Skript zur Vorlesung

Automatentheorie und Formale Sprachen

Sommersemester 2018

Prof. Dr. Steffen Reith
Steffen.Reith@hs-rm.de

Hochschule RheinMain
Fachbereich Design Informatik Medien

Erstellt von:	Steffen Reith
Zuletzt überarbeitet von:	Steffen Reith
Email:	steffen.reith@hs-rm.de
Erste Version vollendet:	Februar 2007
Version:	24b002f
Datum:	2018-08-01 23:39:08 +0200

Es ist nicht das Wissen, sondern das Lernen,
nicht das Besitzen, sondern das Erwerben,
nicht das Dasein, sondern das Hinkommen,
was den größten Genuß gewährt.

Carl Friedrich Gauss

Jede Art von Anfang ist schwer.

Paul Atreides

No! Try not. Do, or do not. There is no try.

Yoda

Dieses Skript ist aus der Vorlesung „Automatentheorie und Formale Sprachen“ (früher Informatik 2) des Bachelor- und Diplom-Studiengangs „Allgemeine Informatik“ an der Fachhochschule Wiesbaden hervorgegangen. Ich danke allen Hörern dieser Vorlesung für konstruktive Anmerkungen und Verbesserungen. Insbesondere Kim Stebel,Thomas Frase, Patrick Vogt, Carlos Manrique, Tim Kohlstadt und Fabio Campos haben viele Fehler und Unklarheiten angemerkt. Herr Harald Heckmann fand einen Typo im CYK-Algorithmus uns Frau Savina Diez zeigte mir einen Bug bei der Verwendung des Pumping Lemmas. Danke! Alle weiteren Fehler liegen in der Verantwortung des Autors.

Naturgemäß ist ein Skript nie fehlerfrei (ganz im Gegenteil!) und es ändert (mit Sicherheit!) sich im Laufe der Zeit. Deshalb bin ich auf weitere Verbesserungsvorschläge angewiesen und freue mich sehr über weitere Anregungen und Ergänzungen durch die Hörer.

Inhaltsverzeichnis

Abbildungsverzeichnis

Liste der Algorithmen

1. Einleitung

In natürlichen Sprachen legen die Regeln einer Grammatik fest, welche Aneinanderreihung von Worten und Satzzeichen korrekte Sätze ($\triangleq$ *Syntax*) bilden. Ein solche Grammatik macht keinerlei Aussagen über den Sinn/Bedeutung ($\triangleq$ *Semantik*) dieser syntaktisch korrekten Sätze. So ist z.B. der Satz „Wiesbaden wohnt weiterhin weich“ syntaktisch korrekt, allerdings trägt dieser Satz keinen Sinn.

Der amerikanische Linguist Noam Chomsky[1], hatte die Idee (siehe [Cho56]), alle syntaktisch korrekten Sätze einer (natürlichen) Sprache durch ein (endliches) System von formalen Regeln zu erzeugen. Die von Chomsky vorgeschlagenen Regelsysteme nennt man heute auch *generative Grammatiken*. Obwohl diese Idee bis heute in den Sprachwissenschaften umstritten ist, hat sie sich in der Informatik als sehr fruchtbar erwiesen. Schon wesentlich früher, wurden die so genannten *Semi-Thue-Systeme* von Axel Thue[2] untersucht, die in der Informatik auch eine wichtige Bedeutung erlangt haben.

Im Laufe der Zeit haben sich in der Theoretischen Informatik folgende Sprechweisen eingebürgert: Eine endliche Menge Σ heißt *Alphabet* und die Elemente der Menge Σ werden *Buchstaben* genannt. Eine beliebige Folge von Buchstaben ($\triangleq$ Elementen aus Σ) nennt man *Wort* über Σ, oder nur Wort, wenn das verwendete Alphabet aus dem Kontext hervorgeht. Eine (formale) *Sprache* (über Σ) ist dann eine beliebige Menge von Worten.

Beispiel 1: *Sei* $\Sigma = \{(,), +, -, *, /, x\}$*, dann ist die Menge aller korrekten arithmetischen Ausdrücke EXPR eine formale Sprache über* Σ *und es gilt:*

- $(x - x) \in \text{EXPR}$
- $((x + x) * x)/x \in \text{EXPR}$
- $)((x-) * x \notin \text{EXPR}$

(Formale) Sprachen sind meist unendliche Objekte, d.h. in einer Sprache sind im Allgemeinen unendlich viele Worte enthalten. Aus diesem Grund brauchen wir Regeln, um (algorithmisch) mit formalen Sprachen umgehen zu können. Wir brauchen also geeignete Grammatiken und Erkenner ($\triangleq$ Automaten), die feststellen können, ob ein Wort zu einer Sprache gehört oder nicht.

Beispiel 2 (Teil einer natürlichen Sprache):
Gegeben seien die folgenden Regeln:

Satz	$\rightarrow$	**Subjekt Pradikat Objekt**
Subjekt	$\rightarrow$	**Artikel Attribut Substantiv**
Artikel	$\rightarrow$	ϵ | der | die | das
Attribut	$\rightarrow$	ϵ
Attribut	$\rightarrow$	**Adjektiv**
Attribut	$\rightarrow$	**Adjektiv Attribut**
Adjektiv	$\rightarrow$	kleine | bissige | verschlafene
Substantiv	$\rightarrow$	Student | Katze
Pradikat	$\rightarrow$	jagt | betritt
Objekt	$\rightarrow$	**Artikel Attribut Substantiv**

Dabei wird das Symbol „|“ dazu verwendet, verschiedene Alternativen zu markieren, d.h. $\mathbf{A} \rightarrow \mathbf{B} \mid \mathbf{C}$ ist die Abkürzung für die zwei Regeln $\mathbf{A} \rightarrow \mathbf{B}$ und $\mathbf{A} \rightarrow \mathbf{C}$.

[1] *1928 in Philadelphia (USA)
[2] *1863 in Tönsberg (Norwegen) - †1922 in Oslo (Norwegen)

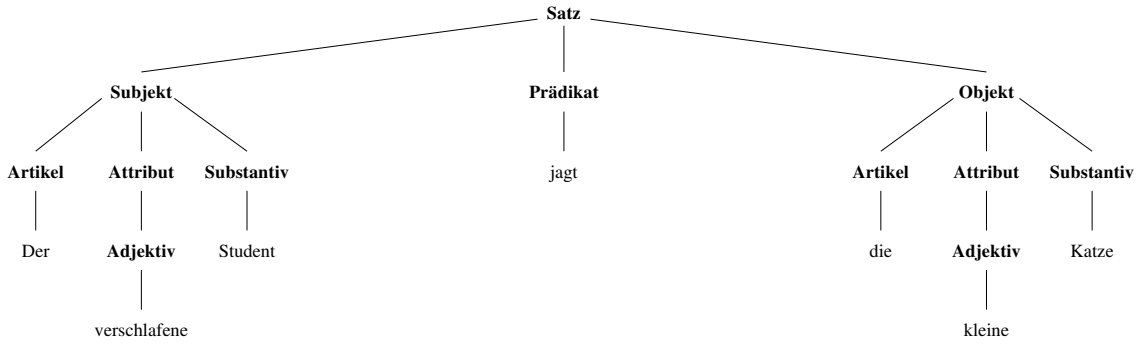


Abbildung 1: Beispiel für einen Syntaxbaum

Mit Hilfe der obigen Regeln können z.B. die folgenden Sätze gebildet werden:

- Der kleine bissige Student betritt die verschlafene Mensa
- Der verschlafene Student jagt die kleine Katze

Wie schon erwähnt, müssen solche Sätze keinen Sinn ergeben. Mit Hilfe von so genannten „*Syntaxbäumen*“ (siehe Abbildung 1), kann man die „Ableitungsschritte“ ($\triangleq$ Erzeugung des Satzes) veranschaulichen.

1.1. L-Systeme

Die *L-Systeme* wurden 1968 von Aristid Lindenmayer[3] als mathematisches Modell des Pflanzenwachstums eingeführt. Die L-Systeme gehören, ähnlich wie die Semi-Thue-Systeme, zu den Termersetzungssystemen. In diesem Abschnitt soll die einfachste Klasse von L-Systemen verwendet werden, die als D0L-System bezeichnet werden. Bei D0L-System sind alle Regeln deterministisch ($\triangleq$ es kann immer nur genau eine Regel für einen Buchstaben angewendet werden) und kontextfrei ($\triangleq$ die Ersetzung hängt nicht von den umgebenden Buchstaben ab).

Bevor wir D0L-Systeme einsetzen können, benötigen wir einige Definitionen:

Definition 3 (D0L-Systeme):

i) Sei Σ *ein* Alphabet*, dann bezeichnen wir mit* Σ^* *die Menge aller* Wörter *über* Σ *(inkl. dem leeren Wort* ϵ*).*

ii) Ein 0L-System G *ist ein Tripel* $G = (\Sigma, w, P)$*, wobei* Σ *das Alphabet des L-Systems ist,* w *das* Axiom *und* $P \subseteq \Sigma \times \Sigma^*$ *die Menge der* Produktionen*.*

iii) Eine Produktion $(a, \chi) \in P$ *wird auch als* $a \to \chi$ *geschrieben, wobei der Buchstabe* a *als* Vorgänger *und* χ *als* Nachfolger *dieser Produktion bezeichnet wird.*

iv) Für jeden Buchstaben $a \in \Sigma$ *existiert eine Produktion* $(a, \chi) \in P$*. Geben wir aus Bequemlichkeit für einen Buchstaben* a *keine explizite Produktion an, so vereinbaren wir implizit die Produktion* $(a, a) \in P$*.*

v) Ein 0L-System heißt deterministisch*, wenn es für jeden Buchstaben* $a \in \Sigma$ *nur* genau eine *Produktion* $(a, \chi) \in P$ *gibt. Solche Systeme werden als* D0L-Systeme *bezeichnet.*

[3] *1925 in Budapest (Ungarn) - †1989 evtl. Utrecht (Niederlande)

vi) Sei $\mu = a_1 \dots a_m$ ein beliebiges Wort über Σ, dann kann das Wort $\nu = \chi_1 \dots \chi_m$ direkt von μ abgeleitet *werden (in Zeichen: $\mu \vdash \nu$), wenn für alle $i = 1, \dots, m$ eine Produktion $a_i \to \chi_i$ existiert. Ein Wort ν heißt von G* generiert*, wenn es durch eine endliche Anzahl von Schritten aus dem Axiom abgeleitet werden kann.*

Wird ein Wort ν von dem Axiom w abgeleitet, dann können wir $w \vdash \mu_1 \vdash \mu_2 \vdash \dots \vdash \mu_m = \nu$ schreiben.

Bemerkung 4: *Bei L-Systemen werden alle Produktionen „gleichzeitig" auf die Buchstaben eines Wort angewendet. Aus diesem Grund war es auch wichtig, dass für jeden Buchstaben $a \in \Sigma$ eine Produktion der Form (a, χ) existiert.*

Beispiel 5: *Sei $G = (\Sigma, w, P)$, wobei*

- $\Sigma = \{a, b, c\}$,
- $w = abc$ *und*
- $P = \{a \to aa, b \to bb, c \to cc\}$.

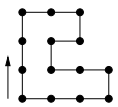
Mit Hilfe dieses D0L-Systems können die Worte $a^{2^n} b^{2^n} c^{2^n}$ für $n \geq 0$ erzeugt werden. Dabei ist a^m die abkürzende Schreibweise für $\underbrace{aa \dots a}_{m\text{-}mal}$.

1.2. Die Turtle-Interpretation von Strings

Sei δ ein beliebiger Winkel, dann werden die Buchstaben F, f, $+$ und $-$ als Kommandos wie folgt interpretiert:

F	$\triangleq$	Bewege den Stift einen Schritt der Länge d und zeichne eine Linie
f	$\triangleq$	Bewege den Stift einen Schritt der Länge d und zeichne keine Linie
$-$	$\triangleq$	drehe um δ Grad nach rechts
$+$	$\triangleq$	drehe um δ Grad nach links

Bei einem Winkel von $\delta = 90°$ wird der String $\texttt{FFF} - \texttt{FF} - \texttt{F} - \texttt{F} + \texttt{F} + \texttt{FF} - \texttt{F} - \texttt{FFF}$ in die Graphik



umgesetzt, wobei an der linken unteren Ecke gestartet wird.

Beispiel 6 (Kochsche Schneeflocke): *Gegeben sei das L-System $G = (\Sigma, w, P)$ mit*

- $\Sigma = \{\texttt{F}, +, -\}$,
- $w = \texttt{F}$ *und*
- $P = \{\texttt{F} \to \texttt{F} + \texttt{F} - - \texttt{F} + \texttt{F}\}$

Legen wir den Winkel $\delta = 45°$ fest, dann haben die einzelnen Ableitungsschritte die folgende graphische Repräsentationen:

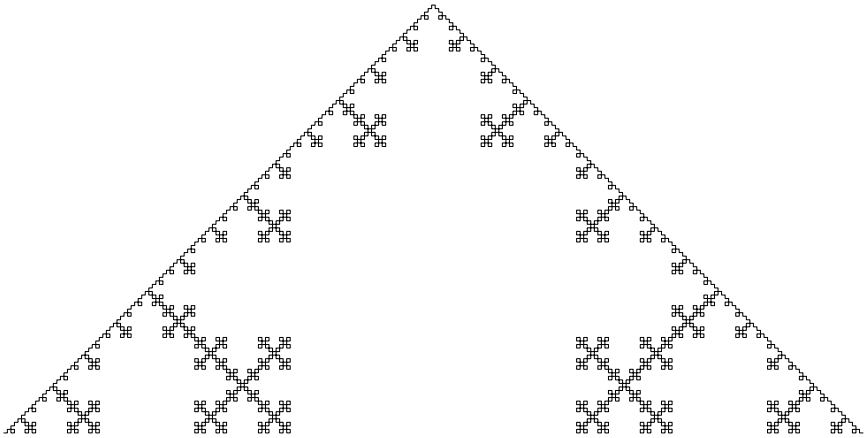
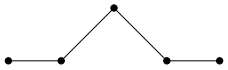
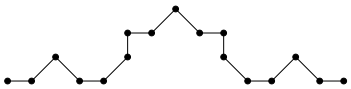


Abbildung 2: Beispiel für eine durch ein L-System erzeugte Graphik

#Ableitungsschritte	*Abgeleitetes Wort*	*Graphische Repräsentation*
$n = 0$	$\mathtt{F}$	
$n = 1$	$\mathtt{F+F--F+F}$	
$n = 2$	$\mathtt{F+F--F+F+F+F}$ $\mathtt{--F+F--F+F--}$ $\mathtt{F+F+F+F--F+F}$	

Die sich ergebende Kurve ist als „Kochsche Schneeflocke“ bekannt.

Beispiel 7: *Sei das L-System* $G = (\{\mathtt{F}, -, +\}, -\mathtt{F}, \{\mathtt{F} \to \mathtt{F+F-F-F+F}\})$ *gegeben. Wählen wir* $\delta = 90°$*, dann ergibt sich mit der Turtle-Interpretation Abbildung 2.*

Beispiel 8 (Drachenkurve): *Gegeben sei* $\delta = 90°$ *und das L-System* $G = (\{\mathtt{F_r}, \mathtt{F_l}, +, -\}, \mathtt{F_l}, \{\mathtt{F_l} \to \mathtt{F_l + F_r}+, \mathtt{F_r} \to -\mathtt{F_l} - \mathtt{F_r}\})$*, dann ergibt sich mit der Turtle-Interpretation Abbildung 3, wobei sowohl* $\mathtt{F_l}$ *als auch* $\mathtt{F_r}$ *als „Bewege den Stift einen Schritt der Länge d und zeichne eine Linie“ interpretiert wird.*

2. Die Chomsky Hierarchie

In den 1950er Jahren hatte der Linguist Noam Chomsky die Idee, alle syntaktisch korrekten Sätze einer natürlichen Sprache durch ein System von Regeln zu erzeugen. Dazu definierte er verschiedene Klassen von Sprachen, die heute als *Chomsky Hierarchie* bekannt sind. In der Informatik hat sich diese Idee als sehr fruchtbar erwiesen und bildet heute die Grundlagen für die Entwicklung von Programmiersprachen, XML und zahllosen weiteren Anwendungen. Bevor wir die Klassen der Chomsky Hierarchie einführen können, müssen wir, um Unklarheiten und Verwechslungen zu vermeiden, einige Schreib- und Sprechweisen genau festlegen.

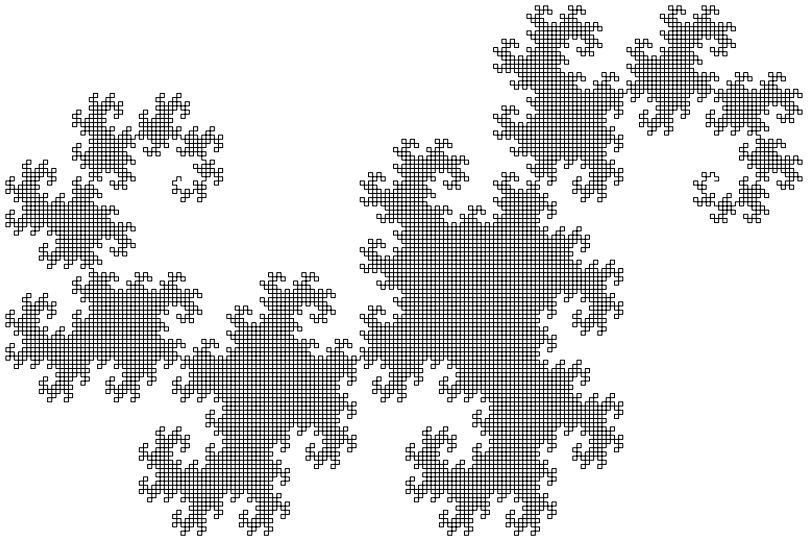


Abbildung 3: Die Drachenkurve

2.1. Einige grundlegende Definitionen und Schreibweisen

Definition 9:

- *Ein* Alphabet *ist eine endliche Menge von* Buchstaben.
  Bezeichnung meist: Σ, Γ, Δ
- *Ein* Wort *über einem Alphabet* Σ *ist eine endliche Aneinanderreihung von beliebigen Buchstaben aus* Σ.
  Bezeichnung meist: u, v, w, x, y, z
  Schreibweise: $z = a_1 \dots a_n$, *wobei* $a_1, \dots, a_n \in \Sigma$
- *Die* Konkatenation *zweier Worte* $u = a_1 \dots a_n$ *und* $v = b_1 \dots b_m$ *ist festgelegt durch* $uv =_{\text{def}} a_1 \dots a_n b_1 \dots b_m$. *Oft schreibt man auch* $u \cdot v$ *statt* uv.
- *Das* leere Wort ϵ *wird durch* $u\epsilon = \epsilon u = u$ *definiert (*neutrales Element *bzgl. der Konkatenation).*
- *Das Wort* $\underbrace{aa \dots a}_{n-\text{mal}}$ *wird mit* a^n *abgekürzt. Zusätzlich legen wir fest, dass* $a^0 =_{\text{def}} \epsilon$.
- *Die* Menge aller Worte *über* Σ *(inkl. dem leeren Wort* ϵ*) wird mit* Σ^* *bezeichnet. Wir legen fest* $\Sigma^+ =_{\text{def}} \Sigma^* \backslash \{\epsilon\}$ *(Menge der Worte über* Σ *ohne das leere Wort).*
- *Eine Teilmenge* L *von* Σ^* *heißt* (formale) Sprache *(über* Σ*).*
- *Seien* L_1 *und* L_2 *Sprachen über* Σ, *dann definieren wir die* Konkatenation *von* L_1 *und* L_2 *als* $L_1 \cdot L_2 =_{\text{def}} \{w \in \Sigma^* \mid$ *es gibt ein* $u \in L_1$, *ein* $v \in L_2$ *und* $w = uv\}$.
- *Die* Länge *eines Wortes* w *ist definiert als die Anzahl der Buchstaben in* w. *Aus diesem Grund ist die Länge des leeren Worts* ϵ *gleich* 0.
  Schreibweise: $|w|$
- *Die* Menge aller Worte (über Σ) der Länge n *ist durch* $\Sigma^n =_{\text{def}} \{w \in \Sigma^* \mid$ *wobei* $|w| = n\}$ *definiert und die* Menge aller Worte (über Σ) die nicht länger als n *sind, wird durch* $\Sigma^{\leq n} =_{\text{def}} \{w \in \Sigma^* \mid$ *wobei* $|w| \leq n\}$ *festgelegt.*

- *Ist $w \in \Sigma^*$ und $a \in \Sigma$, so bezeichnet $|w|_a$ die Anzahl des Auftretens von a im Wort w.*
- *Ist $w \in \Sigma^*$ mit $w = a_1 \dots a_m$, dann ist $w^R =_{\text{def}} a_m \dots a_1$ (w^R ist das Wort w von rechts nach links gelesen).*

Beispiel 10: *Sei das Alphabet $\Sigma = \{0, 1\}$ gegeben, dann ist $\Sigma^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$ und $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \dots\}$. Die Menge von Worten $\{w \mid |w|_0$ ist gerade$\} = \{\epsilon, 1, 00, 11, 001, 010, 100, 111, \dots\}$ ist eine formale Sprache (über $\{0, 1\}$).*

Nachdem wir festgelegt haben, dass jede Menge $L \subseteq \Sigma^*$ eine (formale) Sprache über Σ ist, stellt sich die Frage, wie wir die Worte der Sprache mit Hilfe von Regeln erzeugen. Dazu benötigen wir die folgenden Festlegungen:

Definition 11 (Chomsky Grammatiken):
Eine (generative) Grammatik *G ist ein 4-Tupel $G = (\Sigma, N, P, S)$, wobei*

- *Σ das Alphabet der* Terminalsymbole*,*
- *N eine endliche Menge von* Nichtterminalen *(mit $\Sigma \cap N = \emptyset$),*
- *$S \in N$ das* Startsymbol *und*
- *P die endliche Menge der* Produktionen *mit $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$.*
  Schreibweise: Eine Produktion $(u, v) \in P$ schreibt man auch als $u \to v$.

Definition 12 (Ableiten von Worten):
Sei $G = (\Sigma, N, P, S)$ eine Grammatik und $u_1, u_2, v, w \in (\Sigma \cup N)^*$, dann gilt:

- *Wenn $v \to w \in P$, dann heißt $u_1 w u_2$ (von G) aus $u_1 v u_2$* in einem Schritt erzeugt *oder* abgeleitet.
  Schreibweise: $u_1 v u_2 \vdash u_1 w u_2$
- *Das Wort w heißt aus v (von G)* in t Schritten erzeugt (abgeleitet)*, falls es eine Folge von Worten $x_0, \dots, x_t$ gibt, mit $v = x_0 \vdash x_1 \vdash \dots \vdash x_t = w$.*
  Schreibweise: $v \vdash^t w$
- *Das Wort w heißt aus v von G* erzeugt *oder* abgeleitet*, falls es ein $t \in \mathbb{N}$ gibt mit $v \vdash^t w$.*
  Schreibweise: $v \vdash^* w$
- *Die Sprache $L(G) =_{\text{def}} \{w \in \Sigma^* \mid S \vdash^* w\}$ heißt die von der Grammatik G* erzeugte Sprache*.*

2.2. Die Klassen der Chomsky Hierarchie

Sei $G = (\Sigma, N, P, S)$ eine Grammatik, dann gilt:

- Jede solche Grammatik heißt auch Grammatik vom *Typ 0*.
- G heißt Grammatik vom *Typ 1* oder *kontextsensitive Grammatik*, falls jede Produktion der Form $u_1 A u_2 \to u_1 w u_2$ mit $A \in N, u_1, u_2, w \in (\Sigma \cup N)^*$ und $|w| \geq 1$ entspricht.
- G heißt Grammatik vom *Typ 2* oder *kontextfreie Grammatik*, falls jede Produktion der Form $A \to w$ mit $A \in N, w \in (N \cup \Sigma)^*$ und $|w| \geq 1$ entspricht.

- G heißt Grammatik vom *Typ 3* oder *reguläre Grammatik*, falls jede Produktion der Form $A \to a\,B$ oder $A \to a$ mit $A, B \in N$ und $a \in \Sigma$ entspricht.

Sei $i \in \{0, 1, 2, 3\}$, dann heißt eine Sprache L auch *Sprache vom Typ* i, falls es eine Grammatik G vom Typ i mit $L(G) = L \backslash \{\epsilon\}$ gibt.

- Die Sprachen vom Typ 0 heißen auch *rekursiv aufzählbare Sprachen.*
- Die Sprachen vom Typ 1 heißen auch *kontextsensitive Sprachen.*
- Die Sprachen vom Typ 2 heißen auch *kontextfreie Sprachen.*
- Die Sprachen vom Typ 3 heißen auch *reguläre Sprachen.*

Bemerkung 13: *Die Sprache $L(G)$ besteht nur aus Terminalsymbolworten. Die Nichtterminale sind also „Hilfszeichen“ oder „Variablen“, die ersetzt werden. Der Ersetzungsprozess ist erst dann zu Ende, wenn keine Nichtterminale oder keine passenden Regeln mehr vorhanden sind. In einem Ableitungsschritt wird* genau eine *Produktion angewendet.*

Definition 14 (Äquivalenz von Grammatiken): *Zwei Grammatiken G und G' heißen* äquivalent*, falls $L(G) = L(G')$ gilt.*

Im Allgemeinen ist es nicht einfach die die Äquivalenz von Grammatiken fest zu stellen. Für reguläre Grammatiken ist das noch verhältnismäßig einfach, aber schon für kontextfreie Grammatiken ist das Problem unentscheidbar (vgl. Definition 93 auf Seite 43), d.h. man kann *beweisen*, dass kein Algorithmus existiert, der dieses Problem löst.

Offensichtlich ist für $i \in \{0, 1, 2\}$ jede Grammatik vom Typ $i + 1$ auch eine Grammatik vom Typ i. Analog gilt das auch für die Sprachen, die durch eine generative Grammatiken erzeugt werden. Eine graphische Übersicht gibt Abbildung 4.

Eigenschaft 15: *Sei $\mathbf{L}_i =_{\text{def}} \{L \mid L$ ist eine Sprache vom Typ $i\}$, dann gilt, wie leicht einsichtig ist, die folgende Eigenschaft:*

$$\mathbf{L}_3 \subseteq \mathbf{L}_2 \subseteq \mathbf{L}_1 \subseteq \mathbf{L}_0$$

Die Folge $\mathbf{L}_3 \subseteq \mathbf{L}_2 \subseteq \mathbf{L}_1 \subseteq \mathbf{L}_0$ von Sprachklassen nennen wir *Chomsky Hierarchie*. Wir werden noch sehen, dass diese Inklusionen echt sind, d.h. es gilt sogar $\mathbf{L}_3 \subset \mathbf{L}_2 \subset \mathbf{L}_1 \subset \mathbf{L}_0$.

Beispiel 16: *Sei $G_1 = (\{a, b\}, \{S\}, \{S \to aSa, S \to bSb, S \to a, S \to b, S \to \epsilon\}, S)$. Die Grammatik erzeugt die Menge der Palindrome[4] über dem Alphabet $\{a, b\}$. Also ist die Sprache $L(G_1) = \{w \in \{a, b\}^* \mid v_1 \dots v_m = w = w^R = v_m \dots v_1\}$. Wir können z.B. $S \vdash aSa \vdash abSba \vdash abba$ oder $S \vdash^* abSba \vdash ababa$ erzeugen.*

Beispiel 17: *Sei $G_2 = (\{a, b\}, \{S\}, \{S \to \epsilon, S \to aSb\}, S)$. Die Grammatik G_2 erzeugt die Sprache $L(G_2) = \{w \in \{a, b\}^* \mid w = a^n b^n$ mit $n \in \mathbb{N}\}$.*

Beispiel 18: *$G_3 = (\{), (, x, +, *\}, \{E, T, F\}, P, E)$, wobei:*

$$\begin{aligned} P = \{ \quad & E \to T \\ & E \to E + T, \\ & T \to F, \\ & T \to T * F, \\ & F \to x, \\ & F \to (E) \quad \} \end{aligned}$$

[4] Palindrome sind Zeichenketten, die vorwärts und rückwärts gelesen gleich sind (z.B. „leohortetrohoel“ oder „abba“).

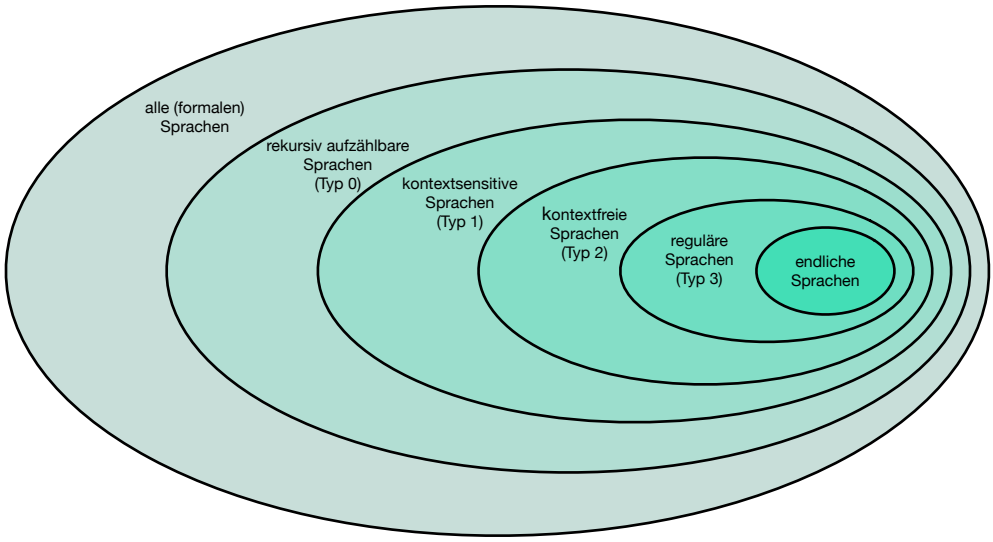


Abbildung 4: Die Chomsky-Hierarchie

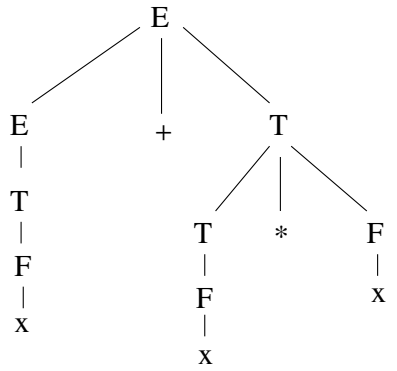


Abbildung 5: Syntaxbaum für $x + x * x$

Diese Grammatik beschreibt einfache geklammerte arithmetische Ausdrücke, wie man leicht ausprobieren kann (Interpretiere E als „Expression“, T als „Term“ und F als „Faktor“).

$$E \vdash E + T \vdash E + T * F \vdash E + T * x \vdash T + T * x \vdash^{*} x + x * x$$

Der Erzeugungsprozess von $x + x * x$ wird graphisch als Syntaxbaum *in Abbildung 5 dargestellt. Weiterhin können wir $E \vdash T \vdash E + T \vdash E + T * F \vdash T + T * T \vdash^{*} F + F * F \vdash^{*} x + x * x$ und $E \vdash T \vdash E + T \vdash E + x \vdash E + T + x \vdash^{*} x + x * x$ ableiten, d.h. es kann passieren, dass Wörter einer Sprache auf verschiedenen „Ableitungswegen“ erzeugt werden können („Mehrdeutigkeit“).*

Beispiel 19: *Welche Sprache wird durch $G_4 = (\{a, b, c\}, \{S, B\}, \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\}, S)$ erzeugt? Hinweis: Nachdenken geht schneller als probieren, wenn man die Anzahl der Buchstaben a, b und c in den erzeugbaren Worten zählt.*

2.3. Das Wortproblem

Wollen wir in der Praxis (generative) Grammatiken dazu verwenden, um von einer Eingabe zu prüfen, ob sie syntaktisch korrekt ist, so müssen wir das folgende Problem lösen:

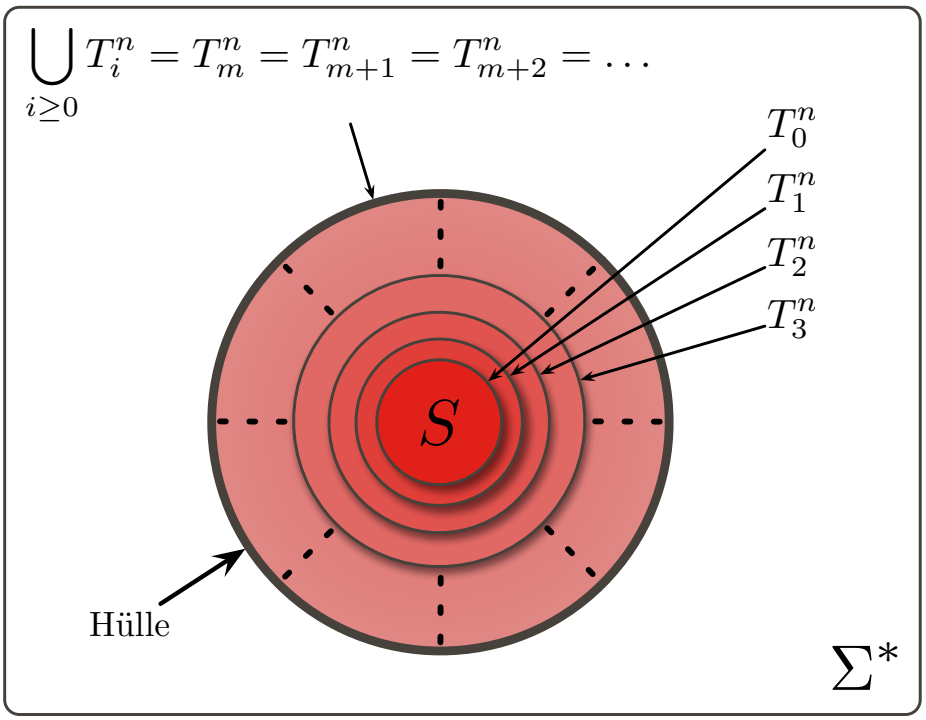


Abbildung 6: Hüllenkonstruktion für Typ 1 - Grammatiken

PROBLEM:	MEMBER
EINGABE:	Grammatik G und Wort w
FRAGE:	Gilt $w \in L(G)$?

Dieses Problem ist als „*Wortproblem*“ bekannt. In der Praxis tritt es z.B. auf, wenn ein Compiler entscheiden muss, ob eine Eingabe ein gültiges Programm darstellt oder wenn ein XML-Parser ein Dokument validieren soll. Wir wollen nun zeigen, dass es möglich ist das Wortproblem für kontextsensitive Sprachen mit Hilfe eines Algorithmus zu lösen:

Satz 20: *Es gibt einen Algorithmus, der das Wortproblem für Typ 1 - Grammatiken löst.*

Beweis: Bevor wir mit dem eigentlichen Beweis beginnen, soll eine wichtige Beobachtung formuliert werden: Anhand der Definition von Typ 1 - Grammatiken sieht man sofort ein, dass für jede Produktion $u \to v$ einer kontextsensitiven Grammatik $|u| \leq |v|$ gilt. Dies bewirkt, dass die „Zwischenschritte“ bei einem Ableitungsprozess nie in ihrer Länge schrumpfen.

Sei $G = (\Sigma, N, P, S)$ eine Typ 1 - Grammatik und $n, m \in \mathbb{N}$, dann definieren wir die Mengen T_m^n wie folgt:

$$T_m^n =_{\text{def}} \{w \in (N \cup \Sigma)^* \mid S \vdash^* w \text{ in } \leq m \text{ Schritten und } |w| \leq n\}$$

Nun definieren wir die Menge von „Erzeugnissen“ der Länge maximal n, die durch einen Ableitungsschritt aus der Menge von Zwischenergebnissen hervorgehen kann:

$$\text{Abl}_n(X) =_{\text{def}} X \cup \{w \in (N \cup \Sigma)^* \mid w' \vdash w \text{ für ein } w' \in X \text{ und } |w| \leq n\}$$

Die Mengen T_i^n können dann induktiv wie folgt berechnet werden (siehe Abbildung 6):

(IA) $T_0^n = \{S\}$ (Startsymbol)

Algorithmus 1: Member - Algorithmus für das Wortproblem von Typ 1 - Sprachen

Data: Typ 1 - Grammatik G und Wort w
Result: `true` falls $w \in L(G)$ und `false` sonst

$T = \{S\}$;
repeat
  $T_{\text{OLD}} = T$;
  $T = \text{Abl}_n(T_{\text{OLD}})$;
until *(*$(w \notin T)$ && $(T \neq T_{\text{OLD}})$*)*;
if *(*$w \in T$*)* **then**
  return `true`;
else
  return `false`;
end

(IS) $T^n_{m+1} = \text{Abl}_n(T^n_m)$

Diese Konstruktion funktioniert bei Typ 0 - Grammatiken, die nicht vom Typ 1 sind, nicht mehr notwendigerweise, da hier „verkürzende“ Produktionen existieren, d.h. es gibt Produktionen mit $u \to v$ und $|u| > |v|$.

Es ist leicht einzusehen, dass es nur endlich viele Worte der Länge $\leq n$ über $(N \cup \Sigma)^*$ gibt und damit muss ein $m \in \mathbb{N}$ existieren mit

$$T^n_m = T^n_{m+1} = T^n_{m+2} = T^n_{m+3} = \ldots \qquad (\text{„Fixpunkt“ / Hülle von } S)$$

Damit ist $w \in L(G), |w| = n$ genau dann, wenn gilt $w \in \bigcup_{m \geq 0} T^n_m$, d.h. das Wortproblem kann gelöst werden, wenn schrittweise die Mengen T^n_m berechnet werden. Somit ergibt sich Algorithmus 1 zur Lösung des Wortproblem von Typ 1 - Grammatiken. #

Bemerkung 21: *Dieser Algorithmus hat ein schlechtes Laufzeitverhalten (exponentielle Laufzeit) und ist damit für praktische Anwendungen nicht brauchbar, zeigt aber auch, dass das Wortproblem für Grammatiken vom Typ 2 und Typ 3 auch algorithmisch lösbar ist.*

Beispiel 22: *Welche Worte der Länge* ≤ 4 *existieren in* $L(G_4)$*? Durch die Anwendung von* $\text{Abl}_n(\cdot)$ *ergibt sich:*

$$\begin{aligned} T^4_0 &= \{S\} \\ T^4_1 &= \{S, aSBc, abc\} \\ T^4_2 &= \{S, aSBc, abc\} = T^4_3 = T^4_4, \end{aligned}$$

d.h. $L(G_4) \cap T^4_2 = \{abc\}$.

