Prof. Dr. Jürgen Dassow
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

G R U N D L A G E N D E R
T H E O R E T I S C H E N
I N F O R M A T I K

T E I L I

Vorlesungsmanuskript

Magdeburg, Wintersemester 2007/2008

Einleitung

Dieses Manuskript ist aus Vorlesungen zur Theoretischen Informatik hervorgegangen, die ich für Studenten der Fachrichtung Informatik mehrfach an der Otto-von-Guericke-Universität Magdeburg gelesen habe. Es ist aber auch ein völlig neues Skript, da es diesmal das Begleitmaterial zu der Vorlesung im Bachelor-Studium der Fachrichtungen Informatik, Computervisualistik und Ingenieurinformatik ist, die bisher innerhalb des Diplomstudiums verschiedene Kurse mit unterschiedlicher Schwerpunktsetzung hatten. Schon der Stundenumfang weicht in allen Fällen von den bisherigen Lehrveranstaltungen ab.
Daher werden in dieser Vorlesung einige Gebiete innerhalb dieser Vorlesung eine verkürzte Darstellung erfahren, die zum Teil dann für die Studierenden der Informatik im zweiten Teil eine Ergänzung finden wird. Zum anderen wird aber auch auf eine Vollständigkeit in den Beweisen innerhalb der Vorlesung verzichtet, wenn die Beweismethodik vorher schon exemplifiziert wurde; die (vollständigen) Beweise sind dann aber in diesem Skript zu finden.

Die Informatik wird heutzutage oft in vier große Teilgebiete unterteilt: Theoretische Informatik, Technische Informatik, Praktische Informatik, Angewandte Informatik. Dabei sind die Grenzen zwischen diesen Disziplinen fließend und nicht in jedem Fall ist eine eindeutige Einordnung eines Problems oder Sachverhalts möglich.
Die Theoretische Informatik beschäftigt sich im Wesentlichen mit Objekten, Methoden und Problemfeldern, die bei der Abstraktion von Gegenständen und Prozessen der anderen Teilgebiete der Informatik und benachbarter Wissenschaften entstanden sind. So werden im Rahmen der Theorie der formalen Sprachen, einem klassischen Bestandteil der Theoretischen Informatik, solche Grammatiken und Sprachen behandelt, die aus Beschreibungen der Syntax natürlicher Sprachen und Programmiersprachen hervorgegangen sind. Die dabei entwickelten Methoden sind so allgemein, dass sie über diese beiden Anwendungsfelder weit hinausgehen und heute z.B. auch in der theoretischen Biologie bei der Beschreibung der Entwicklung von Organismen benutzt werden.
Natürlich ist es unmöglich im Rahmen dieser Einführung alle Gebiete zu berühren, die der Theoretischen Informatik zugerechnet werden. Es wurde hier eine Konzentration auf die folgenden Problemkreise vorgenommen:

- Im ersten Kapitel werden verschiedene Aspekte des Algorithmenbegriffs, der für die gesamte Informatik ein zentrales Konzept darstellt, behandelt. Es werden Präzisierungen des Begriffs Algorithmus angegeben und gezeigt, dass es Probleme gibt, die mittels Algorithmen nicht zu lösen sind (die also auch von Computern nicht gelöst werden können, da Computer nur implementierte Algorithmen realisieren).

- Das zweite Kapitel ist der Theorie der formalen Sprachen gewidmet. Es werden

verschiedene Klassen von Sprachen durch Grammatiken, Automaten und algebraische Eigenschaften charakterisiert. Ferner werden die verschiedenen Sprachklassen miteinander verglichen und ihre Eigenschaften hinsichtlich Entscheidbarkeitsfragen diskutiert.

- Im dritten Kapitel wird eine Einführung in die Komplexitätstheorie gegeben. Es werden Maße für die Güte von Algorithmen eingeführt. Außerdem wird das Verhältnis von Determinismus und Nichtdeterminismus auf der Basis der Qualität von Algorithmen untersucht.

Von den Gebieten, die trotz ihrer Bedeutung nicht behandelt werden können, seien hier folgende genannt (diese Liste ist nicht vollständig, die Reihenfolge ist mehr zufällig denn eine Rangfolge):

- Theorie der Booleschen Funktionen und Schaltkreise (welche Eingabe-/Ausgabeverhalten lassen sich mittels welcher Schaltkreise beschreiben; wieviel Schaltkreise sind zur Erzeugung gewisser Funktionen notwendig),
- formale Semantik,
- Codierungstheorie und Kryptographie,
- Fragen der Parallelisierung.

Die Ergebnisse und Definitionen sind in jedem Kapitel nummeriert, wobei eine durchgängige Zählung für Sätze, Lemmata, Folgerungen usw. erfolgt. Das Ende eines Beweises wird durch □ angegeben; wird auf den Beweis verzichtet bzw. folgt er direkt aus den vorher gegebenen Erläuterungen, so steht □ am Ende der Formulierung der Aussage.

Jedes Kapitel endet mit eine Serie von Übungsaufgaben zum Gegenstand des Kapitels. Diese sollen es zum einen der Leserin / dem Leser ermöglichen, ihren/seinen Wissensstand zu kontrollieren. Zum anderen geben sie in einigen Fällen zusätzliche Kenntnisse, auf die teilweise im Text verwiesen wird (beim Verweis erfolgt nur die Nennung der Aufgabennummer, wenn die Aufgabe zum gleichen Kapitel gehört; sonst wird auch das Kapitel angeben).

Mein Dank gilt meinen Mitarbeitern Dr. B. Reichel, Dr. H. Bordihn, Dr. Ralf Stiebe und Dr. Bianca Truthe für die vielfältigen Diskussionen zur Darstellung des Stoffes und für das sorgfältige Lesen der Vorgängermanuskripte; ihre Vorschläge zu inhaltlichen Ergänzungen und Umgestaltungen und ihre Hinweise auf Fehler und notwendige Änderungen in Detailfragen führten zu zahlreichen Verbesserungen sowohl des Inhalts selbst und der Anordnung des Stoffes als auch der didaktischen Gestaltung. Herrn Ronny Harbich danke ich für seine vielfältigen Hinweise auf Fehler in einer früheren Variante des Skriptes.

Jürgen Dassow Magdeburg/Lostau, Herbst/Winter 2007/2008

Vorbemerkungen

Im Folgenden setzen wir voraus, dass der Leser über mathematische Kenntnisse verfügt, wie sie üblicherweise in einer Grundvorlesung Mathematik vermittelt werden. Das erforderliche Wissen besteht im Wesentlichen aus Formeln bei kombinatorischen Anzahlproblemen und Summen, Basiswissen in Zahlentheorie, linearer und abstrakter Algebra und Graphentheorie.

Wir verwenden folgende Bezeichnungen für Zahlbereiche:

- $\mathbf{N}$ für die Menge der natürlichen Zahlen $\{1, 2, \ldots\}$,
- $\mathbf{N}_0 = \mathbf{N} \cup \{0\}$,
- $\mathbf{Z}$ für die Menge der ganzen Zahlen,
- $\mathbf{Q}$ für die Menge der rationalen Zahlen,
- $\mathbf{R}$ für die Menge der reellen Zahlen.

Eine Funktion $f : M \to N$ ist stets als eine eindeutige Abbildung aus der Menge M in die Menge N zu verstehen. Wir bezeichnen mit $dom(f)$ und $rg(f)$ den Definitionsbereich bzw. Wertevorrat einer Funktion f. Falls der Definitionsbereich von f mit M identisch ist, sprechen wir von einer *totalen* Funktion, sonst von einer *partiellen* Funktion. Falls M das kartesische Produkt von n Mengen ist, so sprechen wir von einer n-stelligen Funktion (im Fall $n = 0$ ist f also eine Abbildung aus $\{\emptyset\}$ und daher stets total).

Unter einem Alphabet verstehen wir eine endliche nichtleere Menge. Die Elemente eines Alphabets heißen Buchstaben. Endliche Folgen von Buchstaben des Alphabets V nennen wir Wörter über V; Wörter werden durch einfaches Hintereinanderschreiben der Buchstaben angegeben. Unter der Länge $|w|$ eines Wortes w verstehen wir die Anzahl der in w vorkommenden Buchstaben, wobei jeder Buchstabe sooft gezählt wird, wie er in w vorkommt. λ bezeichnet das Leerwort, das der leeren Folge entspricht, also aus keinem Buchstaben besteht und die Länge 0 hat. Mit V^* bezeichnen wir die Menge aller Wörter über V (einschließlich λ) und setzen $V^+ = V^* \setminus \{\lambda\}$.
In V^* definieren wir ein Produkt $w_1 w_2$ der Wörter w_1 und w_1 durch einfaches Hintereinanderschreiben. Für alle Wörter $w, w_1, w_2, w_3 \in V^*$ gelten dann folgende Beziehungen:

$$\begin{aligned} w_1(w_2w_3) &= (w_1w_2)w_3 = w_1w_2w_3 \quad \text{(Assoziativgesetz)}, \\ w\lambda &= \lambda w, \\ |w_1w_2| &= |w_1| + |w_2|. \end{aligned}$$

Dagegen gilt im Allgemeinen nicht $w_1w_2 \neq w_2w_1$ (entsprechend der Definition von Wörtern als Folgen müssen w_1w_2 und w_2w_1 als Folgen gleich sein, was z.B. für $w_1 = ab$, $w_2 = ba$ und damit $w_1w_2 = abba$, $w_2w_1 = baab$ nicht gegeben ist).

Inhaltsverzeichnis

Kapitel 1

Berechenbarkeit und Algorithmen

1.1 Berechenbarkeit

Ziel dieses Kapitels ist die Fundierung des Begriffs des Algorithmus. Dabei nehmen wir folgende intuitive Forderungen an einen Algorithmus als Grundlage. Ein Algorithmus

- überführt Eingabedaten in Ausgabedaten (wobei die Art der Daten vom Problem, das durch den Algorithmus gelöst werden soll, abhängig ist),
- besteht aus einer Folge von Anweisungen mit folgenden Eigenschaften:
  - es gibt eine eindeutig festgelegte Anweisung, die als erste auszuführen ist,
  - nach Abarbeitung einer Anweisung gibt es eine eindeutig festgelegte Anweisung, die als nächste abzuarbeiten ist, oder die Abarbeitung des Algorithmus ist beendet und liefert eindeutig bestimmte Ausgabedaten,
  - die Abarbeitung einer Anweisung erfordert keine Intelligenz (ist also prinzipiell durch eine Maschine realisierbar).

Mit diesem intuitiven Konzept lässt sich leicht feststellen, ob ein Verfahren ein Algorithmus ist. Betrachten wir als Beispiel die schriftliche Addition. Als Eingabe fungieren die beiden gegebenen zu addierenden Zahlen; das Ergebnis der Addition liefert die Ausgabe. Der Algorithmus besteht im Wesentlichen aus der sukzessiven Addition der entsprechenden Ziffern unter Beachtung des jeweils entstehenden Übertrags, wobei mit den „letzten“ Ziffern angefangen wird. Zur Ausführung der Addition von Ziffern ist keine Intelligenz notwendig (obwohl wir in der Praxis dabei das scheinbar Intelligenz erfordernde Kopfrechnen benutzen), da wir eine Tafel benutzen können, in der alle möglichen Additionen von Ziffern enthalten sind (und wir davon ausgehen, dass das Ablesen eines Resultats aus einer Tafel oder Liste ohne Intelligenz möglich ist). In ähnlicher Weise kann man leicht überprüfen, dass z.B.

