 0 \text{ ist}}, x)$$

„⇒“: Sei f durch μ-Operator definiert - also $f = \mu g$.
Ziel: *WHILE*-Programm angeben, das f berechnet.

$x_0 := 0;$
$y := g(0,x);$
WHILE $y \neq 0$ *DO* $x_0 := x_0 + 1; y := g(x_0,x);$ *END*

In x_0 steht am Ende $f(x) = \mu g(x)$. □

Satz 4.12 (Normalformsatz von Kleene)
Berechenbare Funktionen benötigen den μ-Operator maximal einmal.

4.3 Die Ackermann Funktion

$$
\begin{aligned}
ack :& \mathbb{N}_0^2 \to \mathbb{N}_0 \\
ack(0, y) &= y + 1 \\
ack(x, 0) &= ack(x - 1, 1) \\
ack(x, y) &= ack(x - 1, ack(x, y - 1))
\end{aligned}
$$

$ack(x,y)$	$y \to$					
	0	1	2	3	4	5
0	1	2	3	4	5	6
1	2	3	4	5	6	7
$x \downarrow$ 2	3	5	7	9	11	13
3	5	13	29	61	125	253
4	13	65533	$2^{65536} - 3$	$2^{2^{65536}} - 3$	$2^{2^{2^{65536}}} - 3$	$2^{2^{2^{2^{65536}}}} - 3$
5	65533	$ack(4, 65533)$	$\cdots$	$\cdots$	$\cdots$	$\cdots$

Es gilt:

- $ack(1, y) = y + 2$
- $ack(2, y) = 2 \cdot y + 3$
- $ack(x, y)$ ist Turing-berechenbar und damit auch μ-rekursiv.
- $ack(x, y)$ ist total.
- $ack(x, y)$ ist nicht primitiv rekursiv.

Für ein *LOOP*-Programm sei

$$
f_P(n) = \max \left\{ \sum_{i \geq 0} n_i' \;\middle|\; \begin{array}{l} \sum_{i \geq 0} n_i \leq n \ . \\ \text{Die } n_i \text{ sind die konkreten Eingaben für } P. \\ \text{Die } n_i' \text{ sind die Inhalte der Variablen, nachdem } P \\ \text{mit den } n_i \text{ gestartet worden ist und fertig ist.} \end{array} \right\} .
$$

Lemma 4.13 *Für jedes LOOP-Programm P gibt es eine Zahl k_p, so dass für alle n gilt: $f_P(n) \lneqq ack(k_P, n)$ ($\Leftrightarrow \forall P \in \{$LOOP-Programme$\} \exists k_P \forall n \in \mathbb{N}_0 : f_P(n) \lneqq ack(k_P, n)$).*

BEWEIS: Induktiv über den Aufbau von *LOOP*-Programmen.

- Falls P von der Form $x_i := -x_j \pm c$ ist:
  Ohne Beschränkung der Allgemeinheit gilt $c \in \{0, 1\}$

  $$f_P(n) \leq 2n + 1 \lneqq ack(2, n)$$

  Wähle $k_P = 2$.

- Falls P von der Form $P_1; P_2$ ist:
  Per Induktionsannahme gibt es k_{P_1} und k_{P_2} mit $f_{P_1}(n) \lneqq ack(k_{P_1}, n)$ und $f_{P_2}(n) \lneqq ack(k_{P_2}, n)$.
  Mit $k_3 := \max\{k_{P_1} - 1, k_{P_2}\}$:

$$\begin{aligned} f_P(n) \leq & f_{P_2}(f_{P_1}(n)) \\ \lneqq & ack(k_{P_2}, ack(k_{P_1}, n)) \\ \leq & ack(k_3, ack(k_3 + 1, n)) \\ = & ack(k_3 + 1, n + 1) \\ \leq & ack(k_3 + 2, n) \ . \end{aligned}$$

  Wähle $k_P = \max\{k_{P_1} - 1, k_{P_2}\} + 2$.

- Falls P von der Form *LOOP* x_i *DO* Q *END* ist:
  Per Induktionsannahme gibt es k_Q mit $f_Q(n) \lneqq ack(k_Q, n)$.
  Ohne Beschränkung der Allgemeinheit kommt x_i in Q nicht vor.
  Sei m $(m \leq n)$ die Zahl, so dass m Schleifendurchläufe die größte Summe $\sum_{i \geq 0} n'_i$ erzeugt.

$$\begin{aligned} m = 0: \quad & f_P(n) = n \lneqq ack(0, n) \\ m = 1: \quad & f_P(n) \leq f_Q(n) \lneqq ack(k_Q, n) \\ m \geq 2: \quad & f_P(n) \leq \underbrace{f_Q(f_Q(\ldots f_Q(f_Q}_{m \text{ mal}}(n - m)) \ldots)) + \underbrace{m}_{\text{Inhalt von } x_i} \\ & \lneqq ack(k_Q, f_Q(\ldots f_Q(f_Q(n - m)) \ldots)) + m \\ & \vdots \\ & \lneqq \underbrace{ack(k_Q, ack(k_Q, \ldots ack(k_Q, ack}_{m \text{ mal}}(k_Q, n - m)) \ldots)) + m \\ & \lneqq \underbrace{ack(k_Q, ack(k_Q, \ldots ack(k_Q, ack}_{m \text{ mal}}(k_Q + 1, n - m)) \ldots)) \\ & = \underbrace{ack(k_Q, ack(k_Q, \ldots ack}_{(m-1) \text{ mal}}(k_Q + 1, n - m + 1) \ldots)) \\ & \vdots \\ & = ack(k_Q + 1, n) \end{aligned}$$

  Wähle $k_P = k_Q + 1$. □

Satz 4.14 *$ack(x, y)$ ist nicht LOOP-berechenbar und somit auch nicht primitiv rekursiv.*

BEWEIS: Diagonalisierung:

Annahme $ack(x, y)$ ist *LOOP*-berechenbar durch P.

Dann ist auch $g(n) = ack(n, n)$ durch P' *LOOP*-berechenbar.

Nach Lemma 4.13 gibt es eine Konstante $k_{P'}$ sodass $g(n) \leq f_{P'}(n) \lneqq ack(k_{P'}, n)$.

Für $n = k_{P'}$ gilt damit $g(k_{P'}) \leq f_{P'}(k_{P'}) \lneqq ack(k_{P'}, k_{P'}) = g(k_{P'})$ ↯ □