Bemerkung 23: *Das Wortproblem für Typ 1 - Grammatiken ist* **NP***-hart und sogar* **PSPACE***-vollständig. Ergebnisse aus der Komplexitätstheorie (siehe Abschnitt 6) sagen, dass wir hier nicht auf einen effizienten Algorithmus hoffen dürfen. Allerdings werden wir noch sehen, dass wesentlich schnellere Algorithmen für das Wortproblem von Grammatiken des Typs 2 bzw. des Typs 3 existieren. Sprachen dieser Typen sind für Programmiersprachen völlig ausreichend, so dass Typ 1 - Grammatiken für praktische Belange eine untergeordnete Rolle spielen.*

3. Endliche Automaten und reguläre Sprachen

In diesem Abschnitt sollen die Sprachen vom Typ 3 näher untersucht werden. Dazu werden (nicht)deterministische endliche Automaten als schnelle Erkenner vorgestellt, die das Wortproblem für Typ 3 - Grammatiken wesentlich schneller lösen können als Algorithmus 1. Eine weitere Möglichkeit reguläre Sprachen darzustellen sind *reguläre Ausdrücke*. Typ 3 - Sprachen, endliche Automaten und reguläre Ausdrücke haben in der Praxis der Informatik eine große Bedeutung und werden in vielen verschiedenen Anwendungen eingesetzt (z.B. bash, (f)lex - Scannergenerator, PHP, Perl, emacs, im VHDL-Design, XML, etc.)

3.1. Deterministische endliche Automaten

Definition 24: *Ein (deterministischer)* endlicher Automat *M (kurz: DEA oder DFA) ist ein* 5*-Tupel $M = (Z, \Sigma, \delta, z_0, E)$, wobei*

- *Z - die endliche Menge der* Zustände *ist,*
- *Σ - das* Alphabet *mit $Z \cap \Sigma = \emptyset$,*
- *δ - die* Überführungsfunktion *mit $\delta\colon Z \times \Sigma \to Z$.*
- *z_0 - der* Startzustand*, wobei $z_0 \in Z$ und*
- *E - die Menge der* Endzustände *mit $E \subseteq Z$.*

Die erweiterte Überführungsfunktion *$\hat{\delta}\colon Z \times \Sigma^* \to Z$ wird induktiv wie folgt definiert:*

(IA) $\hat{\delta}(z, \epsilon) =_{\text{def}} z$
(IS) $\hat{\delta}(z, aw) =_{\text{def}} \hat{\delta}(\delta(z, a), w)$ *für alle $a \in \Sigma$, $w \in \Sigma^*$ und $z \in Z$.*

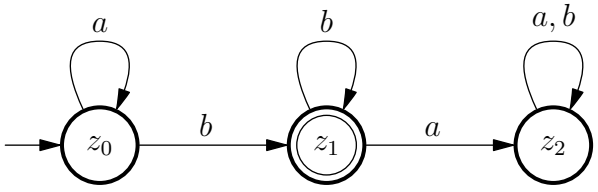
Die von M akzeptierte Sprache *ist dann*

$$L(M) =_{\text{def}} \{w \in \Sigma^* \mid \hat{\delta}(z_0, w) \in E\}$$

Beispiel 25: *Gegeben sei der endliche Automat $M_1 = (\{z_0, z_1, z_2\}, \{a, b\}, \delta, z_0, \{z_1\})$, wobei die totale Funktion δ wie folgt festgelegt wird:*

$$\begin{array}{lll} \delta(z_0, a) = z_0, & \delta(z_1, a) = z_2, & \delta(z_2, a) = z_2, \\ \delta(z_0, b) = z_1, & \delta(z_1, b) = z_1, & \delta(z_2, b) = z_2. \end{array}$$

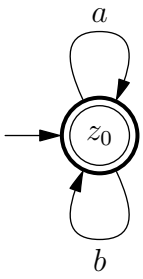
Graphisch kann man diesen Automaten wie folgt darstellen:



Dabei wird der Startzustand durch einen Pfeil markiert und die Endzustände durch doppelte Kreise.

Es gilt $L(M_1) = \{w \in \{a, b\}^* \mid w = a^n b^m, n \geq 0, m \geq 1\}$ und $abb \in L(M_1)$, da $\hat{\delta}(\delta(z_0, a), bb) = \hat{\delta}(z_0, bb) = \hat{\delta}(\delta(z_0, b), b) = \hat{\delta}(z_1, b) = \hat{\delta}(\delta(z_1, b), \epsilon) = \hat{\delta}(z_1, \epsilon) = z_1$, d.h. die Eingabe abb wird von M_1 akzeptiert.

Beispiel 26: *Gegeben sei der endliche Automat* $M_2 = (\{z_0\}, \{a, b\}, \delta, z_0, \{z_0\})$*, wobei die totale Funktion* δ *durch* $\delta(z_0, a) = z_0$ *und* $\delta(z_0, b) = z_0$ *festgelegt ist. Damit gilt* $L(M_2) = \{a, b\}^*$ *und graphisch kann man den Automaten* M_2 *wie folgt darstellen:*



3.2. Nichtdeterministische endliche Automaten

Bei einem DEA ist der Übergang für einen Zustand bei gegebenen Eingabezeichen eindeutig festgelegt, d.h. es gibt genau eine Folge von Zuständen bei der Abarbeitung einer Eingabe (siehe Abbildung 7(a)). Dies soll nun verallgemeinert werden. Dazu wollen wir die nichtdeterministischen endlichen Automaten einführen, die sich zu einem Zeitpunkt gleichzeitig in mehreren Zuständen befinden können.

Definition 27: *Ein* nichtdeterministischer endlicher Automat *M (kurz: NEA oder NFA) ist ein* 5*-Tupel* $M = (Z, \Sigma, \delta, z_0, E)$*, wobei*

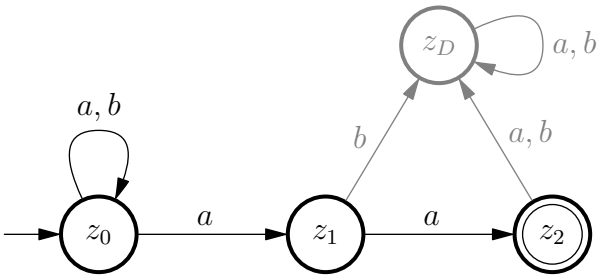
- Z *- die endliche Menge der* Zustände *ist,*
- Σ *-* das Alphabet *mit* $Z \cap \Sigma = \emptyset$*,*
- δ *- die* Überführungsfunktion *mit* $\delta\colon Z \times \Sigma \to \mathcal{P}(Z)$*.*
- z_0 *- der* Startzustand*, wobei* $z_0 \in Z$ *und*
- E *- die Menge der* Endzustände *mit* $E \subseteq Z$*.*

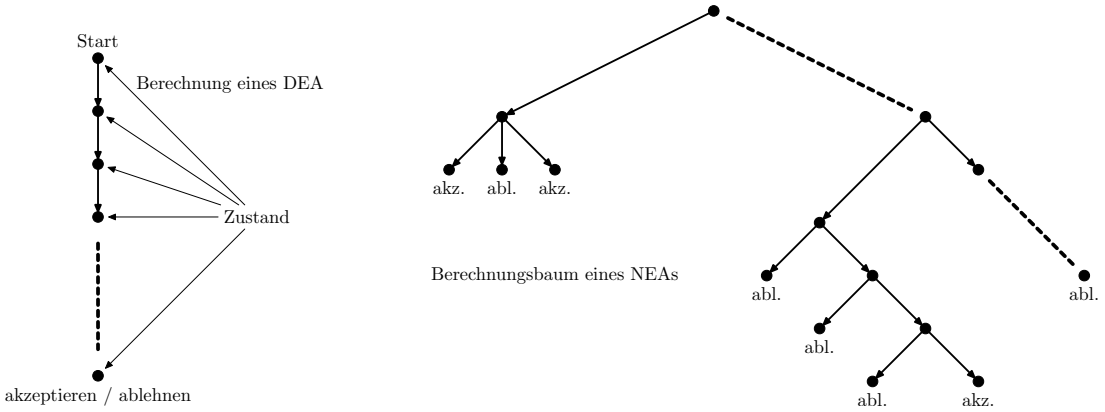
Dabei bezeichnet $\mathcal{P}(Z)$ *die* Potenzmenge *von* Z *(siehe Abschnitt A.1.3). Analog zu den DFAs definieren wir die* erweiterte Überführungsfunktion $\hat{\delta}\colon \mathcal{P}(Z) \times \Sigma^* \to \mathcal{P}(Z)$ *induktiv wie folgt:*
(IA) $\hat{\delta}(Z', \epsilon) =_{\text{def}} Z'$ *für alle* $Z' \in \mathcal{P}(Z)$
(IS) $\hat{\delta}(Z', ax) =_{\text{def}} \bigcup_{z \in Z'} \hat{\delta}(\delta(z, a), x)$ *für* $a \in \Sigma$ *und* $x \in \Sigma^*$*.*
Die von M akzeptierte Sprache *ist dann*

$$L(M) =_{\text{def}} \{w \in \Sigma^* \mid \hat{\delta}(\{z_0\}, w) \cap E \neq \emptyset\}$$

Mit Definition 27 ist es nicht schwer einzusehen, dass die NEAs eine direkte Verallgemeinerung des Konzepts der deterministischen endlichen Automaten darstellen.

Beispiel 28: *Sei* $M_3 = (\{z_0, z_1, z_2\}, \{a, b\}, \delta, z_0, \{z_2\})$*, wobei die Überführungsfunktion* δ *implizit durch die folgende graphische Darstellung gegeben ist:*





(a) Deterministische Berechnung (b) Nichtdeterministische Berechnung

Abbildung 7: Ein Beispiel für mögliche Berechnungspfade eines DEAs bzw. NEAs.

Damit ergibt sich $L(M_3) = \{w \in \{a,b\}^* \mid w \text{ endet mit } aa\}$. Da die Übergangsfunktion δ nicht total ist, können wir z.B. einen neuen „Dummy"-Zustand z_D (grau) einführen, der bewirkt, dass die Übergangsfunktion zum einen total wird und sich zum anderen die akzeptierte Sprache nicht verändert. Dazu werden für alle fehlenden Übergänge Kanten zum Zustand z_D neu eingeführt. Im Zustand z_D selbst, werden dann alle Eingaben immer verworfen.

Da die DEAs einen Spezialfall der NEAs darstellen, gilt offensichtlich:

Beobachtung 29: *Jede Sprache die durch einen DEA akzeptiert wird, wird auch durch einen geeigneten NEA akzeptiert.*

Beobachtung 30: *Anschaulich arbeitet ein NEA wie folgt:*

i) Lese ein Zeichen der Eingabe.

ii) Für jede Möglichkeit der Übergangsfunktion wird eine neue Kopie des NEA erzeugt und jede dieser Alternativen wird parallel verfolgt.

iii) Gibt es einen Pfad, der zu einem Zustand $z \in E$ führt, dann akzeptiert der NEA.

Arbeitet ein NEA eine Eingabe ab, so kann er sich zu einem Zeitpunkt gleichzeitig in mehreren Zuständen befinden. Nichtdeterminismus[5] kann also als eine Art paralleler Berechnung betrachtet werden, bei der mehrere „Prozesse" oder „Threads" parallel rechnen (siehe Abbildung 7). Für den NEA M_3 ergibt sich für die Eingabe babaa der Berechnungsbaum *in Abbildung 8.*

Wir wollen nun zeigen, dass jede Sprache die durch einen NEA akzeptiert wird, durch einen geeigneten DEA erkannt wird. Dazu geben wir an, wie man aus einem gegebenen NEA einen äquivalenten DEA konstruiert. Diese Technik ist als *Potenzmengenkonstruktion* (oder auch *Teilmengenkonstruktion*) bekannt. Die Hauptidee dabei ist es, die Menge von Zuständen in denen sich ein NEA zu einem Zeitpunkt aufhalten kann, durch einen einzigen Zustand eines DEAs zu ersetzen.

[5] Das Konzept der „nichtdeterministischen Berechnung" kann auch auf andere Maschinentypen übertragen und studiert werden. Ein schönes Beispiel sind „Turingmaschinen". Hier führt die Definition direkt zum Studium von Problemen aus der Klasse **NP** (siehe Abbildung 20 auf Seite 52).

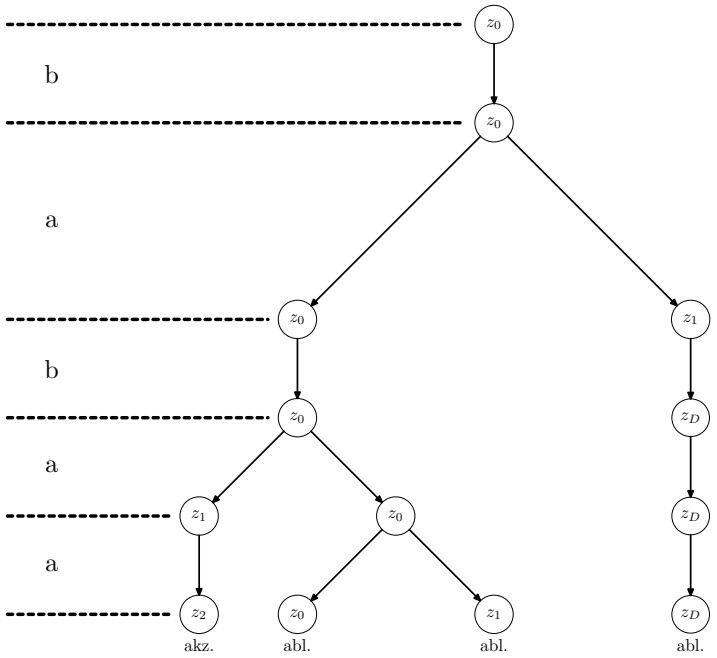


Abbildung 8: Berechnungsbaum von M_3 bei der Eingabe *babaa*

Satz 31 („Potenzmengenkonstruktion“): *Jede von einem NEA akzeptierte Sprache ist auch durch einen DEA akzeptierbar.*

Beweis: Die Sprache $L \subseteq \Sigma^*$ wird durch den NEA $M = (Z, \Sigma, \delta, z_0, E)$ akzeptiert, d.h. $L = L(M)$. Gesucht ist ein DEA M' mit $L(M') = L = L(M)$.

Idee: Ein Zustand von M' entspricht einer Menge von Zuständen von M, d.h. die Zustände von M' entsprechen Elementen aus $\mathcal{P}(Z)$, wobei $\mathcal{P}(Z)$ die *Potenzmenge* von Z (siehe Abschnitt A.1.3) ist.

Sei $M' = (Z', \Sigma, \delta', z_0', E')$ ein deterministischer endlicher Automat mit

- $Z' =_{\text{def}} \mathcal{P}(Z)$
- $z_0' =_{\text{def}} \{z_0\}$
- $E' =_{\text{def}} \{F \subseteq Z \mid F \cap E \neq \emptyset\} = \{F \in \mathcal{P}(Z) \mid F \cap E \neq \emptyset\}$
- $\delta'(F, a) =_{\text{def}} \underbrace{\bigcup_{z \in F} \delta(z, a)}_{\text{Zustand von } M'} = \hat{\delta}(F, a)$, wobei $F \in Z'$

Sei nun $w = a_1 \dots a_n \in \Sigma^*$, dann gilt:

$w \in L(M)$	gdw.	$\hat{\delta}(\{z_0\}, w) \cap E \neq \emptyset$
	gdw.	es gibt eine Folge von Teilmengen $Z_1, \dots, Z_n$ von Z mit $\delta'(\{z_0\}, a_1) = \hat{\delta}(\{z_0\}, a_1) = Z_1, \delta'(Z_1, a_2) = Z_2, \dots, \delta'(Z_{n-1}, a_n) = Z_n$ und $Z_n \cap E \neq \emptyset$
	gdw.	$\hat{\delta}'(\{z_0\}, w) \in E'$
	gdw.	$w \in L(M')$.

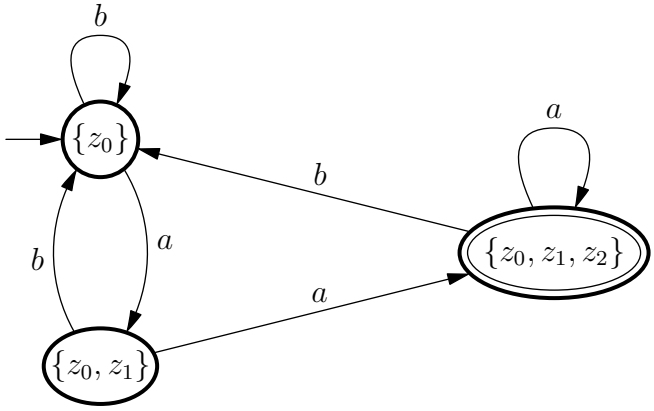
Damit haben wir gezeigt, dass der deterministische Automat M' die gleiche Sprache wie M akzeptiert. #

Dieser Beweis zeigt sehr anschaulich, wie ein Algorithmus konstruiert werden kann, der einen beliebigen NEA in einen DEA umwandelt. Varianten dieses Algorithmus kommen in der Praxis z.B. bei Perl oder PHP zum Einsatz.

Beispiel 32: *Bestimmen Sie einen DEA M_3' mit $L(M_3') = L(M_3)$. Dazu legen wir fest: $M_3' = (\mathcal{P}(\{z_0, z_1, z_2\}), \{a, b\}, \delta', \{z_0\}, \{\{z_0, z_1, z_2\}\})$ und bestimmen mit der Potenzmengenkonstruktion[6].*

$$\begin{array}{ll} \delta'(\{z_0\}, a) = \{z_0, z_1\} & \delta'(\{z_0\}, b) = \{z_0\} \\ \delta'(\{z_0, z_1\}, a) = \{z_0, z_1, z_2\} & \delta'(\{z_0, z_1\}, b) = \{z_0\} \\ \delta'(\{z_0, z_1, z_2\}, a) = \{z_0, z_1, z_2\} & \delta'(\{z_0, z_1, z_2\}, b) = \{z_0\}. \end{array}$$

Dies führt zu der folgenden graphischen Darstellung von M_3':

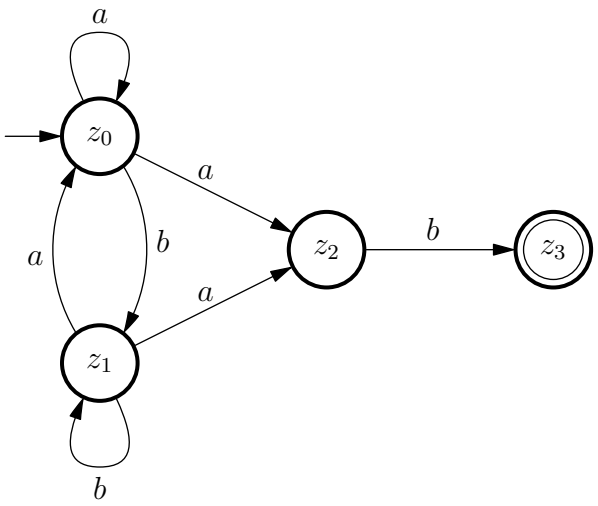


Dieser neue Automat akzeptiert ebenfalls die Sprache $\{w \in \{a, b\}^* \mid w$ endet mit $aa\}$ und damit gilt $L(M_3') = L(M_3)$.

Bemerkung 33: *Man kann auch alle möglichen Zustände des zu konstruierenden DEAs aufschreiben und für jeden Zustand die Übergangsfunktion ermitteln. Bei dieser Methode können sich aber auch nicht erreichbare Zustände ergeben, die dann noch entfernt werden sollten.*

Beispiel 34: *Sei $M_4 = (\{z_0, z_1, z_2, z_3\}, \{a, b\}, \delta, z_0, \{z_3\})$ durch die folgende graphische Repräsentation gegeben:*

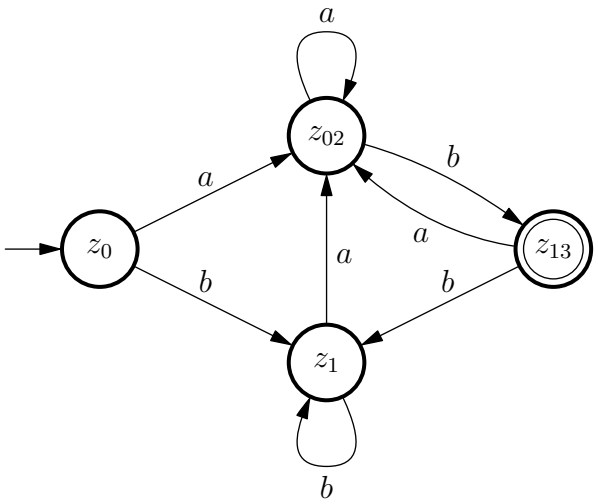
[6] Eigentlich sind in der Menge der Zustände nach Definition noch mehr Elemente enthalten. Wie die Konstruktion von M_3' zeigt, werden diese nie erreicht, weshalb sie entfernt wurden.



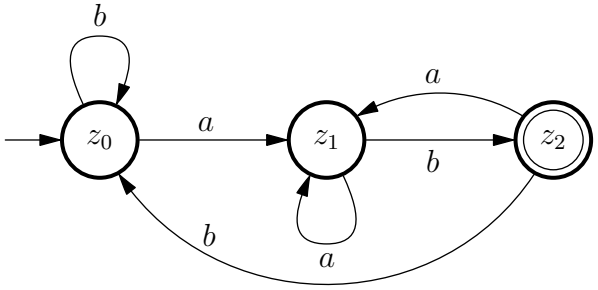
Es gilt $L(M_4) = \{w \in \{a,b\} \mid w \text{ endet mit } ab\}$*. Bestimmen Sie mit der Potenzmengenkonstruktion einen DEA* M_4' *mit* $L(M_4') = L(M_4)$*. Mit Satz 31 ergibt sich also* $M_4' = (\mathcal{P}(\{z_0, z_1, z_2, z_3\}), \{a,b\}, \delta', \{\{z_1, z_3\}\})$ *und*

$$\begin{array}{lll} \delta'(\{z_0\}, a) = \{z_0, z_2\}, & \delta'(\{z_0\}, b) = \{z_1\}, & \delta'(\{z_0, z_2\}, a) = \{z_0, z_2\}, \\ \delta'(\{z_0, z_2\}, b) = \{z_1, z_3\}, & \delta'(\{z_1\}, a) = \{z_0, z_2\}, & \delta'(\{z_1\}, b) = \{z_1\}, \\ \delta'(\{z_1, z_3\}, a) = \{z_0, z_2\}, & \delta'(\{z_1, z_3\}, b) = \{z_1\}. & \end{array}$$

Wir kürzen die Menge $\{z_0\}$ *mit* z_0*,* $\{z_0, z_2\}$ *mit* z_{02}*,* $\{z_1\}$ *mit* z_1 *und* $\{z_1, z_3\}$ *mit* z_{13} *ab. Nun ergibt sich die folgende graphische Darstellung des Automaten* M_4'*:*



Allerdings gibt es einen äquivalenten DEA mit weniger Zuständen:



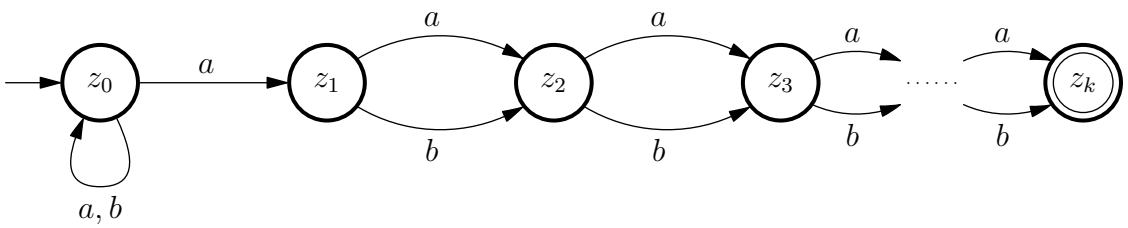
Dies zeigt, dass die Potenzmengenkonstruktion nicht unbedingt den DEA mit den wenigsten Zuständen (≙ Minimalautomat) liefert.

Nun stellt sich natürlich sofort die Frage, wie viele Zustände ein DFA haben kann, der bei der Potenzmengenkonstruktion aus einem NEA entsteht. Der nächste Satz wird zeigen, dass ein äquivalenter DEA exponentiell größer sein kann als ein gegebener NEA.

Satz 35: *Sei* $L_k =_{\text{def}} \{w \in \{a,b\}^* \mid |w| \geq k$ *und das* k*-letzte Zeichen ist* $a\}$*,* $k \geq 1$*. Die Sprache* L_k *kann durch einen NEA mit* $k+1$ *Zuständen akzeptiert werden, aber es gibt keinen DEA mit weniger als* 2^k *Zuständen, der* L_k *akzeptiert.*

Dieser Satz zeigt, das die Umwandlung eines NEAs in einen DEA sehr aufwändig sein kann, d.h. für einen gegebenen NEA mit n Zuständen müssen $O(2^n)$ viele Zustände ausgegeben werden. In einem solchen Fall rechnet die Potenzmengenkonstruktion mindestens $O(2^n)$ Schritte. Damit stellt sich natürlich sofort die Frage, ob dieser Algorithmus für die Praxis überhaupt geeignet ist. Dies ist aber der Fall, denn es zeigt sich in der praktischen Benutzung, dass dieser worst-case (fast) nie auftritt und damit (oft) vernachlässigt werden kann.

Beweis: Der folgende NEA hat das gewünschte Verhalten:



Behauptung: Es gibt keinen DEA mit weniger als 2^k Zuständen, der L_k akzeptiert.

Annahme: Es gibt einen DEA $M = (Z, \{a,b\}, \delta, z_0, E)$ mit weniger als 2^k Zuständen, dann müssen zwei verschiedene Worte $w, w' \in \{a,b\}^k$ existieren, so dass $\hat{\delta}(z_0, w) = \hat{\delta}(z_0, w')$ gilt. Dies ist mit Hilfe des Schubfachprinzips (siehe Abschnitt C.4) einsichtig, weil es 2^k Wörter w über $\{a,b\}^k$ gibt, aber nur $\#\{z \mid \hat{\delta}(z_0, w) = z \text{ und } w \in \{a,b\}^k\} < 2^k$ verschiedene Zustände.

Sei $y \in \{a,b\}^{i-1}$ beliebig und sei i die erste Position an der sich w und w' unterscheiden, wobei $1 \leq i \leq k$. Dann gilt[7] o.B.d.A. $wy = \underbrace{uav}_{w} y$ und $w'y = \underbrace{ubv'}_{w'} y$, wobei $|v| = |v'| = k - i$ und $|u| = i - 1$. Das a (bzw. b) in wy (bzw. $w'y$) sitzt an der k-letzten Stelle, d.h. $wy \in L_k$ aber $w'y \notin L_k$. Weiterhin gilt:

$$\begin{aligned} \hat{\delta}(z_0, wy) &= \hat{\delta}(\hat{\delta}(z_0, w), y) \\ &= \hat{\delta}(\hat{\delta}(z_0, w'), y) \\ &= \hat{\delta}(z_0, w'y) \end{aligned}$$

Damit ist auch $w'y \in L_k$, was ein Widerspruch zu der Beobachtung $w'y \notin L_k$ ist. Also muss unsere ursprüngliche Annahme falsch sein, d.h. es gibt keinen DEA mit weniger als 2^k Zuständen der L_k akzeptiert. #

Bemerkung 36: *Anschaulich gesprochen bedeutet dies, dass sich endliche Automaten nicht lange „erinnern" können, d.h. Informationen können nur in den Zuständen gemerkt werden. Im obigen Beispiel hatte der Automat z.B. alle Informationen über das* k*-letzte Zeichen vergessen und konnte deshalb die Fälle* wy *bzw.* $w'y$ *nicht mehr unterscheiden. Diese Vergesslichkeit wird sich auch im Pumping Lemma für reguläre Sprachen wieder zeigen.*

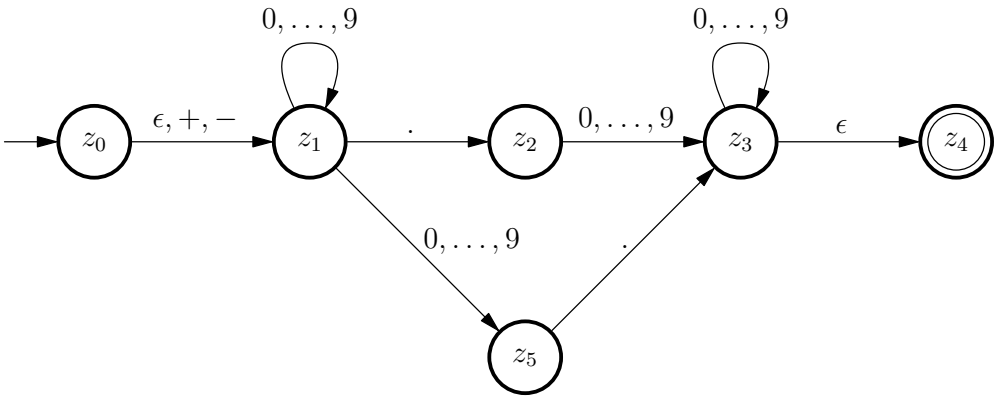
[7] o.B.d.A. ist die Abkürzung für „ohne Beschränkung der Allgemeinheit"

3.2.1. Endliche Automaten mit ϵ-Übergängen

Für praktische Zwecke sind NEAs mit so genannten ϵ-Übergängen ($\triangleq$ spontaner Zustandsübergang) oft sehr nützlich, denn mit diesen Automaten lassen sich manche reguläre Sprachen leichter beschreiben (vgl. Behandlung des Vorzeichens im nächsten Automaten).

Anschaulich gesprochen sind ϵ-Übergänge Zustandsübergänge, die nicht mit einem Buchstaben $a \in \Sigma$ beschriftet sind, sondern mit dem leeren Wort ϵ markiert wurden. Solche Übergänge konsumieren also keinen Buchstaben der Eingabe, was auch den Begriff des *spontanen Zustandsübergangs* erklärt, denn ein solcher Automat kann einen ϵ-Übergang „kostenlos“ durchlaufen. Damit sind solche Automaten aber sicherlich als nichtdeterministisch anzusehen, denn sie können sich für eine gelesene Eingabe gleichzeitig in mehreren Zuständen aufhalten.

Beispiel 37: *Der folgende NEA mit ϵ-Übergängen akzeptiert Dezimalzahlen, wobei auch Abkürzungen wie „.4“ und „3.“ zulässige Eingaben sind:*



Definition 38 (NEA mit ϵ-Übergängen): *Ein NEA mit ϵ-Übergängen entspricht einen normalen NEA, wobei lediglich die Übergangsfunktion δ wie folgt erweitert wird:*

$$\delta \colon Z \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Z)$$

NEAs mit ϵ-Übergängen verhalten sich ähnlich wie einfache NEAs. Insbesondere funktioniert auch die Potenzmengenkonstruktion:

Satz 39 („Potenzmengenkonstruktion für NEAs mit ϵ-Übergängen“): *Jede von einem NEA mit ϵ-Übergängen akzeptierte Sprache ist auch durch einen DEA akzeptierbar.*

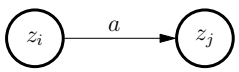
Beweis: Übung #

3.3. Reguläre Grammatiken und endliche Automaten

In diesem Abschnitt wollen wir nun zeigen, dass die Menge der regulären Sprachen und die Menge der Sprachen die durch endliche Automaten akzeptiert werden, gleich sind.

Satz 40: *Jede durch einen endlichen Automaten akzeptierbare Sprache ist vom Typ 3.*

Die Kernidee hier ist, jedem Zustandsübergang



eine Produktion $A \to aA'$ zuzuordnen.

Beweis: Werde L durch den DEA $M = (Z, \Sigma, \delta, z_0, E)$ akzeptiert. Nach obiger Idee wird aus einem Zustandsübergang von z nach $\delta(z,a)$ die Produktion $z \to a\delta(z,a)$. Genauer ergibt sich die Grammatik $G = (\Sigma, Z, P, z_0)$ und

$$P =_{\text{def}} \{z \to a\delta(z,a) \mid z \in Z, a \in \Sigma\} \cup \{z \to a \mid \delta(z,a) \in E, z \in Z \text{ und } a \in \Sigma\}$$

Damit ergibt sich

$a_1 \dots a_n \in L(M)$	gdw.	es gibt eine Folge von Zuständen $z_1, \dots, z_n \in Z$ und $z_n \in E$, so dass $\delta(z_0, a_1) = z_1, \delta(z_1, a_2) = z_2, \dots \delta(z_{n-1}, a_n) = z_n$
	gdw.	es eine Folge von Nichtterminalen $z_0, \dots, z_{n-1}$ der Grammatik G gibt mit $z_0 \to a_1 z_1, z_1 \to a_2 z_2, \dots, z_{n-1} \to a_n$, $z_0 \vdash a_1 z_1 \vdash a_1 a_2 z_2 \vdash^* a_1 \dots a_n$ und damit $a_1 \dots a_n \in L(G)$

Damit gilt $L(G) = L \backslash \{\epsilon\}$ und deshalb ist die Sprache L vom Typ 3. #

Nachdem wir gezeigt haben, dass jede Sprache, die durch einen endlichen Automaten akzeptiert werden kann, vom Typ 3 ist, wollen wir auch die umgekehrte Richtung untersuchen. Dazu zeigen wir dass wir aus einer Typ 3 - Grammatik einen äquivalenten NEA konstruieren können.

Satz 41: *Für jede reguläre Sprache L, die von der regulären Grammatik G erzeugt wird, existiert ein NEA M mit $L = L(M)$.*

Beweis: Sei die reguläre Grammatik $G = (\Sigma, N, P, S)$ gegeben. Der gewünschte Automat $M = (Z, \Sigma, \delta, S, E)$ ist wie folgt festgelegt:

$$Z = N \cup \{X\}, \text{ wobei } \{X\} \cap N = \emptyset$$

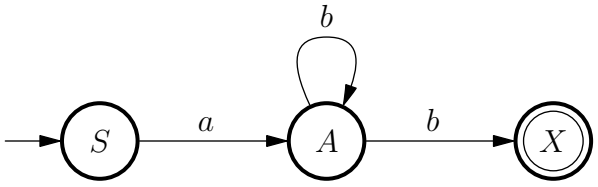
$$E = \begin{cases} \{S, X\}, \text{ falls } \epsilon \in L \\ \{X\}, \text{ sonst} \end{cases}$$

Weiterhin ist $B \in \delta(A,a)$, falls $A \to aB \in P$ und $X \in \delta(A,a)$, falls $A \to a \in P$. Für $n \geq 1$ gilt

$a_1 \dots a_n \in L(G)$	gdw.	es gibt eine Folge von Nichtterminalen $A_1, \dots, A_{n-1} \in N$ und $S \vdash a_1 A_1 \vdash a_1 a_2 A_2 \vdash \dots \vdash a_1 \dots a_{n-1} A_{n-1} \vdash a_1 \dots a_n$
	gdw.	es eine Folge von Zuständen $A_1, \dots, A_{n-1}$ des Automaten M gibt mit $A_1 \in \delta(S, a_1), A_2 \in \delta(A_1, a_2), \dots, X \in \delta(A_{n-1}, a_n)$
	gdw.	$a_1 \dots a_n \in L(M)$

Deshalb folgt $L(G) = L(M)$. #

Beispiel 42: *Sei die Typ 3 Grammatik $G = (\{a,b\}, \{S, A\}, P, S)$ mit $P = \{S \to aA, A \to bA, A \to b\}$ gegeben, dann ergibt sich der folgende Automat:*



Kombinieren wir bisher erzielten Ergebnisse, so ergibt die Folgerung:

Folgerung 43: *Die regulären Sprachen sind genau die Sprachen, die durch einen DEA oder einen NEA akzeptiert werden, und es ergibt sich das in Abb. 9 gezeigte Bild.*

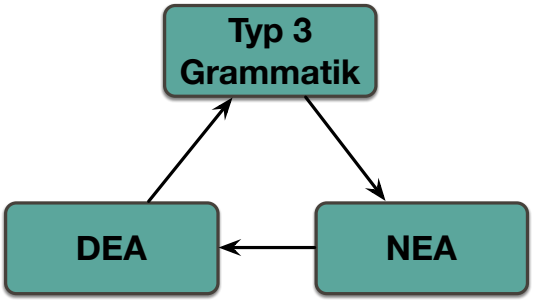


Abbildung 9: Endliche Automaten und reguläre Sprachen

3.4. Reguläre Ausdrücke

Reguläre Ausdrücke sind ein spezieller Formalismus mit dem wir Sprachen definieren können. In diesem Abschnitt sehen wir, dass die so definierbaren Sprachen genau mit den Typ 3 - Sprachen übereinstimmen. In der Praxis werden reguläre Sprachen in einer Vielzahl von Werkzeugen angewendet, wie z.B. perl, PHP, grep oder awk[8]. Auf moderne Skriptsprachen wie Python oder Ruby und die funktionale Sprache Scala unterstützen solche regulären Ausdrücke.