- der Gaußsche Algorithmus zur Lösung von linearen Gleichungssystemen (über den rationalen Zahlen),
- Kochrezepte (mit Zutaten und Kochgeräten als Eingabe und dem fertigen Gericht als Ausgabe),

- Bedienungsanweisungen für Geräte,
- PASCAL-Programme

Algorithmen sind.
Jedoch ist andererseits klar, dass dieser Algorithmenbegriff nicht ausreicht, um zu klären, ob es für ein Problem einen Algorithmus zur Lösung gibt. Falls man einen Algorithmus zur Lösung hat, so sind nur obige Kriterien zu testen. Um aber zu zeigen, dass es keinen Algorithmus gibt, ist es erforderlich, eine Kenntnis aller möglichen Algorithmen zu haben; und dafür ist der obige intuitive Begriff zu unpräzise. Folglich wird es unsere erste Aufgabe sein, eine Präzisierung des Algorithmenbegriffs vorzunehmen, die es gestattet, in korrekter Weise Beweise führen zu können.
Intuitiv gibt es zwei mögliche Wege zur Formalisierung des Algorithmenbegriffs.

1. Wir betrachten einige Basisfunktionen, die wir als Algorithmen ansehen (d.h. wir gehen davon aus, dass die Transformation einer Eingabe in eine Ausgabe ohne Intelligenz in einem Schritt möglich ist). Ferner betrachten wir einige Operationen, mittels derer die Basisfunktionen verknüpft werden können, um weitere Funktionen zu erhalten, die dann ebenfalls als Algorithmen angesehen werden.

2. Wir definieren Maschinen, deren elementare Schritte als algorithmisch realisierbar gelten, und betrachten die Überführung der Eingabe in die Ausgabe durch die Maschine als Algorithmus.

1.1.1 LOOP/WHILE-Berechenbarkeit

In diesem Abschnitt wollen wir eine Präzisierung des Algorithmenbegriffs auf der Basis einer Konstruktion, die Programmiersprachen ähnelt, geben.
Als Grundsymbole verwenden wir

$$0,\ S,\ P,\ \textbf{LOOP},\ \textbf{WHILE},\ \textbf{BEGIN},\ \textbf{END},\ :=,\ \neq,\ ;,\ (,\ )$$

und eine unendliche Menge von Variablen (genauer Variablensymbolen)

$$x_1, x_2, \ldots, x_n, \ldots.$$

Definition 1.1 *i) Eine Wertzuweisung ist ein Ausdruck, der eine der folgenden vier Formen hat:*

$$\begin{aligned} x_i &:= 0 && \text{für } i \in \mathbf{N}, \\ x_i &:= x_j && \text{für } i \in \mathbf{N}, j \in \mathbf{N} \\ x_i &:= S(x_j) && \text{für } i \in \mathbf{N}, j \in \mathbf{N} \\ x_i &:= P(x_j) && \text{für } i \in \mathbf{N}, j \in \mathbf{N} \end{aligned}$$

Jede Wertzuweisung ist ein Programm.
ii) Sind Π, Π_1 *und* Π_2 *Programme und* x_i *eine Variable,* $i \in \mathbf{N}$*, so sind auch die folgenden Ausdrücke Programme:*

Π_1; Π_2 ,
LOOP x_i **BEGIN** Π **END** ,
WHILE $x_i \neq 0$ **BEGIN** Π **END** .

Wir geben nun einige Beispiele.

Beispiel 1.2

a) **LOOP** x_2 **BEGIN** $x_1 := S(x_1)$ **END** ,

b) $x_3 := 0$;
LOOP x_1 **BEGIN**
LOOP x_2 **BEGIN** $x_3 := S(x_3)$ **END**
END

c) **WHILE** $x_1 \neq 0$ **BEGIN** $x_1 := x_1$ **END** ,

d) $x_3 := 0$; $x_3 := S(x_3)$;
WHILE $x_2 \neq 0$ **BEGIN**
$x_1 := 0$; $x_1 := S(x_1)$; $x_2 := 0$; $x_3 := 0$
END ;
WHILE $x_3 \neq 0$ **BEGIN** $x_1 := 0$; $x_3 := 0$ **END**.

Durch Definition 1.1 ist nur festgelegt, welche Ausdrücke syntaktisch richtige Programme sind. Wir geben nun eine semantische Interpretation der einzelnen Bestandteile von Programmen.
Die Variablen werden mit natürlichen Zahlen aus $\mathbf{N}_0$ belegt.
Bei der Wertzuweisung $x_i := 0$ wird die Variable x_i mit dem Wert 0 belegt, und bei $x_i := x_j$ wird der Variablen x_i der Wert der Variablen x_j zugewiesen. S und P realisieren die Funktionen

$$\begin{aligned} S(x) &= x+1, \\ P(x) &= \begin{cases} x-1 & x \geq 1 \\ 0 & x = 0 \end{cases} . \end{aligned}$$

Π_1; Π_2 wird als Nacheinanderausführung der Programme Π_1 und Π_2 interpretiert.
LOOP x_i **BEGIN** Π **END** beschreibt die x_i-malige aufeinanderfolgende Ausführung des Programms Π, wobei eine Änderung von x_i während der x_i-maligen Abarbeitung unberücksichtigt bleibt.
Bei **WHILE** $x_i \neq 0$ **BEGIN** Π **END** wird das Programm Π solange ausgeführt, bis die Variable x_i den Wert 0 annimmt (hierbei wird also die Änderung von x_i durch die Ausführung von Π berücksichtigt).

Definition 1.3 *Es sei* Π *ein Programm mit n Variablen. Für* $1 \leq i \leq n$ *bezeichnen wir mit* $\Phi_{\Pi,i}(a_1, a_2, \ldots, a_n)$ *den Wert, den die Variable* x_i *nach Abarbeitung des Programms* Π *annimmt, wobei die Variable* x_j, $1 \leq j \leq n$, *als Anfangsbelegung den Wert* a_j *annimmt. Dadurch sind durch* Π *auch n Funktionen* $\Phi_{\Pi,i}(x_1, x_2, \ldots, x_n) : \mathbf{N}_0^n \to \mathbf{N}_0$, $1 \leq i \leq n$, *definiert.*

Beispiel 1.1 (Fortsetzung) Wir berechnen nun die Funktionen, die aus den Programmen in Beispiel 1.1 resultieren.
a) Wir bemerken zuerst, dass der Wert von x_2 bei der Abarbeitung des Programms unverändert bleibt. Der Wert der Variablen x_1 wird dagegen entsprechend der Semantik der **LOOP**-Anweisung so oft um 1 erhöht, wie der Wert der Variablen x_2 angibt. Dies liefert

$$\begin{aligned} \Phi_{\Pi,1}(x_1, x_2) &= x_1 + x_2, \\ \Phi_{\Pi,2}(x_1, x_2) &= x_2. \end{aligned}$$

b) Nach Teil a) liefert die innere **LOOP**-Anweisung die Addition vom Wert von x_2 zum Wert von x_3. Diese Addition hat nach der Definition der äußeren **LOOP**-Anweisung so oft zu erfolgen, wie der Wert von x_1 angibt. Unter Beachtung der Wertzuweisung zu Beginn des Programms ergibt sich

$$\begin{aligned} \Phi_{\Pi,1}(x_1, x_2, x_3) &= x_1, \\ \Phi_{\Pi,2}(x_1, x_2, x_3) &= x_2, \\ \Phi_{\Pi,3}(x_1, x_2, x_3) &= 0 + \underbrace{x_2 + x_2 + \ldots + x_2}_{x_1-mal} = x_1 \cdot x_2. \end{aligned}$$

c) Falls x_1 den Wert 0 hat, so wird die **WHILE**-Anweisung nicht durchlaufen; und folglich hat x_1 auch nach Abarbeitung des Programms den Wert 0. Ist dagegen der Wert von x_1 von 0 verschieden, so wird die **WHILE**-Anweisung immer wieder durchlaufen, da die darin enthaltene Wertzuweisung den Wert von x_1 nicht ändert; somit wird kein Ende der Programmabarbeitung erreicht und daher kein Wert von x_1 nach Abarbeitung des Programms definiert. Dies ergibt

$$\Phi_{\Pi,1}(x_1) = \begin{cases} 0 & x_1 = 0 \\ nicht\ definiert & \text{sonst} \end{cases}.$$

d) Dieses Programm realisiert die Funktionen

$$\begin{aligned} \Phi_{\Pi,1}(x_1, x_2, x_3) &= \begin{cases} 0 & x_2 = 0 \\ 1 & \text{sonst} \end{cases}, \\ \Phi_{\Pi,2}(x_1, x_2, x_3) &= 0, \\ \Phi_{\Pi,3}(x_1, x_2, x_3) &= 0. \end{aligned}$$

Wir bemerken hier, dass durch dieses Programm die folgende Anweisung

$$\textbf{IF } x_2 = 0 \textbf{ THEN } x_1 := 0 \textbf{ ELSE } x_1 := 1,$$

die der Funktion $\Phi_{\Pi,1}$ entspricht, beschrieben wird. Der Einfachheit halber haben wir hier eine sehr spezielle **IF-THEN-ELSE**-Konstruktion angegeben, obwohl jede derartige Anweisung realisiert werden kann (siehe Übungsaufgabe 2).

Definition 1.4 *Eine Funktion $f : \mathbf{N_0^n} \to \mathbf{N_0}$, $n \in \mathbf{N}_0$, heißt,* **LOOP/WHILE**-*berechenbar, wenn es ein Programm Π mit m Variablen, $m \geq n$, derart gibt, dass*

$$\Phi_{\Pi,1}(x_1, x_2, \ldots, x_n, 0, 0, \ldots, 0) = f(x_1, x_2, \ldots, x_n)$$

für alle $x_i \in N_0$, $1 \leq i \leq n$, gilt.

Wir sagen dann auch, dass Π die Funktion f berechnet.

Entsprechend dieser Definition kann das Programm Π mehr Variable als die Funktion f haben, aber die zusätzlichen Variablen $x_{n+1}, x_{n+2}, \ldots, x_m$ müssen bei Beginn der Programmabarbeitung mit dem Wert 0 belegt sein.
Die in Definition 1.4 gegebene Festlegung auf die erste Variable durch die Auswahl von $\Phi_{\Pi,1}$ ist nur scheinbar eine Einschränkung, da durch Hinzufügen der Wertzuweisung $x_1 := x_i$ als letzte Anweisung des Programms $\Phi_{\Pi,1} = \Phi_{\Pi,i}$ erreicht werden kann.

Aufgrund der Beispiele wissen wir bereits, dass die Addition und Multiplikation zweier Zahlen und die konstanten Funktionen **LOOP**/**WHILE**-berechenbar sind. Wir geben nun ein weiteres Beispiel.

Beispiel 1.5 Die nach dem italienischen Mathematiker Fibonacci, der sie im Zusammenhang mit der Vermehrung von Kaninchen als erster untersucht hat, benannte Folge hat die Anfangsglieder

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ \ldots$$

und das Bildungsgesetz

$$a_{i+2} = a_i + a_{i+1} \text{ für } i \geq 0.$$

Wir wollen nun zeigen, dass die Funktion $f : \mathbf{N} \to \mathbf{N}$ mit

$$f(i) = a_i$$

LOOP/**WHILE**-berechenbar ist. Wir haben also ein Programm Π zu konstruieren, dessen Funktion $\Phi_{\Pi,1}$ mit f übereinstimmt.
Entsprechend dem Bildungsgesetz der Fibonacci-Folge haben wir für $i \geq 2$ jeweils $i - 1$ Additionen durchzuführen. Dies lässt sich durch eine **WHILE**-Anweisung realisieren, die bei $i-1$ beginnen muss und bei jedem Durchlauf den Wert von i um 1 senkt. Weiterhin ist innerhalb dieser Anweisung eine Addition (wie in Beispiel 1.1 a) gezeigt) durchzuführen und die Summanden sind stets umzubenennen, damit beim nächsten Durchlauf die korrekten Summanden addiert werden (der zweite Summand der durchgeführten Addition ist der erste Summand der durchzuführenden, der zweite Summand der durchzuführenden Addition ist das Ergebnis der durchgeführten). Ferner sind die beiden Anfangswerte als $a_0 = a_1 = 1$ zu setzen. Wir werden die Summanden mit den Variablen x_2 und x_3 bezeichnen; diese fungieren auch als die beiden Anfangswerte, da dies die Summanden der ersten Addition sind. x_1 wird sowohl für i verwendet, als auch für das Ergebnis (aufgrund von Definition 1.4). Formal ergibt sich entsprechend diesen Überlegungen das folgende Programm:

$x_2 := 0$; $x_2 := S(x_2)$; $x_3 := x_2$; $x_1 := P(x_1)$;
WHILE $x_1 \neq 0$ **BEGIN**
LOOP x_3 **BEGIN** $x_2 := S(x_2)$ **END** ;
$x_4 := x_2$; $x_2 := x_3$; $x_3 := x_4$; $x_1 := P(x_1)$
END ;
$x_1 := x_3$

Wir geben nun einige Methoden an, mit denen aus bekannten **LOOP/WHILE**-berechenbaren Funktionen neue ebenfalls **LOOP/WHILE**-berechenbare Funktionen erzeugt werden können. Diese Methoden sind die Superposition oder Einsetzung von Funktionen, die Rekursion und die Bildung gewisser Minima, die alle von großer Bedeutung in der Informatik sind (Genaueres dazu wird im zweiten Teil der Vorlesung vermittelt).

Satz 1.6 *Es seien f eine m-stellige* **LOOP/WHILE**-*berechenbare Funktion und f_i eine n-stellige* **LOOP/WHILE**-*berechenbare Funktion für $1 \leq i \leq m$. Dann ist auch die n-stellige Funktionen g, die durch*

$$g(x_1, x_2, \ldots, x_n) = f(f_1(x_1, \ldots, x_n), f_2(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$$

definiert ist, eine **LOOP/WHILE**-*berechenbare Funktion.*

Beweis. Nach Voraussetzung gibt es Programme $\Pi, \Pi_1, \Pi_2, \ldots, \Pi_m$ derart, dass

$$\Phi_{\Pi,1} = f \text{ und } \Phi_{\Pi_i,1} = f_i \text{ für } 1 \leq i \leq m$$

gelten. Nun prüft man leicht nach, dass das Programm

$x_{n+1} := x_1; x_{n+2} := x_2; \ldots; x_{2n} := x_n;$
$\Pi_1; x_{2n+1} := x_1;$
$x_1 := x_{n+1}; x_2 := x_{n+2}; \ldots; x_n := x_{2n}; \Pi_2; x_{2n+2} := x_1;$
...
$x_1 := x_{n+1}; x_2 := x_{n+2}; \ldots; x_n := x_{2n}; \Pi_m; x_{2n+m} := x_1;$
$x_1 := x_{2n+1}; x_2 := x_{2n+2}; \ldots; x_m := x_{2n+m}; \Pi$

die Funktion g berechnet (die Setzungen $x_{n+i} := x_i$ stellen ein Abspeichern der Eingangswerte für die Variablen x_i dar; durch die Anweisungen $x_i := x_{n+i}$ wird jeweils gesichert, dass die Programme Π_j mit der Eingangsbelegung der x_i arbeiten, denn bei der Abarbeitung von Π_{j-1} kann die Belegung der x_i geändert worden sein; die Setzungen $x_{2n+j} := x_1$ speichern die Werte $f_j(x_1, x_2, \ldots, x_n)$, die durch die Programme Π_j bei der Variablen x_1 entsprechend der berechneten Funktion erhalten werden; mit diesen Werten wird dann aufgrund der Anweisungen $x_j := x_{2n+j}$ das Programm Π gestartet und damit der gewünschte Wert berechnet). □

Satz 1.7 *Es seien f und h eine $(n-1)$-stellige bzw. $(n+1)$-stellige* **LOOP/WHILE**-*berechenbare Funktionen. Dann ist auch die n-stellige Funktionen g, die durch*

$$\begin{aligned} g(x_1, x_2, \ldots, x_{n-1}, 0) &= f(x_1, x_2, \ldots, x_{n-1}), \\ g(x_1, x_2, \ldots, x_{n-1}, S(x_n)) &= h(x_1, x_2, \ldots, x_{n-1}, x_n, g(x_1, x_2, \ldots, x_{n-1}, x_n)) \end{aligned}$$

definiert ist, eine **LOOP/WHILE**-*berechenbare Funktion.*

Beweis. Nach Voraussetzung gibt es Programme Π über den Variablen $x_1, x_2, \ldots x_{n-1}$ und Π' über den Variablen $x_1, x_2, \ldots x_{n-1}, y, z$ mit $\Phi_{\Pi,1} = f$ und $\Phi_{\Pi',1} = h$ (wobei wir zur Vereinfachung nicht nur Variable der Form x_i, wie in Abschnitt 1.1.1 gefordert, verwenden). Wir betrachten das folgende Programm:

$y := 0; x_n := x_1; x_{n+1} := x_2; \ldots x_{2n-2} := x_{n-1}; \Pi; z := x_1;$
LOOP y' **BEGIN** $x_1 := x_n; \ldots x_{n-1} := x_{2n-2}; \Pi'; z := x_1; y := S(y)$ **END**;
$x_1 := z$

und zeigen, dass dadurch der Wert $g(x_1, x_2, \ldots, x_n, y')$ berechnet wird.
Erneut wird durch die Variablen x_{n+i-1}, $1 \leq i \leq n-1$ die Speicherung der Anfangsbelegung der Variablen x_i gewährleistet.
Ist $y' = 0$, so werden nur die erste und dritte Zeile des Programms realisiert. Daher ergibt sich der Wert von Π bei der ersten Variablen, und weil Π die Funktion f berechnet, erhalten wir $f(x_1, x_2, \ldots, x_{n-1})$, wie bei der Definition von g gefordert wird.
Ist dagegen $y' > 0$, so wird innerhalb der **LOOP**-Anweisung mit $z = g(x_1, x_2, \ldots, x_{n-1}, y)$ der Wert $g(x_1, x_2, \ldots x_{n-1}, y+1)$ berechnet und die Variable y um Eins erhöht. Da dies insgesamt von $y = 0$ und $g(x_1, x_2, \ldots, x_{n-1}, 0) = f(x_1, x_2, \ldots, x_{n-1})$ (aus der ersten Zeile) ausgehend, y'-mal zu erfolgen hat, wird tatsächlich $f(x_1, x_2, \ldots, x_n, y')$ als Ergebnis geliefert. □

Satz 1.8 *Es sei h eine totale $(n+1)$-stellige* **LOOP/WHILE***-berechenbare Funktion. Dann ist auch die n-stellige Funktionen g, die durch*

$$g(x_1, x_2, \ldots, x_n) = \begin{cases} \min\{m \mid h(x_1, x_2, \ldots, x_n, m) = 0\} & \textit{falls } 0 \in rg(h) \\ \textit{nicht definiert} & \textit{sonst} \end{cases}$$

definiert ist, eine **LOOP/WHILE***-berechenbare Funktion.*

Beweis. Es sei Π ein Programm, das f berechnet. Um den minimalen Wert m zu berechnen, berechnen wir der Reihe nach die Werte

$$h(x_1, x_2, \ldots, x_n, 0),\ h(x_1, x_2, \ldots, x_n, 1),\ h(x_1, x_2, \ldots, x_n, 2),\ \ldots$$

und testen jeweils, ob das aktuelle Ergebnis von Null verschieden ist. Formal ergibt sich folgendes Programm für g:

$y := 0; x_{n+1} := x_1; \ldots x_{2n} := x_n; y' := S(y);$
WHILE $y' \neq 0$ **BEGIN** $x_1 := x_{n+1}, \ldots, x_n := x_{2n}; \Pi; y' := x_1; y := S(y)$ **END**;
$x_1 := P(y)$

(die Setzung $y' := S(y)$ sichert, dass die Schleife mindestens einmal durchlaufen wird, d.h., dass mindestens $h(x_1, x_2, \ldots, x_n, 0)$ berechnet wird). □

Wir definieren nun die *Tiefe* eines **LOOP/WHILE**-Programms, die sich im Folgenden als nützliches Hilfsmittel erweisen wird.

Definition 1.9 *Die Tiefe $t(\Pi)$ eines Programms Π wird induktiv wie folgt definiert:*
i) Für eine Wertzuweisung Π gilt $t(\Pi) = 1$,
ii) $t(\Pi_1; \Pi_2) = t(\Pi_1) + t(\Pi_2)$,
*iii) $t($***LOOP** x_i **BEGIN** Π **END**$) = t(\Pi) + 1$,*
*iv) $t($***WHILE** $x_i \neq 0$ **BEGIN** Π **END**$) = t(\Pi) + 1$.*

Beispiel 1.1 (Fortsetzung) Für das unter a) betrachtete Programm ergibt sich die Tiefe 2, da aufgrund der Definition die Tiefe um 1 größer ist als die des Programms innerhalb der **LOOP**-Anweisung, das als Wertzuweisung die Tiefe 1 hat.
Aus gleicher Überlegung resultiert auch die Tiefe 2 für das Programm aus c). Dagegen haben b) bzw. d) die Tiefe 4 bzw. 10.