Reguläre Ausdrücke werden induktiv wie folgt definiert:

Definition 44 (Syntax von regulären Ausdrücken): *Sei Σ ein beliebiges Alphabet, dann gilt:*
(IA) *$\emptyset$, ϵ und alle $a \in \Sigma$ sind reguläre Ausdrücke*
(IS) *Wenn α und β reguläre Ausdrücke sind, dann auch $\alpha\beta$, $(\alpha \mid \beta)$ und $(\alpha)^*$.*

Der $*$-Operator ist auch als *Kleene*-Stern bekannt und wurde nach dem Mathematiker Stephen Kleene[9] benannt. Bis jetzt haben die Ausdrücke aus Definition 44 noch keinerlei Bedeutung. Die Bedeutung soll mit Hilfe der nächsten Definition festgelegt werden, indem wir für jeden regulären Ausdruck α die von ihm festgelegte Sprache $L(\alpha)$ ($\triangleq$ Bedeutung) induktiv definieren:

Definition 45 (Semantik von regulären Ausdrücken):
(IA) $L(\emptyset) =_{\text{def}} \emptyset$, $L(\epsilon) =_{\text{def}} \{\epsilon\}$ *und* $L(a) =_{\text{def}} \{a\}$
(IS)

- *Sei $\gamma = \alpha\beta$, dann ist $L(\gamma) =_{\text{def}} L(\alpha) \cdot L(\beta)$ („Konkatenation[10]“.)*
- *Sei $\gamma = (\alpha \mid \beta)$, dann ist $L(\gamma) =_{\text{def}} L(\alpha) \cup L(\beta)$ („Vereinigung“)*
- *Sei $\gamma = (\alpha)^*$, dann ist $L(\gamma) =_{\text{def}} L(\alpha)^* = \{\epsilon\} \cup L(\alpha) \cup L(\alpha) \cdot L(\alpha) \cup L(\alpha) \cdot L(\alpha) \cdot L(\alpha) \cup \ldots$ („alle Worte, die sich durch (mehrfache) Konkatenation von Worten aus $L(\alpha)$ bilden lassen“)*

Beispiel 46: *Einige kleine Beispiele für Sprachen die mit Hilfe von regulären Ausdrücken definiert werden:*

Durch $\gamma = (a \mid (a \mid b)^*aa)$ wird die Sprache $L(\gamma) = \{w \in \{a,b\}^* \mid w \text{ endet mit } aa\} \cup \{a\}$ definiert.

[8] siehe z.B. http://www.pcre.org/
[9] *1909 in Hartford (USA) - †1994 in Madison (USA)
[10] Für die genaue Bedeutung der Konkatenation von zwei Sprachen siehe Abschnitt A.1.4

$L(a^*bc^*) = \{w \in \{a,b,c\}^* \mid w = a^n bc^m, m,n \geq 0\}$

Bemerkung 47: *Oft wird auch der reguläre Ausdruck α^+ verwendet. Wir legen fest, dass α^+ eine Abkürzung für den Ausdruck $\alpha\alpha^*$ ist. Weiterhin sei $\alpha?$ die Abkürzung von $(\emptyset|\alpha)$. In der Praxis werden noch andere Operatoren benutzt, die man aber auch mit Hilfe der drei regulären Grundoperationen ausdrücken kann. Deshalb macht es Sinn, diese nicht in die grundlegende Definition der regulären Ausdrücke aufzunehmen und so Beweise zu vereinfachen.*

Satz 48 (Kleene): *Die Menge der Sprachen, die durch reguläre Ausdrücke beschrieben werden können, entspricht genau der Menge der Typ 3 Sprachen.*

Beweisskizze: Es sind zwei Richtungen zu zeigen:

„⇒“: Für jeden beliebigen regulären Ausdruck R gibt es einen NEA mit ϵ-Übergängen M mit $L(R) = L(M)$.

„⇐“: Für jede Typ 3 Grammatik G gibt es einen regulären Ausdruck R mit $L(G) = L(R)$.

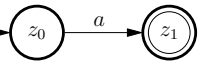
In der Praxis ist es besonders wichtig aus einem regulären Ausdruck einen endlichen Automaten konstruieren zu können, um z.B. feststellen zu können, ob ein gegebenes Wort zu einem gegebenen regulären Ausdruck passt. Aus diesem Grund zeigen wir zuerst diese Richtung.

„⇒“: Die zentrale Idee hier ist es, einen Induktionsbeweis über die Struktur der regulären Ausdrücke zu führen. Durch die induktive Definition von regulären Ausdrücken ist dies in diesem Fall relativ einfach. Sei γ ein regulärer Ausdruck, dann gilt

(IA)

- Der Ausdruck $\gamma = \emptyset$ wird zu 

- Der Ausdruck $\gamma = \epsilon$ wird zu 

- Der Ausdruck $\gamma = a$ wird zu 

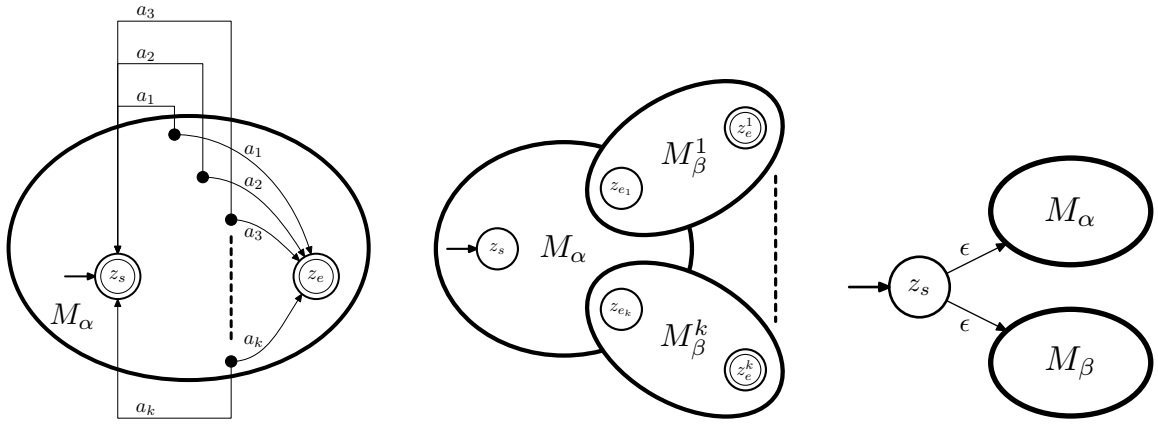
(IV) Sei α ein regulärer Ausdruck, dann existiert ein NFA M_α mit $L(\alpha) = L(M_\alpha)$.

(IS) Entsprechend dem induktiven Aufbau der regulären Ausdrücke müssen drei Fälle unterschieden werden.

Fall $\gamma = \alpha \cdot \beta$: Nach Induktionsvoraussetzung existiert ein Automaten M_α mit k Endzuständen und ein Automat M_β, wobei $L(\alpha) = L(M_\alpha)$ bzw. $L(\beta) = L(M_\beta)$. Nun werden die k Endzustände $z_{e_1}, \ldots, z_{e_k}$ von M_α mit den Startzuständen von Kopien $M_\beta^1, \ldots, M_\beta^k$ von M_β zu einem neuen Zustand „verklebt“. Der dabei entstehende NEA mit ϵ-Übergängen M_γ hat die Eigenschaft $L(M_\gamma) = L(\gamma)$ (siehe Abbildung 10(b)). („Serienschaltung von Automaten“)

Fall $\gamma = \alpha \mid \beta$: Nach Induktionsvoraussetzung existieren Automaten M_α und M_β mit $L(\alpha) = L(M_\alpha)$ bzw. $L(\beta) = L(M_\beta)$. Wir führen einen neuen Zustand ein, der direkt zu den Startzuständen von M_α bzw. M_β verzweigt. Dieser Automat sei M_γ (siehe Abbildung 10(c)). („Parallelschaltung von Automaten“)

Fall $\gamma = (\alpha)^*$: Nach Induktionsvoraussetzung existiert ein Automat M_α, der die Sprache $L(\alpha)$ akzeptiert. Wir konstruieren aus M_α einen neuen Automat, indem wir den Startzustand in die Menge der Endzustände aufnehmen, denn $\epsilon \in L((\alpha)^*)$ und führen „Rückwärtskanten“ ein, die von den Endzuständen zu den Startzuständen führen. Der dabei entstehende Automat ist M_γ (siehe Abbildung 10(a)).



(a) Automat $\gamma = (\alpha)^*$ (b) Automat für $\gamma = \alpha \cdot \beta$ (c) Automat für $\gamma = \alpha \mid \beta$

Abbildung 10: Automaten bei der Umwandlung von regulären Ausdrücken

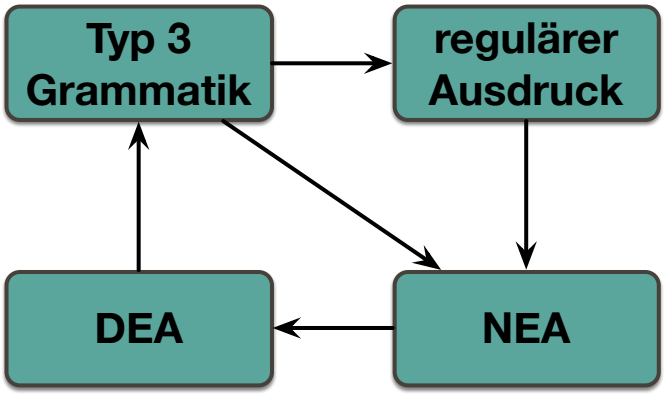


Abbildung 11: Endliche Automaten, reguläre Sprachen und reguläre Ausdrücke

In der Praxis würde man eine rekursive Funktion entwickeln, die nach obigen Schema einen regulären Ausdruck in einen NFA umwandelt. Dabei sind die Fälle aus dem Induktionsanfang der Rekursionsabbruch.

„⇐“: (sehr grobe Idee) Sei $G = (\Sigma, N, P, S)$ vom Typ 3. Nun konstruieren wir einen regulären Ausdruck R mit $L(G) = L(R)$ wie folgt:

Sei $A \in N$ und $A \to a_1 B_1, A \to a_2 B_2, \ldots, A \to a_m B_m, A \to b_1, \ldots, A \to b_k$ alle Regeln in P mit A auf der linken Seite. Wir definieren $L_A =_{\text{def}} \{w \in \Sigma^* \mid A \vdash^* w\} = a_1 L_{B_1} \cup a_2 L_{B_2} \cup \cdots \cup a_m L_m \cup \{b_1, \ldots, b_k\}$. Solch einen Ausdruck kann man in einen regulären Ausdruck umwandeln, wenn man die Ausdrücke für $L_{B_1}, \ldots, L_{B_m}$ hat. D.h. das Ziel ist es, ein Gleichungssystem für alle Nichtterminale von G aufzustellen und dann dieses Gleichungssystem in einen regulären Ausdruck umzuwandeln (siehe z.B. [Sch01, Abschnitt 1.2.3, Seite 38]). #

Damit haben wir gezeigt, dass die Menge der durch reguläre Ausdrücke ausdrückbaren Sprachen genau mit den Typ 3 Sprachen übereinstimmt. Ein Überblick gibt Abb. 11.

3.5. Das Pumping Lemma

Bis jetzt können wir von einer Sprache L nur zeigen, dass sie regulär ist. Dazu müssen wir einen endlichen Automaten M (bzw. einen regulären Ausdruck oder eine Typ 3-Grammatik) finden, so dass $L(M) = L$ gilt.

Nun stellt sich die Frage, ob die Sprache $L_K = \{w \in \{a,b\}^* \mid w = a^m b^m, m \geq 1\}$ regulär ist. Offensichtlich ist diese Sprache vom Typ 2 (kontextfrei), denn für $G = (\{a,b\}, \{S\}, P, S)$ mit $P = \{S \to ab, S \to aSb\}$ gilt $L(G) = L_K$. Wir werden nun zeigen, dass keine Typ 3 Grammatik G' mit $L(G') = L_K$ existiert, d.h. L_K ist nicht regulär. Um dies zu erreichen, benötigen wir das so genannte *Pumping-Lemma*, das auch als *uvw-Theorem* bekannt ist.

Satz 49 (Pumping-Lemma / uvw-Theorem): *Sei L eine reguläre Sprache, dann gibt es eine Zahl $n \in \mathbb{N}$ so, dass sich alle Wörter $x \in L$ mit $|x| \geq n$ in $x = uvw$ zerlegen lassen, wobei folgende Eigenschaften erfüllt sind:*

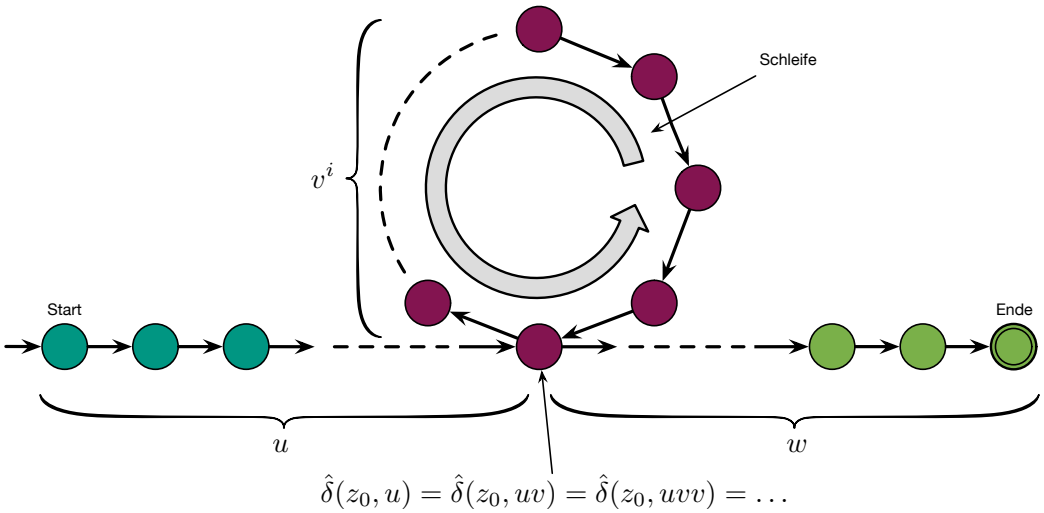
- $|v| \geq 1$
- $|uv| \leq n$
- *Für alle $i \in \mathbb{N}$ gilt: $uv^i w \in L$*

Die Zahl $n \in \mathbb{N}$ aus Satz 49 wird auch *Pumpingkonstante* genannt.

Bemerkung 50: *Die Aussage des Pumping-Lemmas ist* keine *„genau-dann-wenn“ Beziehung, sondern eine Aussage der Form „Wenn L regulär, dann $\exists n \in \mathbb{N}$: $\forall uvw = x \in L, |x| \geq n, |v| \geq 1, |uv| \leq n$ gilt $\forall i \in \mathbb{N}$ $uv^i w \in L$“. Die umgekehrte Richtung muss nicht gelten.*

Beweis: Da L regulär ist, gibt es einen DEA $M = (Z, \Sigma, \delta, z_0, E)$ mit $L = L(M)$. Wir definieren $n =_{\text{def}} \#Z$, d.h. n ist die Anzahl der Zustände von M.

Sei nun x ein beliebiges von M akzeptiertes Wort mit $|x| \geq n$. Beim Abarbeiten von x besucht M genau $|x| + 1$ Zustände (Startzustand mitzählen!). Da $|x| \geq n$ ist, können diese Zustände nicht alle verschieden sein, d.h. mindestens ein Zustand wird zweimal besucht (Schubfachschluss, siehe Abschnitt C.4). Es ergibt sich das folgende Bild:



Die Zerlegung von $x = uvw$ wird nun so gewählt, dass der Zustand nach dem Lesen von u und uv gleich ist, d.h. $\hat{\delta}(z_0, u) = \hat{\delta}(z_0, uv)$. Dabei gilt, dass

- $|v| \geq 1$ („Schleife nicht leer“)
- $|uv| \leq n$ („Die Schleife muss spätestens nach n Zeichen / Zuständen auftreten“)

Weiterhin gilt dann auch: Wenn $uvw \in L$, dann sind auch $uv^0w = uw, uv^1w = uvw, uv^2w$, $uv^3w, \ldots$ in L enthalten, d.h. für alle $i \in \mathbb{N}$ gilt $uv^iw \in L$. Damit ist die Aussage des Pumping-Lemmas gezeigt. #

Bemerkung 51: *Wir wenden das Pumping-Lemma* immer *nach dem Prinzip des indirekten Beweises (siehe Abschnitt C.3 auf Seite 76) an:*

i) ***Annahme:*** *Die gegebene Sprache ist regulär*

ii) Es existiert ein n ($\triangleq$ Pumping Konstante) und es gelten die drei Aussagen des Pumping-Lemmas.

iii) Es wird ein Widerspruch gezeigt, d.h. die Annahme war falsch und damit ist L nicht regulär.

Der Wert n spielt für unsere Argumentation gar keine Rolle und aufgrund der Konstruktion unseres Beweises existiert ein solcher Wert ja auch gar nicht, da die Sprache nicht regulär ist.

Bemerkung 52: *Da festgelegt wurde, dass $0 \in \mathbb{N}$ (siehe Beispiel 97 auf Seite 64) ist dann auch das (kürzere) Wort uw in der Sprache L enthalten. Diese Tatsache kann man in einigen Fällen dazu verwenden einfacher einen Widerspruch zu erzeugen, wenn man das Pumping Lemma benutzt.*

Bemerkung 53: *Das Pumping-Lemma kann auch mit Hilfe von Typ 3 Grammatiken bewiesen werden (Hinweis: Welche Form haben die Syntaxbäume, die sich für Typ 3 Grammatiken ergeben? Wo kann das Schubfachprinzip angewendet werden?)*

Beispiel 54: *Die Sprache $L_K = \{w \in \{a,b\}^* \mid w = a^m b^m, m \geq 1\}$ ist nicht regulär.*

Annahme: *L_K ist regulär. Dann existiert eine Zahl n und jedes Wort $x \in L_K$ mit $|x| \geq n$ lässt sich wie im Pumping-Lemma beschrieben zerlegen.*
Wir wählen speziell das Wort $x = a^n b^n$, also ein Wort der Länge $2n$. Damit existiert eine Zerlegung von x, wobei v nicht leer ist und $|uv| \leq n$. Der String v besteht also nur aus dem Zeichen a. Nach Pumping-Lemma ist dann auch $uv^0w = a^{n-|v|}b^n \in L_K$. Dies ist aber ein Widerspruch zur Definition von L_K, da immer gleich viele Buchstaben a und b in einem Wort aus L_K enthalten sein müssen, aber es gilt $n - |v| < n$. Also war die Annahme falsch und damit ist gezeigt, dass L_K nicht regulär ist.

Damit ist die Frage, die zu Beginn dieses Abschnitts gestellt wurde, beantwortet und es ist nun klar, dass keine Typ 3 Grammatik für die Sprache L_K existieren kann. Weiterhin ergibt sich (vgl. Eigenschaft 15) nun

Folgerung 55: *Die Menge der regulären Sprachen ist eine echte Teilmenge der kontextfreien Sprachen, d.h.*

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

Diese Erkenntnis hat weitreichende Auswirkungen in der praktischen Informatik, denn die Sprache L_K ist ein Modell für die korrekte Klammerung[11] von (mathematischen) Ausdrücken oder

[11] Eine korrekte Klammerung hat immer die Eigenschaft, dass die Anzahl der öffnenden Klammern mit der Anzahl der schließenden Klammern übereinstimmt.

z.B. von Anweisungsblöcken, die in Programmiersprachen zur Strukturierung eingesetzt werden. Aus diesem Grund ist die korrekte Blockklammerung von C, C++ oder Java durch „{“ und „}“ oder die Blockstrukturen von HTML/XML nicht mit endlichen Automaten testbar oder mit regulären Ausdrücken beschreibbar.

Beispiel 56: *Die Sprache* $L = \{w \in \{0\}^* \mid w = 0^p,\ p \text{ ist Primzahl}\} = \{0^2, 0^3, 0^5, 0^7, 0^{11}, \dots\}$ *ist nicht regulär.*

Annahme: *L ist regulär, dann existiert eine Zahl n und jedes* $x \in L$ *mit* $|x| \geq n$ *lässt sich wie im Pumping-Lemma beschrieben zerlegen. Es ist bekannt, dass unendlich viele Primzahlen existieren, also auch eine Primzahl* $p \geq n + 2$*. Dann muss nach Definition* $0^p \in L$ *gelten. Also müssen nach dem Pumping-Lemma auch alle Wörter der Form* $0^{|uw|+i|v|}$ *in L enthalten sein. Wir wählen* $i = |uw|$ *und erhalten* $|uw| + |uw| \cdot |v| = |uw| \cdot (\underbrace{1 + |v|}_{Faktor})$*. Dies zeigt aber, dass die Zahl* $|uw| + |uw| \cdot |v|$ *keine Primzahl sein kann. Insbesondere bedeutet dies, dass* $0^{|uw|+|uw|\cdot|v|} \notin L$*. Dies ist ein Widerspruch, was zeigt, dass die Annahme falsch wahr. Damit kann die Sprache L nicht regulär sein.*

Beispiel 57: *Die Sprache* $L = \{w \in \{0\}^* \mid w = 0^m,\ m \text{ ist eine Quadratzahl}\}$ *kann nicht durch einen regulären Ausdruck beschrieben werden. („Hausaufgabe“)*

4. Kontextfreie Sprachen

Nachdem wir uns nun ausführlich mit regulären Sprachen beschäftigt haben, wollen wir nun die Typ 2 - Sprachen näher untersuchen. Dazu haben wir schon definiert, dass eine Grammatik $G = (\Sigma, N, P, S)$ vom Typ 2 ist, gdw. jede Produktion der Form $A \to w$ mit $A \in N$, $w \in (N \cup \Sigma)^*$ und $|w| \geq 1$ entspricht. Weiterhin haben wir bereits gezeigt, dass es kontextfreie Sprachen gibt, die nicht regulär sind.

4.1. Die Chomsky Normalform

Definition 58: *Eine kontextfreie Grammatik heißt in* Chomsky Normalform*, falls alle Regeln entweder die Form* $A \to BC$ *mit* $A, B, C \in N$ *oder* $A \to a$ *mit* $A \in N$ *und* $a \in \Sigma$ *haben.*

Bemerkung 59: *Die Chomsky Normalform ist interessant, weil die Syntaxbäume (bis auf den letzten Schritt) Binärbäume sind (evtl. unbalanciert).*

Lemma 60: *Sei* $G = (\Sigma, N, P, S)$ *in Chomsky Normalform. Wenn* $w \in L(G)$*, dann* $S \overset{t}{\vdash} w$ *mit* $t = 2 \cdot |w| - 1$*.*

Beweisidee: Zeichnet man den Ableitungsbaum und betrachtet diesen Baum von den Blättern zur Wurzel, dann kann man zählen wieviele Schritte man braucht, um alle Terminalsymbole in Nichtterminalsymbole zu überführen und wieviele Worte man braucht, um zum Startsymbol zurück zu kommen. #

Satz 61: *Zu jeder kontextfreien Grammatik G gibt es eine Grammatik* G' *in Chomsky Normalform mit* $L(G) = L(G')$*.*

Beweis: Wir führen drei Schritte durch:

Eliminieren von Terminalsymbolen an der falschen Position: Wir formen die Grammatik G in eine Grammatik G_1 um, so dass Terminale nur noch in Regeln der Form $A \to a$ vorkommen.

Ersetze in jeder Regel $A \to w, |w| \geq 2$ jedes Terminal $a \in \Sigma$ durch das neue Nichtterminal D_a, wobei $D_a \notin N$ und füge die Regel $D_a \to a$ zu P hinzu. Offensichtlich gilt: $L(G_1) = L(G)$.

Entferne Kettenregeln: Regeln der Form $A \to B$ heißen *Kettenregeln*. Wir erzeugen eine Grammatik G_2 mit $L(G_2) = L(G_1)$ bei der nur noch Regeln der Form $A \to B_1 B_2 \dots B_k$, $k \geq 2$ und $A \to a$, wobei $A, B_1, \dots B_k \in N$ und $a \in \Sigma$, vorkommen.

Jede Folge $A_1 \to A_2, A_2 \to A_3, \dots, A_{m-1} \to A_m$ und $A_m \to B_1 B_2 \dots B_k$ mit $m \geq 2$ und $A_1, \dots A_m, B_1, \dots B_m \in N$ wird überführt in $A_1 \to B_1 B_2 \dots B_k$. Analog fügen wir für $A_1 \to A_2, A_2 \to A_3, \dots, A_{m-1} \to A_m, A_m \to a$, wobei $A_1, \dots, A_m \in N$ und $a \in \Sigma$ die neue Regel $A_1 \to a$ ein. Zum Schluss werden die Kettenregeln entfernt. Offensichtlich gilt: $L(G_2) = L(G_1)$.

Regeln kürzen: Nun wird die Grammatik G' erzeugt. Alle zu langen Regeln der Form $A \to B_1 B_2 \dots B_k$, $k \geq 3$ werden in kurze Regeln „aufgebrochen“.

Füge für jede zu lange Regel $A \to B_1 B_2 \dots B_k$, $k \geq 3$ *neue* Nichtterminale $D_2, \dots, D_{k-1}$ ein, und ersetze die zu lange Regel durch $A \to B_1 D_2, D_2 \to B_2 D_3, \dots, D_i \to B_i D_{i+1}$ für $2 \leq i \leq k-2$ und $D_{k-1} \to B_{k-1} B_k$. Wieder gilt: $L(G') = L(G_2)$.

Nach diesen drei Schritten ist die umgeformte Grammatik in Chomsky Normalform. #

Beispiel 62: *Sei $G = (\{a, b, c\}, \{S, A, B\}, P, S)$ mit $P = \{S \to AB, A \to aAb, A \to ab, B \to cB, B \to c\}$, dann formen wir die Grammatik in Chomsky Normalform wie folgt um:*

Eliminieren von Terminalsymbolen an der falschen Position:

- *$A \to aAb$ wird zu $A \to A_a A A_b$, $A_a \to a$ und $A_b \to b$.*
- *$A \to ab$ wird zu $A \to A_a A_b$.*
- *$B \to cB$ wird zu $B \to A_c B$, $A_c \to c$.*

Entferne Kettenregeln: *Entfällt, da es keine Kettenregeln gibt.*

Regeln kürzen: *Die Regel $A \to A_a A A_b$ wird zu $A \to A_a D_2$ und $D_2 \to A A_b$.*

Damit erhalten wir die Grammatik $G' = (\{a, b, c\}, \{S, A, B, A_a, A_b, A_c, D_2\}, P', S)$ mit $P' = \{S \to AB, A \to A_a D_2, D_2 \to A A_b, A_a \to a, A_b \to b, A_c \to c, A \to A_a A_b, B \to A_c B, B \to c\}$ die in Chomsky Normalform ist.

4.2. Das Pumping Lemma für kontextfreie Sprachen

Mit ähnlichen Ideen wie im Fall des Pumping Lemmas für reguläre Sprachen beweisen wir nun das *Pumping Lemma für kontextfreie Sprachen.*

Lemma 63: *Sei B ein Binärbaum so, dass jeder Knoten entweder keinen oder genau zwei Kinder hat. Wenn $B \geq 2^k$ Blätter hat, dann hat B mindestens einen Pfad der Länge $\geq k$.*

Beweisidee: Induktion über die Länge k des längsten Pfads oder mit Hilfe eines indirekten Beweises in Verbindung mit dem Abzählen der Anzahl aller Pfade der Länge $\leq k$. #

Satz 64: *Sei L eine kontextfreie Sprache. Dann gibt es eine Zahl $n \in \mathbb{N}$, so dass sich alle Wörter $z \in L$ mit $|z| \geq n$ zerlegen lassen in $z = uvwxy$ wobei:*

1. $|vx| \geq 1$
2. $|vwx| \leq n$
3. *Für alle $i \in \mathbb{N}$ gilt: $uv^iwx^iy \in L$*

Beweis: O.B.d.A. sei G eine Grammatik in Chomsky Normalform und $L(G) = L\backslash\{\epsilon\}$. Wir wählen $n =_{\text{def}} 2^{\#N}$ als *Pumping-Konstante*. Sei nun $z \in L(G)$ und $|z| \geq n$. Der Syntaxbaum ist dann, bis auf den letzten Ableitungsschritt, ein Binärbaum. Da $|z| \geq n$ hat dieser Baum mindestens $n = 2^{\#(N)}$ Blätter und damit gibt es mit Lemma 63 einen Pfad der Länge mindestens $\#N$. Halten wir nun diesen längsten Pfad fest, dann kommen, einschließlich des Startsymbols, mindestens $\#N + 1$ Nichtterminale auf diesem Pfad vor.

Deshalb muss ein Nichtterminal (in Abbildung 12 ist dieses Nichtterminal mit A bezeichnet) doppelt vorkommen (Schubfachprinzip). Wir folgen diesen Pfad von unten nach oben bis ein Nichtterminal zum zweiten Mal auftaucht. Damit ist das zweite Nichtterminal maximal $\#N$ Schritte von der Blattebene entfernt (sonst wäre es schon früher aufgetaucht), d.h. es ergibt sich die Situation in Abbildung 12.

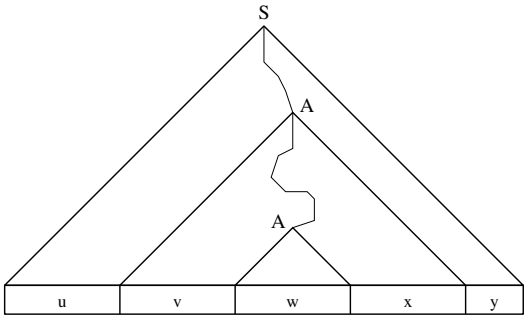
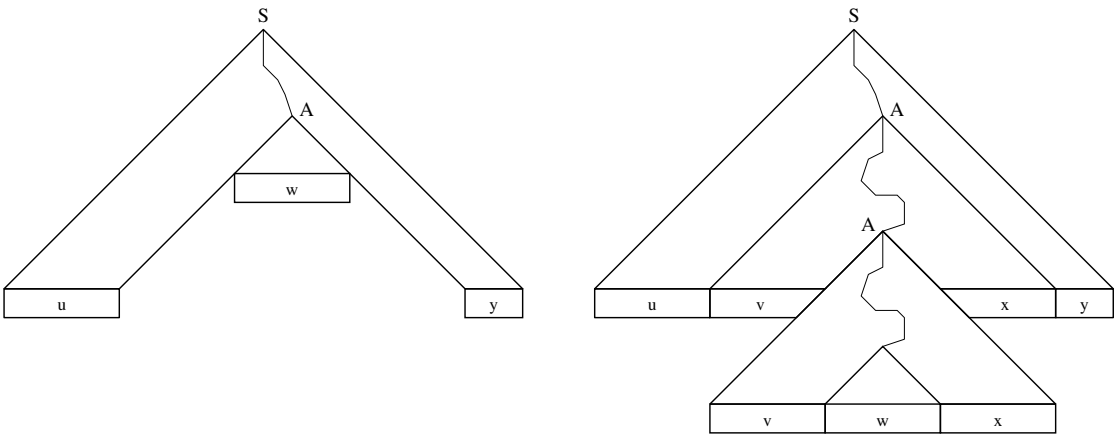


Abbildung 12: Ein Syntaxbaum

Weil G in Chomsky Normalform ist, gilt für das obere A, dass $A \to BC$. Nun geht entweder v aus B hervor, oder x aus C, d.h. vx kann nicht das leere Wort sein, also $|vx| \geq 1$.

Da das zweite A maximal $\#N$ Abstand von der Blattebene hat, können maximal $2^{\#N} = n$ Blätter erzeugt werden, d.h. $|vwx| \leq n$. Weiterhin kommt ein Nichtterminal doppelt vor, d.h. der Syntaxbaum kann nun z.B. wie folgt modifiziert werden:



Damit ist $uv^0wx^0y \in L$, $uv^2wx^2y \in L$ und allgemein gilt für alle $i \in \mathbb{N}$ auch $uv^iwx^iy \in L$. #

Bemerkung 65: *Da $0 \in \mathbb{N}$ (siehe Beispiel 97 auf Seite 64) gilt auch hier, ähnlich zum Pumping Lemma für reguläre Sprachen (vgl. Satz 49 auf Seite 23), dass das (kürzere) Wort uwy zur Sprache L gehört.*

Bemerkung 66: *Es gilt wieder: Ist L kontextfrei, dann existiert eine Pumping-Konstante n, jedes Wort $z \in L$ mit $|z| \geq n$ kann wie in Satz 64 in $z = uvwxy$ zerlegt werden und für jedes $i \in \mathbb{N}$ gilt auch $uv^iwx^iy \in L$. Die Umkehrung muß nicht wahr sein.*

Beispiel 67: *Die Sprache $L = \{z \in \{a,b,c\}^* \mid z = a^m b^m c^m \text{ mit } m \geq 1\}$ ist nicht kontextfrei.*

Annahme: *L ist kontextfrei, dann gibt es nach dem Pumping Lemma eine Zahl n, so dass sich alle Wörter $z = a^m b^m c^m$, $|z| \geq n$ zerlegen lassen in $uvwxy$ mit den Eigenschaften (1), (2) und (3) aus Theorem 64. Wir wählen speziell $z = a^n b^n c^n$ mit der Länge $3n$. Wegen Eigenschaft (2) kann vx nicht alle drei Arten von Buchstaben enthalten, denn $|vwx| \leq n$. Es gilt aber auch $|vx| \geq 1$ (Eigenschaft (1)). Wegen Eigenschaft (3) muss aber auch $z' = uv^0wx^0y = uwy \in L$ gelten, d.h. mindestens eine Art von Buchstaben in z' kommt öfters vor als die anderen Arten. Dies ist ein Widerspruch, denn solch ein Wort ist nicht in der Sprache L enthalten.*

Folgerung 68: *Die Menge der kontextfreien Sprachen ist eine echte Teilmenge der kontextsensitiven Sprachen, d.h.*

$$\mathbf{L}_2 \subset \mathbf{L}_1.$$

Beispiel 69: *Die Sprache $L = \{w \in \{0\}^* \mid w = 0^p,\ p \text{ ist Primzahl}\} = \{0^2, 0^3, 0^5, 0^7, 0^{11}, \dots\}$ ist nicht kontextfrei.*

Annahme: *L ist kontextfrei, dann existiert eine Zahl n und jedes Wort $z \in L$ mit $|z| \geq n$ kann in $z = uvwxy$ zerlegt werden, wobei die Bedingungen aus Satz 64 gelten. Wir wählen $z = 0^p$, wobei p eine Primzahl mit $p \geq n$ ist. Eine solche Primzahl existiert auf jeden Fall, da es unendlich viele Primzahlen gibt.*

Es existiert also eine Zerlegung $z = uvwxy$ und alle Wörter uv^iwx^iy, $i \geq 0$ sind in L enthalten. Wir wählen $i = p + 1$. Also $z' = uv^{p+1}wx^{p+1}y \in L$. Es gilt $|z'| = |u| + |v| \cdot (p+1) + |w| + |x| \cdot (p+1) + |y| = \underbrace{|u| + |v| + |w| + |x| + |y|}_{p} + |v| \cdot p + |x| \cdot p = p + |v| \cdot p + |x| \cdot p = p \cdot (1 + |v| + |x|)$, da $|vx| \geq 1|$ gilt, ist $(1 + |v| + |x|) \geq 2$. Also ist $|z'|$ keine Primzahl, was einen Widerspruch bedeutet, da $z' \in L$. Damit kann L nicht kontextfrei sein.

4.3. Der CYK-Algorithmus

Wir haben schon gezeigt, dass das Wortproblem für Typ1-, Typ2- und Typ3-Sprachen lösbar ist (siehe Algorithmus 1 auf Seite 10). Für Typ3-Sprachen kennen wir schon eine schnelle Möglichkeit um zu testen, ob $z \in L$ gilt, wenn wir einen DEA gegeben haben, der die Sprache L akzeptiert.

Nun soll ein „schneller“ Algorithmus für das Wortproblem für kontextfreie Sprachen angegeben werden, der als *CYK-Algorithmus* bekannt ist und der nach seinen Erfindern Cocke, Younger und Kasami benannt wurde. Die grundlegende Idee dabei ist, dass für jede kontextfreie Sprache L eine Grammatik G in Chomsky-Normalform existiert mit $L = L(G)$ (siehe Satz 61). Damit ergibt sich:

Worte der Länge 1: Es kommt nur eine Regel der Form $A \to a$ in Frage um das Wort zu erzeugen.

Worte der Länge > 1: Wenn $A \overset{*}{\vdash} x$ gilt, dann wurde im ersten Schritt eine Regel der Form $A \to BC$ angewendet. Sei nun $x = a_1 \dots a_n$, dann wird ein Teil von x aus B erzeugt und das Reststück aus C. Es muss also ein $1 \leq k \leq n$ geben, sodass sich der Syntaxbaum aus Abbildung 13 ergibt. Mit Hilfe dieser Idee können wir das Wortproblem für die Länge n auf zwei Probleme der Länge k und $n - k$ zurückführen. Dabei ist allerdings das k (also der Index an dem das Wort in zwei Teile zerlegt wird) zu diesem Zeitpunkt unbekannt. Aus diesem Grund müssen alle Möglichkeiten von 1 bis $n - 1$ untersucht werden.

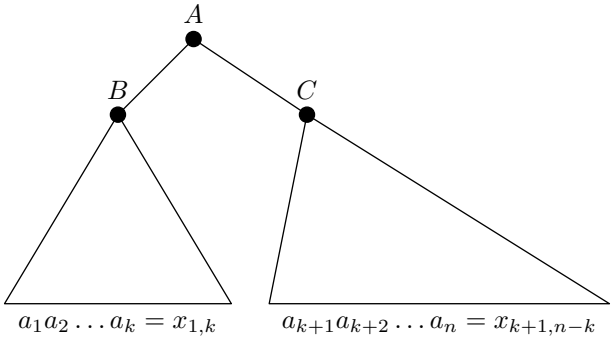


Abbildung 13: Struktur eines Syntaxbaums bei Anwendung einer Regel $A \to BC$

Um das Problem der richtigen „Unterteilungsposition“ k zu lösen, verwenden wir die Methode des *dynamischen Programmierens* und die folgende Notation:

Sei $x = a_1 \dots a_n$, dann ist $x_{i,j}$ das Teilwort von x, das an *Position i startet* und die *Länge j* hat.
Der CYK-Algorithmus (siehe Algorithmus 2 auf Seite 30) verwendet eine zweidimensionale Matrix $T[1 \dots n][1 \dots n]$, wobei aber nur die obere Dreiecksmatrix zum Einsatz kommt. Im Tabelleneintrag $T[i][j]$ sind alle die Nichtterminale notiert, aus denen $x_{i,j}$ abgeleitet werden kann. Offensichtlich gilt dann $x = a_1 \dots a_n \in L(G)$ gdw. $S \in T[1][n]$.