Wir bemerken, dass alle Programme der Tiefe 1 eine Wertzuweisung sind. Weiterhin haben alle Programme der Tiefe 2 eine der folgenden Formen:

$x_i := A; x_r := B$,
LOOP x_k **BEGIN** $x_i := A$ **END**,
WHILE $x_k \neq 0$ **BEGIN** $x_i := A$ **END**

mit

$$A \in \{0, x_j, S(x_j), P(x_j),\ B \in \{0, x_s, S(x_s), P(x_s) \text{ und } i, j, k, r, s \in \mathbf{N}\,,$$

denn es müssen zwei Programme der Tiefe 1 nacheinander ausgeführt werden oder es muss eine der beiden Schleifen auf ein Programm der Tiefe 1 angewendet werden. Analog haben Programme der Tiefe 3 eine der folgenden Formen:

$x_i := A'; x_r := B'; x_u := C'$,
$x_i := A'$; **LOOP** x_k **BEGIN** $x_r := B'$ **END**,
$x_i; = A'$; **WHILE** $x_k \neq 0$ **BEGIN** $x_r = B'$ **END**,
LOOP x_k **BEGIN** $x_r := B'$ **END**; $x_i := A'$,
WHILE $x_k \neq 0$ **BEGIN** $x_r = B'$ **END**; $x_i := A'$,
LOOP x_k **BEGIN** $x_i := A'; x_r := B'$ **END**,
WHILE $x_k \neq 0$ **BEGIN** $x_i = A'; x_r := B'$ **END**,
LOOP x_k **BEGIN LOOP** x_i **BEGIN** $x_r := B'$ **END END**,
WHILE $x_k \neq 0$ **BEGIN LOOP** x_i **BEGIN** $x_r := B'$ **END END**,
LOOP x_k **BEGIN WHILE** $x_i \neq 0$ **BEGIN** $x_r := B'$ **END END**,
WHILE $x_k \neq 0$ **BEGIN WHILE** $x_i \neq 0$ **BEGIN** $x_r := B'$ **END END**

mit

$$A' \in \{0, x_j, S(x_j), P(x_j),\ B' \in \{0, x_s, S(x_s), P(x_s),\ C' \in \{0, x_v, S(x_v), P(x_v),$$
$$i, j, k, r, s, u, v \in \mathbf{N}\,.$$

Wir kommen nun zu einem der Hauptresultate dieses Abschnitts. Es besagt, dass nicht jede Funktion, die einem Tupel natürlicher Zahlen wieder eine natürliche Zahl zuordnet, durch ein **LOOP**/**WHILE**-Programm berechnet werden kann. Somit zeigt der Satz Grenzen des bisher gegebenen Berechenbarkeitsbegriffs.

Satz 1.10 *Es gibt (mindestens) eine totale Funktion, die nicht* **LOOP**/ **WHILE**-*berechenbar ist.*

Beweis. Wir geben zwei Beweise für diese Aussage.

a) Wir erinnern zuerst an folgende aus der Mathematik bekannten Fakten:

- Die Vereinigung abzählbar vieler abzählbarer Mengen ist wieder abzählbar.
- Sind die Mengen M und N abzählbar, so ist auch $M \times N$ abzählbar.

Wir zeigen nun, dass die Menge Q aller **LOOP**/**WHILE**-Programme abzählbar ist.
Für $k \in \mathbf{N}$ sei Q_k die Menge aller **LOOP**/**WHILE**-Programme der Tiefe $k \geq 1$. Dann gilt offenbar

$$Q = \bigcup_{k \geq 1} Q_k \,.$$

Wegen des oben genannten ersten Faktes reicht es also zu beweisen, dass Q_k für $k \in \mathbf{N}$ abzählbar ist. Dies beweisen mittels Induktion über die Tiefe k.
Ist $k = 1$, so besteht das Programm nur aus einer Wertzuweisung. Da es eine eineindeutige Abbildungen gibt, die jeder Zahl $i \in \mathbf{N}$ die Anweisung $x_i := 0$ zuordnet bzw. jedem Paar (i, j) eine Anweisung $x_i := x_j$ bzw. $x_i := S(x_j)$ bzw. $x_i := P(x_j)$ zuordnet, ist nach obigen Fakten Q_1 als Vereinigung von vier abzählbaren Mengen wieder abzählbar.
Hat ein Programm $\Pi_1; \Pi_2$ die Tiefe $k + 1$, so haben Π_1 und Π_2 eine Tiefe $\leq k$. Damit kann dieses Programm eineindeutig auf ein Tupel $(\Pi_1, \Pi_2) \in Q_i \times Q_j$ mit $i \in \mathbf{N}$, $j \in \mathbf{N}$ und $i + j = k + 1$ abgebildet werden. Daher ergibt sich, dass die Menge aller Programme dieser Form gleichmächtig zu

$$\bigcup_{i \in \mathbf{N}, j \in \mathbf{N}, i+j=k+1} Q_i \times Q_j$$

ist, die nach obigen Fakten abzählbar ist.
Hat **LOOP** x_i **BEGIN** Π **END** die Tiefe $k + 1$, so hat Π die Tiefe k und kann folglich auf das Tupel (i, Π) mit $\Pi \in Q_k$ abgebildet werden. Damit ist die Menge aller **LOOP**-Anweisungen der Tiefe $k+1$ gleichmächtig zu $\mathbf{N} \times Q_k$. Analoges gilt auch für die **WHILE**-Anweisung.
Folglich ist Q_{k+1} als Vereinigung dreier abzählbarer Mengen selbst abzählbar.
Da zwei **LOOP**/**WHILE**-Programme die gleiche Funktion berechnen können, gibt es höchstens soviele **LOOP**/**WHILE**-berechenbare Funktionen wie **LOOP**/**WHILE**-Programme. Somit gibt es nur abzählbar viele **LOOP**/**WHILE**-berechenbare Funktionen.
Andererseits zeigen wir nun, dass es bereits überabzählbar viele einstellige Funktion von $\mathbf{N}_0$ in $\mathbf{N}_0$ gibt. Sei nämlich die Menge E dieser Funktionen abzählbar, so gibt es eine eineindeutige Funktion von $\mathbf{N}_0$ auf E. Für $i \in \mathbf{N}$ sei f_i das Bild von i. Dann können wir die Elemente von E als unendliche Matrix schreiben, wobei die Zeilen den Funktionen und die Spalten den Argumenten entsprechen (siehe Abbildung 1.1). Wir definieren nun die

$$\begin{array}{cccccc}
f_0(0) & f_0(1) & f_0(2) & \ldots & f_0(r) & \ldots \\
f_1(0) & f_1(1) & f_1(2) & \ldots & f_1(r) & \ldots \\
f_2(0) & f_2(1) & f_2(2) & \ldots & f_2(r) & \ldots \\
 & \ldots & & \ldots & & \ldots \\
f_r(0) & f_r(1) & f_r(2) & \ldots & f_r(r) & \ldots \\
 & \ldots & & \ldots & & \ldots
\end{array}$$

Abbildung 1.1: Matrixdarstellung von der Menge E der einstelligen Funktionen

Funktion $f \in E$ mittels der Setzung $f(r) = f_r(r) + 1$ für $r \in \mathbf{N}_0$. Offenbar ist f nicht eine der Funktionen der Matrix, da für jedes $t \in \mathbf{N}_0$ die Beziehung $f(t) = f_t(t) + 1 \neq f_t(t)$ gilt. Dies liefert einen Widerspruch, da die Matrix alle Funktionen nach Konstruktion enthält. Folglich kann E nicht abzählbar sein.

Wir bemerken, dass dieser Beweis nicht konstruktiv ist und keinen Hinweis auf eine nicht **LOOP/WHILE**-berechenbare Funktion liefert.

b) Der zweite Beweis besteht in der Angabe einer Funktion, die nicht **LOOP/WHILE**-berechenbar ist. (Allerdings scheint die Funktion keinerlei praktische Relevanz zu haben. Deshalb geben wir im Abschnitt 1.2. weitere Beispiele nicht **LOOP/WHILE**-berechenbarer Funktionen, die von Bedeutung in der Informatik sind.)

Wir betrachten dazu die Funktion f, bei der $f(n)$ die größte Zahl ist, die mit einem **LOOP/WHILE**-Programm der Tiefe $\leq n$ auf der Anfangsbelegung $x_1 = x_2 = \ldots = 0$ berechnet werden kann.

Aus der obigen Bestimmung der **LOOP/WHILE**-Programme der Tiefen 1,2 und 3 sieht man sofort, dass sich der maximale Wert immer dann ergibt, wenn nur die Variable x_1 vorkommt und jede Anweisung die Inkrementierung $x_i := S(x_i)$ ist. Damit gelten $f(1) = 1$, $f(2) = 2$ und $f(3) = 3$. Um zu zeigen, dass f nicht die Identität ist, betrachten wir das Programm

$x_1 := S(x_1);\ x_1 := S(x_1);\ x_1 := S(x_1);\ x_1 := S(x_1);$
$x_2 := S(x_2);\ x_2 := S(x_2);\ x_2 := S(x_2);$
LOOP x_1 **BEGIN**
LOOP x_2 **BEGIN** $x_3 := S(x_3)$ **END**
END;
$x_1 := x_3$

das die Tiefe 11 hat und auf der Anfangsbelegung 0 für alle Variablen das Produkt $4 \cdot 3 = 12$ berechnet. Folglich gilt $f(11) \geq 12 > 11$.

Aus der Definition von f folgt sofort, dass f auf allen natürlichen Zahlen definiert ist.

Wir beweisen zuerst, dass f eine streng monotone Funktion ist, d.h., dass $f(n) < f(m)$ für $n < m$ gilt. Offenbar reicht es, $f(n) < f(n+1)$ für alle natürlichen Zahlen zu zeigen. Sei dazu Π ein Programm der Tiefe n mit

$$\Phi_{\Pi,1}(0, 0, \ldots, 0) = k = f(n),$$

d.h. k ist der maximale durch Programme der Tiefe n berechenbare Wert. Dann gelten für das Programm Π', das durch Hintereinanderausführung von Π und $S(x_1)$ entsteht,

$$t(\Pi') = n + 1 \quad \text{und} \quad \Phi_{\Pi',1}(0, 0, \ldots, 0) = k + 1 \leq f(n+1).$$

Entsprechend der Definition von $f(n+1)$ als maximalen Wert, der durch Programme der Tiefe $n+1$ berechnet werden kann, erhalten wir die gewünschte Relation

$$f(n+1) \geq k + 1 > k = f(n).$$

Wir zeigen nun indirekt, dass f nicht **LOOP/WHILE**-berechenbar ist. Dazu nehmen wir an, dass f durch das Programm Π_0 berechnet wird und betrachten die Funktion g, die durch

$$g(n) = f(2n)$$

definiert ist. Offenbar ist auch g auf allen natürlichen Zahlen definiert. Ferner ist g auch **LOOP/WHILE**-berechenbar, denn entsprechend den Beispielen gibt es ein Programm Π_1, dass die Funktion $u(n) = 2n$ berechnet, und somit berechnet das Programm

$$\Pi_2 = \Pi_1; \Pi_0$$

die Funktion g. Es sei

$$k = t(\Pi_2).$$

Weiterhin sei h eine beliebige Zahl. Dann betrachten wir das Programm

$$\Pi_3 = \underbrace{x_1 := S(x_1); x_1 := S(x_1); \ldots; x_1 := S(x_1);}_{h \ mal} \Pi_2.$$

Dann gelten

$$t(\Pi_3) = k + h \quad \text{und} \quad \Phi_{\Pi_3,1}(0, 0, \ldots, 0) = g(h)\,.$$

Wegen der Forderung nach dem Maximalwert in der Definition von f folgt $f(h+k) \geq g(h)$. Wir wählen nun h so, dass $k < h$ und damit auch $h+k < 2h$ gilt. Aufgrund der Definition von g und der strengen Monotonie von f erhalten wir dann

$$f(h + k) \geq g(h) = f(2h) > f(h + k),$$

wodurch offensichtlich ein Widerspruch gegeben ist. □

Für spätere Anwendungen benötigen wir die folgende Modifikation von Satz 1.10.

Folgerung 1.11 *Es gibt eine Funktion f mit folgenden Eigenschaften:*
- f ist total,
- der Wertebereich von f ist $\{0, 1\}$,
- f ist nicht **LOOP/WHILE***-berechenbar.*

Beweis. Nach Satz 1.10 gibt es eine totale Funktion $f : \mathbf{N}_0^n \to \mathbf{N}_0$, die nicht **LOOP/ WHILE**-berechenbar ist. Wir konstruieren nun die Funktion $g : \mathbf{N}_0^{n+1} \to \mathbf{N}_0$ mit

$$g(x_1, x_2, \ldots, x_n, x_{n+1}) = \begin{cases} 0 & f(x_1, x_2, \ldots, x_n) = x_{n+1} \\ 1 & \text{sonst} \end{cases}.$$

Offenbar genügt g den ersten beiden Forderungen aus Folgerung 1.11. Wir zeigen nun indirekt, dass auch die dritte Forderung erfüllt ist.
Dazu nehmen wir an, dass es ein Programm Π mit $\Phi_{\Pi,1} = g$ gibt, und konstruieren das Programm Π':

$x_{n+1} := 0; x_{n+2} := x_1; x_{n+3} := x_2; \ldots; x_{2n+1} := x_n; x_1 := 0; x_1 := S(x_1);$
WHILE $x_1 \neq 0$ **BEGIN**
$x_1 := x_{n+2}; x_2 := x_{n+3}; \ldots; x_n := x_{2n+1}; \Pi; x_{n+1} = S(x_{n+1})$
END;
$x_1 := P(x_{n+1})$.

Dieses Programm berechnet die Funktion f, was aus folgenden Überlegungen folgt: Die Variablen $x_{n+2}, x_{n+3}, \ldots, x_{2n+1}$ dienen der Speicherung der Werte, mit denen die Variablen $x_1, x_2, \ldots, x_n$ zu Beginn belegt sind. Durch die anschließende Setzung $x_1 := 0; x_1 := S(x_1)$ wird $x_1 \neq 0$ gesichert, womit die **WHILE**-Anweisung mindestens einmal durchlaufen wird. Aufgrund der Wertzuweisung $x_{n+1} := S(x_{n+1})$ und durch die stets erfolgende Setzung der Variablen $x_1, x_2, \ldots, x_n$ auf die Werte der Anfangsbelegung werden mittels der **WHILE**-Anweisung der Reihe nach die Werte

$$g(x_1, x_2, \ldots, x_n, 0), g(x_1, x_2, \ldots, x_n, 1), g(x_1, x_2, \ldots, x_n, 2), \ldots$$

berechnet, bis i mit

$$g(x_1, x_2, \ldots, x_n, i) = 0$$

erreicht wird. Dann wird durch die letzte Wertzuweisung des Programms x_1 mit i belegt. Andererseits gilt nach Definition von g auch

$$f(x_1, x_2, \ldots, x_n) = i.$$

Damit haben wir ein Programm Π' mit $\Phi_{\Pi',1} = f$ erhalten. Dies ist aber unmöglich, da f so gewählt war, dass f nicht durch **LOOP/WHILE**-Programme berechnet werden kann. Dieser Widerspruch besagt, dass unsere Annahme, dass g **LOOP/WHILE**-berechenbar ist, falsch ist. □

Aus Beispiel 1.1 c) ist bekannt, dass **LOOP/WHILE**-berechenbare Funktionen nicht immer auf der Menge aller natürlichen Zahlen definiert sein müssen. Wir wollen nun eine Einschränkung der zugelassenen Programme so vornehmen, dass die davon erzeugten Funktionen total sind. Hierfür gestatten wir die Verwendung von Wertzuweisungen, Hintereinanderausführung von Programmen und die **LOOP**-Anweisung. Formal wird dies durch die folgende Definition und Satz 1.13 gegeben.

Definition 1.12 *Eine Funktion* $f : \mathbf{N_0^n} \to \mathbf{N_0}$ *heißt* **LOOP***-berechenbar, wenn es ein Programm* Π *mit* m *Variablen,* $m \geq n$*, derart gibt, dass in* Π *keine* **WHILE***-Anweisung vorkommt und* Π *die Funktion* f *berechnet.*

Satz 1.13 *Der Definitionsbereich jeder* n*-stelligen* **LOOP***-berechenbaren Funktion ist die Menge* $\mathbf{N}^n$*, d.h. jede* **LOOP***-berechenbare Funktion ist total.*

Beweis. Wir beweisen den Satz mittels vollständiger Induktion über die Tiefe der Programme. Für Programme der Tiefe 1 ist die Aussage sofort klar, da derartige Programme aus genau einer Wertzuweisung bestehen, und nach Definition sind die von Wertzuweisungen berechneten Funktionen total.
Sei nun Π ein Programm der Tiefe $t > 1$. Dann tritt einer der folgenden Fälle ein:
Fall 1. $\Pi = \Pi_1; \Pi_2$ mit $t(\Pi_1) < t$ und $t(\Pi_2) < t$.
Nach Induktionsvoraussetzung sind daher die von Π_1 und Π_2 berechneten Funktionen total, und folglich ist die von Π als Hintereinanderausführung von Π_1 und Π_2 berechnete Funktion ebenfalls total.
Fall 2. $\Pi =$ **LOOP** x_i **BEGIN** Π' **END** mit $t(\Pi') = t - 1$. Nach Definition ist das Programm Π' sooft hintereinander auszuführen, wie der Wert von der Variablen angibt. Da die von Π' berechnete Funktion nach Induktionsvoraussetzung total definiert ist, gilt dies auch für die von Π berechnete Funktion. □

Unter Beachtung von Beispiel 1.1 c) ergibt sich sofort die folgende Folgerung.

Folgerung 1.14 *Die Menge der* **LOOP***-berechenbaren Funktionen ist echt in der Menge der* **LOOP/WHILE***-berechenbaren Funktionen enthalten.*

Die bisherigen Ausführungen belegen, dass die **WHILE**-Schleife nicht mittels **LOOP**-Schleifen simuliert werden kann. Umgekehrt berechnet das Programm

$x_{n+1} := x_i$;
WHILE $x_{n+1} \neq 0$ **BEGIN** $\Pi; x_{n+1} := P(x_{n+1})$ **END**

die gleiche Funktion wie

LOOP x_i **BEGIN** Π **END**

(wobei n die Anzahl der in Π vorkommenden Variablen ist).

1.1.2 TURING-Maschinen

Die **LOOP**/**WHILE**-Berechenbarkeit basiert auf einer Programmiersprache, die üblicherweise durch einen Rechner bzw. eine Maschine abgearbeitet wird. Wir wollen nun eine Formalisierung des Berechenbarkeitsbegriffs auf der Basis einer Maschine selbst geben. Dabei streben wir eine möglichst einfache Maschine an. Sie soll im Wesentlichen nur die Inhalte von Speicherzellen in Abhängigkeit von den ihr zur Verfügung stehenden Informationen ändern. In den Zellen werden nur Buchstaben eines Alphabets gespeichert. Die Operation besteht dann im Ersetzen eines Buchstaben durch einen anderen. Ferner kann nach Änderung des Inhalts einer Zelle nur zu den beiden benachbarten Zellen gegangen werden. Ein solches Vorgehen wurde erstmals vom englischen Mathematiker ALAN TURING (1912-1954) im Jahre 1935 untersucht. Wir geben nun die formale Definition einer TURING-Maschine.

Definition 1.15 *Eine* TURING-*Maschine ist ein Quintupel*

$$M = (X, Z, z_0, Q, \delta),$$

wobei
- X *und* Z *Alphabete sind,*
- $z_0 \in Z$ *und* $\emptyset \subseteq Q \subseteq Z$ *gelten,*
- δ *eine Funktion von* $(Z \setminus Q) \times (X \cup \{*\})$ *in* $Z \times (X \cup \{*\}) \times \{R, L, N\}$ *ist, und* $* \notin X$ *gilt.*

Um den Begriff „Maschine“ zu rechtfertigen, geben wir folgende Interpretation. Eine TURING-Maschine besteht aus einem beidseitig unendlichen, in Zellen unterteilten Band und einem „Rechenwerk“ mit einem Lese-/Schreibkopf. In jeder Zelle des Bandes steht entweder ein Element aus X oder das Symbol $*$; insgesamt stehen auf dem Band höchstens endlich viele Elemente aus X. Der Lese-/Schreibkopf ist in der Lage, das auf dem Band in einer Zelle stehende Element zu erkennen (zu lesen) und in eine Zelle ein neues Element einzutragen (zu schreiben). Das „Rechenwerk“ kann intern Informationen in Form von Elementen der Menge Z, den Zuständen, speichern. z_0 bezeichnet den Anfangszustand, in dem sich die Maschine zu Beginn ihrer Arbeit befindet. Q ist die Menge der Zustände, in denen die Maschine ihre Arbeit stoppt.
Ein Arbeitsschritt der Maschine besteht nun in Folgendem: Die Maschine befindet sich in einem Zustand z, ihr Kopf befindet sich über einer Zelle i und liest deren Inhalt x; hiervon ausgehend berechnet die Maschine einen neuen Zustand z', schreibt in die Zelle i

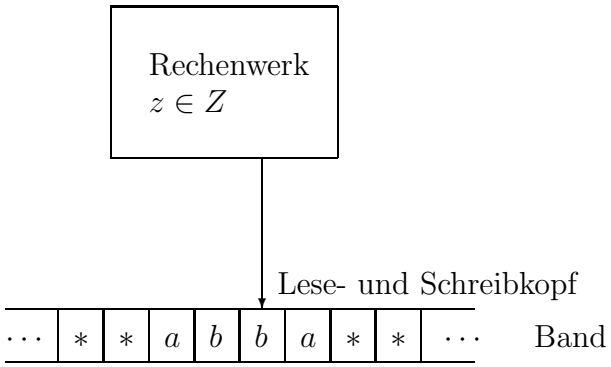


Abbildung 1.2: TURING-Maschine

ein aus z und x berechnetes Element x' und bewegt den Kopf um eine Zelle nach rechts (R) oder nach links (L) oder bewegt den Kopf nicht (N). Dies wird durch

$$\delta(z,x) = (z', x', r) \text{ mit } z \in Z \setminus Q, z' \in Z, x, x' \in X \cup \{*\}, r \in \{R, L, N\}$$

beschrieben.
Die aktuelle Situation, in der sich eine TURING-Maschine befindet, wird also durch das Wort (die Wörter) über X auf dem Band, den internen Zustand und die Stelle an der der Kopf steht, beschrieben. Formalisiert wird dies durch folgende Definition erfasst.