Beispiel 70: *Die Sprache $L = \{a^n b^n c^m \mid n, m \geq 1\}$ ist kontextfrei vermöge der Grammatik $G' = (\{a, b, c\}, \{S, A, B\}, \{S \to AB, A \to ab, A \to aAb, B \to c, B \to cB\}, S)$. Umformen in Chomsky Normalform ergibt $G' = (\{a, b, c\}, \{S, A, B, C, D, E, F\}, P, S)$, wobei*

$$P = \left\{ \begin{array}{ll} S \to AB, & \\ A \to CD, & A \to CF, \\ B \to c, & B \to EB, \\ C \to a, & \\ D \to b, & \\ E \to c, & \\ F \to AD & \end{array} \right\}$$

Sei nun $w = aaabbbcc$, dann wird die Tabelle aus Abbildung 14 erzeugt. Dies zeigt $w \in L$.

Beispiel 71: *Gegeben sei die Grammatik $G' = (\{), (, \wedge, \vee, \neg, x\}, \{F\}, P', F)$ mit der Menge der Produktionen $P' = \{F \to (F \wedge F), F \to (F \vee F), F \to \neg F, F \to x\}$. Diese Grammatik ist offensichtlich nicht in Chomsky Normalform. Wir konstruieren eine äquivalente Grammatik G in Chomsky Normalform:*

- *$F \to (F \wedge F)$ wird zu $F \to AFUFZ$*
- *$F \to (F \vee F)$ wird zu $F \to AFOFZ$*

Diese Regeln müssen noch gekürzt werden, wobei wir eine kleine Optimierung für die Umwandlung der Regeln $F \to AFUFZ$ bzw. $F \to AFOFZ$ verwenden:

$$\begin{array}{llll} F \to AD_2, & D_2 \to FD_3, & D_3 \to UD_4, & D_4 \to FZ \\ & & D_3 \to OD_4 & \end{array}$$

Algorithmus 2: Der CYK-Algorithmus

Data: Grammatik $G = (\Sigma, N, P, S)$, Wort $x = a_1, \ldots, a_n$ und Länge n
Result: `true` falls $x \in L(G)$ und `false` sonst

for *($i = 1$; $i \leq n$; i++)* **do**
  `/* Worte der Länge 1 */`
  $T[i][1] = \{A \in N \mid A \to a_i \ \in P\}$;
end
`/* Bearbeite alle möglichen Längen */`
for *($j = 2$; $j \leq n$; j++)* **do**
  `/* Bearbeite alle möglichen Startpositionen */`
  for *($i = 1$; $i \leq n + 1 - j$; i++)* **do**
    `/* Initialisiere die Mengenvariable */`
    $T[i][j] = \emptyset$;
    `/* Teste mögliche Zerlegungen für Worte der Länge` j `an Position` i `*/`
    for *($k = 1$; $k \leq j - 1$; k++)* **do**
      $T[i][j] = T[i][j] \cup \{A \in N \mid A \to BC \in P, B \in T[i][k] \text{ und } C \in T[i + k][j - k]\}$;
    end
  end
end
`/* Teste` $x \stackrel{?}{\in} L$ (`≙ das Wort` x `kann aus` S `erzeugt werden) */`
if *($S \in T[1][n]$)* **then**
  return `true`;
else
  return `false`;
end

Länge j \ Position i	a	a	a	b	b	b	c	c
1	C	C	C	D	D	D	E, B	E, B
2	$\emptyset$	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$	B	
3	$\emptyset$	$\emptyset$	F	$\emptyset$	$\emptyset$	$\emptyset$		
4	$\emptyset$	A	$\emptyset$	$\emptyset$	$\emptyset$			
5	$\emptyset$	F	$\emptyset$	$\emptyset$				
6	A	$\emptyset$	$\emptyset$					
7	S	$\emptyset$						
8	S							

Abbildung 14: Ein Beispiel für den CYK-Algorithmus

Länge j \ Position i	(	(	x	$\vee$	x	)	$\wedge$	$\neg$	x	)
	A	A	F	O	F	Z	U	N	F	Z
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	D_4	$\emptyset$	$\emptyset$	F	D_4	
	$\emptyset$	$\emptyset$	$\emptyset$	D_3	$\emptyset$	$\emptyset$	$\emptyset$	D_4		
	$\emptyset$	$\emptyset$	D_2	$\emptyset$	$\emptyset$	$\emptyset$	D_3			
	$\emptyset$	F	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$				
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$					
	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$						
	$\emptyset$	$\emptyset$	$\emptyset$							
	$\emptyset$	D_2								
	F									

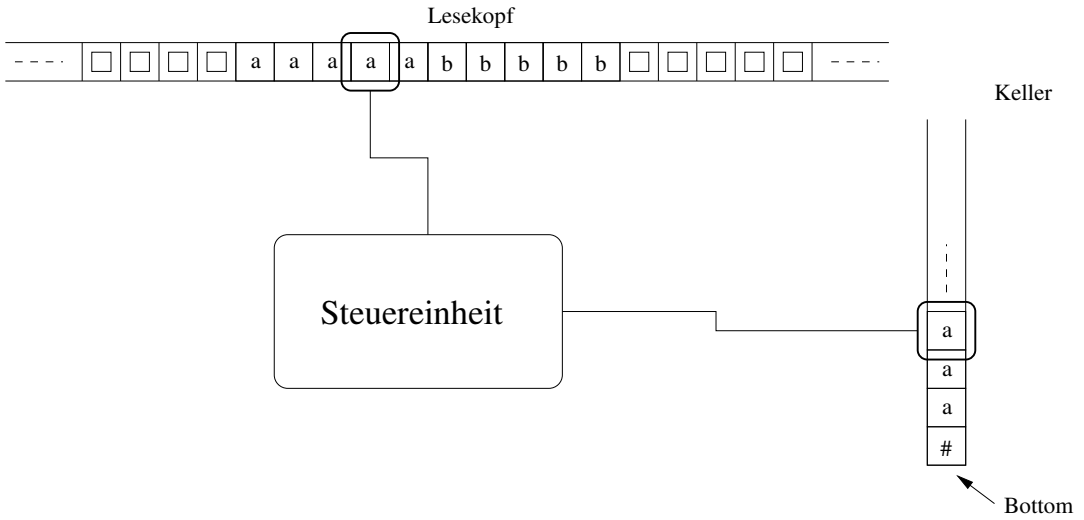
Abbildung 15: Ein weiteres Beispiel für den CYK-Algorithmus

Zusätzlich haben wir noch weitere Regeln: $F \to x$, $N \to \neg$, $U \to \wedge$, $O \to \vee$, $A \to (, Z \to)$, $F \to NF$. *Also ist die gesuchte Grammatik* $G = (\{), (, \wedge, \vee, \neg, x\}, \{F, A, D_2, D_3, D_4, Z, U, O, N\}, F, P)$, *wobei die Menge* P *der Produktionen schon beschrieben wurde. Es gilt* $F \overset{*}{\vdash} (F \vee F) \overset{*}{\vdash} ((F \vee F) \wedge F) \overset{*}{\vdash} ((F \vee F) \wedge \neg F) \overset{*}{\vdash} ((x \vee x) \wedge \neg x)$.

Wir wollen via CYK-Algorithmus testen, ob $((x \vee x) \wedge \neg x) \in L(G')$ *bzw.* $((x \vee x) \wedge \neg x) \in L(G)$. *Sei* $w = ((x \vee x) \wedge \neg x)$, *dann ergibt sich die Tabelle aus Abbildung 15. Damit ist* $w \in L$ *gezeigt.*

4.4. Kellerautomaten

Wir haben schon gesehen, dass die Sprache $a^m b^m$ nicht durch einen endlichen Automaten akzeptiert werden kann. Wie muss das Modell des endlichen Automaten erweitert werden, damit wir diese Sprache akzeptieren können?
Idee: Wir brauchen einen zusätzlichen Kellerspeicher:



Definition 72 ((nichtdeterministischer) Kellerautomat): *Ein* (nichtdeterministischer) Kellerautomat *(engl. pushdown automaton, kurz PDA) ist ein* $6-Tupel$

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$$

- Z *die endliche Menge der Zustände*
- Σ *das Eingabealphabet*
- Γ *das Kelleralphabet*
- $\delta\colon Z \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Z \times \Gamma^*)$ *die Überführungsfunktion*
- z_0 *der Startzustand*
- $\#$ *das unterste Kellerzeichen*

Intuitiv bedeutet $\delta(z, a, A) \ni (z', B_1 \dots B_k)$, dass, wenn sich der Automat M im Zustand z befindet ein a liest und A oben im Keller liegt, dann *kann*[12] M in Zustand z' übergehen und dass A auf dem Keller durch $B_1 \dots B_k$ ersetzen, wobei B_1 dann oberstes Symbol im Keller ist.

Definition 73 (deterministischer Kellerautomat): *Ein nichtdeterministischer Kellerautomat M heißt* deterministisch*, falls in jeder möglichen Situation von M höchstens ein Befehl/Übergang möglich ist.*

Das Verhältnis von nichtdeterministischen und deterministischen Kellerautomaten ist vergleichbar zu den nichtdeterministischen bzw. deterministischen endlichen Automaten (siehe Abbildung 7 auf Seite 13).

Zu Beginn der Berechnung eines Kellerautomats M findet sich immer das Kellerendesymbol $\#$ im Keller. Ein Kellerautomat *akzeptiert* die Eingabe gdw. der Keller leer ist und die Eingabe vollständig gelesen wurde. Wir legen fest:

$$L(M) =_{\text{def}} \{w \in \Sigma^* \mid w \text{ wird von } M \text{ akzeptiert}\}$$

Definition 74: *Eine* Konfiguration *eines Kellerautomaten ist gegeben durch ein Tripel $k \in Z \times \Sigma^* \times \Gamma^*$. Intuitiv ist die Konfiguration eines Kellerautomaten eine vollständige „Momentaufnahme“, „Save-Game“ oder „core-dump“ eines Kellerautomaten und besteht aus dem* aktuellen Zustand*, der noch* nicht verarbeiteten Eingabe *und dem* vollständigen Kellerinhalt*.*

Beispiel 75: *Sei $M = (\{z_0, z_1\}, \{a, b\}, \{A, \#\}, \delta, z_0, \#)$ und δ wird durch folgende Tabelle angegeben:*

$$\begin{array}{llllll} z_0\ a\ \# & \to & z_0\ A\# & z_1\ b\ A & \to & z_1\ \epsilon \\ z_0\ a\ A & \to & z_0\ AA & z_1\ \epsilon\ \# & \to & z_1\ \epsilon \\ z_0\ b\ A & \to & z_1\ \epsilon & & & \end{array}$$

Für die Eingabe aaabbb ergibt sich die folgende Berechnung des Automaten M:

$$\begin{array}{llllll} (z_0, aaabbb, \#) & \vdash & (z_0, aabbb, A\#) & \vdash & (z_0, abbb, AA\#) & \vdash & (z_0, bbb, AAA\#) \\ & \vdash & (z_1, bb, AA\#) & \vdash & (z_1, b, A\#) & \vdash & (z_1, \epsilon, \#) \\ & \vdash & (z_1, \epsilon, \epsilon) & & & & \end{array}$$

Damit wird ersichtlich, dass der Automat M die Eingabe akzeptiert und es gilt $L(M) = \{a^m b^m \mid m \geq 0\}$ und insbesondere $aaabbb \in L(M)$. Weiterhin ist dieser Automat sogar deterministisch, denn in jedem Berechnungsschritt gibt es für eine gegebene Eingabe nur einen möglichen Nachfolgeschritt.

[12] Die hier beschriebenen Kellerautomaten arbeiten nichtdeterministisch, d.h. es werden evtl. mehrere Berechnungspfade nachvollzogen.

Der folgende Satz zeigt, dass für jede kontextfreie Sprache L ein Kellerautomat M mit $L = L(M)$ existiert, d.h. die Kellerautomaten spielen für die kontextfreien Sprachen die gleiche Rolle wie die (nicht)deterministischen Automaten für die regulären Sprachen.

Satz 76: *Eine Sprache L ist kontextfrei gdw. L von einem Kellerautomaten akzeptiert wird.*

Beweis: Um diesen Satz zu beweisen, müssen wir zwei verschiedene Richtungen zeigen:

- Gegeben eine kontextfreie Grammatik G, dann konstruiere aus G einen Kellerautomaten M mit $L(G) = L(M)$.
- Gegeben ein Kellerautomat M', dann konstruiere aus M' eine kontextfreie Grammatik G' mit $L(M') = L(G')$.

„⇒“: Sei $G = (\Sigma, N, P, S)$ eine Typ2 - Grammatik mit $L = L(G)$. Wir geben einen Kellerautomaten an, der die Ersetzungsschritte von G mit Hilfe des Kellerinhaltes simuliert.

Das Kelleralphabet von M ist $\Gamma =_{\text{def}} N \cup \Sigma$, wobei das unterste Kellersymbol S ist. Wird dieses mit einer Pop-Operation entfernt, dann akzeptiert M. Wir haben also:

$$M = (\{z\}, \Sigma, \Gamma, \delta, z, S), \text{wobei } \delta \text{ wie folgt definiert ist:}$$

Für jede Regel $A \to \alpha \in P$ und $\alpha \in (N \cup \Sigma)^*$ setze $\delta(z, \epsilon, A) \ni (z, \alpha)$ und $\delta(z, a, a) \ni (z, \epsilon)$ für $a \in \Sigma$.

Damit wird immer dann, wenn das oberste Kellerzeichen ein Nichtterminal einer Grammatik ist, nichtdeterministisch eine Regel aus P angewendet, ohne ein Eingabesymbol zu lesen. Immer dann, wenn ein Terminalsymbol oben auf dem Keller liegt, dann wird dieses Terminalsymbol mit einer Pop-Operation entfernt gdw. es mit dem aktuellen Eingabezeichen übereinstimmt.

Also gilt $x \in L(G)$ gdw. x kann aus S erzeugt werden (d.h. $S \overset{*}{\underset{G}{\vdash}} x$) gdw. es gibt eine Folge von Konfigurationen von M mit $(z, x, S) \overset{*}{\underset{M}{\vdash}} \ldots \overset{*}{\underset{M}{\vdash}} (z, \epsilon, \epsilon)$ gdw. $x \in L(M)$, d.h. M akzeptiert x.

„⇐“: Sei M ein PDA mit mit $M = (Z, \Sigma, \Gamma, \delta, z_0, \#)$ und $L = L(M)$. Gesucht ist eine Grammatik G vom Typ 2 mit $L = L(G)$.

O.B.d.A. können wir annehmen, dass für jede Zeile der δ-Funktion $zaA \to z'B_1 \ldots B_k$ mit $k \leq 2$ gilt, denn wenn $k > 2$, dann ersetzen wir diese Zeile äquivalent durch

$$\left.\begin{array}{rcl} zaA & \to & z_1 B_{k-1} B_k \\ z\epsilon B_{k-1} & \to & z_2 B_{k-2} B_{k-1} \\ \vdots & \vdots & \vdots \\ z_{k-2}\epsilon B_2 & \to & z' B_1 B_2, \end{array}\right\} \text{Aufbrechen einer „langen“ Zeile der } \delta\text{-Funktion}$$

wobei $z_1, \ldots, z_{k-2}$ neue Zustände sind. Wir konstruieren nun eine Grammatik, die die Schritte von M simuliert, wobei die Nichtterminale der Grammatik aus mehreren Bestandteilen zusammengesetzt sind (via Kreuzprodukt). Sei $G = (\Sigma, N, P, S)$ und

$$N = \{S\} \cup (Z \times \Gamma \times Z).$$

Die erste Komponente eines Tripel aus $Z \times \Gamma \times Z$ steht dabei für den vorherigen Zustand, die zweite Komponente für das verarbeitete Kellersymbol und die letzte Komponente symbolisiert den erreichten Zustand. Nun definiert wir

$$\begin{aligned} P = \quad & \{S \to \overbrace{(z_0,\ \#, z)}^{\text{Nichtterminal}} \mid z \in Z\} \\ \cup \quad & \{(z, A, z') \to a \mid \delta(z, a, A) \ni (z', \epsilon) \text{ und } z' \in Z\} \\ \cup \quad & \{(z, A, z') \to a(z_1, B, z') \mid \delta(z, a, A) \ni (z_1, B) \text{ und } z' \in Z\} \\ \cup \quad & \{(z, A, z') \to a(z_1, B, z_2)(z_2, C, z') \mid \delta(z, a, A) \ni (z_1, BC) \text{ und } z', z_2 \in Z\} \end{aligned}$$

Diese Konstruktion kann Regeln der Form $A \to \epsilon$ erzeugen, die nach unserer Definition eigentlich illegal sind. Regeln dieser Form können aber nachträglich leicht aus der Grammatik entfernt werden ($\Rightarrow$ Übungsaufgabe). Wir zeigen nun $(z, A, z') \overset{*}{\vdash_G} x$ gdw. $(z, x, A) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$. Dazu beobachten wir zuerst, dass für $a \in \Sigma \cup \{\epsilon\}$

$$(z, A, z') \vdash_G a \quad \text{gdw.} \quad (z, A, z') \to a \in P \; (\star\star)$$

gilt. Nun beweisen wir durch Induktion über die Anzahl der Induktionsschritte, dass falls $(z, A, z') \overset{*}{\vdash_G} x$ gilt, dann auch $(z, x, A) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$.

(IA) Wird direkt durch Beobachtung $(\star\star)$ gezeigt.

(IS) Es gilt $(z, ay, A) \vdash_M (z_1, y, \alpha) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$, wobei $a \in \Sigma \cup \{\epsilon\}$, $z_1 \in Z$ und $\alpha \in \Gamma^*$. Wir unterscheiden drei denkbare Kellerinhalte:

$\alpha = \epsilon$**:** Dies ist nicht möglich, da (z_1, γ, ϵ) keine Folgekonfiguration hat, d.h. die Berechnung würde abbrechen.

$\alpha = B$**:** Nach Induktionsvoraussetzung gilt dann $(z_1, B, z') \overset{*}{\vdash_G} y$. Außerdem existiert in P eine Regel der Form $(z, A, z') \to a(z_1, B, z')$. Zusammen ergibt sich dann $(z, A, z') \vdash_G a(z_1, B, z') \underbrace{ay}_{=x}$.

$\alpha = BC$**:** Wir zerlegen $(z_1, y, BC) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$ in zwei Abschnitte $(z_1, y, BC) \overset{*}{\vdash_M} (z_1, y_2, C)$ und $(z_2, y_2, C) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$, so dass $y = y_1 y_2$. Für das Anfangsstück y_1 gilt $(z_1, y, B) \overset{*}{\vdash_M} (z_2, \epsilon, \epsilon)$. Nach **(IV)** gilt also $(z_1, B, z_2) \overset{*}{\vdash_G} y_1$, $(z_2, X, z') \overset{*}{\vdash_G} y_2$ und es muss eine Regel $(z, A, z') \to a(z_1 B, z_2)(z_2, C, z') \in P$ existieren. Zusammen gilt dann $(z, A, z') \vdash_G a(z_1, B, z_2)(z_2, B, z') \overset{*}{\vdash_G} a y_1 (z_2, C, z') \overset{*}{\vdash_G} \underbrace{a y_1 y_2}_{=x}$.

Abschließend zeigen wir noch durch Induktion über die Anzahl der Induktionsschritte, dass wenn $(z, x, A) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$ gilt, dann auch $(z, A, z') \overset{*}{\vdash_G} x$.

(IA) Siehe Beobachtung $(\star\star)$.

(IS) Die Induktionsschritt teilt sich in drei Teilbeweise auf:

Fall 1: Falls $(z, A, z') \vdash_G a \vdash_G x$, dann $x = a \in \Sigma$. Dies ist nicht möglich, da $k > 1$ und $|x| = 1$.

Fall 2: Falls $(z, a, z') \vdash_G a(z_1, B, z') \overset{*}{\vdash_G} ay$, wobei $ay = x$. Dann ist $\delta(z, a, A) \ni (z_1, B)$ und nach **(IV)** gilt $(z_1, B, z') \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$. Zusammen gilt also $(z, ay, A) \vdash_M (z_1, y, B) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$.

Fall 3: Wenn $(z, A, z') \vdash_G a(z_1, B, z_2)(z_2, C, z') \overset{*}{\vdash_G} ay$, wobei $ay = x$, dann ist $\delta(z, a, A) \ni (z_1, BC)$ und nach **(IV)** gilt sowohl $(z_1, y_1, B) \overset{*}{\vdash_M} (z_2, \epsilon, \epsilon)$ als auch $(z_2, y_2, C) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$. Zusammen gilt also $(z, a y_1 y_2, A) \vdash_M (z_1, y_1, y_2, BC) \overset{*}{\vdash_M} (z_2, y_2, C) \overset{*}{\vdash_M} (z', \epsilon, \epsilon)$.

Es gilt also $L(M) = L(G)$ und

$$\begin{aligned} x \in L(M) \quad & \text{gdw.} \quad (z_0, x, \#) \overset{*}{\vdash_M} (z, \epsilon, \epsilon) \text{ für } z \in Z. \\ & \text{gdw.} \quad S \vdash_G (z_0, \#, z) \overset{*}{\vdash_G} x \text{ für } z \in Z. \\ & \text{gdw.} \quad x \in L(G) \end{aligned}$$

Damit ist gezeigt, dass eine Sprache L kontextfrei ist gdw. L sie von einem Kellerautomaten akzeptiert wird. #

Bemerkung 77: *Die Kellerautomaten stellen ein weiteres Werkzeug dar, mit dem man nachweisen kann, dass eine gegebene Sprache kontextfrei ist, denn man muss nur einen geeigneten Automaten angeben, der diese Sprache akzeptiert.*

Weiterhin wird nun auch einsichtig, warum die Sprache $L_1 =_{\text{def}} \{ww^R \mid w \in \Sigma^*\}$ *kontextfrei ist und weshalb die Sprache* $L_2 =_{\text{def}} \{ww \mid w \in \Sigma^*\}$ *nicht kontextfrei sein kann.*

Bemerkung 78: *Der Beweis von Satz 76 ist deshalb von besonderem Interesse, weil er uns eine Methode liefert aus einer gegebenen Grammatik einen Kellerautomaten zu konstruieren (vgl. Richtung „⇒“). Damit haben wir eine Methode an der Hand, aus einer gegebenen Grammatik G einen „Erkenner“ (Parser) zu konstruieren, der für uns entscheidet, ob* $w \in L(G)$ *gilt. Leider sind diese Automaten nicht immer deterministisch und man kann sogar zeigen, dass eine mit der Potenzmengenkonstruktion vergleichbare Methode für die kontextfreien Sprachen nicht existieren kann. Damit müssen wir uns zumindest auf eine echte Teilmenge der kontextfreien Sprachen, die so genannten* deterministischen kontextfreien Sprachen *, einschränken. In der Praxis schränkt man sich noch weiter ein und setzt so genannte* LR(1)- *oder* LL(1)*-Sprachen zur Konstruktion von Programmiersprachen ein.*

5. Kontextsensitive- und Typ0-Sprachen

Wir haben definiert, dass eine Grammatik vom Typ1 ist gdw. alle Regeln die Form

$$u_1 A u_2 \to u_1 w u_2 \text{ mit } u_1, u_2, w \in (\Sigma \cup N)^* \text{ und } |w| \geq 1$$

haben. Leicht beobachtet man, dass für alle Produktionen einer kontextsensitiven Grammatik $|u_1 A u_2| \leq |u_1 w u_2|$ gilt. Wir wollen nun sogar zeigen, dass Typ1-Sprachen auch wie folgt durch Grammatiken charakterisiert werden können:

Definition 79: *Eine Grammatik ist vom* Typ1′*, falls alle Regeln* $w_1 \to w_2 \in P$ *die Eigenschaft* $|w_1| \leq |w_2|$ *(⋆) haben („Regeln sind nicht schrumpfend“).*

Satz 80: *Die Menge von Sprachen vom Typ1 entspricht der Menge von Sprachen vom Typ1', d.h.* $\mathbf{L}_1 = \mathbf{L}_{1'}$.

Beweis:

⊆: Aus der obigen Beobachtung (⋆) folgt direkt $\mathbf{L}_1 \subseteq \mathbf{L}_{1'}$.

⊇: Man kann zeigen (Übung), dass jede Grammatik vom Typ1′ so umgeformt werden kann, das alle Produktionen einer der vier Formen

$$\underbrace{A \to a}_{\text{(I)}}, \quad \underbrace{A \to B}_{\text{(II)}}, \quad \underbrace{A \to BC}_{\text{(III)}} \text{ oder } \underbrace{AB \to CD}_{\text{(IV)}}$$

entspricht, wobei $A, B, C, D \in N$ und $a \in \Sigma$. Grammatiken, bei denen jede Produktion einer der Formen (I) − (IV) entspricht, sind in *Kuroda Normalform*. Es ist unmittelbar einsichtig, dass alle Regeln der Art $(I) - (III)$ zu dem Schema von Typ1-Grammatiken passen (wähle $u_1 = u_2 = \epsilon$), da sie kontextfrei sind. Regeln der Form $AB \to CD$ werden durch $AB \to AD$ und $AD \to CD$ ersetzt, die damit auch dem Typ1-Schema entsprechen, d.h. $\mathbf{L}_{1'} \subseteq \mathbf{L}_1$. #

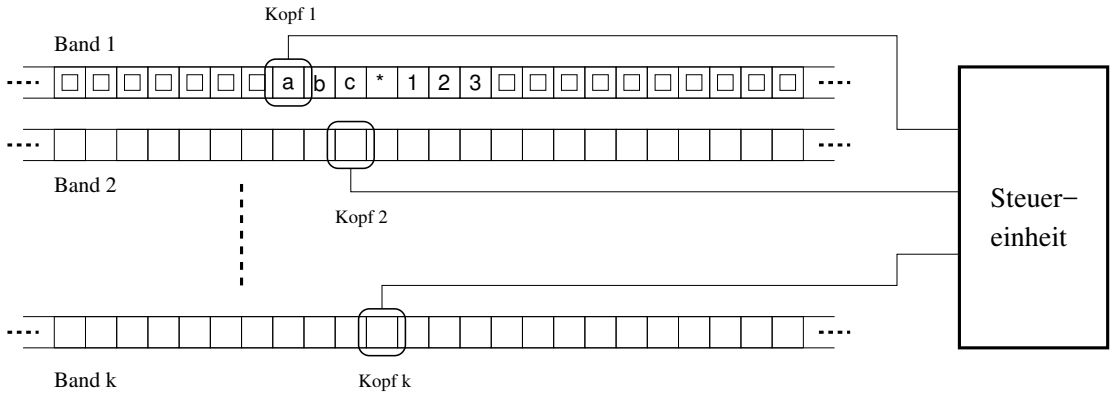


Abbildung 16: Graphische Darstellung einer Turingmaschine

5.1. Turingmaschinen

Die *Turingmaschine* (kurz *TM*) wurde 1936 (vgl. [Tur36]) von dem britischen Mathematiker Alan M. Turing[13] als „Modell des menschlichen Rechnens“ eingeführt.

Definition 81 (Turingmaschine): *Eine Turingmaschine besteht aus folgenden Teilen (eine graphische Darstellung findet sich in Abbildung 16):*

- *Mindestens ein beidseitig unendlich langes* Band *($k \geq 1$). Jedes Feld enthält einen Buchstaben aus dem (Band)-Alphabet Σ. Das besondere Zeichen $\square \in \Sigma$ („*Blanksymbol*“) sagt aus, dass in diesem Feld nichts steht.*
- *Jeder Kopf kann Feld für Feld auf dem Band bewegt werden und dabei den Inhalt des aktuellen Felds lesen und ändern.*
- *Die Steuereinheit befindet sich in einem der* endlich *vielen Zustände und steuert mit Hilfe des Bandinhalts die Aktivitäten.*
- *Es gibt zwei besondere Zustände, den Start- und den Stopzustand.*

Arbeitsweise: Eine TM arbeitet taktweise. In einem Takt kann sie in Abhängigkeit

- *vom aktuellen Zustand und*
- *den von den k Köpfen gelesenen k Buchstaben*

gleichzeitig

- *in einen neuen Zustand übergehen*
- *die k Bandsymbole unter dem Kopf ändern und*
- *jeden Kopf maximal um ein Feld verschieben.*

Genauer: Eine k-Band Turingmaschine M ist ein 5-Tupel $M = (\Sigma, Z, \delta, z_0, z_1)$, wobei

- *Σ das Bandalphabet*
- *Z die Menge der Zustände,*

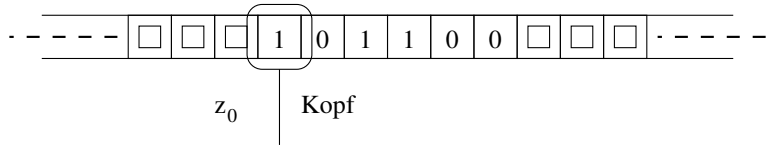
[13] *1912 in London (England) - †1954 in Wilmslow (England)

- $\delta\colon Z \times \Sigma^k \to \mathcal{P}(Z \times \Sigma^k \times \{L, N, R\}^k)$,
- z_0 *der Startzustand und*
- z_1 *der Endzustand ist.*

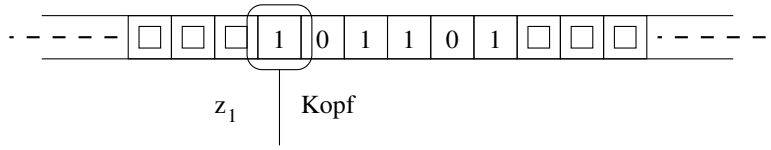
Dabei legt $\delta(z, a_1, \ldots, a_k) \ni (z', a'_1, \ldots, a'_k, \sigma_1, \ldots, \sigma_k)$ *(kurz:* $za_1 \ldots a_k \to z'a'_1 \ldots a'_k\sigma_1 \ldots \sigma_k$*) das Verhalten der TM wie folgt fest. Ist* M *im Zustand* z *und die Köpfe* $1, \ldots, k$ *lesen* $a_1, \ldots, a_k$*, dann geht* M *in den Zustand* z' *über, die* k *Köpfe schreiben* $a'_1, \ldots, a'_k$ *und* M *bewegt danach den Kopf* i *nach* $\sigma_i \in \{L, N, R\}$*. Dabei bedeutet* L *(bzw.* R*) eine Kopfbewegung nach* Links *(bzw.* Rechts*) und* N *bedeutet, dass der Kopf* nicht *bewegt wird.*

Zusätzlich legen wir folgende Standardsituationen fest:
Die Eingabe befindet sich auf Band 1, Kopf 1 ist am Anfang ganz links auf der Eingabe und alle anderen Bänder sind leer (d.h. sie enthalten nur □-Symbole). Die TM hält mit der Ausgabe auf Band 1, wobei sich Kopf 1 ganz links auf der Ausgabe befindet. Alle anderen Bänder sind leer.

Beispiel 82: *Wir entwickeln eine Turingmaschine, die zu einer Zahl in Binärdarstellung den Wert* $+1$ *addiert. Unsere* 1*-Band TM startet in der folgenden Situation:*



addiert 1 *und endet in folgendem Zustand:*



Dabei ist $k = 1$*, das Bandalphabet* $\Sigma = \{\square, 0, 1\}$*, die Zustandsmenge* $Z = \{z_0, \ldots, z_3\}$ *und die Übergangsfunktion* δ *(in verkürzter Schreibweise):*

z_0	1	$\to$	z_0	1	R	z_2	0	$\to$	z_2	0	L	z_3	0	$\to$	z_2	1	L
z_0	0	$\to$	z_0	0	R	z_2	1	$\to$	z_2	1	L	z_3	1	$\to$	z_3	0	L
z_0	□	$\to$	z_3	□	L	z_2	□	$\to$	z_1	□	R	z_3	□	$\to$	z_1	*1*	N
nach rechts wandern						*kein Übertrag*						*Übertrag beachten*					

Beispiel 83: *Erkenne ob die Eingabe ein Palindrom ist, d.h. ob die Buchstaben von links nach rechts gelesen das gleiche Wort bilden, wie von rechts nach links gelesen. Wir sollen also ein Turing-Programm für die folgende Funktion angeben:*

$$\text{palindrom}(x) =_{\text{def}} \begin{cases} a, & \textit{falls } x \textit{ ein Palindrom} \\ b, & \textit{sonst} \end{cases}$$

Idee:

- *Merke Buchstabe links im Zustand und lösche ihn.*
- *Vergleich mit Buchstaben rechts:*

  - *Wenn die Buchstaben (ganz links und ganz rechts) nicht übereinstimmen, dann alles löschen und b schreiben.*
  - *Bei Übereinstimmung letztes Zeichen löschen und ganz nach links wandern. Danach wird dieser Vergleichsprozess wiederholt.*

Die Zustände haben die folgende Bedeutung:

z_a*: Buchstabe a gemerkt,*

z_b*: Buchstabe b gemerkt,*

z'_a*: Ein Schritt nach links und a testen,*

z'_b*: Ein Schritt nach links und b testen,*

z_0*: Startzustand, d.h. „Merkprozess“ starten,*

z_1*: Endzustand,*

z_2*: Test positiv, d.h. nach links laufen und Prozess wiederholen und*

z_3*: Test negativ, alles löschen.*

Nun geben wir die Turingmaschine $M = (\Sigma, Z, \delta, z_0, z_1)$ *an, die die Funktion* palindrom$(\cdot)$ *berechnet. Dabei ist*

- $\Sigma = \{a, b, \square\}$,
- $Z = \{z_a, z_b, z'_a, z'_b, z_0, z_1, z_2, z_3\}$,
- δ *siehe Abbildung 17,*
- z_0 *Startzustand und*
- z_1 *Endzustand.*

5.2. Turing-Berechenbarkeit

Mit Hilfe von Turingmaschinen können wir eine (mathematische) Präzisierung des Algorithmen- und Berechenbarkeitsbegriffs geben:

- Ein *Algorithmus* ist das was mit einem Turing-Programm aufgeschrieben werden kann.
- Eine Funktion heißt *berechenbar*, wenn sie mit Hilfe einer Turingmaschine berechnet werden kann.

Der letzte Punkt der (informalen) Beschreibung des Berechenbarkeitsbegriffs soll nun genauer definiert werden:

$$
\begin{array}{llllll}
z_0 & \square & \to & z_1 & a & N \quad // \text{ leeres Wort} \\
z_0 & a & \to & z_a & \square & R \quad // \text{ a merken nach rechts} \\
z_0 & b & \to & z_b & \square & R \quad // \text{ b merken nach rechts} \\
 & & & & & \\
z_a & a & \to & z_a & a & R \quad // \text{ nach rechts, a gemerkt} \\
z_a & b & \to & z_a & b & R \\
z_a & \square & \to & z'_a & \square & L \quad // \text{ rechtes Ende gefunden} \\
z'_a & a & \to & z_2 & \square & L \quad // \text{ Vergleich positiv} \\
z'_a & b & \to & z_3 & \square & L \quad // \text{ Vergleich negativ} \\
 & & & & & \\
z'_a & \square & \to & z_1 & a & N \quad // \text{ War Palindrom} \\
 & & & & & \\
z_b & a & \to & z_b & a & R \quad // \text{ Nach rechts, b gemerkt} \\
z_b & b & \to & z_b & b & R \\
z_b & \square & \to & z'_b & \square & L \quad // \text{ rechtes Ende gefunden} \\
z'_b & b & \to & z_2 & \square & L \quad // \text{ Vergleich positiv} \\
z'_b & a & \to & z_3 & \square & L \quad // \text{ Vergleich negativ} \\
 & & & & & \\
z'_b & \square & \to & z_1 & a & N \quad // \text{ War Palindrom} \\
 & & & & & \\
z_2 & a & \to & z_2 & a & L \quad // \text{ Nach links und} \\
z_2 & b & \to & z_2 & b & L \quad // \text{ neuen Vergleich starten} \\
z_2 & \square & \to & z_0 & \square & R \\
 & & & & & \\
z_3 & a & \to & z_3 & \square & L \quad // \text{ Band löschen und nach links} \\
z_3 & b & \to & z_3 & \square & L \\
z_3 & \square & \to & z_1 & b & N
\end{array}
$$

Abbildung 17: Ein Turingprogramm zum Erkennen von Palindromen

Definition 84 (Turingberechenbarkeit): *Sei $n \in \mathbb{N}$ und $f\colon \mathbb{N}^n \to \mathbb{N}$. Die Funktion f heißt mit einer TM M* berechenbar*, wenn für die Eingaben $x_1, \ldots, x_n \in \mathbb{N}$ von f gilt:*

$$f(x_1, \ldots, x_n) = \begin{cases} \textit{Inhalt von Band } 1 \textit{ in Binärdarstellung} & \textit{falls, } M \textit{ in der Standardsituation mit den Argumenten } \mathrm{bin}(x_1) * \cdots * \mathrm{bin}(x_n) \textit{ auf Band } 1 \textit{ startet und nach endlich vielen Schritten in der Standardsituation hält, wobei sich die Ausgabe auf Band } 1 \textit{ befindet.} \\ \textit{nicht definiert} & \textit{sonst} \end{cases}$$

Mit $bin(x)$ bezeichnen wir die Binärdarstellung von x, d.h. z.B. $\mathrm{bin}(11) = 1011$*. Der Stern $*$ ist ein Trennsymbol um die einzelnen Teile der Eingabe zu separieren, d.h. eine Eingabe $5, 3, 7$ für eine dreistellige Funktion würde auf Band 1 als $\cdots \square 101 * 11 * 111 \square \cdots$ angegeben werden.*

Wir sagen eine Funktion $f\colon \mathbb{N}^n \to \mathbb{N}$ ist Turing-berechenbar*, falls es eine TM M gibt, die f berechnet und legen fest:*

$$\mathcal{TM} =_{\text{def}} \{f \mid f \textit{ ist Turing-berechenbar}\}$$

Bemerkung 85: *Die schon bekannten Funktionen*

$$\begin{aligned} \mathrm{inc}(x) &=_{\text{def}} x + 1 \\ \mathrm{palindrom}(x) &=_{\text{def}} \begin{cases} a, & \textit{falls } x \textit{ Palindrom} \\ b, & \textit{sonst} \end{cases} \end{aligned}$$

sind Turing-berechenbar, d.h. $\mathrm{inc}, \mathrm{palindrom} \in \mathcal{TM}$*.*

Bemerkung 86:

- *Sehr ähnlich zu der Definition der Turing-Berechenbarkeit können wir den Begriff der JAVA- bzw. C-Berechenbarkeit definieren (Eine Funktion f ist JAVA- bzw. C-berechenbar, wenn es ein JAVA- bzw. C-Programm gibt, das für jede Eingabe x den Funktionswert $f(x)$ liefert). Man kann mit einigem Aufwand (siehe [Wag03]) zeigen, dass die Menge der Turing-berechenbaren Funktionen gleich der Menge der JAVA- bzw. C-berechenbaren Funktionen ist.*
- *Für theoretische Studien reicht es also, Turing-Maschinen zu untersuchen, die wesentlich leichter zu analysieren sind als C-Programme oder JAVA-Programme.*

Wir können uns sogar auf den Fall $k = 1$ einschränken, ohne Berechnungskraft zu verlieren, denn es gilt der folgende Satz:

Satz 87: *Zu jeder Mehrband-TM M gibt es eine Einband-TM M', so dass M' die gleiche Funktion wie M berechnet.*

Beweis: siehe z.B. in [Sch01]. #

Aufgrund dieser Äquivalenzen und der Tatsache, dass (bisher) niemand einen mächtigeren Berechnungsbegriff finden konnte (trotz großer Anstrengungen), wird die folgende von Alonzo Church[14] aufgestellte These (immer noch) akzeptiert:

These 88 (Churchsche These): *Die durch die formale Definition der Turing-Berechenbarkeit (C/JAVA-Berechenbarkeit) erfasste Klasse von Funktionen entspricht genau der Klasse der im intuitiven Sinn berechenbaren Funktionen.*

Da wir den Begriff „intuitiv“ nicht formalisieren können, ist die Churchsche These nicht beweisbar, aber man geht heute von ihrer Richtigkeit aus.