Definition 1.16 *Es sei M eine* TURING*-Maschine wie in Definition 1.15. Eine Konfiguration K der* TURING*-Maschine M ist ein Tripel*

$$K = (w_1, z, w_2),$$

wobei w_1 und w_2 Wörter über $X \cup \{*\}$ sind und $z \in Z$ gilt.
Eine Anfangskonfiguration liegt vor, falls $w_1 = \lambda$ und $z = z_0$ gelten.
Eine Endkonfiguration ist durch $z \in Q$ gegeben.

Wir interpretieren dies wie folgt: Auf dem Band steht das Wort w_1w_2; alle Zellen vor und hinter denjenigen, in denen w_1w_2 steht, sind mit $*$ belegt; der Kopf steht über der Zelle, in der der erste Buchstabe von w_2 steht; und die Maschine befindet sich im Zustand z.
Wir bemerken, dass eine Situation durch mehrere Konfigurationen beschrieben werden kann, z.B. beschreiben (λ, z, ab), $(*, z, ab)$ und $(**, z, ab*)$ alle die Situation, dass auf dem Band ab steht und der Kopf über a positioniert ist. Bei den nachfolgenden Definitionen und Beispielen wird jeweils unter den verschiedenen Konfigurationen, die die gleiche Situation des Bandes beschreiben, eine geeignete Konfiguration ausgewählt. [1]
Die folgende Definition formalisiert nun die Konfigurationsänderung, wenn die Maschine einen Schritt entsprechend δ ausführt.

[1]Formal können wir eine Äquivalenzrelation $\equiv$ dadurch definieren, dass wir $K_1 \equiv K_2$ genau dann setzen, wenn K_1 und K_2 die gleiche Situation beschreiben, und wir wählen stets einen geeigneten Repräsentanten der Äquivalenzklasse zur Beschreibung einer Situation.

Definition 1.17 *Es sei M eine* TURING*-Maschine wie in Definition 1.15, und $K_1 = (w_1, z, w_2)$ und $K_2 = (v_1, z', v_2)$ seien Konfigurationen von M. Wir sagen, dass K_1 durch M in K_2 überführt wird (und schreiben dafür $K_1 \models K_2$), wenn eine der folgenden Bedingungen erfüllt ist:*

$$v_1 = w_1, w_2 = xu, v_2 = x'u, \delta(z, x) = (z', x', N)$$

oder

$$w_1 = v, v_1 = vx', w_2 = xu, v_2 = u, \delta(z, x) = (z', x', R)$$

oder

$$w_1 = vy, v_1 = v, w_2 = xu, v_2 = yx'u, \delta(z, x) = (z', x', L)$$

für gewisse $x, x', y \in X \cup \{*\}$ und $u, v \in (X \cup \{*\})^*$.

Offenbar kann eine Endkonfiguration in keine weitere Konfiguration überführt werden, da die Funktion δ für Zustände aus Q und beliebige $x \in X \cup \{*\}$ nicht definiert ist.

Definition 1.18 *Es sei M eine* TURING*-Maschine wie in Definition 1.15. Die durch M induzierte Funktion f_M aus X^* in X^* ist wie folgt definiert: $f_M(w) = v$ gilt genau dann, wenn es für die Anfangskonfiguration $K = (\lambda, z_0, w)$ eine Endkonfiguration $K' = (v_1, q, v_2)$, natürliche Zahlen r, s und t und Konfigurationen $K_0, K_1, \ldots, K_t$ derart gibt, dass $*^r v *^s = v_1 v_2$ und*

$$K = K_0 \models K_1 \models K_2 \models \ldots \models K_t = K'$$

gelten.

Interpretiert bedeutet dies, dass sich durch mehrfache Anwendung von Überführungsschritten aus der Anfangskonfiguration, bei der w auf dem Band steht, eine Endkonfiguration ergibt, in der v auf dem Band steht. Falls in der Endkonfiguration (v_1, q, v_2) der Kopf über einer Zelle von v steht, so gelten $v = v_1 v_2$ und $r = s = 0$; steht der Kopf dagegen r Zellen vor v bzw. s Zellen hinter v, so gelten $*^r v = v_1 v_2$, $v_1 = \lambda$ und $s = 0$ bzw. $v *^s = v_1 v_2$, $v_2 = \lambda$ und $r = 0$.
Wir bemerken ferner, dass für solche Wörter w, bei denen die Maschine nie ein Stopzustand aus Q erreicht, kein zugeordneter Funktionswert $f_M(w)$ definiert ist. Somit kann f_M auch eine partielle Funktion sein.

Beispiel 1.19 Um eine TURING-Maschine zu beschreiben, werden wir nachfolgend die Funktion δ immer durch eine Tabelle angeben, bei der im Schnittpunkt der zu $x \in X$ bzw. $*$ gehörenden Zeile und der zu $z \in Z \setminus Q$ gehörenden Spalte das Tripel $\delta(z, x)$ steht.
a) Es sei

$$M_1 = (\{a, b\}, \{z_0, q, z_a, z_b\}, z_0, \{q\}, \delta)$$

eine TURING-Maschine, und es sei δ durch die Tabelle

δ	z_0	z_a	z_b
$*$	$(q, *, N)$	(q, a, N)	(q, b, N)
a	$(z_a, *, R)$	(z_a, a, R)	(z_b, a, R)
b	$(z_b, *, R)$	(z_a, b, R)	(z_b, b, R)

gegeben.
Wir starten mit dem Wort *abba* auf dem Band. Dann ergeben sich die folgenden Konfigurationen mittels Überführungen (um Übereinstimmung mit Definition 1.17 zu erreichen, haben wir die Konfiguration immer in die Form umgewandelt, die benötigt wird):

$$\begin{aligned}(\lambda, z_0, abba) &\models (*, z_a, bba) \models (*b, z_a, ba) = (b, z_a, ba) \models (bb, z_a, a) = (bb, z_a, a*) \\ &\models (bba, z_a, *) \models (bba, q, a).\end{aligned}$$

Folglich gilt

$$f_{M_1}(abba) = bbaa.$$

Ausgehend von *bab* erhalten wir

$$(\lambda, z_0, bab) \models (*, z_b, ab) \models (a, z_b, b) \models (ab, z_b, *) \models (ab, q, b)$$

und damit

$$f_{M_1}(bab) = abb.$$

Allgemein ergibt sich

$$f_{M_1}(x_1x_2 \dots x_n) = x_2x_3 \dots x_nx_1$$

(den zu Beginn gestrichenen Buchstaben x_1 merkt sich die Maschine in Form des Zustandes z_{x_1} und schreibt ihn an das Ende des Wortes).
b) Es sei

$$M_2 = (\{a, b\}, \{z_0, z_1, q\}, z_0, \{q\}, \delta),$$

wobei δ durch

δ	z_0	z_1
$*$	$(z_0, *, N)$	$(q, *, N)$
a	(z_1, a, R)	(z_0, a, R)
b	(z_1, b, R)	(z_0, b, R)

gegeben sei.
Für *abb* und *abba* ergeben sich

$$(\lambda, z_0, abb) \models (a, z_1, bb) \models (ab, z_0, b) \models (abb, z_1, *) \models (abb, q, *)$$

und

$$\begin{aligned}(\lambda, z_0, abba) &\models (a, z_1, bba) \models (ab, z_0, ba) \models (abb, z_1, b) \\ &\models (abba, z_0, *) \models (abba, z_0, *) \models (abba, z_0, *) \models \dots .\end{aligned}$$

Folglich gilt

$$f_{M_2}(abb) = abb,$$

und $f_{M_2}(abba)$ ist nicht definiert. Es gilt

$$f_{M_2}(x_1x_2 \dots x_n) = \begin{cases} x_1x_2 \dots x_n & n \text{ ungerade} \\ nicht\ definiert & \text{sonst.} \end{cases}$$

c)Wir betrachten die Turing-Maschine

$$M_3 = (\{a, b, c, d\}, \{z_0, z_1, z_2, z_3, q, z_a, z_b\}, z_0, \{q\}, \delta)$$

mit

δ	z_0	z_1	z_2	z_3	z_a	z_b
$*$	$(z_0, *, N)$	$(z_1, *, N)$	$(z_3, *, L)$			
a	(z_0, a, N)	(z_1, a, N)	(z_2, a, R)	$(z_a, *, L)$	(z_a, a, L)	(z_a, b, L)
b	(z_0, b, N)	(z_1, b, N)	(z_2, b, R)	$(z_b, *, L)$	(z_b, a, L)	(z_b, b, L)
c	(z_1, c, R)	(z_1, c, N)	(z_2, c, N)			
d	(z_0, d, N)	(z_2, d, R)	(z_2, d, N)	(z_3, d, N)	(q, a, N)	(q, b, N)

Für diese TURING-Maschine ergibt sich

$$f_{M_3}(w) = \begin{cases} cx_1x_2\ldots x_n & \text{für } w = cdx_1x_2\ldots x_n, x_i \in \{a,b\}, 1 \leq i \leq n, n \geq 0 \\ \textit{nicht definiert} & \text{sonst} \end{cases}$$

Zur Begründung merken wir folgendes an: Wir laufen zuerst über das Wort, ändern den Zustand in z_1 bzw. z_2 wenn wir als ersten bzw. zweiten Buchstaben ein c bzw. ein d lesen und bleiben im Zustand z_2, wenn wir danach nur Buchstaben aus $\{a, b\}$ lesen. Damit wissen wir, dass das Eingabewort die Form hat, bei der eine Ausgabe definiert ist. Bei Erreichen des Wortendes gehen wir in den Zustand z_3. Jetzt laufen wir von rechts nach links über das Wort, merken uns jeweils einen gelesenen Buchstaben und schreiben diesen in die links davon befindliche Zelle. Dadurch verschieben wir das Wort über $\{a, b\}$ um eine Zelle nach links. Nach Lesen des d stoppt die Maschine.

Wir bemerken, dass M_3 im Wesentlichen den Buchstaben d löscht und die dadurch entstehende Lücke durch Verschiebung wieder füllt. Wir werden im Folgenden mehrfach davon Gebrauch machen, dass Streich-, Auffüll- und Verschiebungsoperationen von TURING-Maschinen realisiert werden können, ohne dies dann explizit auszuführen.

d) Wir wollen eine TURING-Maschine konstruieren, deren induzierte Funktion die Nachfolgerfunktion (bzw. die Addition von 1) ist, wobei wir die Dezimaldarstellung für Zahlen verwenden wollen.