[14] *1903 in Washington (USA) - †1995 in Hudson (USA)

5.3. Die Unentscheidbarkeit des Halteproblems

5.3.1. Die Entscheidbarkeit von Mengen

Definition 89 (Entscheidbarkeit von Mengen): *Eine Sprache $L \subseteq \Sigma^*$ heißt* entscheidbar*, wenn die charakteristische Funktion $c_L \colon \Sigma^* \to \{0,1\}$ von L berechenbar ist. Dabei gilt:*

$$c_L(w) =_{\text{def}} \begin{cases} 1, & \textit{falls } w \in L \\ 0, & \textit{falls } w \notin L \end{cases}$$

Beispiel 90: *Seien die folgenden Mengen gegeben:*

- $\mathsf{EVEN} =_{\text{def}} \{w \in \{0,1\}^+ \colon w \textit{ ist die Binärdarstellung einer geraden Zahl}\}$,
- $\mathsf{ODD} =_{\text{def}} \overline{\mathsf{EVEN}}$,
- $\mathsf{PARITY} =_{\text{def}} \{w \in \{0,1\}^+ \mid \textit{die } |w|_1 \textit{ ist ungerade}\}$ *und*
- $\mathsf{PRIM} =_{\text{def}} \{w \in \{0,1\}^+ \mid w \textit{ ist die Binärdarstellung einer Primzahl}\}$.

Die Mengen EVEN, ODD, PARITY *und* PRIM *sind entscheidbar.*

Definition 91 (Semi-Entscheidbarkeit von Mengen): *Eine Sprache $L \subseteq \Sigma^*$ heißt* semi-entscheidbar*, wenn die Funktion $c'_L \colon \Sigma^* \to \{0,1\}$ („halbe charakteristische Funktion“) von L berechenbar ist. Dabei gilt:*

$$c'_L(w) =_{\text{def}} \begin{cases} 1, & \textit{falls } w \in L \\ \text{undefiniert}, & \textit{falls } w \notin L \end{cases}$$

D.h. stoppt ein Algorithmus bei der Berechnung von c'_L, dann gilt $w \in L$. Stoppt die Berechnung für sehr lange Zeit nicht, so ist unklar, ob der Fall $w \notin L$ gilt (die Maschine hält nie an), oder ob nicht doch $w \in L$ gilt (und nur noch länger gerechnet werden muss).
Diese unbefriedigende Situation ist das Beste, was für viele praktisch relevante Probleme erreichbar ist. Im nächsten Abschnitt soll ein solches Problem genauer untersucht werden.

5.3.2. Die Gödelisierung von Turingmaschinen

Aus technischen Gründen wollen wir Turingmaschinen in natürliche Zahlen umwandeln.

Sei $M = (\Sigma, Z, \delta, z_0, z_1)$ eine 1-Band TM mit $\Sigma = \{a_0, \ldots, a_k\}$ und $Z = \{z_0, \ldots, z_l\}$, dann kann man die Übergangsfunktion δ wie folgt kodieren:

$$\delta(z_i, a_j) \ni (z_{i'}, a_{j'}, \sigma)$$

wird zu

$$\#\#\text{bin}(i)\#\text{bin}(j)\#\text{bin}(i')\#\text{bin}(j')\#\text{bin}(m),$$

wobei

$$m = \begin{cases} 0, & \text{falls } \sigma = L \\ 1, & \text{falls } \sigma = R \\ 2, & \text{falls } \sigma = N. \end{cases}$$

Dabei ist $\text{bin}(x)$ wieder die Binärkodierung von x. Die Befehle unserer Turingmaschine schreiben wir in einer festgelegten Reihenfolge auf und bekommen so ein Wort über $\{\#, 0, 1\}$. Nun kodieren wir

$$\begin{array}{lcl} 0 & \mapsto & 00, \\ 1 & \mapsto & 01 \text{ und} \\ \# & \mapsto & 11. \end{array}$$

Die so berechnete Zahl nennen wir *Gödelnummer*[15] der Maschine M. Die Konstruktion verdeutlicht, dass für jede beliebige Turingmaschine leicht ihre Gödelnummer berechenbar ist. Ebenfalls einsichtig ist, dass man auch aus einer Gödelnummer leicht wieder die ursprüngliche Turingmaschine gewinnen kann. Weiterhin kann man sich leicht überlegen, dass nicht jede natürliche Zahl in binärer Darstellung eine Turingmaschine beschreibt. Dieses Problem können wir aber relativ leicht reparieren. Sei $\hat{M}$ eine beliebige aber festgelegte Turingmaschine, dann sei

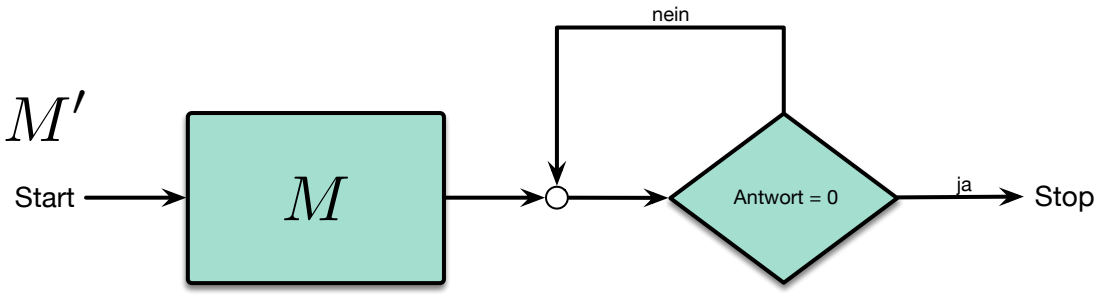
$$M_w =_{\text{def}} \begin{cases} M, & \text{falls } w \text{ die Gödelnummer der Turingmaschine } M \text{ ist} \\ \hat{M}, & \text{sonst.} \end{cases}$$

Mit der *Gödelisierung* ($\triangleq$ Kodierung einer Turingmaschine durch eine Gödelnummer) haben wir erreicht, dass eine Turingmaschine die Eingabe einer anderen Turingmaschine M sein kann, d.h. die Turingmaschine M kann Berechnungen durchführen, die bestimmte Eigenschaften dieser Turingmaschine testet. Mit Hilfe dieser Idee definieren wir

Definition 92 (Spezielles Halteproblem):

$$H =_{\text{def}} \{w \in \{0,1\}^* \mid M_w \textit{ hält mit der Eingabe } w \textit{ an}\}$$

Beweis: Angenommen H wäre entscheidbar, dann wäre die charakteristische Funktion c_H mit Hilfe der TM M berechenbar. Wir bauen nun die Turingmaschine M in die Maschine M' wie folgt um:



Damit ergibt sich die folgende Situation: Die Maschine M' stoppt gdw. M die Antwort 0 ausgibt. Berechnet M die Antwort 1, dann gelangt M' in eine Endlosschleife und hält nicht an. Sei w' die Gödelnummer von M', dann gilt:

M' mit Eingabe w' hält	gdw.	M mit Eingabe w' gibt 0 aus
	gdw.	$c_H(w') = 0$
	gdw.	$w' \notin H$
	gdw.	$M_{w'}$ hält bei Eingabe w' nicht an
	gdw.	M' mit Eingabe w' hält nicht.

Dies ist ein Widerspruch zu der ursprünglichen Annahme, d.h. die Annahme war falsch und es kann keine Turingmaschine geben, die die charakteristische Funktion c_H berechnet. Damit ist das spezielle Halteproblem H nicht entscheidbar. #

[15] Gödelnummern sind nach dem österreichisch-ungarischen Mathematiker Kurt Gödel (*1906 in Brno (Tschechien) - †1978 in Princeton (USA)) benannt, der Gödelnummern in der Logik einführte.

5.3.3. Eine (unvollständige) Liste nicht entscheidbarer Probleme

In diesem Abschnitt sollen einige Probleme vorgestellt werden, von den bekannt ist, dass sie nicht entscheidbar sind.

Zunächst benötigen wir noch folgende Notation für den Definitionsbereich einer Turingmaschine, der analog zu dem Definitionsbereich einer Funktion definiert wird:

Definition 93: *Sei $e \in \mathbb{N}$, dann heißt*

$$W_e =_{\text{def}} \{x \mid \textit{Maschine } M_e \textit{ hält auf Eingabe } x\}$$

Definitionsbereich *der Maschine M_e.*

PROBLEM:	H′ (Halteproblem)
EINGABE:	Gödelnummer $e \in \mathbb{N}$ und ein $x \in \mathbb{N}$
FRAGE:	Hält $M_e(x)$?

PROBLEM:	H (spezielles Halteproblem)
EINGABE:	Eine Gödelnummer $e \in \mathbb{N}$
FRAGE:	Hält $M_e(e)$?

PROBLEM:	SEQ
EINGABE:	Zwei Gödelnummern $e, d \in \mathbb{N}$
FRAGE:	Gilt $W_d \subseteq W_e$?

PROBLEM:	EQ
EINGABE:	Zwei Gödelnummern $e, d \in \mathbb{N}$
FRAGE:	Gilt $W_d = W_e$?

PROBLEM:	TOT
EINGABE:	Eine Gödelnummer $e \in \mathbb{N}$
FRAGE:	Berechnet M_e totale Funktion?

PROBLEM:	FIN
EINGABE:	Eine Gödelnummer $e \in \mathbb{N}$
FRAGE:	Ist W_e endlich?

PROBLEM:	COF
EINGABE:	Eine Gödelnummer $e \in \mathbb{N}$
FRAGE:	Ist $\overline{W_e}$ endlich?

PROBLEM:	HILBERT10
EINGABE:	Eine Diophantische Gleichung $p(x_1, \ldots, x_n) = 0$
FRAGE:	Hat die Gleichung eine Lösung?

PROBLEM:	PCP
EINGABE:	Eine endliche Menge $\{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$ von Wortpaaren
FRAGE:	Gibt es eine Folge von Indizes $i_1, \ldots, i_n \in \{1, 2, \ldots, k\}$ und $n \geq 1$, sodass $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$?

PROBLEM:	CFG − CUT
EINGABE:	Zwei kontextfreie Grammatiken G_1 und G_2
FRAGE:	$L(G_1) \cap L(G_2) = \emptyset$?

PROBLEM:	CFG − INF
EINGABE:	Zwei kontextfreie Grammatiken G_1 und G_2
FRAGE:	$\lvert L(G_1) \cap L(G_2) \rvert = \infty$?

PROBLEM:	CFG − SEQ
EINGABE:	kontextfreie Grammatiken G_1, G_2
FRAGE:	$L(G_1) \subseteq L(G_2)$?

PROBLEM:	CFG − EQ
EINGABE:	kontextfreie Grammatiken G_1, G_2
FRAGE:	$L(G_1) = L(G_2)$?

PROBLEM:	ISCFG
EINGABE:	Zwei kontextfreie Grammatiken G_1 und G_2
FRAGE:	Gibt es eine kontextfreie Grammatik G mit $L(G) = L(G_1) \cap L(G_2)$?

Dabei ist PCP die Abkürzung für Post's Correspondence Problem. EMIL LEON POST[16] war ein Mathematiker und Logiker, der bedeutende Ergebnisse in der Berechenbarkeits- und in der Gruppentheorie

[16]*1897 in Augustów (Polen) - † 1954 in New York (USA)

erzielt hat.

5.3.4. Ein Beispiel für eine nicht berechenbare Funktion

Nachdem wir mit dem Halteproblem ein erstes nicht entscheidbares Problem kennen gelernt haben, soll hier noch ein Beispiel für eine nicht berechenbare Funktion beschrieben werden. Diese Funktion wurde von T. Rado 1962 als Beispiel für eine nicht berechenbare Funktion angegeben (vgl. [Rad62]). Heute ist sie unter dem Namen *Busy-Beaver Funktion* bekannt.

Eine TM M mit einem Band und dem Alphabet $\Sigma = \{|, \square\}$ wird als *Busy-Beaver-Kandidat* (kurz *BBK*) bezeichnet.

$$\beta(n) \quad =_{\text{def}} \max\{k \mid \text{es gibt einen BBK mit } n \text{ Zuständen, der mit leerer Eingabe hält und } k \text{ Striche „|“ auf sein Band schreibt}\}$$

Satz 94: *Die Busy-Beaver Funktion $\beta(n)$ ist nicht berechenbar.*

Beweis: Wir nehmen an, dass die Funktion $\beta(n)$ mit Hilfe der TM M berechenbar ist, die k Zustände hat, d.h. die Maschine M hält, wenn Sie mit n Strichen auf dem Band gestartet wird, mit $\beta(n)$ Strichen auf dem Band.

- Sei M' die TM, die m Zustände hat und m Striche auf das **leere** Band schreibt.
- Sei M'' die TM, die c Zustände hat und die die Anzahl von Strichen auf dem Band verdoppelt.

Wenn wir mit ϵ die leere Eingabe bezeichnen, so gilt

$$\underbrace{M(M''(\underbrace{M'(\epsilon)}_{\text{schreibe } m \text{ Striche}}))}_{\text{verdopple die Anzahl}} = \beta(2 \cdot m)$$

Nun wählen wir $m > c + k$. Dann gilt $\beta(2 \cdot m) > \beta(m + c + k)$. Da $\beta(m + c + k)$ die maximale Anzahl von Strichen ist, die mit $m+c+k$ Zuständen geschrieben werden kann, kann die Maschine $M \circ M'' \circ M'$[17] nicht mehr Striche auf das Band schreiben.

Das ist aber ein Widerspruch, denn wäre unsere Annahme richtig, dann würde die Maschine $M \circ M'' \circ M'$ ja $\beta(2 \cdot m)$ Striche auf das Band schreiben. Damit kann die Maschine $M \circ M'' \circ M'$ nicht existieren. Da aber die Maschinen M' und M'' sicherlich leicht gebaut werden können, kann nur die Maschine M nicht existieren. Damit ist aber unsere Annahme falsch, d.h. $\beta(n)$ kann mit Hilfe der TM M nicht berechnet werden. #

Wir haben gesehen, dass keine Turingmaschine existieren kann, die die Busy-Beaver Funktion $\beta(n)$ berechnet. Damit gibt es auch kein C-Programm für die β-Funktion. Für einige kleine n ist der Wert der β-Funktion bekannt:

n	$\beta(n)$	Autor
1	1	Lin und Rado
2	4	Lin und Rado
3	6	Lin und Rado
4	13	Brady
5	$\geq$ 4098	Marxen und Buntrock
6	$\geq$ 95 524 079	Marxen

[17] Dies ist die TM, die durch Verkettung der drei Maschinen M, M' und M'' entsteht

Für $1 \leq n \leq 4$ sind die Busy-Beaver bekannt. Für $n = 5$ ist nur die untere Schranke 4098 bekannt. Es wäre sehr interessant diese untere Schranke[18] zu verbessern.

6. Komplexität

Für viele ständig auftretende Berechnungsprobleme, wie Sortieren, die arithmetischen Operationen (Addition, Multiplikation, Division), Fourier-Transformation etc., sind sehr effiziente Algorithmen konstruiert worden. Für wenige andere praktische Probleme weiß man, dass sie nicht oder nicht effizient algorithmisch lösbar sind. Im Gegensatz dazu ist für einen sehr großen Teil von Fragestellungen aus den verschiedensten Anwendungsbereichen wie Operations Research, Netzwerkdesign, Programmoptimierung, Datenbanken, Betriebssystem-Entwicklung und vielen mehr jedoch nicht bekannt, ob sie effiziente Algorithmen besitzen (vgl. die Abbildungen 22 und 23). Diese Problematik hängt mit der seit über vierzig Jahren[19] offenen $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage zusammen, wahrscheinlich gegenwärtig das wichtigste ungelöste Problem der theoretischen Informatik. Es wurde sogar vor einiger Zeit auf Platz 1 der Liste der so genannten *Millennium Prize Problems* des Clay Mathematics Institute gesetzt[20]. Diese Liste umfasst sieben offene Probleme aus der gesamten Mathematik. Das Clay Institute zahlt jedem, der eines dieser Probleme löst, eine Million US-Dollar.

In diesem Abschnitt werden die wesentlichen Begriffe aus dem Kontext des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems und des Begriffes der **NP**-Vollständigkeit erläutert, um so die Grundlagen für das Verständnis derartiger Probleme zu schaffen und deren Beurteilung zu ermöglichen.

6.1. Effizient lösbare Probleme: die Klasse P

Jeder, der schon einmal mit der Aufgabe konfrontiert wurde, einen Algorithmus für ein gegebenes Problem zu entwickeln, kennt die Hauptschwierigkeit dabei: Wie kann ein effizienter Algorithmus gefunden werden, der das Problem mit möglichst wenigen Rechenschritten löst? Um diese Frage beantworten zu können, muss man sich zunächst einige Gedanken über die verwendeten Begriffe, nämlich „Problem“, „Algorithmus“, „Zeitbedarf“ und „effizient“, machen.

Was ist ein „Problem“? Jedem Programmierer ist diese Frage intuitiv klar: Man bekommt geeignete Eingaben, und das Programm soll die gewünschten Ausgaben ermitteln. Ein einfaches Beispiel ist das Problem MULT. (Jedes Problem soll mit einem eindeutigen Namen versehen und dieser in Großbuchstaben geschrieben werden.) Hier bekommt man zwei ganze Zahlen als Eingabe und soll das Produkt beider Zahlen berechnen, d.h. das Programm berechnet einfach eine zweistellige Funktion. Es hat sich gezeigt, dass man sich bei der Untersuchung von Effizienzfragen auf eine abgeschwächte Form von Problemen beschränken kann, nämlich sogenannte *Entscheidungsprobleme*. Hier ist die Aufgabe, eine gewünschte Eigenschaft der Eingaben zu testen. Hat die aktuelle Eingabe die gewünschte Eigenschaft, dann gibt man den Wert 1 ($\triangleq$ `true`) zurück (man spricht dann auch von einer *positiven Instanz* des Problems), hat die Eingabe die Eigenschaft nicht, dann gibt man den Wert 0 ($\triangleq$ `false`) zurück. Oder anders formuliert: Das Programm berechnet eine Funktion, die den Wert 0 oder 1 zurück gibt und partitioniert damit

[18] Man suche eine TM mit 5 Zuständen, die mehr als 4098 Striche auf das Band schreibt und dann anhält, um so „Buntrock & Marxen“ zu schlagen. Als Alternative zeigt man, dass die Maschine von Buntrock & Marxen ein Busy Beaver mit 5 Zuständen ist.

[19] Interessanterweise reicht die Geschichte des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems viel weiter in die Geschichte zurück. So fragte 1956 Kurt Gödel in einem Brief an John von Neumann schon „Given a first-order formula F and a natural number n, determine if there is a proof of F of length n“. Gödel war an der Anzahl von Schritten interessiert, die eine Turing-Maschine benötigt, um dieses Problem zu lösen. Dazu fragte er, ob dies in linearer oder quadratischer Zeit möglich ist, was man als eine Frühform der $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Frage auffassen könnte.

[20] siehe `http://www.claymath.org/millennium-problems`

die Menge der möglichen Eingaben in zwei Teile: die Menge der Eingaben mit der gewünschten Eigenschaft und die Menge der Eingaben, die die gewünschte Eigenschaft nicht besitzen. Folgendes Beispiel soll das Konzept verdeutlichen:

PROBLEM: PARITY
EINGABE: Positive Integerzahl x
AUSGABE: Ist die Anzahl der Ziffern 1 in der Binärdarstellung von x ungerade?

Es soll also ein Programm entwickelt werden, das die Parität einer Integerzahl x berechnet. Eine mögliche Entscheidungsproblem-Variante des Problems MULT ist die folgende:

PROBLEM: MULT_D
EINGABE: Integerzahlen x, y, positive Integerzahl i
AUSGABE: Ist das i-te Bit in $x \cdot y$ gleich 1?

Offensichtlich sind die Probleme MULT und MULT_D gleich schwierig (oder leicht) zu lösen.

Im Weiteren wollen wir uns hauptsächlich mit Problemen beschäftigen, die aus dem Gebiet der Graphentheorie stammen. Das hat zwei Gründe. Zum einen können, wie sich noch zeigen wird, viele praktisch relevante Probleme mit Hilfe von Graphen modelliert werden, und zum anderen sind sie anschaulich und oft relativ leicht zu verstehen. Ein (ungerichteter) Graph G besteht aus einer Menge von Knoten V und einer Menge von Kanten E, die diese Knoten verbinden. Man schreibt: $G = (V, E)$. Ein wohlbekanntes Beispiel ist der Nikolausgraph: $G_N = \big(\{1,2,3,4,5\}, \{(1,2),(1,3),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)\}\big)$. Es gibt also fünf Knoten $V = \{1,2,3,4,5\}$, die durch die Kanten in $E = \{(1,2),(1,3),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5)\}$ verbunden werden (siehe Abbildung 18).

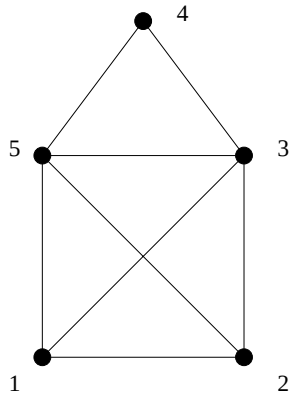


Abbildung 18: Der Graph G_N

Ein prominentes Problem in der Graphentheorie ist es, eine sogenannte Knotenfärbung zu finden. Dabei wird jedem Knoten eine Farbe zugeordnet, und man verbietet, dass zwei Knoten, die durch eine Kante verbunden sind, die gleiche Farbe zugeordnet wird. Natürlich ist die Anzahl der Farben, die verwendet werden dürfen, durch eine feste natürliche Zahl k beschränkt. Genau wird das Problem, ob ein Graph k-färbbar ist, wie folgt beschrieben:

PROBLEM: kCOL
EINGABE: Ein Graph $G = (V, E)$
AUSGABE: Hat G eine Knotenfärbung mit höchstens k Farben?

Offensichtlich ist der Beispielgraph G_N nicht mit drei Farben färbbar (aber mit 4 Farben, wie man leicht ausprobieren kann), und jedes Programm für das Problem 3COL müsste ermitteln, dass G_N die gewünschte Eigenschaft (3-Färbbarkeit) nicht hat.

Man kann sich natürlich fragen, was das künstlich erscheinende Problem 3COL mit der Praxis zu tun hat. Das folgende einfache Beispiel soll das verdeutlichen. Man nehme das Szenario an, dass ein großer Telefonprovider in einer Ausschreibung drei Funkfrequenzen für einen neuen Mobilfunkstandard erworben hat. Da er schon über ein Mobilfunknetz verfügt, sind die Sendemasten schon gebaut. Aus technischen Gründen dürfen Sendemasten, die zu eng stehen, nicht mit der gleichen Frequenz funken, da sie sich sonst stören würden. In der graphentheoretischen Welt modelliert man die Sendestationen mit Knoten eines Graphen, und „nahe“ zusammenstehende Sendestationen symbolisiert man mit einer Kante zwischen den Knoten, für die sie stehen.

Die Aufgabe des Mobilfunkplaners ist es nun, eine 3-Färbung für den entstehenden Graphen zu finden. Offensichtlich kann das Problem verallgemeinert werden, wenn man sich nicht auf drei Farben/Frequenzen festlegt; dann aber ergeben sich genau die oben definierten Probleme kCOL für beliebige Zahlen k.

Als nächstes ist zu klären, was unter einem „Algorithmus“ zu verstehen ist. Ein Algorithmus ist eine endliche, formale Beschreibung einer Methode, die ein Problem löst (z.B. ein Programm in einer beliebigen Programmiersprache). Diese Methode muss also alle Eingaben mit der gesuchten Eigenschaft von den Eingaben, die diese Eigenschaft nicht haben, unterscheiden können. Man legt fest, dass der Algorithmus für erstere den Wert 1 und für letztere den Wert 0 ausgeben soll. Wie soll die „Laufzeit“ eines Algorithmus gemessen werden? Um dies festlegen zu können, muss man sich zunächst auf ein sogenanntes *Berechnungsmodell* festlegen. Das kann man damit vergleichen, welche Hardware für die Implementation des Algorithmus verwendet werden soll. Für die weiteren Analysen soll das folgende einfache C-artige Modell verwendet werden: Es wird (grob!) die Syntax von C verwendet und festgelegt, dass jede Anweisung in einem Schritt abgearbeitet werden kann. Gleichzeitig beschränkt man sich auf zwei Datentypen: einen Integer-Typ und zusätzlich Arrays dieses Integer-Typs (wobei Array-Grenzen nicht deklariert werden müssen, sondern sich aus dem Gebrauch ergeben). Dieses primitive Maschinenmodell ist deshalb geeignet, weil man zeigen kann, dass jeder so formulierte Algorithmus auf realen Computern implementiert werden kann, ohne eine substantielle Verlangsamung zu erfahren. (Dies gilt zumindest, wenn die verwendeten Zahlen nicht übermäßig wachsen, d.h., wenn alle verwendeten Variablen nicht zu viel Speicher belegen. Genaueres zu dieser Problematik – man spricht von der Unterscheidung zwischen *uniformem Komplexitätsmaß* und *Bitkomplexität* – findet sich in [Sch01, S. 62f].)

Umgekehrt kann man ebenfalls sagen, dass dieses einfache Modell die Realität genau genug widerspiegelt, da auch reale Programme ohne allzu großen Zeitverlust auf diesem Berechnungsmodell simuliert werden können. Offensichtlich ist die *Eingabe* der Parameter, von dem die Rechenzeit für einen festen Algorithmus abhängt. In den vergangenen Jahrzehnten, in denen das Gebiet der Analyse von Algorithmen entstand, hat die Erfahrung gezeigt, dass die Länge der Eingabe, also die Anzahl der Bits, die benötigt werden um die Eingabe zu speichern, ein geeignetes und robustes Maß ist, in der die Rechenzeit gemessen werden kann. Auch der Aufwand, die Eingabe selbst festzulegen (zu konstruieren), hängt schließlich von ihrer Länge ab, nicht davon, ob sich irgendwo in der Eingabe eine 0 oder 1 befindet.

6.1.1. Das Problem der 2-Färbbarkeit

Das Problem der 2-Färbbarkeit ist wie folgt definiert:

Problem:	2COL
Eingabe:	Ein Graph $G = (V, E)$
Ausgabe:	Hat G eine Knotenfärbung mit höchstens 2 Farben?

Es ist bekannt, dass dieses Problem mit einem sogenannten *Greedy-Algorithmus* gelöst werden kann: Beginne mit einem beliebigen Knoten in G (z.B. v_1) und färbe ihn mit Farbe 1. Färbe dann die Nachbarn dieses Knoten mit 2, die Nachbarn dieser Nachbarn wieder mit 1, usw. Falls G aus mehreren Komponenten (d.h. zusammenhängenden Teilgraphen) besteht, muss dieses Verfahren für jede Komponente wiederholt werden. G ist schließlich 2-färbbar, wenn bei obiger Prozedur keine inkorrekte Färbung entsteht. Diese Idee führt zu Algorithmus 3.

Die Laufzeit von Algorithmus 3 kann wie folgt abgeschätzt werden: Die erste **for**-Schleife benötigt n Schritte. In der **while**-Schleife wird entweder mindestens ein Knoten gefärbt und die

Schleife dann erneut ausgeführt, oder es wird kein Knoten gefärbt und die Schleife dann verlassen; also wird diese Schleife höchstens n-mal ausgeführt. Innerhalb der **while**-Schleife finden sich drei ineinander verschachtelte **for**-Schleifen, die alle jeweils n-mal durchlaufen werden, und eine **while**-Schleife, die maximal n-mal durchlaufen wird.

Damit ergibt sich also eine Gesamtlaufzeit der Größenordnung n^4, wobei n die Anzahl der Knoten des Eingabe-Graphen G ist. Wie groß ist nun die Eingabelänge, also die Anzahl der benötigten Bits zur Speicherung von G? Sicherlich muss jeder Knoten in dieser Speicherung vertreten sein, d.h. also, dass mindestens n Bits zur Speicherung von G benötigt werden. Die Eingabelänge ist also mindestens n. Daraus folgt, dass die Laufzeit des Algorithmus also höchstens von der Größenordnung N^4 ist, wenn N die Eingabelänge bezeichnet.

Tatsächlich sind (bei Verwendung geeigneter Datenstrukturen wie Listen oder Queues) wesentlich effizientere Verfahren für 2COL möglich. Aber auch schon das obige einfache Verfahren zeigt: 2COL hat einen Polynomialzeitalgorithmus, also 2COL $\in \mathbf{P}$.

Alle Probleme für die Algorithmen existieren, die eine Anzahl von Rechenschritten benötigen, die durch ein beliebiges Polynom beschränkt ist, bezeichnet man mit **P** („**P**“ steht dabei für „**P**olynomialzeit“). Auch dabei wird die Rechenzeit in der Länge der Eingabe gemessen, d.h. in der Anzahl der Bits, die benötigt werden, um die Eingabe zu speichern (zu kodieren). Die Klasse **P** wird auch als Klasse der *effizient lösbaren Probleme* bezeichnet. Dies ist natürlich wieder eine idealisierte Auffassung: Einen Algorithmus mit einer Laufzeit n^{57}, wobei n die Länge der Eingabe bezeichnet, kann man schwer als effizient bezeichnen. Allerdings hat es sich in der Praxis gezeigt, dass für fast alle bekannten Probleme in **P** auch Algorithmen existieren, deren Laufzeit durch ein Polynom kleinen[21] Grades beschränkt ist.

In diesem Licht ist die Definition der Klasse **P** auch für praktische Belange von Relevanz. Dass eine polynomielle Laufzeit etwas substanziell Besseres darstellt als exponentielle Laufzeit (hier beträgt die benötigte Rechenzeit $2^{c \cdot n}$ für eine Konstante c, wobei n wieder die Länge der Eingabe bezeichnet), zeigt die Tabelle „Rechenzeitbedarf von Algorithmen“. Zu beachten ist, dass bei einem Exponentialzeit-Algorithmus mit $c = 1$ eine Verdoppelung der „Geschwindigkeit“ der verwendeten Maschine (also Taktzahl pro Sekunde) es nur erlaubt, eine um höchstens 1 Bit längere Eingabe in einer bestimmten Zeit zu bearbeiten. Bei einem Linearzeit-Algorithmus hingegen verdoppelt sich auch die mögliche Eingabelänge; bei einer Laufzeit von n^k vergrößert sich die mögliche Eingabelänge immerhin noch um den Faktor $\sqrt[k]{2}$. Deswegen sind Probleme, für die nur Exponentialzeit-Algorithmen existieren, praktisch nicht lösbar; daran ändert sich auch nichts Wesentliches durch die Einführung von immer schnelleren Rechnern.

Nun stellt sich natürlich sofort die Frage: Gibt es für jedes Problem einen effizienten Algorithmus? Man kann relativ leicht zeigen, dass die Antwort auf diese Frage „Nein“ ist. Die Schwierigkeit bei dieser Fragestellung liegt aber darin, dass man von vielen Problemen *nicht weiß*, ob sie effizient lösbar sind. Ganz konkret: Ein effizienter Algorithmus für das Problem 2COL ist Algorithmus 3. Ist es möglich, ebenfalls einen Polynomialzeitalgorithmus für 3COL zu finden? Viele Informatiker beschäftigen sich seit den 60er Jahren des letzten Jahrhunderts intensiv mit dieser Frage. Dabei kristallisierte sich heraus, dass viele praktisch relevante Probleme, für die kein effizienter Algorithmus bekannt ist, eine gemeinsame Eigenschaft besitzen, nämlich die der *effizienten Überprüfbarkeit* von geeigneten Lösungen. Auch 3COL gehört zu dieser Klasse von Problemen, wie sich in Kürze zeigen wird. Aber wie soll man zeigen, dass für ein Problem kein effizienter Algorithmus existiert? Nur weil kein Algorithmus bekannt ist, bedeutet das noch nicht, dass keiner existiert.

Es ist bekannt, dass die oberen Schranken (also die Laufzeit von bekannten Algorithmen) und die unteren Schranken (mindestens benötigte Laufzeit) für das Problem PARITY (und einige we-

[21] Aktuell *scheint* es kein praktisch relevantes Problem aus **P** zu geben, für das es keinen Algorithmus mit einer Laufzeit von weniger als n^{12} gibt.

Algorithmus 3: Algorithmus zur Berechnung einer 2-Färbung eines Graphen

```
Data: Graph G = ({v_1, ... v_n}, E);
Result: 1 wenn es eine 2-Färbung für G gibt, 0 sonst
begin
    for (i = 1 to n) do
        Farbe[i] = 0;
    end
    Farbe[1] = 1;
    repeat
        aktKompoBearbeiten = false;
        for (i = 1 to n) do
            for (j = 1 to n) do
                                          /* Kante mit noch ungefärbten Knoten? */
                if (((v_i, v_j) ∈ E) und (Farbe[i] ≠ 0) und (Farbe[j] = 0)) then
                                          /* v_j bekommt eine andere Farbe als v_i */
                    Farbe[j] = 3 − Farbe[i];
                    aktKompoBearbeiten = true;
                                        /* Alle direkten Nachbarn von v_j prüfen */
                    for (k = 1 to n) do
                                          /* Kollision beim Färben aufgetreten? */
                        if (((v_j, v_k) ∈ E) und (Farbe[i] = Farbe[k])) then
                                            /* Kollision! Graph nicht 2-färbbar */
                            return 0;
                        end
                    end
                end
            end
        end
                /* Ist die aktuelle Zusammenhangskomponente völlig gefärbt? */
        if (not(aktKompoBearbeiten)) then
            i = 1;
                 /* Suche nach einer weiteren Zusammenhangskomponente von G */
            repeat
                  /* Liegt v_i in einer neuen Zusammenhangskomponente von G? */
                if (Farbe[i] = 0) then
                    Farbe[i] = 1;
                                    /* Neue Zusammenhangskomponente bearbeiten */
                    aktKompoBearbeiten = true;
                        /* Suche nach neuer Zusammenhangskomponente abbrechen */
                    weiterSuchen = false;
                end
                i = i + 1;
            until (not(weiterSuchen) und (i ≤ n));
        end
    until (aktKompoBearbeiten);
                                                        /* 2-Färbung gefunden */
    return 1.
end
```

Anzahl der Takte	Eingabelänge n					
	10	20	30	40	50	60
n	0,00001 Sekunden	0,00002 Sekunden	0,00003 Sekunden	0,00004 Sekunden	0,00005 Sekunden	0,00006 Sekunden
n^2	0,0001 Sekunden	0,0004 Sekunden	0,0009 Sekunden	0,0016 Sekunden	0,0025 Sekunden	0,0036 Sekunden
n^3	0,001 Sekunden	0,008 Sekunden	0,027 Sekunden	0,064 Sekunden	0,125 Sekunden	0,216 Sekunden
n^5	0,1 Sekunden	3,2 Sekunden	24,3 Sekunden	1,7 Minuten	5,2 Minuten	13,0 Minuten
2^n	0,001 Sekunden	1 Sekunde	17,9 Minuten	12,7 Tage	35,7 Jahre	366 Jahrhunderte
3^n	0,059 Sekunden	58 Minuten	6,5 Jahre	3855 Jahrhunderte	$2 \cdot 10^8$ Jahrhunderte	$1,3 \cdot 10^{13}$ Jahrhunderte

Abbildung 19: Rechenzeitbedarf von Algorithmen auf einem „1-MIPS“-Rechner

nige weitere, ebenfalls sehr einfach-geartete Probleme) sehr nahe zusammen liegen. Das bedeutet also, dass nur noch unwesentliche Verbesserungen der Algorithmen für des PARITY-Problem erwartet werden können. Beim Problem 3COL ist das ganz anders: Die bekannten unteren und oberen Schranken liegen extrem weit auseinander. Deshalb ist nicht klar, ob nicht doch (extrem) bessere Algorithmen als heute bekannt im Bereich des Möglichen liegen. Aber wie untersucht man solch eine Problematik? Man müsste ja über unendlich viele Algorithmen für das Problem 3COL Untersuchungen anstellen. Dies ist äußerst schwer zu handhaben und deshalb ist der einzige bekannte Ausweg, das Problem mit einer Reihe von weiteren (aus bestimmten Gründen) interessierenden Problemen zu vergleichen und zu zeigen, dass unser zu untersuchendes Problem nicht leichter zu lösen ist als diese anderen. Hat man das geschafft, ist eine untere Schranke einer speziellen Art gefunden: Unser Problem ist nicht leichter lösbar, als alle Probleme der Klasse von Problemen, die für den Vergleich herangezogen wurden. Nun ist aus der Beschreibung der Aufgabe aber schon klar, dass auch diese Aufgabe schwierig zu lösen ist, weil ein Problem nun mit unendlich vielen anderen Problemen zu vergleichen ist. Es zeigt sich aber, dass diese Aufgabe nicht aussichtslos ist. Bevor diese Idee weiter ausgeführt wird, soll zunächst die Klasse von Problemen untersucht werden, die für diesen Vergleich herangezogen werden sollen, nämlich die Klasse **NP**.