Offenbar muss das Eingabealphabet aus den Ziffern $0, 1, 2, \ldots, 9$ bestehen. Wir werden als Grundidee die schriftliche Addition verwenden, d.h. wir verwenden den Anfangszustand z_0, um das Wortende zu finden, indem wir bis zum ersten $*$ nach rechts laufen; danach verwenden wir einen Zustand $+$ zur Addition von 1 bei der Ziffer, über der der Kopf gerade steht; die Addition kann abgebrochen werden, falls die Addition nicht zur Ziffer 9 erfolgt, bei der der entstehende Übertrag 1 zur Fortsetzung der Addition von 1 zu der links davon stehenden Ziffer notwendig wird. Formal ergibt sich die Maschine

$$M_+ = (\{0, 1, 2, \ldots, 9\}, \{z_0, +, q\}, z_0, \{q\}, \delta)$$

mit

δ	z_0	$+$
$*$	$(+, *, L)$	$(q, 1, N)$
0	$(z_0, 0, R)$	$(q, 1, N)$
1	$(z_0, 1, R)$	$(q, 2, N)$
2	$(z_0, 2, R)$	$(q, 3, N)$
3	$(z_0, 3, R)$	$(q, 4, N)$
4	$(z_0, 4, R)$	$(q, 5, N)$
5	$(z_0, 5, R)$	$(q, 6, N)$
6	$(z_0, 6, R)$	$(q, 7, N)$
7	$(z_0, 7, R)$	$(q, 8, N)$
8	$(z_0, 8, R)$	$(q, 9, N)$
9	$(z_0, 9, R)$	$(+, 0, L)$

Wir definieren die TURING-Berechenbarkeit nun dadurch, dass wir eine Funktion für berechbar erklären, wenn sie durch eine TURING-Maschine induziert werden kann. Formalisiert führt dies zu der folgenden Definition.

Definition 1.20 *Eine Funktion $f : X_1^* \to X_2^*$ heißt* TURING*-berechenbar, wenn es eine* TURING*-Maschine $M = (X, Z, z_0, Q, \delta)$ derart gibt, dass $X_1 \subseteq X$, $X_2 \subseteq X$ und*

$$f_M(x) = \begin{cases} f(x) & \text{falls } f(x) \text{ definiert ist} \\ nicht\ definiert & \text{sonst} \end{cases}$$

gelten.

Wir weisen darauf hin, dass das Eingabealphabet der TURING-Maschine, die f induziert, umfassender als die Alphabete sein kann, über denen die Wörter des Definitionsbereichs bzw. Wertevorrats von f gebildet werden. Es ist aber so, dass die TURING-Maschine auf Wörtern, die eines der zusätzlichen Symbole aus $X \setminus X_1$ enthalten nicht stoppt, also kein Ergebnis liefern kann.
Wir geben nun eine Normalform für TURING-Maschinen, d.h. wir geben eine eingeschränkte Form für TURING-Maschinen, die aber trotzdem in der Lage sind, alle TURING-berechenbaren Funktionen zu induzieren.

Lemma 1.21 *Zu jeder* TURING*-berechenbaren Funktion f gibt es eine* TURING*-Maschine M, die genau einen Stoppzustand hat, stets über dem dem ersten Buchstaben des Ergebnisses stoppt und $f = f_M$ erfüllt.*

Beweis. Da f TURING-berechenbar ist, gibt es eine TURING-Maschine $M = (X, Z, z_0, Q, \delta)$ mit $f_M = f$. Wir konstruieren die TURING-Maschine

$$M' = (X \cup \{\S, \#\}, Z \cup (Z \times \{\#\}) \cup (Z \times \{\S\}) \cup \{z_0', z_0'', q_1, q_2, q_3, q'\}, z_0', \{q'\}, \delta'),$$

wobei δ' wie folgt definiert ist:

(1) $\delta'(z_0', x) = (z_0', x, R)$ für $x \in X$,
$\delta'(z_0', \S) = (z_0', \S, N)$,

$\delta'(z_0', \#) = (z_0', \#, N)$,
$\delta'(z_0', *) = (z_0'', \#, L)$,
$\delta'(z_0'', x) = (z_0'', x, L)$ für $x \in X$,
$\delta'(z_0'', *) = (z_0, \S, R)$,

(2) $\delta'(z, x) = (z', x', r)$ für $x \in X \cup \{*\}, z \in Z \setminus Q, \delta(z, x) = (z', x', r), r \in \{R, L, N\}$,

(3) $\delta'(z, \S) = ((z, \S), *, L)$ für $z \in Z$,
$\delta'((z, \S), *) = (z, \S, R)$ für $z \in Z$,
$\delta'(z, \#) = ((z, \#), *, R)$ für $z \in Z$,
$\delta'((z, \#), *) = (z, \#, L)$ für $z \in Z$,

(4) $\delta'(q, x) = (q, x, R)$ für $x \in X \cup \{*\}, q \in Q$,
$\delta'(q, \#) = (q_1, *, L)$,
$\delta'(q_1, *) = (q_1, *, L)$,
$\delta'(q_1, x) = (q_2, x, L)$ für $x \in X$,
$\delta'(q_1, \S) = (q', *, N)$,
$\delta'(q_2, x) = (q_2, x, L)$ für $x \in X \cup \{*\}$,
$\delta'(q_2, \S) = (q_3, *, R)$,
$\delta'(q_3, *) = (q_3, *, R)$,
$\delta'(q_3, x) = (q', x, N)$ für $x \in X$

(für die Paare, für die δ' nicht definiert ist, kann ein beliebiges Tripel als Wert festgelegt werden, da die Arbeitsweise von M' sichert, dass solche Paare nicht erreicht werden können). Dass M' allen Bedingungen genügt, die in Lemma 1.21 gefordert werden, ist aus folgenden Überlegungen zu ersehen: Entsprechend der Definition von δ' im Teil (1) wird zuerst getestet, ob das Wort w auf dem Band ein § oder ein # enthält. Ist dies der Fall, so wird eine Schleife erreicht (die Konfiguration wird stets in sich selbst überführt) und damit kein Ergebnis von $f_{M'}$ erreicht. Ist kein § und kein # in w, so wird hinter das Wort ein # und vor das Wort ein § auf das Band geschrieben. Danach verhält sich die TURING-Maschine M' wegen der Definition von δ' in (2) genauso wie M', wobei mittels der Festlegungen in (3) gesichert wird, dass die Anfangsmarkierung § und die Endmarkierung # stets um eine Zelle nach links bzw. rechts verschoben wird, wenn dies erforderlich ist (d.h. wird ein § erreicht, so wird es durch $*$ ersetzt wird und danach wird die Arbeit in der Zelle, in der ursprünglich § stand und jetzt $*$ steht, mit dem Zustand z fortgesetzt, den die Maschine hatte, als sie diese Zelle betrat, da z in der ersten Komponente von $(z, \S)$ gespeichert wurde; analog wird bei # verfahren). Während M in Zuständen aus Q ihre Arbeit beendet, bewegt M' in diesen Zuständen den Kopf nach rechts, bis # erreicht wird, löscht #, geht dann nach links, bis § erreicht wird. M' löscht § und stoppt, falls zwischen § und # kein Symbol aus X stand, oder geht nach rechts bis zum ersten Buchstaben aus X und stoppt. □

Wir beweisen nun, dass TURING-berechenbare Funktionen eine Eigenschaft haben, die in Satz 1.6 bereits für **LOOP**/**WHILE**-berechenbare Funktionen nachgewiesen wurde.

Lemma 1.22 *Sind $f_1 : X^* \to X^*$ und $f_2 : X^* \to X^*$ zwei* TURING*-berechenbare Funktionen, so ist auch deren Komposition $f : X^* \to X^*$ mit $f(w) = f_2(f_1(w))$ eine* TURING*-berechenbare Funktion.*

Beweis. Nach Definition 1.20 und Lemma 1.21 gibt es TURING-Maschinen

$$M_1 = (X_1, Z_1, z_{0,1}, \{q_1\}, \delta_1) \text{ und } M_2 = (X_2, Z_2, z_{0,2}, \{q_2\}, \delta_2)$$

mit $X \subseteq X_1$ und $X \subseteq X_2$ mit

$$f_{M_1}(w) = \begin{cases} f_1(w) & \text{für } w \in dom(f_1), \\ nicht\ definiert & \text{sonst} \end{cases}$$

und

$$f_{M_2}(w) = \begin{cases} f_2(w) & \text{für } w \in dom(f_2) \\ nicht\ definiert & \text{sonst} \end{cases}$$

und der Eigenschaft, dass beide TURING-Maschinen über dem ersten Buchstaben des Ergebnisses stoppen. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass Z_1 und Z_2 kein Element gemeinsam haben, und betrachten die TURING-Maschine

$$M = (X_1 \cup X_2, Z_1 \cup Z_2, z_{0,1}, \{q_2\}, \delta)$$

mit

$$\delta(z,x) = \begin{cases} \delta_1(z,x) & \text{für } z \in Z_1, z \neq q_1, x \in X_1 \\ (z_{0,2}, x, N) & \text{für } z = q_1 \\ \delta_2(z,x) & \text{für } z \in Z_2, z \neq q_2, x \in X_2 \end{cases}.$$

Da der Anfangszustand von M in Z_1 liegt, beginnt M auf der Eingabe w wie M_1 zu arbeiten, bis der Endzustand q_1 von M_1 erreicht wird und damit die Konfiguration $(\lambda, q_1, f_{M_1}(w))$ vorliegt. Für M ist q_1 kein Endzustand und erreicht nach Definition von δ die Konfiguration $(\lambda, z_{0,2}, f_{M_1}(w))$, die gerade die Anfangskonfiguration von M_2 bei Eingabe von $f_{M_1}(w)$ ist. Nun verhält sich M wie M_2 und stoppt mit der Konfiguration $(\lambda, q_2, f_{M_2}(f_{M_1}(w)))$. Damit ergibt sich

$$f_M(w) = f_{M_2}(f_{M_1}(w))) = f_2(f_1(w)) = f(w).$$

Daher wird f von einer TURING-Maschine induziert und ist damit TURING-berechenbar. □

1.1.3 Äquivalenz der Berechenbarkeitsbegriffe

Wir haben oben gesehen, dass sowohl für TURING-berechenbare als auch **LOOP/WHILE**-berechenbare Funktionen gilt, dass durch Komposition (Einsetzung) stets wieder Funktionen entstehen, die berechenbar sind. Wir wollen nun zeigen, dass dies kein Zufall ist, denn durch beide Berechenbarkeitsbegriffe wird im Wesentlichen die gleiche Menge von Funktionen beschrieben.
Die gerade vorgenommene Einschränkung auf ,,im Wesentlichen" ist sicher erforderlich, denn nach Definition sind der Definitionsbereich und Wertevorrat von **LOOP/WHILE**-berechenbaren Funktionen kartesische Produkte von $\mathbf{N}_0$, während bei TURING-berechenbaren Funktionen Definitionsbereich und Wertevorrat Mengen von Wörtern über gewissen

Alphabeten sind. Dieser Unterschied zeigt schon, dass eine wirkliche Gleichheit der beiden Berechenbarkeitsbegriffe im Sinne übereinstimmender Mengen berechenbarer Funktionen nicht gegeben sein kann.
Um die genannten Unterschiede in den Definitionsbereichen und Wertevorraten auszugleichen, benötigen wir offenbar Verfahren, die (Tupel von) Zahlen in Wörter und umgekehrt überführen.
Nach Definition ist jede natürliche Zahl eine Äquivalenzklasse gleichmächtiger Mengen. Wenn wir Zahlen aber aufschreiben, so benutzen wir stets eine Darstellung von Zahlen in einem Zahlensystem. Zum Beispiel wird durch die Dezimaldarstellung jede natürliche Zahl als Folge von Ziffern aus der Menge $\{0, 1, 2, \ldots, 9\}$ geschrieben. Eine derartige Folge von Ziffern ist aber offensichtlich ein Wort über $\{0, 1, 2, \ldots, 9\}$. Damit haben wir also eine Möglichkeit gefunden, Zahlen in Wörter zu überführen.
Mit $dec(n)$ bezeichnen wir die Dezimaldarstellung der Zahl $n \in \mathbf{N}_0$.
Der folgende Satz gibt nun an, dass bis auf die Transformation der Zahlen in ihre Dezimaldarstellung jede **LOOP/WHILE**-berechenbare Funktion auch TURING-berechenbar ist.