6.2. Effizient überprüfbare Probleme: die Klasse NP

Wie schon erwähnt, gibt es eine große Anzahl verschiedener Probleme, für die kein effizienter Algorithmus bekannt ist, die aber eine gemeinsame Eigenschaft haben: die *effiziente Überprüfbarkeit von Lösungen* für dieses Problem. Diese Eigenschaft soll an dem schon bekannten Problem 3COL veranschaulicht werden: Angenommen, man hat einen beliebigen Graphen G gegeben; wie bereits erwähnt ist kein effizienter Algorithmus bekannt, der entscheiden kann, ob der Graph G eine 3-Färbung hat (d.h., ob der fiktive Mobilfunkprovider mit 3 Funkfrequenzen auskommt). Hat man aber aus irgendwelchen Gründen eine *potenzielle* Knotenfärbung vorliegen, dann ist es leicht, diese potenzielle Knotenfärbung zu überprüfen und festzustellen, ob sie eine *korrekte* Färbung des Graphen ist, wie Algorithmus 4 zeigt.

Das Problem 3COL hat also die Eigenschaft, dass eine potenzielle Lösung leicht daraufhin überprüft werden kann, ob sie eine tatsächliche, d.h. korrekte, Lösung ist. Viele andere praktisch

Algorithmus 4: Ein Algorithmus zur Überprüfung einer potentiellen Färbung

Data: Graph $G = (\{v_1, \dots v_n\}, E)$ und eine potenzielle Knotenfärbung
Result: 1 wenn die Färbung korrekt, 0 sonst

```
begin
    /* Teste systematisch alle Kanten */
    for (i = 1 to n) do
        for (j = 1 to n) do
            /* Test auf Verletzung der Knotenfärbung */
            if (((v_i, v_j) ∈ E) und (v_i und v_j sind gleich gefärbt)) then
                return 0;
            end
        end
    end
    return 1;
end
```

relevante Probleme, für die kein effizienter Algorithmus bekannt ist, besitzen ebenfalls diese Eigenschaft. Dies soll noch an einem weiteren Beispiel verdeutlicht werden, dem so genannten *Hamiltonkreis-Problem*.

Sei wieder ein Graph $G = \big(\{v_1, \dots v_n\}, E\big)$ gegeben. Diesmal ist eine Rundreise entlang der Kanten von G gesucht, die bei einem Knoten v_{i_1} aus G startet, wieder bei v_{i_1} endet und jeden Knoten genau einmal besucht. Genauer wird diese Rundreise im Graphen G durch eine Folge von n Knoten $(v_{i_1}, v_{i_2}, v_{i_3}, \dots, v_{i_{n-1}}, v_{i_n},)$ beschrieben, wobei gelten soll, dass alle Knoten $v_{i_1}, \dots, v_{i_n}$ verschieden sind und die Kanten $(v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n})$ und (v_{i_n}, v_{i_1}) in G vorkommen. Eine solche Folge von Kanten wird als *Hamiltonscher Kreis* bezeichnet. Ein Hamiltonscher Kreis in einem Graphen G ist also ein Kreis, der jeden Knoten des Graphen genau einmal besucht. Das Problem, einen Hamiltonschen Kreis in einem Graphen zu finden, bezeichnet man mit HAMILTON:

PROBLEM:	HAMILTON
EINGABE:	Ein Graph G
FRAGE:	Hat G einen Hamiltonschen Kreis?

Auch für dieses Problem ist kein effizienter Algorithmus bekannt. Aber auch hier ist offensichtlich: Bekommt man einen Graphen gegeben und eine Folge von Knoten, dann kann man sehr leicht überprüfen, ob sie ein Hamiltonscher Kreis ist – dazu ist lediglich zu testen, ob alle Knoten genau einmal besucht werden und auch alle Kanten im gegebenen Graphen vorhanden sind.

Hat man erst einmal die Beobachtung gemacht, dass viele Probleme die Eigenschaft der effizienten Überprüfbarkeit haben, ist es naheliegend, sie in einer Klasse zusammenzufassen und gemeinsam zu untersuchen. Die Hoffnung dabei ist, dass sich alle Aussagen, die man über diese Klasse herausfindet, sofort auf alle Probleme anwenden lassen. Solche Überlegungen führten zur Geburt der Klasse **NP**, in der man alle effizient überprüfbaren Probleme zusammenfasst. Aber wie kann man solch eine Klasse untersuchen? Man hat ja noch nicht einmal ein Maschinenmodell (oder eine Programmiersprache) zur Verfügung, um solch eine Eigenschaft zu modellieren. Um ein Programm für effizient überprüfbare Probleme zu schreiben, braucht man erst eine Möglichkeit, die zu überprüfenden möglichen Lösungen zu ermitteln und sie dann zu testen, d.h. man muss die Programmiersprache für **NP** in einer geeigneten Weise mit mehr „Berechnungskraft“ ausstatten.

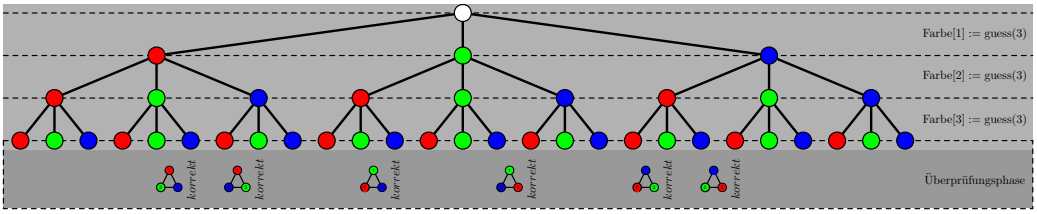


Abbildung 20: Ein Berechnungsbaum für das 3COL-Problem

Die erste Lösungsidee für **NP**-Probleme, nämlich alle in Frage kommenden Lösungen in einer **for**-Schleife aufzuzählen, führt zu Exponentialzeit-Lösungsalgorithmen, denn es gibt im Allgemeinen einfach so viele potenzielle Lösungen. Um erneut auf das Problem 3COL zurückzukommen: Angenommen, G ist ein Graph mit n Knoten. Dann gibt es 3^n potenzielle Färbungen, die überprüft werden müssen, denn es gibt 3 Möglichkeiten den ersten Knoten zu färben, 3 Möglichkeiten den zweiten Knoten zu färben, usw., und damit 3^n viele zu überprüfende potenzielle Färbungen. Würde man diese in einer **for**-Schleife aufzählen und auf Korrektheit testen, so führte das also zu einem Exponentialzeit-Algorithmus. Auf der anderen Seite gibt es aber Probleme, die in Exponentialzeit gelöst werden können, aber nicht zu der Intuition der effizienten Überprüfbarkeit der Klasse **NP** passen. Das Berechnungsmodell für **NP** kann also nicht einfach so gewonnen werden, dass exponentielle Laufzeit zugelassen wird, denn damit wäre man über das Ziel hinausgeschossen.

Hätte man einen Parallelrechner zur Verfügung mit so vielen Prozessoren wie es potenzielle Lösungen gibt, dann könnte man das Problem schnell lösen, denn jeder Prozessor kann unabhängig von allen anderen Prozessoren eine potenzielle Färbung überprüfen. Es zeigt sich aber, dass auch dieses Berechnungsmodell zu mächtig wäre. Es gibt Probleme, die wahrscheinlich nicht im obigen Sinne effizient überprüfbar sind, aber mit solch einem Parallelrechner trotzdem (effizient) gelöst werden könnten. In der Praxis würde uns ein derartiger paralleler Algorithmus auch nichts nützen, da man einen Rechner mit exponentiell vielen Prozessoren, also enormem Hardwareaufwand, zu konstruieren hätte. Also muss auch dieses Berechnungsmodell wieder etwas schwächer gemacht werden.

Eine Abschwächung der gerade untersuchten Idee des Parallelrechners führt zu folgendem Vorgehen: Man „rät“ für den ersten Knoten eine beliebige Farbe, dann für den zweiten Knoten auch wieder eine beliebige Farbe, solange bis für den letzten Knoten eine Farbe gewählt wurde. Danach überprüft man die geratene Färbung und akzeptiert die Eingabe, wenn die geratene Färbung eine korrekte Knotenfärbung ist. Die Eingabe ist eine positive Eingabeinstanz des **NP**-Problems 3COL, falls es eine potenzielle Lösung (Färbung) gibt, die sich bei der Überprüfung als korrekt herausstellt, d.h. im beschriebenen Rechnermodell: falls es eine Möglichkeit zu raten gibt, sodass am Ende akzeptiert (Wert 1 ausgegeben) wird. Man kann also die Berechnung durch einen Baum mit 3-fachen Verzweigungen darstellen (vgl. Abbildung 20). An den Kanten des Baumes findet sich das Resultat der Rateanweisung der darüberliegenden Verzweigung. Jeder Pfad in diesem sogenannten *Berechnungsbaum* entspricht daher einer Folge von Farbzuordnungen an die Knoten, d.h. einer potenziellen Färbung. Der Graph ist 3-färbbar, falls sich auf mindestens einem Pfad eine korrekte Färbung ergibt, falls also auf mindestens einem Pfad die Überprüfungsphase erfolgreich ist; der Beispielgraph besitzt sechs korrekte 3-Färbungen, ist also eine positive Instanz des 3COL-Problems.

Eine weitere, vielleicht intuitivere Vorstellung für die Arbeitsweise dieser **NP**-Maschine ist die, dass bei jedem Ratevorgang 3 verschiedene unabhängige Prozesse gestartet werden, die aber nicht miteinander kommunizieren dürfen. In diesem Sinne hat man es hier mit einem eingeschränkten Parallelrechner zu tun: Beliebige Aufspaltung (fork) ist erlaubt, aber keine

Kommunikation zwischen den Prozessen ist möglich. Würde man Kommunikation zulassen, hätte man erneut den allgemeinen Parallelrechner mit exponentiell vielen Prozessoren von oben, der sich ja als zu mächtig für **NP** herausgestellt hat.

Es hat sich also gezeigt, dass eine Art „Rateanweisung“ benötigt wird. In der Programmiersprache für **NP** verwendet man dazu das neue Schlüsselwort **guess**(m), wobei m die Anzahl von Möglichkeiten ist, aus denen eine geraten wird, und legt fest, dass auch die Anweisung **guess**(m) nur einen Takt Zeit für ihre Abarbeitung benötigt. Berechnungen, die, wie soeben beschrieben, verschiedene Möglichkeiten raten können, heißen *nichtdeterministisch*. Es sei wiederholt, dass *festgelegt* (definiert) wird, dass ein nichtdeterministischer Algorithmus bei einer Eingabe den Wert 1 berechnet, falls *eine Möglichkeit* geraten werden kann, sodass der Algorithmus auf die Anweisung „**return** 1“ stößt. Die Klasse **NP** umfasst nun genau die Probleme, die von nichtdeterministischen Algorithmen mit polynomieller Laufzeit gelöst werden können. „**NP**“ steht dabei für „**n**ichtdeterministische **P**olynomialzeit“, nicht etwa, wie mitunter zu lesen, für „Nicht-Polynomialzeit“. (Eine formale Präsentation der Äquivalenz zwischen effizienter Überprüfbarkeit und Polynomialzeit in der **NP**-Programmiersprache findet sich z.B. in [GJ79, Kapitel 2.3].)

Mit Hilfe eines nichtdeterministischen Algorithmus kann das 3COL-Problem in Polynomialzeit gelöst werden (siehe Algorithmus 5). Die zweite Phase von Algorithmus 5, die *Überprüfungsphase*, entspricht dabei genau dem oben angegebenen Algorithmus zum effizienten Überprüfen von möglichen Lösungen des 3COL-Problems (vgl. Algorithmus 4).

Algorithmus 5: Ein nichtdeterministischer Algorithmus für 3COL

```
Data: Graph G = ({v_1, ... v_n}, E)
Result: 1 wenn eine Färbung existiert, 0 sonst
begin
    /* Ratephase */
    for (i = 1 to n) do
        Farbe[i] = guess(3);
    end
    /* Überprüfungsphase */
    for (i = 1 to n) do
        for (j = 1 to n) do
            if (((v_i, v_j) ∈ E) und (v_i und v_j sind gleich gefärbt)) then
                return 0;
            end
        end
    end
    return 1;
end
```

Dieser nichtdeterministische Algorithmus läuft in Polynomialzeit, denn man benötigt für einen Graphen mit n Knoten mindestens n Bits, um ihn zu speichern (kodieren), und der Algorithmus braucht im schlechtesten Fall $O(n)$ (Ratephase) und $O(n^2)$ (Überprüfungsphase), also insgesamt $O(n^2)$ Takte Zeit. Damit ist gezeigt, dass 3COL in der Klasse **NP** enthalten ist, denn es wurde ein nichtdeterministischer Polynomialzeitalgorithmus gefunden, der 3COL löst. Ebenso einfach könnte man nun einen nichtdeterministischen Polynomialzeitalgorithmus entwickeln, der das Problem HAMILTON löst: Der Algorithmus wird in einer ersten Phase eine Knotenfolge raten und dann in einer zweiten Phase überprüfen, dass die Bedingungen, die an einen Hamiltonschen Kreis gestellt werden, bei der geratenen Folge erfüllt sind. Dies zeigt, dass auch HAMILTON in

der Klasse **NP** liegt.

Dass eine nichtdeterministische Maschine nicht gebaut werden kann, spielt hier keine Rolle. Nichtdeterministische Berechnungen sollen hier lediglich als Gedankenmodell für unsere Untersuchungen herangezogen werden, um Aussagen über die (Nicht-) Existenz von effizienten Algorithmen machen zu können.

6.3. Schwierigste Probleme in NP: der Begriff der NP-Vollständigkeit

Es ist nun klar, was es bedeutet, dass ein Problem in **NP** liegt. Es liegt aber auch auf der Hand, dass alle Probleme aus **P** auch in **NP** liegen, da bei der Einführung von **NP** ja nicht verlangt wurde, dass die **guess**-Anweisung verwendet werden muss. Damit ist jeder deterministische Algorithmus automatisch auch ein (eingeschränkter) nichtdeterministischer Algorithmus. Nun ist aber auch schon bekannt, dass es Probleme in **NP** gibt, z.B. 3COL und weitere Probleme, von denen nicht bekannt ist, ob sie in **P** liegen. Das führt zu der Vermutung, dass $\mathbf{P} \neq \mathbf{NP}$.

Es gibt also in **NP** anscheinend unterschiedlich schwierige Probleme: einerseits die **P**-Probleme (also die leichten Probleme), und andererseits die Probleme, von denen man nicht weiß, ob sie in **P** liegen (die schweren Probleme). Es liegt also nahe, eine allgemeine Möglichkeit zu suchen, Probleme in **NP** bezüglich ihrer Schwierigkeit zu vergleichen. Ziel ist, wie oben erläutert, eine Art von unterer Schranke für Probleme wie 3COL: Es soll gezeigt werden, dass 3COL mindestens so schwierig ist, wie jedes andere Problem in **NP**, also in gewissem Sinne ein *schwierigstes Problem in* **NP** ist.

Für diesen Vergleich der Schwierigkeit ist die erste Idee natürlich, einfach die Laufzeit von (bekannten) Algorithmen für das Problem heranzuziehen. Dies ist jedoch nicht erfolgversprechend, denn was soll eine „größte“ Laufzeit sein, die Programme für „schwierigste“ Probleme in **NP** ja haben müssten? Außerdem hängt die Laufzeit eines Algorithmus vom verwendeten Berechnungsmodell ab. So kennen Turingmaschinen keine Arrays im Gegensatz zu der hier verwendeten C-Variante. Also würde jeder Algorithmus, der Arrays verwendet, auf einer Turingmaschine mühsam simuliert werden müssen und damit langsamer abgearbeitet werden, als bei einer Hochsprache, die Arrays enthält. Obwohl sich die Komplexität eines Problems nicht ändert, würde man sie verschieden messen, je nachdem welches Berechnungsmodell verwendet würde. Ein weiterer Nachteil dieses Definitionsversuchs wäre es, dass die Komplexität (Schwierigkeit) eines Problems mit bekannten Algorithmen gemessen würde. Das würde aber bedeuten, dass jeder neue und schnellere Algorithmus Einfluss auf die Komplexität hätte, was offensichtlich so keinen Sinn macht. Aus diesen und anderen Gründen führt die erste Idee nicht zum Ziel.

Eine zweite, erfolgversprechendere Idee ist die folgende: Ein Problem A ist nicht (wesentlich) schwieriger als ein Problem B, wenn man A mit der Hilfe von B (als Unterprogramm) effizient lösen kann. Ein einfaches Beispiel ist die Multiplikation von n Zahlen. Angenommen, man hat schon ein Programm, dass zwei Zahlen multiplizieren kann; dann ist es nicht wesentlich schwieriger, auch n Zahlen zu multiplizieren, wenn die Routine für die Multiplikation von zwei Zahlen verwendet wird. Dieser Ansatz ist unter dem Namen *relative Berechenbarkeit* bekannt, der genau den oben beschriebenen Sachverhalt widerspiegelt: Multiplikation von n Zahlen (so genannte *iterierte Multiplikation*) ist relativ zur Multiplikation zweier Zahlen (leicht) berechenbar.

Da das Prinzip der relativen Berechenbarkeit so allgemein gehalten ist, gibt es innerhalb der theoretischen Informatik sehr viele verschiedene Ausprägungen dieses Konzepts. Für die **P**-**NP**-Problematik ist folgende Version der relativen Berechenbarkeit, d.h. die folgende Art von erlaubten „Unterprogrammaufrufen“, geeignet:

Seien zwei Probleme A und B gegeben. Das Problem A ist nicht schwerer als B, falls es eine effizient zu berechnende Transformation T gibt, die Folgendes leistet: Wenn x eine Eingabeinstanz von Problem A ist, dann ist $T(x)$ eine Eingabeinstanz für B. Weiterhin gilt: x ist *genau dann* eine positive Instanz von A (d.h. ein Entscheidungsalgorithmus für A muss den Wert 1

für Eingabe x liefern), wenn $T(x)$ eine positive Instanz von Problem B ist. Erneut soll „effizient berechenbar“ hier bedeuten: in Polynomialzeit berechenbar. Es muss also einen Polynomialzeitalgorithmus geben, der die Transformation T ausführt. Das Entscheidungsproblem A ist damit effizient transformierbar in das Problem B. Man sagt auch: A ist reduzierbar auf B; oder intuitiver: A ist nicht schwieriger als B, oder B ist mindestens so schwierig wie A. Formal schreibt man dann $A \leq B$.

Um für dieses Konzept ein wenig mehr Intuition zu gewinnen, sei erwähnt, dass man sich eine solche Transformation auch wie folgt vorstellen kann: A lässt sich auf B reduzieren, wenn ein Algorithmus für A angegeben werden kann, der ein Unterprogramm U_B für B genau so verwendet wie in Algorithmus 6 gezeigt.

Dabei ist zu beachten, dass das Unterprogramm für B nur genau einmal und zwar am Ende aufgerufen werden darf. Das Ergebnis des Algorithmus für A ist genau das Ergebnis, das dieser Unterprogrammaufruf liefert. Es gibt zwar, wie oben erwähnt, auch allgemeinere Ausprägungen der relativen Berechenbarkeit, die diese Einschränkung nicht haben, diese sind aber für die folgenden Untersuchungen nicht relevant.

Nachdem nun ein Vergleichsbegriff für die Schwierigkeit von Problemen aus **NP** gefunden wurde, kann auch definiert werden, was unter einem „schwierigsten“ Problem in **NP** zu verstehen ist. Ein Problem C ist ein schwierigstes Problem in **NP**, wenn alle anderen Probleme in **NP** höchstens so schwer wie C sind. Formaler ausgedrückt sind dazu zwei Eigenschaften von C nachzuweisen:

(1) C ist ein Problem aus **NP**.

(2) C ist mindestens so schwierig wie jedes andere **NP**-Problem A; d.h.: für alle Probleme A aus **NP** gilt: $A \leq C$.

Algorithmus 6: Algorithmische Darstellung der Benutzung einer Reduktionsfunktion

Data: Instanz x für das Problem A
Result: 1 wenn $x \in A$ und 0 sonst

```
begin
        /* T ist die Reduktionsfunktion (polynomialzeitberechenbar) */
    berechne y = T(x);
                                        /* y ist Instanz des Problems B */
    z = U_B(y);
                                  /* z ist 1 genau dann, wenn x ∈ A gilt */
    return z;
end
```

Solche schwierigsten Probleme in **NP** sind unter der Bezeichnung **NP**-*vollständige Probleme* bekannt. Nun sieht die Aufgabe, von einem Problem zu zeigen, dass es **NP**-vollständig ist, ziemlich hoffnungslos aus. Immerhin ist zu zeigen, dass für alle Probleme aus **NP** – und damit unendlich viele – gilt, dass sie höchstens so schwer sind wie das zu untersuchende Problem, und damit scheint man der Schwierigkeit beim Nachweis unterer Schranken nicht entgangen zu sein. Dennoch konnten der russische Mathematiker Leonid Levin und der amerikanische Mathematiker Stephen Cook Anfang der siebziger Jahre des letzten Jahrhunderts unabhängig voneinander die Existenz von solchen **NP**-vollständigen Problemen zeigen. Hat man nun erst einmal *ein* solches Problem identifiziert, ist die Aufgabe, *weitere* **NP**-vollständige Probleme zu finden, wesentlich leichter. Dies ist sehr leicht einzusehen: Ein **NP**-Problem C ist ein schwierigstes Problem

in **NP**, wenn es ein anderes schwierigstes Problem B gibt, sodass C nicht leichter als B ist. Das führt zu folgendem „Kochrezept“:

Nachweis der NP-Vollständigkeit eines Problems C:

i) Zeige, dass C in **NP** enthalten ist, indem dafür ein geeigneter nichtdeterministischer Polynomialzeitalgorithmus konstruiert wird.

ii) Suche ein geeignetes „ähnliches“ schwierigstes Problem B in **NP** und zeige, dass C nicht leichter als B ist. Formal: Finde ein **NP**-vollständiges Problem B und zeige $B \leq C$ mit Hilfe einer geeigneten Transformation T.

Den zweiten Schritt kann man oft relativ leicht mit Hilfe von bekannten Sammlungen **NP**-vollständiger Problemen erledigen. Das Buch von Garey und Johnson [GJ79] ist eine solche Sammlung (siehe auch die Abbildungen 22 und 23), die mehr als 300 **NP**-vollständige Probleme enthält. Dazu wählt man ein möglichst ähnliches Problem aus und versucht dann eine geeignete Reduktionsfunktion für das zu untersuchende Problem zu finden.

6.3.1. Traveling Salesperson ist NP-vollständig

Wie kann man zeigen, dass Traveling Salesperson **NP**-vollständig ist? Dazu wird zuerst die genaue Definition dieses Problems benötigt:

PROBLEM:	TRAVELING SALESPERSON (TSP)
EINGABE:	Eine Menge von Städten $C = \{c_1, \ldots, c_n\}$ und eine $n \times n$ Entfernungsmatrix D, wobei das Element $D[i, j]$ der Matrix D die Entfernung zwischen Stadt c_i und c_j angibt. Weiterhin eine Obergrenze $k \geq 0$ für die maximal erlaubte Länge der Tour
FRAGE:	Gibt es eine Rundreise, die einerseits alle Städte besucht, aber andererseits eine Gesamtlänge von höchstens k hat?

Nun zum ersten Schritt des Nachweises der **NP**-Vollständigkeit von TSP: Offensichtlich gehört auch das Traveling Salesperson Problem zur Klasse **NP**, denn man kann nichtdeterministisch eine Folge von n Städten raten (eine potenzielle Rundreise) und dann leicht überprüfen, ob diese potenzielle Tour durch alle Städte verläuft und ob die zurückzulegende Entfernung maximal k beträgt. Ein entsprechender nichtdeterministischer Polynomialzeitalgorithmus ist leicht zu erstellen. Damit ist der erste Schritt zum Nachweis der **NP**-Vollständigkeit von TSP getan und Punkt (1) des „Kochrezepts“ abgehandelt.

Als nächstes (Punkt (2)) soll von einem anderen **NP**-vollständigen Problem gezeigt werden, dass es effizient in TSP transformiert werden kann. Geeignet dazu ist das im Text betrachtete Hamitonkreis-Problem, das bekanntermaßen **NP**-vollständig ist. Es ist also zu zeigen: HAMILTON $\leq$ TSP.

Folgende Idee führt zum Ziel: Gegeben ist eine Instanz $G = (V, E)$ von HAMILTON. Transformiere G in folgende Instanz von TSP: Als Städtemenge C wählen wir die Knoten V des Graphen G. Die Entfernungen zwischen den Städten sind definiert wie folgt: $D[i, j] = 1$, falls es in E eine Kante von Knoten i zu Knoten j gibt, ansonsten setzt man $D[i, j]$ auf einen sehr großen Wert, also z.B. $n + 1$, wenn n die Anzahl der Knoten von G ist. Dann gilt klarerweise: Wenn G einen Hamiltonschen Kreis besitzt, dann ist der gleiche Kreis eine Rundreise in C mit Gesamtlänge n. Wenn G keinen Hamiltonschen Kreis besitzt, dann kann es keine Rundreise durch die Städte C mit Länge höchstens n geben, denn jede Rundreise muss mindestens eine Strecke von einer Stadt i nach einer Stadt j zurücklegen, die keiner Kante in G entspricht (denn ansonsten hätte G ja einen Hamiltonschen Kreis). Diese einzelne Strecke von i nach j hat dann aber schon Länge

Aus dem Graphen G links berechnet die Transformation die rechte Eingabe für das TSP. Die dick gezeichneten Verbindungen deuten eine Entfernung von 1 an, wogegen dünne Linien eine Entfernung von 6 symbolisieren. Weil G den Hamiltonkreis $1, 2, 3, 4, 5, 1$ hat, gibt es rechts eine Rundreise $1, 2, 3, 4, 5, 1$ mit Gesamtlänge 5.

Im Gegensatz dazu berechnet die Transformation hier aus dem Graphen G' auf der linken eine Eingabe für das TSP auf der rechten Seite, die, wie man sich leicht überzeugt, keine Rundreise mit einer maximalen Gesamtlänge von 5 hat. Dies liegt daran, dass der ursprüngliche Graph G' keinen Hamiltonschen Kreis hatte.

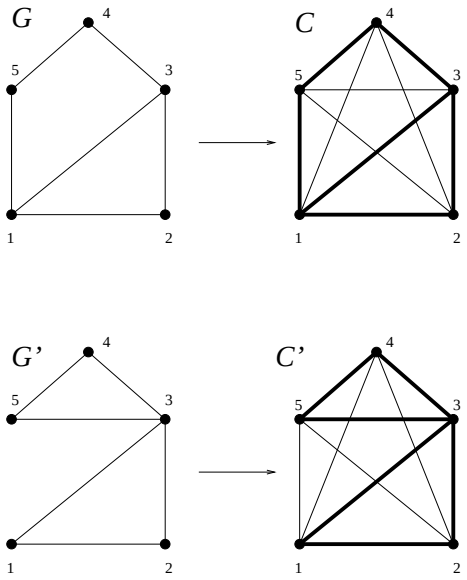


Abbildung 21: Beispiele für die Wirkungsweise von Algorithmus 7

$n+1$ und damit ist eine Gesamtlänge von n oder weniger nicht mehr erreichbar. Die Abbildung 21 zeigt zwei Beispiele für die Wirkungsweise der Transformation, die durch Algorithmus 7 in Polynomialzeit berechnet wird.

6.4. Die Auswirkungen der NP-Vollständigkeit

Welche Bedeutung haben nun die **NP**-vollständigen Probleme für die Klasse **NP**? Könnte jemand einen deterministischen Polynomialzeitalgorithmus $\mathcal{A}_C$ für ein **NP**-vollständiges Problem C angeben, dann hätte man für jedes **NP**-Problem einen Polynomialzeitalgorithmus gefunden (d.h. $\mathbf{P} = \mathbf{NP}$). Diese überraschende Tatsache lässt sich leicht einsehen, denn für jedes Problem A aus **NP** gibt es eine Transformation T mit der Eigenschaft, dass x genau dann eine positive Eingabeinstanz von A ist, wenn $T(x)$ eine positive Instanz von C ist. Damit löst Algorithmus 8 das Problem A in Polynomialzeit. Es gilt also: Ist irgendein **NP**-vollständiges Problem effizient lösbar, dann ist $\mathbf{P} = \mathbf{NP}$.

Sei nun angenommen, dass jemand $\mathbf{P} \neq \mathbf{NP}$ gezeigt hat. In diesem Fall ist aber auch klar, dass dann für kein **NP**-vollständiges Problem ein Polynomialzeitalgorithmus existieren kann, denn sonst würde sich ja der Widerspruch $\mathbf{P} = \mathbf{NP}$ ergeben. Ist das Problem C also **NP**-vollständig, so gilt: C hat genau dann einen effizienten Algorithmus, wenn $\mathbf{P} = \mathbf{NP}$, also wenn jedes Problem in **NP** einen effizienten Algorithmus besitzt. Diese Eigenschaft macht die **NP**-vollständigen Probleme für die Theoretiker so interessant, denn eine Klasse von unendlich vielen Problemen kann untersucht werden, indem man nur ein einziges Problem betrachtet. Man kann sich das auch wie folgt vorstellen: Alle relevanten Eigenschaften aller Probleme aus **NP** wurden in ein einziges Problem „destilliert“. Die **NP**-vollständigen Probleme sind also in diesem Sinn *prototypische* **NP**-Probleme.

Trotz intensiver Bemühungen in den letzten 30 Jahren konnte bisher niemand einen Polynomialzeitalgorithmus für ein **NP**-vollständiges Problem finden. Dies ist ein Grund dafür, dass man heute $\mathbf{P} \neq \mathbf{NP}$ annimmt. Leider konnte auch dies bisher nicht gezeigt werden, aber in der theoretischen Informatik gibt es starke Indizien für die Richtigkeit dieser Annahme, sodass

Algorithmus 7: Ein Algorithmus für die Reduktion von HAMILTON auf TSP

Data: Graph $G = (V, E)$, wobei $V = \{1, \ldots, n\}$
Result: Eine Instanz (C, D, k) für TSP

```
begin
    /* Die Knoten entsprechen den Städten */
    C = V;
    /* Überprüfe alle potentiell existierenden Kanten */
    for (i = 1 to n) do
        for (j = 1 to n) do
            if ((v_i, v_j) ∈ E) then
                /* Kanten entsprechen kleinen Entfernungen */
                D[i][j] = 1;
            else
                /* nicht existierende Kante, dann sehr große Entfernung */
                D[i][j] = n + 1;
            end
        end
    end
    /* Gesamtlänge k der Rundreise ist Anzahl der Städte n */
    k = n;
    /* Gebe die berechnete TSP-Instanz zurück */
    return (C, D, k);
end
```

Algorithmus 8: Ein fiktiver Algorithmus für Problem A

Data: Instanz x für das Problem A
Result: `true`, wenn $x \in A$, `false` sonst

```
begin
    /* T ist die postulierte Reduktionsfunktion */
    y = T(x);
    z = A_C(y);
    return z;
end
```

heute die große Mehrheit der Forscher von $\mathbf{P} \neq \mathbf{NP}$ ausgeht.

Für die Praxis bedeutet dies Folgendes: Hat man von einem in der Realität auftretenden Problem gezeigt, dass es **NP**-vollständig ist, dann kann man getrost aufhören, einen effizienten Algorithmus zu suchen. Wie wir ja gesehen haben, kann ein solcher nämlich (zumindest unter der gut begründbaren Annahme $\mathbf{P} \neq \mathbf{NP}$) nicht existieren.

Nun ist auch eine Antwort für das 3COL-Problem gefunden. Es wurde gezeigt [GJ79], dass kCOL für $k \geq 3$ **NP**-vollständig ist. Der fiktive Mobilfunkplaner hat also Pech gehabt: Es ist unwahrscheinlich, dass er jemals ein korrektes effizientes Planungsverfahren finden wird.

Ein **NP**-Vollständigkeitsnachweis eines Problems ist also ein starkes Indiz für seine praktische Nicht-Handhabbarkeit. Auch die **NP**-Vollständigkeit eines Problems, das mit dem Spiel *Minesweeper* zu tun hat, bedeutet demnach lediglich, dass dieses Problem höchstwahrscheinlich nicht effizient lösbar sein wird. Ein solcher Vollständigkeitsbeweis hat nichts mit einem Schritt in Richtung auf eine Lösung des $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$-Problems zu tun, wie irreführenderweise gelegentlich zu lesen ist. Übrigens ist auch für eine Reihe weiterer Spiele ihre **NP**-Vollständigkeit bekannt. Dazu gehören u.a. bestimmte Puzzle- und Kreuzwortspiele. Typische Brettspiele, wie Dame, Schach oder GO, sind hingegen (verallgemeinert auf Spielbretter der Größe $n \times n$) **PSPACE**-vollständig. Die Klasse **PSPACE** ist eine noch deutlich mächtigere Klasse als **NP**. Damit sind also diese Spiele noch viel komplexer als Minesweeper und andere **NP**-vollständige Probleme.

6.5. Der Umgang mit NP-vollständigen Problemen in der Praxis

Viele in der Praxis bedeutsame Probleme sind **NP**-vollständig (vgl. die Abbildungen 22 und 23). Ein Anwendungsentwickler wird es aber sicher schwer haben, seinem Management mitteilen zu müssen, dass ein aktuelles Projekt nicht durchgeführt werden kann, weil keine geeigneten Algorithmen zur Verfügung stehen (Wahrscheinlich würden in diesem Fall einfach „geeignetere“ Entwickler eingestellt werden!). Es stellt sich daher also die Frage, wie man mit solchen **NP**-vollständigen Problemen in der Praxis umgeht. Zu dieser Fragestellung hat die theoretische Informatik ein ausgefeiltes Instrumentarium entwickelt.

Eine erste Idee wäre es, sich mit Algorithmen zufrieden zu geben, die mit Zufallszahlen arbeiten und die nur mit sehr großer Wahrscheinlichkeit die richtige Lösung berechnen, aber sich auch mit kleiner (vernachlässigbarer) Wahrscheinlichkeit irren dürfen. Solche Algorithmen sind als *probabilistische* oder *randomisierte Algorithmen* bekannt [MR95] und werden beispielsweise in der Kryptographie mit sehr großem Erfolg angewendet. Das prominenteste Beispiel hierfür sind Algorithmen, die testen, ob eine gegebene Zahl eine Primzahl ist und sich dabei fast nie irren. Primzahlen spielen bekanntermaßen im RSA-Verfahren und damit bei PGP und ähnlichen Verschlüsselungen eine zentrale Rolle. Es konnte aber gezeigt werden, dass probabilistische Algorithmen uns bei den **NP**-vollständigen Problemen wohl nicht weiterhelfen. So weiß man heute, dass die Klasse der Probleme, die sich mit probabilistischen Algorithmen effizient lösen lässt, höchstwahrscheinlich nicht die Klasse **NP** umfasst. Deshalb liegen (höchstwahrscheinlich) insbesondere alle **NP**-vollständigen Probleme außerhalb der Möglichkeiten von effizienten probabilistischen Algorithmen.

Nun könnte man auch versuchen, „exotischere“ Computer zu bauen. In der letzten Zeit sind zwei potenzielle Auswege bekannt geworden: DNA-Computer und Quantencomputer.

Es konnte gezeigt werden, dass DNA-Computer (siehe [Pău98]) jedes **NP**-vollständige Problem in Polynomialzeit lösen können. Für diese Berechnungsstärke hat man aber einen Preis zu zahlen: Die Anzahl und damit die Masse der DNA-Moleküle, die für die Berechnung benötigt werden, wächst exponentiell in der Eingabelänge. Das bedeutet, dass schon bei recht kleinen Eingaben mehr Masse für eine Berechnung gebraucht würde, als im ganzen Universum vorhanden ist. Bisher ist kein Verfahren bekannt, wie dieses Masseproblem gelöst werden kann, und

Problemnummern in „[…]“ beziehen sich auf die Sammlung von Garey und Johnson [GJ79].

PROBLEM: CLUSTER [GT19]
EINGABE: Netzwerk $G = (V, E)$, positive Integerzahl K
FRAGE: Gibt es eine Menge von mindestens K Knoten, die paarweise miteinander verbunden sind?

PROBLEM: NETZ-AUFTEILUNG [ND16]
EINGABE: Netzwerk $G = (V, E)$, Kapazität für jede Kante in E, positive Integerzahl K
FRAGE: Kann man das Netzwerk so in zwei Teile zerlegen, dass die Gesamtkapazität aller Verbindungen zwischen den beiden Teilen mindestens K beträgt?

PROBLEM: NETZ-REDUNDANZ [ND18]
EINGABE: Netzwerk $G = (V, E)$, Kosten für Verbindungen zwischen je zwei Knoten aus V, Budget B
FRAGE: Kann G so um Verbindungen erweitert werden, dass zwischen je zwei Knoten mindestens zwei Pfade existieren und die Gesamtkosten für die Erweiterung höchstens B betragen?

PROBLEM: OBJEKTE SPEICHERN [SR1]
EINGABE: Eine Menge U von Objekten mit Speicherbedarf $s(u)$ für jedes $u \in U$; Kachelgröße S, positive Integerzahl K
FRAGE: Können die Objekte in U auf K Kacheln verteilt werden?

PROBLEM: DATENKOMPRESSION [SR8]
EINGABE: Endliche Menge R von Strings über festgelegtem Alphabet, positive Integerzahl K
FRAGE: Gibt es einen String S der Länge höchstens K, sodass jeder String aus R als Teilfolge von S vorkommt?

PROBLEM: K-SCHLÜSSEL [SR26]
EINGABE: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, positive Integerzahl K
FRAGE: Gibt es einen Schlüssel mit höchstens K Attributen?

PROBLEM: BCNF [SR29]
EINGABE: Relationales Datenbankschema, gegeben durch Attributmenge A und funktionale Abhängigkeiten auf A, Teilmenge $A' \subseteq A$
FRAGE: Verletzt die Menge A' die Boyce-Codd-Normalform?

PROBLEM: MP-SCHEDULE [SS8]
EINGABE: Menge T von Tasks, Länge für jede Task, Anzahl m von Prozessoren, positive Integerzahl D („Deadline“)
FRAGE: Gibt es ein m-Prozessor-Schedule für T mit Ausführungszeit höchstens D?

PROBLEM: PREEMPT-SCHEDULE [SS12]
EINGABE: Menge T von Tasks, Länge für jede Task, Präzedenzrelation auf den Tasks, Anzahl m von Prozessoren, positive Integerzahl D („Deadline“)
FRAGE: Gibt es ein m-Prozessor-Schedule für T, das die Präzedenzrelationen berücksichtigt und Ausführungszeit höchstens D hat?

PROBLEM: DEADLOCK [SS22]
EINGABE: Menge von Prozessen, Menge von Ressourcen, aktuelle Zustände der Prozesse und aktuell allokierte Ressourcen
FRAGE: Gibt es einen Kontrollfluss, der zum Deadlock führt?

PROBLEM: K-REGISTER [PO3]
EINGABE: Menge V von Variablen, die in einer Schleife benutzt werden, für jede Variable einen Gültigkeitsbereich, positive Integerzahl K
FRAGE: Können die Schleifenvariablen mit höchstens K Registern gespeichert werden?

PROBLEM: REKURSION [PO20]
EINGABE: Menge A von Prozedur-Identifiern, Pascal-Programmfragment mit Deklarationen und Aufrufen der Prozeduren aus A
FRAGE: Ist eine der Prozeduren aus A formal rekursiv?

Abbildung 22: Eine kleine Sammlung **NP**-vollständiger Probleme (Teil 1)

Problemnummern in „[…]“ beziehen sich auf die Sammlung von Garey und Johnson [GJ79].

PROBLEM:	LR(K)-GRAMMATIK [AL15]
EINGABE:	Kontextfreie Grammatik G, positive Integerzahl K (unär)
FRAGE:	Ist die Grammatik G nicht LR(K)?

PROBLEM:	ZWANGSBEDINGUNG [LO5]
EINGABE:	Menge von Booleschen Constraints, positive Integerzahl K
FRAGE:	Können mindestens K der Constraints gleichzeitig erfüllt werden?

PROBLEM:	INTEGER PROGRAM [MP1]
EINGABE:	Lineares Programm
FRAGE:	Hat das Programm eine Lösung, die nur ganzzahlige Werte enthält?

PROBLEM:	KREUZWORTRÄTSEL [GP15]
EINGABE:	Menge W von Wörtern, Gitter mit schwarzen und weißen Feldern
FRAGE:	Können die weißen Felder des Gitters mit Wörtern aus W gefüllt werden?

Abbildung 23: Eine kleine Sammlung **NP**-vollständiger Probleme (Teil 2)

es sieht auch nicht so aus, als ob es gelöst werden kann, wenn $\mathbf{P} \neq \mathbf{NP}$ gilt. Dieses Problem erinnert an das oben im Kontext von Parallelrechnern schon erwähnte Phänomen: Mit exponentiell vielen Prozessoren lassen sich **NP**-vollständige Probleme lösen, aber solche Parallelrechner haben natürlich explodierende Hardware-Kosten.

Der anderer Ausweg könnten Quantencomputer sein (siehe [Hom08, Gru99]). Hier scheint die Situation zunächst günstiger zu sein: Die Fortschritte bei der Quantencomputer-Forschung verlaufen immens schnell, und es besteht die berechtigte Hoffnung, dass Quantencomputer mittelfristig verfügbar sein werden. Aber auch hier sagen theoretische Ergebnisse voraus, dass Quantencomputer (höchstwahrscheinlich) keine **NP**-vollständigen Probleme lösen können. Trotzdem sind Quantencomputer interessant, denn es ist bekannt, dass wichtige Probleme existieren, für die kein Polynomialzeitalgorithmus bekannt ist und die wahrscheinlich nicht **NP**-vollständig sind, die aber auf Quantencomputern effizient gelöst werden können. Das prominenteste Beispiel hierfür ist die Aufgabe, eine ganze Zahl in ihre Primfaktoren zu zerlegen.

Die bisher angesprochenen Ideen lassen also die Frage, wie man mit **NP**-vollständigen Problemen umgeht, unbeantwortet. In der Praxis gibt es im Moment zwei Hauptansatzpunkte: Die erste Möglichkeit ist die, die Allgemeinheit des untersuchten Problems zu beschränken und eine spezielle Version zu betrachten, die immer noch für die geplante Anwendung ausreicht. Zum Beispiel sind Graphenprobleme oft einfacher, wenn man zusätzlich fordert, dass die Knoten des Graphen in der (Euklidischen) Ebene lokalisiert sind. Deshalb sollte die erste Idee bei der Behandlung von **NP**-vollständigen Problemen immer sein, zu untersuchen, welche Einschränkungen man an das Problem machen kann, ohne die praktische Aufgabenstellung zu verfälschen. Gerade diese Einschränkungen können dann effiziente Algorithmen ermöglichen.

Die zweite Möglichkeit sind sogenannte *Approximationsalgorithmen* (vgl. [ACG$^+$99]). Die Idee hier ist es, nicht die optimalen Lösungen zu suchen, sondern sich mit einem kleinen garantierten Fehler zufrieden zu geben. Dazu folgendes Beispiel. Es ist bekannt, dass das TSP auch dann noch **NP**-vollständig ist, wenn man annimmt, dass die Städte in der Euklidischen Ebene lokalisiert sind, d.h. man kann die Städte in einer fiktiven Landkarte einzeichnen, sodass die Entfernungen zwischen den Städten proportional zu den Abständen auf der Landkarte sind. Das ist sicherlich in der Praxis keine einschränkende Abschwächung des Problems und zeigt, dass die oben erwähnte Methode nicht immer zum Erfolg führen muss: Hier bleibt auch das eingeschränkte Problem **NP**-vollständig. Aber für diese eingeschränkte TSP-Variante ist ein Polynomialzeitalgorithmus bekannt, der immer eine Rundreise berechnet, die höchstens um einen beliebig wählbaren Faktor schlechter ist, als die optimale Lösung. Ein Chip-Hersteller, der bei der Bestückung seiner Platinen die Wege der Roboterköpfe minimieren möchte, kann also beschließen, sich mit einer

Tour zufrieden zu geben, die um 5 % schlechter ist als die optimale. Für dieses Problem existiert ein effizienter Algorithmus! Dieser ist für die Praxis völlig ausreichend.

⋆⋆⋆ Ende ⋆⋆⋆

A. Grundlagen und Schreibweisen

A.1. Mengen

Es ist sehr schwer den fundamentalen Begriff der Menge mathematisch exakt zu definieren. Aus diesem Grund soll uns hier die von Cantor im Jahr 1895 gegebene Erklärung genügen, da sie für unsere Zwecke völlig ausreichend ist:

Definition 95 (Georg Cantor ([Can95])): *Unter einer ‚*Menge*' verstehen wir jede Zusammenfassung M von bestimmten wohlunterschiedenen Objecten m unsrer Anschauung oder unseres Denkens (welche die ‚*Elemente*' von M genannt werden) zu einem Ganzen*[22].

Für die Formulierung „genau dann wenn" verwenden wir im Folgenden die Abkürzung gdw. um Schreibarbeit zu sparen.

A.1.1. Die Elementbeziehung und die Enthaltenseinsrelation

Sehr oft werden einfache große lateinische Buchstaben wie N, M, A, B oder C als Symbole für Mengen verwendet und kleine Buchstaben für die Elemente einer Menge. Mengen von Mengen notiert man gerne mit kalligraphischen Buchstaben wie $\mathcal{A}$, $\mathcal{B}$ oder $\mathcal{M}$.

Definition 96: *Sei M eine beliebige Menge, dann ist*

- $a \in M$ *gdw. a ist ein Element der Menge M,*
- $a \notin M$ *gdw. a ist kein Element der Menge M,*
- $M \subseteq N$ *gdw. aus $a \in M$ folgt $a \in N$ (M ist* Teilmenge *von N),*
- $M \not\subseteq N$ *gdw. es gilt nicht $M \subseteq N$. Gleichwertig: es gibt ein $a \in M$ mit $a \notin N$ (M ist keine Teilmenge von N) und*
- $M \subset N$ *gdw. es gilt $M \subseteq N$ und $M \neq N$ (M ist echte Teilmenge von N).*

Statt $a \in M$ schreibt man auch $M \ni a$, was in einigen Fällen zu einer deutlichen Vereinfachung der Notation führt.

A.1.2. Definition spezieller Mengen

Spezielle Mengen können auf verschiedene Art und Weise definiert werden, wie z.B.

- durch Angabe von Elementen: So ist $\{a_1, \ldots, a_n\}$ die Menge, die aus den Elementen a_1, $\ldots$, a_n besteht, oder
- durch eine Eigenschaft E: Dabei ist $\{a \mid E(a)\}$ die Menge aller Elemente a, die die Eigenschaft[23] E besitzen.

Alternativ zu der Schreibweise $\{a \mid E(a)\}$ wird auch oft $\{a\colon E(a)\}$ verwendet.

[22] Diese Zitat entspricht der originalen Schreibweise von Cantor.
[23] Die Eigenschaft E kann man dann auch als *Prädikat* bezeichnen.

Beispiel 97: *Mengen, die durch die Angabe von Elementen definiert sind:*

- $\mathbb{B} =_{\text{def}} \{0, 1\}$
- $\mathbb{N} =_{\text{def}} \{0, 1, 2, 3, 4, 5, 6, 7, 8, \dots\}$ *(Menge der* natürlichen Zahlen*)*
- $\mathbb{Z} =_{\text{def}} \{\dots, -4, -3, -2, -1, 0, 1, 2, 3, 4, \dots\}$ *(Menge der* ganzen Zahlen*)*
- $2\mathbb{Z} =_{\text{def}} \{0, \pm 2, \pm 4, \pm 6, \pm 8, \dots\}$ *(Menge der geraden ganzen Zahlen)*
- $\mathbb{P} =_{\text{def}} \{2, 3, 5, 7, 11, 13, 17, 19, \dots\}$ *(Menge der* Primzahlen*)*

Beispiel 98: *Mengen, die durch eine Eigenschaft E definiert sind:*

- $\{n \mid n \in \mathbb{N}$ *und* n *ist durch* 3 *teilbar*$\}$
- $\{n \mid n \in \mathbb{N}$ *und* n *ist Primzahl und* $n \leq 40\}$
- $\emptyset =_{\text{def}} \{a \mid a \neq a\}$ *(die leere Menge)*

Aus Definition 96 ergibt sich, dass die leere Menge (Schreibweise: $\emptyset$) Teilmenge jeder Menge ist. Dabei ist zu beachten, dass $\{\emptyset\} \neq \emptyset$ gilt, denn $\{\emptyset\}$ enthält *ein* Element (die leere Menge) und $\emptyset$ enthält *kein* Element.

A.1.3. Operationen auf Mengen

Definition 99: *Seien A und B beliebige Mengen, dann ist*

- $A \cap B =_{\text{def}} \{a \mid a \in A$ *und* $a \in B\}$ *(*Schnitt *von A und B),*
- $A \cup B =_{\text{def}} \{a \mid a \in A$ *oder* $a \in B\}$ *(*Vereinigung *von A und B),*
- $A \setminus B =_{\text{def}} \{a \mid a \in A$ *und* $a \notin B\}$ *(*Differenz *von A und B),*
- $\overline{A} =_{\text{def}} M \setminus A$ *(*Komplement *von A bezüglich einer festen Grundmenge M) und*
- $\mathcal{P}(A) =_{\text{def}} \{B \mid B \subseteq A\}$ *(*Potenzmenge *von A).*

Zwei Mengen A und B mit $A \cap B = \emptyset$ *nennt man* disjunkt*.*

Beispiel 100: *Sei* $A = \{2, 3, 5, 7\}$ *und* $B = \{1, 2, 4, 6\}$*, dann ist* $A \cap B = \{2\}$*,* $A \cup B = \{1, 2, 3, 4, 5, 6, 7\}$ *und* $A \setminus B = \{3, 5, 7\}$*. Wählen wir als Grundmenge die natürlichen Zahlen, also* $M = \mathbb{N}$*, dann ist* $\overline{A} = \{n \in \mathbb{N} \mid n \neq 2$ *und* $n \neq 3$ *und* $n \neq 5$ *und* $n \neq 7\} = \{1, 4, 6, 8, 9, 10, 11, \dots\}$*.*

Als Potenzmenge der Menge A ergibt sich die folgende Menge von Mengen von natürlichen Zahlen $\mathcal{P}(A) = \{\emptyset, \{2\}, \{3\}, \{5\}, \{7\}, \{2, 3\}, \{2, 5\}, \{2, 7\}, \{3, 5\}, \{3, 7\}, \{5, 7\}, \{2, 3, 5\}, \{2, 3, 7\}, \{2, 5, 7\}, \{3, 5, 7\}, \{2, 3, 5, 7\}\}$*.*

Offensichtlich ist die Menge $\{0, 2, 4, 6, 8, \dots\}$ *der geraden natürlichen Zahlen und die Menge* $\{1, 3, 5, 7, 9, \dots\}$ *der ungeraden natürlichen Zahlen disjunkt.*

A.1.4. Gesetze für Mengenoperationen

Für die klassischen Mengenoperationen gelten die folgenden Beziehungen:

$A \cap B$	$=$	$B \cap A$	Kommutativgesetz für den Schnitt
$A \cup B$	$=$	$B \cup A$	Kommutativgesetz für die Vereinigung
$A \cap (B \cap C)$	$=$	$(A \cap B) \cap C$	Assoziativgesetz für den Schnitt
$A \cup (B \cup C)$	$=$	$(A \cup B) \cup C$	Assoziativgesetz für die Vereinigung
$A \cap (B \cup C)$	$=$	$(A \cap B) \cup (A \cap C)$	Distributivgesetz
$A \cup (B \cap C)$	$=$	$(A \cup B) \cap (A \cup C)$	Distributivgesetz
$A \cap A$	$=$	A	Duplizitätsgesetz für den Schnitt
$A \cup A$	$=$	A	Duplizitätsgesetz für die Vereinigung
$A \cap (A \cup B)$	$=$	A	Absorptionsgesetz
$A \cup (A \cap B)$	$=$	A	Absorptionsgesetz
$\overline{A \cap B}$	$=$	$(\overline{A} \cup \overline{B})$	de-Morgansche Regel
$\overline{A \cup B}$	$=$	$(\overline{A} \cap \overline{B})$	de-Morgansche Regel
$\overline{\overline{A}}$	$=$	A	Gesetz des doppelten Komplements

Die „de-Morganschen Regeln“ wurden nach dem englischen Mathematiker Augustus De Morgan[24] benannt.

Als Abkürzung schreibt man statt $X_1 \cup X_2 \cup \cdots \cup X_n$ (bzw. $X_1 \cap X_2 \cap \cdots \cap X_n$) einfach $\bigcup_{i=1}^{n} X_i$ (bzw. $\bigcap_{i=1}^{n} X_i$). Möchte man alle Mengen X_i mit $i \in \mathbb{N}$ schneiden (bzw. vereinigen), so schreibt man kurz $\bigcap_{i \in \mathbb{N}} X_i$ (bzw. $\bigcup_{i \in \mathbb{N}} X_i$).

Oft benötigt man eine Verknüpfung von zwei Mengen, eine solche Verknüpfung wird allgemein wie folgt definiert:

Definition 101 („Verknüpfung von Mengen“): *Seien A und B zwei Mengen und „$\odot$“ eine beliebige Verknüpfung zwischen den Elementen dieser Mengen, dann definieren wir*

$$A \odot B =_{\text{def}} \{a \odot b \mid a \in A \text{ und } b \in B\}.$$

Beispiel 102: *Die Menge $3\mathbb{Z} = \{0, \pm 3, \pm 6, \pm 9, \dots\}$ enthält alle Vielfachen[25] von 3, damit ist $3\mathbb{Z} + \{1\} = \{1, 4, -2, 7, -5, 10, -8, \dots\}$. Die Menge $3\mathbb{Z} + \{1\}$ schreibt man kurz oft auch als $3\mathbb{Z} + 1$, wenn klar ist, was mit dieser Abkürzung gemeint ist.*

A.1.5. Tupel (Vektoren) und das Kreuzprodukt

Seien $A, A_1, \dots, A_n$ im folgenden Mengen, dann bezeichnet

- $(a_1, \dots, a_n) =_{\text{def}}$ die Elemente $a_1, \dots, a_n$ in genau dieser festgelegten *Reihenfolge* und z.B. $(3, 2) \neq (2, 3)$. Wir sprechen von einem n-Tupel.
- $A_1 \times A_2 \times \cdots \times A_n =_{\text{def}} \{(a_1, \dots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \dots, a_n \in A_n\}$ (Kreuzprodukt der Mengen $A_1, A_2, \dots, A_n$),
- $A^n =_{\text{def}} \underbrace{A \times A \times \cdots \times A}_{n\text{-mal}}$ (n-faches Kreuzprodukt der Menge A) und
- speziell gilt $A^1 = \{(a) \mid a \in A\}$.

[24] ⋆1806 in Madurai, Tamil Nadu, Indien - †1871 in London, England

[25] Eigentlich müsste man statt $3\mathbb{Z}$ die Notation $\{3\}\mathbb{Z}$ verwenden. Dies ist allerdings unüblich.

Wir nennen 2-Tupel auch *Paare*, 3-Tupel auch *Tripel*, 4-Tupel auch *Quadrupel* und 5-Tupel *Quintupel*. Bei n-Tupeln ist, im Gegensatz zu Mengen, eine Reihenfolge vorgegeben, d.h. es gilt z.B. immer $\{a, b\} = \{b, a\}$, aber im Allgemeinen $(a, b) \neq (b, a)$.

Beispiel 103: *Sei $A = \{1, 2, 3\}$ und $B = \{a, b, c\}$, dann bezeichnet das Kreuzprodukt von A und B die Menge von Paaren $A \times B = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c), (3, a), (3, b), (3, c)\}$.*

A.1.6. Die Anzahl von Elementen in Mengen

Sei A eine Menge, die endlich viele Elemente[26] enthält, dann ist

$$\#A =_{\text{def}} \text{Anzahl der Elemente in der Menge } A.$$

Beispielsweise ist $\#\{4, 7, 9\} = 3$. Mit dieser Definition gilt

- $\#(A^n) = (\#A)^n$,
- $\#\mathcal{P}(A) = 2^{\#A}$,
- $\#A + \#B = \#(A \cup B) + \#(A \cap B)$ und
- $\#A = \#(A \setminus B) + \#(A \cap B)$.

A.2. Relationen und Funktionen

A.2.1. Eigenschaften von Relationen

Seien $A_1, \ldots, A_n$ beliebige Mengen, dann ist R eine *n-stellige Relation* gdw. $R \subseteq A_1 \times A_2 \times \cdots \times A_n$. Eine zweistellige Relation nennt man auch *binäre Relation*. Oft werden auch Relationen $R \subseteq A^n$ betrachtet, diese bezeichnet man dann als n-stellige Relation über der Menge A.

Definition 104: *Sei R eine zweistellige Relation über A, dann ist R*

- reflexiv *gdw. $(a, a) \in R$ für alle $a \in A$,*
- symmetrisch *gdw. aus $(a, b) \in R$ folgt $(b, a) \in R$,*
- antisymmetrisch *gdw. aus $(a, b) \in R$ und $(b, a) \in R$ folgt $a = b$,*
- transitiv *gdw. aus $(a, b) \in R$ und $(b, c) \in R$ folgt $(a, c) \in R$ und*
- linear *gdw. es gilt immer $(a, b) \in R$ oder $(b, a) \in R$.*

Definition 105: *Sei R eine zweistellige Relation, dann*

- *heißt R* Halbordnung *gdw. R ist reflexiv, antisymmetrisch und transitiv,*
- Ordnung *gdw. R ist eine lineare Halbordnung und*
- Äquivalenzrelation *gdw. R reflexiv, transitiv und symmetrisch ist.*

Beispiel 106: *Die Teilmengenrelation „$\subseteq$“ auf allen Teilmengen von $\mathbb{Z}$ ist eine Halbordnung, aber keine Ordnung.*

[26] Solche Mengen werden als *endliche Mengen* bezeichnet.

Beispiel 107: *Wir schreiben $a \equiv b \bmod n$, falls es eine ganze Zahl q gibt, für die $a - b = qn$ gilt. Für $n \geq 2$ ist die Relation $R_n(a,b) =_{\text{def}} \{(a,b) \mid a \equiv b \bmod n\} \subseteq \mathbb{Z}^2$ eine Äquivalenzrelation.*

A.2.2. Eigenschaften von Funktionen

Seien A und B beliebige Mengen. f ist eine *Funktion* von A nach B (Schreibweise: $f\colon A \to B$) gdw. $f \subseteq A \times B$ und für jedes $a \in A$ gibt es *höchstens* ein $b \in B$ mit $(a,b) \in f$. Ist also $(a,b) \in f$, so schreibt man $f(a) = b$. Ebenfalls gebräuchlich ist die Notation $a \mapsto b$.

Bemerkung 108: *Unsere Definition von Funktion umfasst auch mehrstellige Funktionen. Seien C und B Mengen und $A = C^n$ das n-fache Kreuzprodukt von C. Die Funktion $f\colon A \to B$ ist dann eine n-stellige Funktion, denn sie bildet n-Tupel aus C^n auf Elemente aus B ab.*

Definition 109: *Sei f eine n-stellige Funktion. Möchte man die Funktion f benutzen, aber keine Namen für die Argumente vergeben, so schreibt man auch*

$$f(\underbrace{\cdot, \cdot, \ldots, \cdot}_{n\text{-mal}})$$

Ist also der Namen des Arguments einer einstelligen Funktion $g(x)$ für eine Betrachtung unwichtig, so kann man $g(\cdot)$ schreiben, um anzudeuten, dass g einstellig ist, ohne dies weiter zu erwähnen.

Definition 110: *Sei nun $R \subseteq A_1 \times A_2 \times \cdots \times A_n$ eine n-stellige Relation, dann definieren wir eine Funktion $P_R^n\colon A_1 \times A_2 \times \cdots \times A_n \to \{0,1\}$ wie folgt:*

$$P_R^n(x_1, \ldots, x_n) =_{\text{def}} \begin{cases} 1, & \text{falls } (x_1, \ldots, x_n) \in R \\ 0, & \text{sonst} \end{cases}$$

Eine solche n-stellige Funktion, die „anzeigt“, ob ein Element aus $A_1 \times A_2 \times \cdots \times A_n$ entweder zu R gehört oder nicht, nennt man (n-stelliges) Prädikat.

Beispiel 111: *Sei $\mathbb{P} =_{\text{def}} \{n \in \mathbb{N} \mid n \text{ ist Primzahl}\}$, dann ist $\mathbb{P}$ eine 1-stellige Relation über den natürlichen Zahlen. Das Prädikat $P_{\mathbb{P}}^1(n)$ liefert für eine natürliche Zahl n genau dann 1, wenn n eine Primzahl ist.*

Ist für ein Prädikat P_R^n sowohl die Relation R als auch die Stelligkeit n aus dem Kontext klar, dann schreibt man auch kurz P oder verwendet das Relationensymbol R als Notation für das Prädikat P_R^n.

Nun legen wir zwei spezielle Funktionen fest, die oft sehr hilfreich sind:

Definition 112: *Sei $\alpha \in \mathbb{R}$ eine beliebige reelle Zahl, dann gilt*

- $\lceil \alpha \rceil =_{\text{def}}$ *die kleinste ganze Zahl, die größer oder gleich α ist ($\triangleq$ „Aufrunden“)*
- $\lfloor \alpha \rfloor =_{\text{def}}$ *die größte ganze Zahl, die kleiner oder gleich α ist ($\triangleq$ „Abrunden“)*

Definition 113: *Für eine beliebige Funktion f legen wir fest:*

- *Der* Definitionsbereich *von f ist $D_f =_{\text{def}} \{a \mid$ es gibt ein b mit $f(a) = b\}$.*
- *Der* Wertebereich *von f ist $W_f =_{\text{def}} \{b \mid$ es gibt ein a mit $f(a) = b\}$.*
- *Die Funktion $f\colon A \to B$ ist* total *gdw. $D_f = A$.*

- *Die Funktion $f\colon A \to B$ heißt* surjektiv *gdw. $W_f = B$.*
- *Die Funktion f heißt* injektiv *(oder eineindeutig[27]) gdw. immer wenn $f(a_1) = f(a_2)$ gilt auch $a_1 = a_2$.*
- *Die Funktion f heißt* bijektiv *gdw. f ist injektiv und surjektiv.*

Mit Hilfe der Kontraposition (siehe Abschnitt C.1.1) kann man für die Injektivität alternativ auch zeigen, dass immer wenn $a_1 \neq a_2$, dann muss auch $f(a_1) \neq f(a_2)$ gelten.

Beispiel 114: *Sei die Funktion $f\colon \mathbb{N} \to \mathbb{Z}$ durch $f(n) = (-1)^n \lceil \frac{n}{2} \rceil$ gegeben. Die Funktion f ist surjektiv, denn $f(0) = 0, f(1) = -1, f(2) = 1, f(3) = -2, f(4) = 2, \ldots$, d.h. die ungeraden natürlichen Zahlen werden auf die negativen ganzen Zahlen abgebildet, die geraden Zahlen aus $\mathbb{N}$ werden auf die positiven ganzen Zahlen abgebildet und deshalb ist $W_f = \mathbb{Z}$.*
Weiterhin ist f auch injektiv, denn aus[28] $(-1)^{a_1} \lceil \frac{a_1}{2} \rceil = (-1)^{a_2} \lceil \frac{a_2}{2} \rceil$ folgt, dass entweder a_1 und a_2 gerade oder a_1 und a_2 ungerade, denn sonst würden auf der linken und rechten Seite der Gleichung unterschiedliche Vorzeichen auftreten. Ist a_1 gerade und a_2 gerade, dann gilt $\lceil \frac{a_1}{2} \rceil = \lceil \frac{a_2}{2} \rceil$ und auch $a_1 = a_2$. Sind a_1 und a_2 ungerade, dann gilt $-\lceil \frac{a_1}{2} \rceil = -\lceil \frac{a_2}{2} \rceil$, woraus auch folgt, dass $a_1 = a_2$. Damit ist die Funktion f bijektiv. Weiterhin ist f auch total, d.h. $D_f = \mathbb{N}$.

Definition 115: *Unter einem n-stelligen* Operator *f (auf der Menge Y) versteht man in der Mathematik eine Funktion der Form $f\colon Y^n \to Y$. Einfache Beispiele für zweistellige Operatoren sind der Additions- oder Multiplikationsoperator.*

A.2.3. Hüllenoperatoren

Definition 116: *Sei X eine Menge. Ein einstelliger Operator $\Psi\colon \mathcal{P}(X) \to \mathcal{P}(X)$ heißt* Hüllenoperator*, wenn er die folgenden drei Eigenschaften erfüllt:*

Einbettung: *für alle $A \in \mathcal{P}(X)$ gilt $A \subseteq \Psi(A)$*

Monotonie: *für alle $A, B \in \mathcal{P}(X)$ mit $A \subseteq B$ folgt $\Psi(A) \subseteq \Psi(B)$*

Abgeschlossenheit: *für alle $A \in \mathcal{P}(X)$ gilt $\Psi(\Psi(A)) = \Psi(A)$*

Aufgrund der Monotonieeigenschaft eines Hüllenoperators kann man bei der Abgeschlossenheit die Eigenschaft $\Psi(\Psi(A)) = \Psi(A)$ auch durch $\Psi(\Psi(A)) \subseteq \Psi(A)$ ersetzen. In der Informatik spielen Hüllenoperatoren eine große Rolle. Gute Beispiele hierfür sind z.B. die *transitive Hülle* (vgl. Computergraphik), die Kleene-Hülle (vgl. Formale Sprachen) oder der Abschluss einer Komplexitätsklasse unter Schnitt oder Vereinigung.

A.2.4. Permutationen

Sei S eine beliebige endliche Menge, dann heißt eine bijektive Funktion π der Form $\pi\colon S \to S$ *Permutation*. Das bedeutet, dass die Funktion π Elemente aus S wieder auf Elemente aus S abbildet, wobei für jedes $b \in S$ ein $a \in S$ mit $f(a) = b$ existiert (Surjektivität) und falls $f(a_1) = f(a_2)$ gilt, dann ist $a_1 = a_2$ (Injektivität).

Bemerkung 117: *Man kann den Permutationsbegriff auch auf unendliche Mengen erweitern, aber besonders häufig werden in der Informatik* Permutationen von endlichen Mengen *benötigt. Aus diesem Grund sollen hier nur endliche Mengen S betrachtet werden.*

[27] Achtung: Dieser Begriff wird manchmal unterschiedlich, je nach Autor, in den Bedeutungen „bijektiv“ oder „injektiv“ verwendet.
[28] Für die Definition der Funktion $\lceil \cdot \rceil$ siehe Definition 112.

Sei nun $S = \{1, \ldots, n\}$ (eine endliche Menge) und $\pi \colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine Permutation. Permutationen dieser Art kann man sehr anschaulich mit Hilfe einer Matrix aufschreiben:

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

Durch diese Notation wird klar, dass das Element 1 der Menge S durch das Element $\pi(1)$ ersetzt wird, das Element 2 wird mit $\pi(2)$ vertauscht und allgemein das Element i durch $\pi(i)$ für $1 \leq i \leq n$. In der zweiten Zeile dieser Matrixnotation findet sich also *jedes* (Surjektivität) Element der Menge S genau *einmal* (Injektivität).

Beispiel 118: *Sei $S = \{1, \ldots, 3\}$ eine Menge mit drei Elementen. Dann gibt es, wie man ausprobieren kann, genau* 6 *Permutationen von S:*

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \end{pmatrix} \quad \pi_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \quad \pi_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 1 & 3 \end{pmatrix}$$
$$\pi_4 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \quad \pi_5 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \quad \pi_6 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$$

Satz 119: *Sei S eine endliche Menge mit $n = |S|$, dann gibt es genau $n!$ (Fakultät) verschiedene Permutationen von S.*

Beweis: Jede Permutation π der Menge S von n Elementen kann als Matrix der Form

$$\pi = \begin{pmatrix} 1 & 2 & \ldots & n \\ \pi(1) & \pi(2) & \ldots & \pi(n) \end{pmatrix}$$

aufgeschrieben werden. Damit ergibt sich die Anzahl der Permutationen von S durch die Anzahl der verschiedenen zweiten Zeilen solcher Matrizen. In jeder solchen Zeile muss jedes der n Elemente von S genau einmal vorkommen, da π eine bijektive Abbildung ist, d.h. wir haben für die erste Position der zweiten Zeile der Matrixdarstellung genau n verschiedene Möglichkeiten, für die zweite Position noch $n - 1$ und für die dritte noch $n - 2$. Für die n-te Position bleibt nur noch 1 mögliches Element aus S übrig[29]. Zusammengenommen haben wir also $n \cdot (n-1) \cdot (n-2) \cdot (n-3) \cdot \ldots \cdot 2 \cdot 1 = n!$ verschiedene mögliche Permutationen der Menge S. #

A.3. Summen und Produkte

A.3.1. Summen

Zur abkürzenden Schreibweise verwendet man für Summen das Summenzeichen $\sum$. Dabei ist

$$\sum_{i=1}^{n} a_i =_{\text{def}} a_1 + a_2 + \cdots + a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \leq i \leq n$, dann gilt $\sum_{i=1}^{n} a_i = n \cdot a$ (Summe gleicher Summanden).
- $\sum_{i=1}^{n} a_i = \sum_{i=1}^{m} a_i + \sum_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten einer Summe).

[29] Dies kann man sich auch als die Anzahl der verschiedenen Möglichkeiten vorstellen, die bestehen, wenn man aus einer Urne mit n numerierten Kugeln alle Kugeln *ohne* Zurücklegen nacheinander zieht.

- $\sum_{i=1}^{n}(a_i + b_i + c_i + \dots) = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i + \sum_{i=1}^{n} c_i + \dots$ (Addition von Summen).
- $\sum_{i=1}^{n} a_i = \sum_{i=l}^{n+l-1} a_{i-l+1}$ und $\sum_{i=l}^{n} a_i = \sum_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Summen).
- $\sum_{i=1}^{n}\sum_{j=1}^{m} a_{i,j} = \sum_{j=1}^{m}\sum_{i=1}^{n} a_{i,j}$ (Vertauschen der Summationsfolge).

Manchmal verwendet man keine Laufindizes an ein Summenzeichen, sondern man beschreibt (durch ein Prädikat) welche Zahlen aufsummiert werden sollen. So kann man eine Funktion definieren, die die Summe aller Teiler einer natürlichen Zahl liefert:

$$\sigma(n) =_{\text{def}} \sum_{\substack{t \le n \\ t \text{ teilt } n}} t$$

A.3.2. Produkte

Zur abkürzenden Schreibweise verwendet man für Produkte das Produktzeichen $\prod$. Dabei ist

$$\prod_{i=1}^{n} a_i =_{\text{def}} a_1 \cdot a_2 \cdot \ldots \cdot a_n.$$

Mit Hilfe dieser Definition ergeben sich auf elementare Weise die folgenden Rechenregeln:

- Sei $a_i = a$ für $1 \le i \le n$, dann gilt $\prod_{i=1}^{n} a_i = a^n$ (Produkt gleicher Faktoren).
- $\prod_{i=1}^{n}(ca_i) = c^n \prod_{i=1}^{n} a_i$ (Vorziehen von konstanten Faktoren)
- $\prod_{i=1}^{n} a_i = \prod_{i=1}^{m} a_i \cdot \prod_{i=m+1}^{n} a_i$, wenn $1 < m < n$ (Aufspalten in Teilprodukte).
- $\prod_{i=1}^{n}(a_i \cdot b_i \cdot c_i \cdot \ldots) = \prod_{i=1}^{n} a_i \cdot \prod_{i=1}^{n} b_i \cdot \prod_{i=1}^{n} c_i \cdot \ldots$ (Das Produkt von Produkten).
- $\prod_{i=1}^{n} a_i = \prod_{i=l}^{n+l-1} a_{i-l+1}$ und $\prod_{i=l}^{n} a_i = \prod_{i=1}^{n-l+1} a_{i+l-1}$ (Umnumerierung von Produkten).
- $\prod_{i=1}^{n}\prod_{j=1}^{m} a_{i,j} = \prod_{j=1}^{m}\prod_{i=1}^{n} a_{i,j}$ (Vertauschen der Reihenfolge bei Doppelprodukten).

Ähnlich wie bei Summen kann man bei Produkten auch ohne Laufindex arbeiten. So ist z.B. die *Eulersche ϕ-Funktion* wie folgt definiert:

$$\phi(n) =_{\text{def}} n \prod_{p|n}(1 - \frac{1}{p})$$

In das Produkt gehen also alle Primteiler p von n mit dem Faktor $1 - 1/p$ ein.

Oft werden Summen- oder Produktsymbole verwendet bei denen der Startindex größer als der Stopindex ist. Solche Summen bzw. Produkte sind „leer“, d.h. es wird nichts summiert bzw. multipliziert. Sind dagegen Start- und Endindex gleich, so tritt nur genau ein Wert auf, d.h. das Summen- bzw. Produktsymbol hat in diesem Fall keine Auswirkung. Es ergeben sich also die folgenden Rechenregeln:

- Seien $n, m \in \mathbb{Z}$ und $n < m$, dann

$$\sum_{i=m}^{n} a_i = 0 \text{ und } \prod_{i=m}^{n} a_i = 1$$

- Sei $n \in \mathbb{Z}$, dann

$$\sum_{i=n}^{n} a_i = a_n = \prod_{i=n}^{n} a_i$$

A.4. Logarithmieren, Potenzieren und Radizieren

Die Schreibweise a^b ist eine Abkürzung für

$$a^b =_{\text{def}} \underbrace{a \cdot a \cdot \ldots \cdot a}_{b-\text{mal}}$$

und wird als *Potenzierung* bezeichnet. Dabei ist a die *Basis*, b der *Exponent* und a^b die b-te *Potenz* von a. Seien nun $r, s, t \in \mathbb{R}$ und $r, t \geq 0$ durch die folgende Gleichung verbunden:

$$r^s = t.$$

Dann lässt sich diese Gleichung wie folgt umstellen und es gelten die folgenden Rechenregeln:

Logarithmieren	Potenzieren	Radizieren
$\mathbf{s = \log_r t}$	$\mathbf{t = r^s}$	$\mathbf{r = \sqrt[s]{t}}$
i) $\log_r(\frac{u}{v}) = \log_r u - \log_r v$	i) $r^u \cdot r^v = r^{u+v}$	i) $\sqrt[s]{u} \cdot \sqrt[s]{v} = \sqrt[s]{u \cdot v}$
ii) $\log_r(u \cdot v) = \log_r u + \log_r v$	ii) $\frac{r^u}{r^v} = r^{u-v}$	ii) $\frac{\sqrt[s]{u}}{\sqrt[s]{v}} = \sqrt[s]{\left(\frac{u}{v}\right)}$
iii) $\log_r(t^u) = u \cdot \log_r t$	iii) $u^s \cdot v^s = (u \cdot v)^s$	iii) $\sqrt[u]{\sqrt[v]{t}} = \sqrt[u \cdot v]{t}$
iv) $\log_r(\sqrt[u]{t}) = \frac{1}{u} \cdot \log_r t$	iv) $\frac{u^s}{v^s} = \left(\frac{u}{v}\right)^s$	
v) $\frac{\log_r t}{\log_r u} = \log_u t$ (Basiswechsel)	v) $(r^u)^v = r^{u \cdot v}$	

Zusätzlich gilt: Wenn $r > 1$, dann ist $s_1 < s_2$ gdw. $r^{s_1} < r^{s_2}$ (Monotonie).

Da $\sqrt[s]{t} = t^{\left(\frac{1}{s}\right)}$ gilt, können die Gesetze für das Radizieren leicht aus den Potenzierungsgesetzen abgeleitet werden. Weiterhin legen wir spezielle Schreibweisen für die Logarithmen zur Basis 10, e (Eulersche Zahl) und 2 fest: $\lg t =_{\text{def}} \log_{10} t$, $\ln t =_{\text{def}} \log_e t$ und $\operatorname{lb} t =_{\text{def}} \log_2 t$.

A.5. Gebräuchliche griechische Buchstaben

In der Informatik, Mathematik und Physik ist es üblich, griechische Buchstaben zu verwenden. Ein Grund hierfür ist, dass es so möglich wird mit einer größeren Anzahl von Unbekannten arbeiten zu können, ohne unübersichtliche und oft unhandliche Indizes benutzen zu müssen.

Kleinbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
α	Alpha	β	Beta	γ	Gamma
δ	Delta	ϕ	Phi	φ	Phi
ξ	Xi	ζ	Zeta	ϵ	Epsilon
θ	Theta	λ	Lambda	π	Pi
σ	Sigma	η	Eta	μ	Mu

Großbuchstaben:

Symbol	Bezeichnung	Symbol	Bezeichnung	Symbol	Bezeichnung
Γ	Gamma	Δ	Delta	Φ	Phi
Ξ	Xi	Θ	Theta	Λ	Lambda
Π	Pi	Σ	Sigma	Ψ	Psi
Ω	Omega				

B. Einige (wenige) Grundlagen der elementaren Logik

Aussagen sind entweder *wahr* ($\triangleq 1$) oder *falsch* ($\triangleq 0$). So sind die Aussagen

„Wiesbaden liegt am Mittelmeer“ und „$1 = 7$“

sicherlich falsch, wogegen die Aussagen

„Wiesbaden liegt in Hessen“ und „$11 = 11$“

sicherlich wahr sind. Aussagen werden meist durch *Aussagenvariablen* formalisiert, die nur die Werte 0 oder 1 annehmen können. Oft verwendet man auch eine oder mehrere Unbekannte, um eine Aussage zu parametrisieren. So könnte „$P(x)$“ etwa für „Wiesbaden liegt im Bundesland x“ stehen, d.h. „P(Hessen)“ wäre wahr, wogegen „P(Bayern)“ eine falsche Aussage ist. Solche Konstrukte mit Parameter nennt man auch *Prädikat* oder *Aussageformen*.

Um die Verknüpfung von Aussagen auch formal aufschreiben zu können, werden die folgenden logischen Operatoren verwendet

Symbol	umgangssprachlicher Name	Name in der Logik
$\wedge$	und	Konjunktion
$\vee$	oder	Disjunktion / Alternative
$\neg$	nicht	Negation
$\rightarrow$	folgt	Implikation
$\leftrightarrow$	genau dann wenn (*gdw.*)	Äquivalenz

Zusätzlich werden noch die Quantoren $\exists$ („es existiert“) und $\forall$ („für alle“) verwendet, die z.B. wie folgt gebraucht werden können

$\forall x\colon P(x)$ bedeutet „Für alle x gilt die Aussage $P(x)$“.

$\exists x\colon P(x)$ bedeutet „Es existiert ein x, für das die Aussage $P(x)$ gilt“.

Üblicherweise läßt man sogar den Doppelpunkt weg und schreibt statt $\forall x\colon P(x)$ vereinfachend $\forall x P(x)$. Die Aussageform $P(x)$ wird auch als Prädikat (siehe Definition 110 auf Seite 67) bezeichnet, weshalb man von *Prädikatenlogik* spricht.

Beispiel 120: *Die Aussage „Jede gerade natürliche Zahl kann als Produkt von* 2 *und einer anderen natürlichen Zahl geschrieben werden“ lässt sich dann wie folgt schreiben*

$$\forall n \in \mathbb{N}\colon ((n \textit{ ist gerade}) \rightarrow (\exists m \in \mathbb{N}\colon n = 2 \cdot m))$$

Die folgende logische Formel wird wahr gdw. n eine ungerade natürliche Zahl ist.

$$\exists m \in \mathbb{N}\colon (n = 2 \cdot m + 1)$$

Für die logischen Konnektoren sind die folgenden Wahrheitswertetafeln festgelegt:

p	$\neg p$
0	1
1	0

und

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Jetzt kann man Aussagen auch etwas komplexer verknüpfen:

Beispiel 121: *Nun wird der $\wedge$-Operator verwendet werden. Dazu soll die Aussage „Für alle natürlichen Zahlen n und m gilt, wenn n kleiner gleich m und m kleiner gleich n gilt, dann ist m gleich n“*

$$\forall n, m \in \mathbb{N}\,(((n \leq m) \wedge (m \leq n)) \rightarrow (n = m))$$

Oft benutzt man noch den negierten Quantor $\nexists$ („es existiert kein“).

Cubum autem in duos cubos, aut quadrato-quadratum in duos quadrato-quadratos, et generaliter nullam in infinitum ultra quadratum potestatem in duos eiusdem nominis fas est dividere cuius rei demonstrationem mirabilem sane detexi. Hanc marginis exiguitas non caperet.

Beispiel 122 („Großer Satz von Fermat“): *Die Richtigkeit dieser Aussage konnte erst* 1994 *nach mehr als* 350 *Jahren von Andrew Wiles und Richard Taylor gezeigt werden:*

$$\forall n \in \mathbb{N}\,\nexists a, b, c \in \mathbb{N}\,(((n > 2) \wedge (a \cdot b \cdot c \neq 0)) \rightarrow a^n + b^n = c^n)$$

Für den Fall $n = 2$ hat die Gleichung $a^n + b^n = c^n$ unendlich viele ganzzahlige Lösungen (die so genannten Pythagoräischen Zahlentripel) wie z.B. $3^2 + 4^2 = 5^2$. Diese sind seit mehr als 3500 *Jahren bekannt und haben z.B. geholfen die Cheops-Pyramide zu bauen.*

C. Einige formale Grundlagen von Beweistechniken

Praktisch arbeitende Informatiker glauben oft völlig ohne (formale) Beweistechniken auskommen zu können. Dabei meinen sie sogar, dass formale Beweise keinerlei Berechtigung in der Praxis der Informatik haben und bezeichnen solches Wissen als „in der Praxis irrelevantes Zeug, das nur von und für seltsame Wissenschaftler erfunden wurde“. Studenten in den ersten Semestern unterstellen sogar oft, dass mathematische Grundlagen und Beweistechniken nur als „Filter“ dienen, um die Anzahl der Studenten zu reduzieren. Oft stellen sich beide Gruppen auf den Standpunkt, dass die Korrektheit von Programmen und Algorithmen durch „Lassen wir es doch mal laufen und probieren es aus!“ ($\triangleq$ Testen) belegt werden könne[30]. Diese Einstellung zeigt sich oft auch darin, dass Programme mit Hilfe einer IDE schnell „testweise“ übersetzt werden, in der Hoffnung oder (wesentlich schlimmer) in der Überzeugung, dass jedes übersetzbare Programm ($\triangleq$ syntaktisch korrekt) automatisch auch semantisch korrekt ist. Das diese Einstellung völlig falsch ist zeigen Myriaden in der Praxis auftretende Softwarefehler eindrücklich.

Theoretiker, die sich mit den Grundlagen der Informatik beschäftigen, vertreten oft den Standpunkt, dass die Korrektheit *jedes* Programms rigoros *bewiesen* werden muss. Wahrscheinlich ist die Position zwischen diesen beiden Extremen richtig, denn zum einen ist der formale Beweis von (großen) Programmen oft nicht praktikabel (oder möglich) und zum anderen kann das Testen mit einer (relativ kleinen) Menge von Eingaben sicherlich nicht belegen, dass ein Programm vollständig den Spezifikationen entspricht. Im praktischen Einsatz ist es dann oft mit Eingaben konfrontiert, die zu einer fehlerhaften Reaktion führen oder es sogar abstürzen[31] lassen. Bei einfacher Anwendersoftware sind solche Fehler ärgerlich, aber oft zu verschmerzen. Bei sicherheitskritischer Software (z.B. bei der Regelung von Atomkraftwerken, Airbags und

[30] Diese Bemerkung ist natürlich etwas polemisch, da Testen natürlich die Qualität einer Software verbessert. Weiterhin werden solche Tests systematisch durchgeführt und haben somit eine ganz andere Qualität als „herumprobieren“.

[31] Dies wird eindrucksvoll durch viele Softwarepakete und verbreitete Betriebssysteme im PC-Umfeld belegt.

Bremssystemen in Autos, in der Medizintechnik, bei Finanztransaktionssystemen oder bei der Steuerung von Raumsonden) gefährden solche Fehler menschliches Leben oder führen zu extrem hohen finanziellen Verlusten und müssen deswegen unbedingt vermieden werden.

Für den Praktiker bringen Kenntnisse über formale Beweise aber noch andere Vorteile. Viele Beweise beschreiben direkt den zur Lösung benötigten Algorithmus, d.h. eigentlich wird die Richtigkeit einer Aussage durch die (implizite) Angabe eines Algorithmus gezeigt. Aber es gibt noch einen anderen Vorteil. Ist der umzusetzende Algorithmus komplex (z.B. aufgrund einer komplizierten Schleifenstruktur oder einer verschachtelten Rekursion), so ist es unwahrscheinlich, eine korrekte Implementation an den Kunden liefern zu können, ohne die Hintergründe ($\triangleq$ Beweis) verstanden zu haben. All dies zeigt, dass auch ein praktischer Informatiker Einblicke in Beweistechniken haben sollte. Interessanterweise zeigt die persönliche Erfahrung im praktischen Umfeld auch, dass solches (theoretisches) Wissen über die Hintergründe oft zu klarer strukturierten und effizienteren Programmen führt.

Aus diesen Gründen sollen in den folgenden Abschnitten einige grundlegende Beweistechniken mit Hilfe von Beispielen (unvollständig) kurz vorgestellt werden.

C.1. Direkte Beweise

Um einen *direkten Beweis* zu führen, müssen wir, beginnend von einer initialen Aussage ($\triangleq$ Hypothese), durch Angabe einer Folge von (richtigen) Zwischenschritten zu der zu beweisenden Aussage ($\triangleq$ Folgerung) gelangen. Jeder Zwischenschritt ist dabei entweder unmittelbar klar oder muss wieder durch einen weiteren (kleinen) Beweis belegt werden. Dabei müssen nicht alle Schritte völlig formal beschrieben werden, sondern es kommt darauf an, dass sich dem Leser die eigentliche Strategie erschließt.

Satz 123: *Sei $n \in \mathbb{N}$. Falls $n \geq 4$, dann ist $2^n \geq n^2$.*

Wir müssen also, in Abhängigkeit des Parameters n, die Richtigkeit dieser Aussage belegen. Einfaches Ausprobieren ergibt, dass $2^4 = 16 \geq 16 = 4^2$ und $2^5 = 32 \geq 25 = 5^2$, d.h. intuitiv scheint die Aussage richtig zu sein. Wir wollen die Richtigkeit der Aussage nun durch eine Reihe von (kleinen) Schritten belegen:

Beweis:

Wir haben schon gesehen, dass die Aussage für $n = 4$ und $n = 5$ richtig ist. Erhöhen wir n auf $n+1$, so verdoppelt sich der Wert der linken Seite der Ungleichung von 2^n auf $2 \cdot 2^n = 2^{n+1}$. Für die rechte Seite ergibt sich ein Verhältnis von $(\frac{n+1}{n})^2$. Je größer n wird, desto kleiner wird der Wert $\frac{n+1}{n}$, d.h. der maximale Wert ist bei $n = 4$ mit 1.25 erreicht. Wir wissen $1.25^2 = 1.5625$. D.h. immer wenn wir n um eins erhöhen, verdoppelt sich der Wert der linken Seite, wogegen sich der Wert der rechten Seite um maximal das 1.5625 fache erhöht. Damit muss die linke Seite der Ungleichung immer größer als die rechte Seite sein. #

Dieser Beweis war nur wenig formal, aber sehr ausführlich und wurde am Ende durch das Symbol „#“ markiert. Im Laufe der Zeit hat es sich eingebürgert, das Ende eines Beweises mit einem besonderen Marker abzuschließen. Besonders bekannt ist hier „qed“, eine Abkürzung für die lateinische Floskel „quod erat demonstrandum“, die mit „was zu beweisen war“ übersetzt werden kann. In neuerer Zeit werden statt „qed“ mit der gleichen Bedeutung meist die Symbole „□“ oder „#“ verwendet.

Nun stellt sich die Frage: „Wie formal und ausführlich muss ein Beweis sein?“ Diese Frage kann so einfach nicht beantwortet werden, denn das hängt u.a. davon ab, welche Lesergruppe durch den Beweis von der Richtigkeit einer Aussage überzeugt werden soll und wer den Beweis schreibt. Ein Beweis für ein Übungsblatt sollte auch auf Kleinigkeiten Rücksicht nehmen, wogegen ein solcher Stil für eine wissenschaftliche Zeitschrift vielleicht nicht angebracht wäre,

da die potentielle Leserschaft über ganz andere Erfahrungen und viel mehr Hintergrundwissen verfügt. Nun noch eine Bemerkung zum Thema „Formalismus“: Die menschliche Sprache ist unpräzise, mehrdeutig und Aussagen können oft auf verschiedene Weise interpretiert werden, wie das tägliche Zusammenleben der Geschlechter eindrucksvoll demonstriert. Diese Defizite sollen Formalismen[32] ausgleichen, d.h. die Antwort muss lauten: „So viele Formalismen wie notwendig und so wenige wie möglich!“. Durch Übung und Praxis lernt man die Balance zwischen diesen Anforderungen zu halten und es zeigt sich bald, dass „Geübte“ die formale Beschreibung sogar wesentlich leichter verstehen.

Oft kann man andere, schon bekannte, Aussagen dazu verwenden, die Richtigkeit einer neuen (evtl. kompliziert wirkenden) Aussage zu belegen.

Satz 124: *Sei $n \in \mathbb{N}$ die Summe von 4 Quadratzahlen, die größer als 0 sind, dann muss $2^n \geq n^2$ sein.*

Beweis: Die Menge der Quadratzahlen ist $\mathbb{S} = \{0, 1, 4, 9, 16, 25, 36, \dots\}$, d.h. 1 ist die kleinste Quadratzahl, die größer als 0 ist. Damit muss unsere Summe von 4 Quadratzahlen größer oder gleich als 4 sein. Die Aussage folgt direkt aus Satz 123. #

C.1.1. Die Kontraposition

Mit Hilfe von direkten Beweisen haben wir Zusammenhänge der Form „Wenn Aussage H richtig ist, dann folgt daraus die Aussage C“ untersucht. Manchmal ist es schwierig, einen Beweis für eine solchen Zusammenhang zu finden. Völlig gleichwertig ist die Behauptung „Wenn die Aussage C falsch ist, dann ist die Aussage H falsch“ und oft ist eine solche Aussage leichter zu zeigen.

Die *Kontraposition* von Satz 123 ist also die folgende Aussage: „Wenn nicht $2^n \geq n^2$, dann gilt nicht $n \geq 4$.“. Das entspricht der Aussage: „Wenn $2^n < n^2$, dann gilt $n < 4$.“, was offensichtlich zu der ursprünglichen Aussage von Satz 123 gleichwertig ist.

Diese Technik ist oft besonders hilfreich, wenn man die Richtigkeit einer Aussage zeigen soll, die aus zwei Teilaussagen zusammengesetzt und die durch ein „genau dann wenn“[33] verknüpft sind. In diesem Fall sind zwei Teilbeweise zu führen, denn zum einen muss gezeigt werden, dass aus der ersten Aussage die zweite folgt und umgekehrt muss gezeigt werden, dass aus der zweiten Aussage die erste folgt.

Satz 125: *Eine natürliche Zahl n ist durch drei teilbar genau dann, wenn die Quersumme ihrer Dezimaldarstellung durch drei teilbar ist.*

Beweis: Für die Dezimaldarstellung von n gilt

$$n = \sum_{i=0}^{k} a_i \cdot 10^i, \text{wobei } a_i \in \{0, 1, \dots, 9\} \text{ („Ziffern“) und } 0 \leq i \leq k.$$

Mit $\mathrm{QS}(n)$ wird die Quersumme von n bezeichnet, d.h. $\mathrm{QS}(n) = \sum_{i=0}^{k} a_i$. Mit Hilfe einer einfachen vollständigen Induktion kann man zeigen, dass für jedes $i \geq 0$ ein $b \in \mathbb{N}$ existiert, sodass $10^i = 9b + 1$. Damit gilt $n = \sum_{i=0}^{k} a_i \cdot 10^i = \sum_{i=0}^{k} a_i(9b_i + 1) = \mathrm{QS}(n) + 9\sum_{i=0}^{k} a_i b_i$, d.h. es existiert ein $c \in \mathbb{N}$, so dass $n = \mathrm{QS}(n) + 9c$.

„$\Rightarrow$“: Wenn n durch 3 teilbar ist, dann muss auch $\mathrm{QS}(n)+9c$ durch 3 teilbar sein. Da $9c$ sicherlich durch 3 teilbar ist, muss auch $\mathrm{QS}(n) = n - 9c$ durch 3 teilbar sein.

[32] In diesem Zusammenhang sind Programmiersprachen auch Formalismen, die eine präzise Beschreibung von Algorithmen erzwingen und die durch einen Compiler verarbeitet werden können.

[33] Oft wird „genau dann wenn“ durch *gdw.* abgekürzt.

„⇐“: Dieser Fall soll durch Kontraposition gezeigt werden. Sei nun n nicht durch 3 teilbar, dann darf $\mathrm{QS}(n)$ nicht durch 3 teilbar sein, denn sonst wäre $n = 9c + \mathrm{QS}(n)$ durch 3 teilbar. #

C.2. Der Ringschluss

Oft findet man mehrere Aussagen, die zueinander äquivalent sind. Ein Beispiel dafür ist Satz 126. Um die Äquivalenz dieser Aussagen zu beweisen, müssten jeweils zwei „genau dann wenn“ Beziehungen untersucht werden, d.h. es werden vier Teilbeweise notwendig. Dies kann mit Hilfe eines so genannten *Ringschlusses* abgekürzt werden, denn es reicht zu zeigen, dass aus der ersten Aussage die zweite folgt, aus der zweiten Aussage die dritte und dass schließlich aus der dritten Aussage wieder die erste folgt. Im Beweis zu Satz 126 haben wir deshalb nur drei anstatt vier Teilbeweise zu führen, was zu einer Arbeitsersparnis führt. Diese Arbeitsersparnis wird um so größer, je mehr äquivalente Aussagen zu untersuchen sind. Dabei ist die Reihenfolge der Teilbeweise nicht wichtig, solange die einzelnen Teile zusammen einen Ring bilden.

Satz 126: *Seien A und B zwei beliebige Mengen, dann sind die folgenden drei Aussagen äquivalent:*

i) $A \subseteq B$

ii) $A \cup B = B$

iii) $A \cap B = A$

Beweis: Im folgenden soll ein Ringschluss verwendet werden, um die Äquivalenz der drei Aussagen zu zeigen:
„i) ⇒ ii)“: Da nach Voraussetzung $A \subseteq B$ ist, gilt für jedes Element $a \in A$ auch $a \in B$, d.h. in der Vereinigung $A \cup B$ sind alle Elemente nur aus B, was $A \cup B = B$ zeigt.

„ii) ⇒ iii)“: Wenn $A \cup B = B$ gilt, dann ergibt sich durch Einsetzen und mit den Regeln aus Abschnitt A.1.4 (Absorptionsgesetz) direkt $A \cap B = A \cap (A \cup B) = A$.

„iii) ⇒ i)“: Sei nun $A \cap B = A$, dann gibt es kein Element $a \in A$ für das $a \notin B$ gilt. Dies ist aber gleichwertig zu der Aussage $A \subseteq B$.
Damit hat sich ein Ring von Aussagen „i) ⇒ ii)“, „ii) ⇒ iii)“ und „iii) ⇒ i)“ gebildet, was die Äquivalenz aller Aussagen zeigt. #

C.3. Widerspruchsbeweise

Obwohl die Technik der Widerspruchsbeweise auf den ersten Blick sehr kompliziert erscheint, ist sie meist einfach anzuwenden, extrem mächtig und liefert oft sehr kurze Beweise. Angenommen wir sollen die Richtigkeit einer Aussage „aus der Hypothese H folgt C“ zeigen. Dazu beweisen wir, dass sich ein Widerspruch ergibt, wenn wir, von H und der Annahme, dass C falsch ist, ausgehen. Also war die Annahme falsch, und die Aussage C muss richtig sein.

Anschaulicher wird diese Beweistechnik durch folgendes Beispiel: Nehmen wir einmal an, dass Alice eine bürgerliche Frau ist und deshalb auch keine Krone trägt. Es ist klar, dass jede Königin eine Krone trägt. Wir sollen nun beweisen, dass Alice keine Königin ist. Dazu nehmen wir an, dass Alice eine Königin ist, d.h. Alice trägt eine Krone. Dies ist ein Widerspruch! Also war unsere Annahme falsch, und wir haben gezeigt, dass Alice keine Königin sein kann.

Der Beweis zu folgendem Satz verwendet diese Technik:

Satz 127: *Sei S eine endliche Untermenge einer unendlichen Menge U. Sei T das Komplement von S bzgl. U, dann ist T eine unendliche Menge.*

Beweis: Hier ist unsere Hypothese „S endlich, U unendlich und T Komplement von S bzgl. U“ und unsere Folgerung ist „T ist unendlich“. Wir nehmen also an, dass T eine endliche Menge ist. Da T das Komplement von S ist, gilt $S \cap T = \emptyset$, also ist $\#(S) + \#(T) = \#(S \cap T) + \#(S \cup T) = \#(S \cup T) = n$, wobei n eine Zahl aus $\mathbb{N}$ ist (siehe Abschnitt A.1.6). Damit ist $S \cup T = U$ eine endliche Menge. Dies ist ein Widerspruch zu unserer Hypothese! Also war die Annahme „T ist endlich“ falsch. #

C.4. Der Schubfachschluss

Der *Schubfachschluss* ist auch als *Dirichlets Taubenschlagprinzip* bekannt. Werden $n > k$ Tauben auf k Boxen verteilt, so gibt es mindestens eine Box in der sich wenigstens zwei Tauben aufhalten. Allgemeiner formuliert sagt das Taubenschlagprinzip, dass wenn n Objekte auf k Behälter aufgeteilt werden, dann gibt es mindestens eine Box die mindestens $\lceil \frac{n}{k} \rceil$ Objekte enthält.

Beispiel 128: *Auf einer Party unterhalten sich 8 Personen ($\triangleq$ Objekte), dann gibt es mindestens einen Wochentag ($\triangleq$ Box) an dem $\lceil \frac{8}{7} \rceil = 2$ Personen aus dieser Gruppe Geburtstag haben.*

C.5. Gegenbeispiele

Im wirklichen Leben wissen wir nicht, ob eine Aussage richtig oder falsch ist. Oft sind wir dann mit einer Aussage konfrontiert, die auf den ersten Blick richtig ist und sollen dazu ein Programm entwickeln. Wir müssen also entscheiden, ob diese Aussage wirklich richtig ist, denn sonst ist evtl. alle Arbeit umsonst und hat hohen Aufwand verursacht. In solchen Fällen kann man versuchen, ein einziges Beispiel dafür zu finden, dass die Aussage falsch ist, um so unnötige Arbeit zu sparen.

Wir zeigen, dass die folgenden Vermutungen falsch sind:

Vermutung 129: *Wenn $p \in \mathbb{N}$ eine Primzahl ist, dann ist p ungerade.*

Gegenbeispiel: Die natürliche Zahl 2 ist eine Primzahl und 2 ist gerade. #

Vermutung 130: *Es gibt keine Zahlen $a, b \in \mathbb{N}$, sodass $a \bmod b = b \bmod a$.*

Gegenbeispiel: Für $a = b = 2$ gilt $a \bmod b = b \bmod a = 0$. #

C.6. Induktionsbeweise und das Induktionsprinzip

Sei nun eine Menge von natürlichen Zahlen X mit den folgenden zwei Eigenschaften gegeben:

(IA) $0 \in X$

(IS) Ist eine beliebige natürliche Zahl n ein Element von X, so ist auch die Zahl $n+1$ ein Element von X.

Man kann sich nun leicht überlegen, dass dann X alle natürlichen Zahlen enthält, d.h. es gilt $X = \mathbb{N}$. Mit Hilfe dieser Beobachtung konstruieren wir eine der nützlichsten Beweismethoden in der Informatik bzw. Mathematik: Das *Induktionsprinzip* bzw. die Methode des *Induktionsbeweises*. Die Idee funktioniert mit folgender Beobachtung:

> Angenommen man kann nachweisen, dass 0 die Eigenschaft E hat[34] (kurz: $E(0)$) und weiterhin, dass wenn n die Eigenschaft E hat, dann gilt auch $E(n+1)$. Ist dies der Fall, so muss jede natürliche Zahl die Eigenschaft E haben.

Im Folgenden wollen wir nachweisen, dass für jedes $n \in \mathbb{N}$ eine bestimmte Eigenschaft E gilt. Wir schreiben also abkürzend $E(n)$ für die Aussage „n besitzt die Eigenschaft E“, d.h. der Schreibweise $E(0)$ drücken wir also aus, dass die erste natürliche Zahl 0 die Eigenschaft E besitzt, dann erhalten wir die folgende Vorgehensweise:

Induktionsprinzip: Es gelten
(IA) $E(0)$
(IS) Für $n \geq 0$ gilt, wenn $E(n)$ korrekt ist, dann ist auch $E(n+1)$ richtig.

Sind diese beiden Aussagen erfüllt, so hat jede natürliche Zahl die Eigenschaft E. Dabei ist **IA** die Abkürzung für *Induktionsanfang* und **IS** ist die Kurzform von *Induktionsschritt*. Die Voraussetzung ($\triangleq$ Hypothese) $E(n)$ ist korrekt für n und wird im Induktionsschritt als *Induktionsvoraussetzung* benutzt (kurz: **IV**). Hat man also den Induktionsanfang und den Induktionsschritt gezeigt, dann ist es anschaulich, dass jede natürliche Zahl die Eigenschaft E haben muss.
Es gibt verschiedene Versionen von Induktionsbeweisen. Die bekannteste Version ist die vollständige Induktion, bei der Aussagen über natürliche Zahlen gezeigt werden.

C.6.1. Die vollständige Induktion

Wie in Piratenfilmen üblich, seien Kanonenkugeln in einer Pyramide mit quadratischer Grundfläche gestapelt. Wir stellen uns die Frage, wieviele Kugeln (in Abhängigkeit von der Höhe) in einer solchen Pyramide gestapelt sind.

Satz 131: *Mit einer quadratische Pyramide aus Kanonenkugeln der Höhe $n \geq 1$ als Munition, können wir $\frac{n(n+1)(2n+1)}{6}$ Schüsse abgeben.*

Beweis: Einfacher formuliert: wir sollen zeigen, dass $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$.
(IA) Eine Pyramide der Höhe $n = 1$ enthält $\frac{1 \cdot 2 \cdot 3}{6} = 1$ Kugel, d.h. wir haben die Eigenschaft für $n = 1$ verifiziert.
(IV) Für $k \leq n$ gilt $\sum_{i=1}^{k} i^2 = \frac{k(k+1)(2k+1)}{6}$.
(IS) Wir müssen nun zeigen, dass $\sum_{i=1}^{n+1} i^2 = \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}$ gilt und dabei muss die Induktionsvoraussetzung $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ benutzt werden. Es ergeben sich die folgenden Schritte:

[34] Mit E wird also ein Prädikat oder Aussagenform bezeichnet (siehe Abschnitt A.1.2)

$$\begin{aligned}
\sum_{i=1}^{n+1} i^2 &= \sum_{i=1}^{n} i^2 + (n+1)^2 \\
&\overset{\textbf{(IV)}}{=} \frac{n(n+1)(2n+1)}{6} + (n^2+2n+1) \\
&= \frac{2n^3+3n^2+n}{6} + (n^2+2n+1) \\
&= \frac{2n^3+9n^2+13n+6}{6} \\
&= \frac{(n+1)(2n^2+7n+6)}{6} \quad (\star) \\
&= \frac{(n+1)(n+2)(2n+3)}{6} \quad (\star\star) \\
&= \frac{(n+1)((n+1)+1)(2(n+1)+1)}{6}
\end{aligned}$$

Die Zeile $\star$ (bzw. $\star\star$) ergibt sich, indem man $2n^3+9n^2+13n+6$ durch $n+1$ teilt (bzw. $2n^2+7n+6$ durch $n+2$). #

Das Induktionsprinzip kann man auch variieren. Dazu geht man davon aus, dass die Eigenschaft E für alle Zahlen $k \leq n$ erfüllt ist (Induktionsvoraussetzung). Obwohl dies auf den ersten Blick wesentlich komplizierter erscheint, vereinfacht dieser Ansatz oft den Beweis:

Verallgemeinertes Induktionsprinzip: Wenn die zwei Aussagen (also Induktionsanfang und Induktionsschritt)
(IA) $E(0)$
(IS) Wenn für alle $0 \leq k \leq n$ die Eigenschaft $E(k)$ gilt, dann ist auch $E(n+1)$ richtig,

gelten, dann haben alle natürliche Zahlen die Eigenschaft E. Damit ist das verallgemeinerte Induktionsprinzip eine Verallgemeinerung des weiter oben vorgestellten Induktionsprinzips, wie das folgende Beispiel veranschaulicht:

Satz 132: *Jede natürliche Zahl $n \geq 2$ läßt sich als Produkt von Primzahlen schreiben.*

Beweis: Das verallgemeinerte Induktionsprinzip wird wie folgt verwendet:
(IA) Offensichtlich ist 2 das Produkt von einer Primzahl.
(IV) Jede natürliche Zahl m mit $2 \leq m \leq n$ kann als Produkt von Primzahlen geschrieben werden.
(IS) Nun wird eine Fallunterscheidung durchgeführt:

i) Sei $n+1$ wieder eine Primzahl, dann ist nichts zu zeigen, da $n+1$ direkt ein Produkt von Primzahlen ist.

ii) Sei $n+1$ keine Primzahl, dann existieren mindestens zwei Zahlen p und q mit $2 \leq p, q < n+1$ und $p \cdot q = n+1$. Nach Induktionsvoraussetzung sind dann p und q wieder als Produkt von Primzahlen darstellbar. Etwa $p = p_1 \cdot p_2 \cdot \ldots \cdot p_s$ und $q = q_1 \cdot q_2 \cdot \ldots \cdot q_t$. Damit ist aber $n+1 = p \cdot q = p_1 \cdot p_2 \cdot \ldots \cdot p_s \cdot q_1 \cdot q_2 \cdot \ldots \cdot q_t$ ein Produkt von Primzahlen. #

Induktionsbeweise treten z.B. bei der Analyse von Programmen immer wieder auf und spielen deshalb für die Informatik eine ganz besonders wichtige Rolle.

C.6.2. Induktive Definitionen

Das Induktionsprinzip kann man aber auch dazu verwenden, (Daten-)Strukturen formal zu spezifizieren. Dies macht diese Technik für Anwendungen in der Informatik besonders interessant. Dazu werden in einem ersten Schritt ($\triangleq$ Induktionsanfang) die „atomaren“ Objekte definiert und dann in einem zweiten Schritt die zusammengesetzten Objekte ($\triangleq$ Induktionsschritt). Diese Technik ist als *induktive Definition* bekannt.

Beispiel 133: *Die Menge der* binären Bäume *ist wie folgt definiert:*

(IA) *Ein einzelner Knoten w ist ein* Baum *und w ist die* Wurzel *dieses Baums.*

(IS) *Seien $T_1, T_2, \ldots, T_n$ Bäume mit den Wurzeln $k_1, \ldots, k_n$ und w ein einzelner neuer Knoten. Verbinden wir den Knoten w mit allen Wurzeln $k_1, \ldots, k_n$, dann entsteht ein neuer Baum mit der Wurzel w. Nichts sonst ist ein Baum.*

Beispiel 134: *Die Menge der* arithmetischen Ausdrücke *ist wie folgt definiert:*

(IA) *Jeder Buchstabe und jede Zahl ist ein arithmetischer Ausdruck.*
(IS) *Seien E und F Ausdrücke, so sind auch $E + F$, $E * F$ und $[E]$ Ausdrücke. Nichts sonst ist ein Ausdruck.*

D.h. x, $x + y$, $[2 * x + z]$ sind arithmetische Ausdrücke, aber beispielsweise sind $x+$, yy, $][x + y$ sowie $x + *z$ keine arithmetischen Ausdrücke im Sinn dieser Definition.

Beispiel 135: *Die Menge der* aussagenlogischen Formeln *ist wie folgt definiert:*

(IA) *Jede aussagenlogische Variable $x_1, x_2, x_3, \ldots$ ist eine aussagenlogische Formel.*
(IS) *Seien H_1 und H_2 aussagenlogische Formeln, so sind auch $(H_1 \wedge H_2)$, $(H_1 \vee H_2)$, $\neg H_1$, $(H_1 \leftrightarrow H_2)$, $(H_1 \rightarrow H_2)$ und $(H_1 \oplus H_2)$ aussagenlogische Formeln. Nichts sonst ist eine aussagenlogische Formel.*

Bei diesen Beispielen ahnt man schon, dass solche Techniken zur präzisen und eleganten Definition von Programmiersprachen und Dateiformaten gute Dienste leisten. Es zeigt sich, dass die im Compilerbau verwendeten Chomsky-Grammatiken eine andere Art von induktiven Definitionen darstellen. Darüber hinaus bieten induktive Definitionen noch weitere Vorteile, denn man kann oft relativ leicht Induktionsbeweise konstruieren, die Aussagen über induktiv definierte Objekte belegen / beweisen.

C.6.3. Die strukturelle Induktion

Satz 136: *Die Anzahl der öffnenden Klammern eines arithmetischen Ausdrucks stimmt mit der Anzahl der schließenden Klammern überein.*

Es ist offensichtlich, dass diese Aussage richtig ist, denn in Ausdrücken wie $(x + y)/2$ oder $x + ((y/2) * z)$ muss ja zu jeder öffnenden Klammer eine schließende Klammer existieren. Der nächste Beweis verwendet diese Idee um die Aussage von Satz 136 mit Hilfe einer *strukturellen Induktion* zu zeigen.
Beweis: Wir bezeichnen die Anzahl der öffnenden Klammern eines Ausdrucks E mit $\#_[(E)$ und verwenden die analoge Notation $\#_](E)$ für die Anzahl der schließenden Klammern.
(IA) Die einfachsten Ausdrücke sind Buchstaben und Zahlen. Die Anzahl der öffnenden und schließenden Klammern ist in beiden Fällen gleich 0.
(IV) Sei E ein Ausdruck, dann gilt $\#_[(E) = \#_](E)$.
(IS) Für einen Ausdruck $E+F$ gilt $\#_[(E+F) = \#_[(E)+\#_[(F) \overset{\text{IV}}{=} \#_](E)+\#_](F) = \#_](E+F)$. Völlig analog zeigt man dies für $E * F$. Für den Ausdruck $[E]$ ergibt sich $\#_[([E]) = \#_[(E)+1 \overset{\text{IV}}{=} \#_](E) + 1 = \#_]([E])$. In jedem Fall ist die Anzahl der öffnenden Klammern gleich der Anzahl der schließenden Klammern. #

Mit Hilfe von Satz 136 können wir nun leicht ein Programm entwickeln, das einen Plausibilitätscheck (z.B. direkt in einem Editor) durchführt und die Klammern zählt, bevor die Syntax von arithmetischen Ausdrücken überprüft wird. Definiert man eine vollständige Programmiersprache induktiv, dann werden ganz ähnliche Induktionsbeweise möglich, d.h. man kann die Techniken aus diesem Beispiel relativ leicht auf die Praxis der Informatik übertragen.
Man überlegt sich leicht, dass die natürlichen Zahlen auch induktiv definiert werden können. Damit zeigt sich, dass die vollständige Induktion eigentlich nur ein Spezialfall der strukturellen Induktion ist.

Stichwortverzeichnis

Symbole

A

B

T

U

V

W

Z

Literatur

[ACG+99] G. Ausiello, P. Crescenzi, V. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability.* Springer Verlag, 1999.

[Can95] Georg Cantor. Beiträge zur begründung der transfiniten mengenlehre. *Mathematische Annalen*, 46(4):481–512, 1895.

[Cho56] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 1956.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[Gru99] J. Gruska. *Quantum Computing.* McGraw-Hill, 1999.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley Publishing Company, 2001.

[HMU02] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Pearson Studium, 2002. siehe auch [HMU01].

[Hom08] Matthias Homeister. *Quantum Computing verstehen.* Vieweg, 2008.

[MM06] Christoph Meinel and Martin Mundhenk. *Mathematische Grundlagen der Informatik.* Teubner, 2006.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[Pău98] G. Păun. *Computing with Bio-Molecules.* Springer Series in Discrete Mathematics and Theoretical Computer Science. Springer Verlag, Sinpapore, 1998.

[Rad62] T. Rado. On non-computable functions. *Bell Systeme Technical Journal*, 41:877–884, 1962.

[RV01a] Steffen Reith and Heribert Vollmer. Ist P=NP? Einführung in die Theorie der NP-Vollständigkeit. Technical Report 269, Institut für Informatik, Universität Würzburg, 2001. siehe http://www.informatik.uni-wuerzburg.de/reports/tr.html.

[RV01b] Steffen Reith and Heribert Vollmer. Wer wird Millionär? Komplexitätstheorie: Konzepte und Herausforderungen. *c't Magazin für Computertechnik*, Heft 7, 2001. siehe auch [RV01a].

[Sch01] Uwe Schöning. *Theoretische Informatik - kurzgefasst.* Spektrum Akademischer Verlag, 2001.

[Sip06] Michael Sipser. *Introduction to the Theory of Computation.* Thompson, 2006.

[Tur36] Alan M. Turing. On Computable Numbers with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[Wag03] Klaus W. Wagner. *Theoretische Informatik.* Springer Verlag, 2003.