Satz 1.23 *Es sei f eine n-stellige* **LOOP/WHILE***-berechenbare Funktion.*
Dann ist die Funktion f', die durch

$$f'(w) = \begin{cases} dec(f(x_1, x_2, \ldots, x_n)) & \text{falls } w = dec(x_1)\#dec(x_2)\#\ldots\#dec(x_n) \\ nicht\ definiert & \text{sonst} \end{cases}$$

definiert ist, TURING*-berechenbar.*

Beweis. Wir zeigen sogar etwas mehr. Für jedes **LOOP**/bf WHILE-Programm Π gibt es eine TURING-Maschine M derart, dass

$$f_M(w) = \begin{cases} dec(y_1)\#dec(y_2)\#\ldots\#dec(y_n) & \text{falls } w = dec(x_1)\#dec(x_2)\#\ldots\#dec(x_n) \\ nicht\ definiert & \text{sonst} \end{cases}$$

gilt, wobei $y_i = \Phi_{\Pi,i}(x_1, x_2, \ldots, x_n)$ für $1 \leq i \leq n$ gültig ist. Wir berechnen mit der TURING-Maschine also aus den Eingaben aller Variablen in Dezimaldarstellung auch die Ausgaben an den entsprechenden Stellen in Dezimaldarstellung. Da das Streichen der Symbole # und der Dezimaldarstellungen von $y_2, y_3, \ldots, y_n$ offenbar durch eine TURING-Maschine M' vorgenommen werden kann, ist auch f' nach Lemma 1.22 TURING-berechenbar, da f' aus der Substitution von f_M und $f_{M'}$ entsteht.
Wir geben den Beweis für die Existenz von M durch Induktion über die Tiefe t des Programms Π.
Induktionsanfang ($t = 1$): Dann ist Π eine der Grundanweisungen $x_i := 0$ oder $x_i := x_j$ oder $x_i := S(x_j)$ oder $x_i := P(x_j)$. Wir geben hier nur den Beweis für $x_i := 0$ und überlassen dem Leser den Beweis für die restlichen Fälle; es sind nur leichte Modifikationen und der Gebrauch der TURING-Maschinen für $S(x_j)$ bzw. $P(x_j)$ (die in den Beispielen bzw. Übungsaufgaben behandelt wurden) erforderlich.
Wir haben zu zeigen, dass es eine TURING-Maschine M gibt, die die Eingabe

$$dec(x_1)\#dec(x_2)\#\ldots\#dec(x_{i-1})\#dec(x_i)\#dec(x_{i+1})\#\ldots\#dec(x_n)$$

in die Ausgabe

$$dec(x_1)\#dec(x_2)\#\ldots\#dec(x_{i-1})\#0\#dec(x_{i+1})\#\ldots\#dec(x_n)$$

überführt. Dies leistet die TURING-Maschine

$$M = (\{0,1,2,\ldots,9,\#\}, Z, z_0'', \{q\}, \delta)$$

mit

$$Z = \{z_0'', z_1'', \ldots, z_{i-1}'', z_1', z_2', z_3', z_4', z_5', q\} \cup \{z_a \mid a \in \{0,1,\ldots,9,\#\}\}$$

und

(1)	$\delta(z_j'', \#) = (z_{j+1}'', \#, R)$	für $1 \leq j \leq i-2$,
(2)	$\delta(z_j'', a) = (z_j'', a, R)$	für $1 \leq j \leq i-2$ und $a \in \{0,1,\ldots,9\}$,
(3)	$\delta(z_{i-1}'', a) = (z_1', 0, R)$,	
(4)	$\delta(z_1', \#) = (q, \#, N)$,	
(5)	$\delta(z_1', a) = (z_2', *, R)$	für $a \in \{0,1,\ldots,9\}$,
(6)	$\delta(z_2', a) = (z_2', *, R)$	für $a \in \{0,1,\ldots,9\}$,
(7)	$\delta(z_2', \#) = (z_3', \#, R)$	für $a \in \{0,1,\ldots,9\}$,
(8)	$\delta(z_3', x) = (z_3', x, R)$	für $x \in \{0,1,\ldots,9,\#\}$,
(9)	$\delta(z_3', *) = (z_4', *, L)$	für $a \in \{0,1,\ldots,9\}$,
(10)	$\delta(z_4', a) = (z_a, *, L)$	für $a \in \{0,1,\ldots,9\}$,
(11)	$\delta(z_a, b) = (z_b, a, L)$	für $a, b \in \{0,1,\ldots,9,\#\}$,
(12)	$\delta(z_a, *) = (z_5', a, L)$	für $a \in \{0,1,\ldots,9,\#\}$,
(13)	$\delta(z_5', *) = (z_2', *, R)$	für $a \in \{0,1,\ldots,9\}$,
(14)	$\delta(z_5', 0) = (q, 0, N)$.	

Entsprechend (1) - (2) bewegt sich der Kopf ohne Veränderung des Bandes nach rechts und zählt (im Index) dabei die Anzahl der #, bis er den ersten Buchstaben nach dem $(i-1)$-ten #, also die erste Ziffer von $dec(x_i)$, erreicht hat. Wegen (3) wird diese Ziffer nun durch 0 ersetzt. Ist der nachfolgende Buchstabe ein #, so besteht $dec(x_i)$ aus nur einer Ziffer, die durch 0 ersetzt wurde, und die gewünschte Ausgabe steht auf dem Band. Dies wird durch (4) abgesichert. Besteht $dec(x_i)$ nicht nur aus einer Ziffer, so werden nach (5)-(6) die restlichen Ziffern durch ein $*$ ersetzt. Ist das nächste Trennsymbol # erreicht, wird wegen (7) - (9) in z_3' gewechselt und dann ans Ende des Wortes gelaufen. Nun wird durch (10) - (12) der Teil des Wortes hinter der 0, die auf das Band geschrieben wurde, um eine Stelle nach links verschoben. Falls nun noch weitere $*$ auf dem Band sind, ist eine weitere Verschiebung nach links erforderlich, was dadurch erreicht wird, dass entsprechend (13) wieder in den Zustand z_3' gewechselt wird. Ist dagegen kein $*$ mehr auf dem Band (d.h. die 0 wurde erreicht), so steht dass Ergebnis auf dem Band und der Stoppzustand wird erreicht (siehe (14)).

Seien nun $t \geq 2$ und Π ein **LOOP**/**WHILE**-Programm der Tiefe t. Dann entsteht Π durch Hintereinanderausführung zweier Programme Π_1 und Π_2 oder durch eine **LOOP**- oder **WHILE**-Anweisung, die das Programm Π' verwendet.

Es sei $\Pi = \Pi_1; \Pi_2$. Wir bezeichnen mit $x_1, x_2, \ldots, x_n$ die Eingabewerte von Π_1. Diese werden durch Π_1 in die Ausgaben $y_1, y_2, \ldots, y_n$ überführt. Diese Werte $y_1, y_2, \ldots, y_n$ werden von Π_2 als Eingaben genommen und in die Ausgaben $z_1, z_2, \ldots z_n$ transformiert.

Die Tiefe von Π_1 und Π_2 ist höchstens $t-1$. Folglich gibt es nach Induktionsvoraussetzung Maschinen M_1 und M_2 derart, dass durch M_1 die Eingabe $dec(x_1)\#dec(x_2)\#\ldots\#dec(x_n)$ in die Ausgabe $dec(y_1)\#dec(y_2)\#\ldots\#dec(y_n)$ überführt wird und durch M_2 die Eingabe $dec(y_1)\#dec(y_2)\#\ldots\#dec(y_n)$ in die Ausgabe $dec(z_1)\#dec(z_2)\#\ldots\#dec(z_n)$ transformiert wird. Nach Lemma 1.22 gibt es dann auch eine TURING-Maschine M, die die Eingabe $dec(x_1)\#dec(x_2)\#\ldots\#dec(x_n)$ in die Ausgabe $dec(z_1)\#dec(z_2)\#\ldots\#dec(z_n)$ transformiert. Folglich stimmen die Berechnungen von Π und M bis auf die Repräsentation der Zahlen überein.

Es sei $\Pi =$ **WHILE** $x_i \neq 0$ **BEGIN** Π' **END**. Die Tiefe von Π' ist $t-1$, und daher gibt es nach Induktionsvoraussetzung eine TURING-Maschine $M' = (X', Z', z_0', \{q'\}, \delta')$ (wir können ohne Beschränkung der Allgemeinheit nach Lemma 1.21 annehmen, dass M' nur einen Endzustand hat und mit dem Kopf über dem ersten Buchstaben des Ergebnisses stoppt) derart, dass die Eingabe $dec(x_1)\#dec(x_2)\#\ldots\#dec(x_n)$ in die Ausgabe $dec(y_1)\#dec(y_2)\#\ldots\#dec(y_n)$ überführt wird, wobei $x_1, x_2, \ldots, x_n$ die Eingaben und $y_1, y_2, \ldots, y_n$ die Ausgaben von Π' sind.
Wir konstruieren nun die TURING-Maschine M, die folgende Schritte abarbeitet (die formale Definition von M bleibt dem Leser überlassen): M bewegt sich von ihrem Anfangszustand z_0 zuerst nach rechts, zählt die # und geht über dem ersten Buchstaben von $dec(x_i)$ in dem Zustand z. Liest M nun eine 0, so wird die **WHILE**-Schleife nicht durchlaufen. Die Maschine M beendet daher ihre Arbeit, indem sie in den Stoppzustand q von M übergeht. Liest M dagegen keine 0, so geht M an den Anfang des Wortes zurück und geht über dem ersten Buchstaben in den Zustand z_0' (den Anfangszustand von M') über. Nun schließt sich eine Berechnung an, die völlig der von M' entspricht. Dieses Arbeit wird in q' beendet (womit ein Durchlauf der **WHILE**-Schleife abgearbeitet ist) und der Kopf steht über dem ersten Buchstaben. Nun wird in den Zustand z_0 gewechselt. Hierdurch wird der nächste Durchlauf der **WHILE**-Schleife eingeleitet, sofern er notwendig ist.
Es ist leicht zu sehen, dass M das Programm Π simuliert.

Da jede **LOOP**-Anweisung durch eine **WHILE**-Schleife simuliert werden kann (siehe am Ende des Abschnitts 1.1.1, Seite 19), können wir auf die Konstruktion einer TURING-Maschine für die **LOOP**-Anweisung verzichten. □

Wir kommen nun zur umgekehrten Richtung. Hierfür benötigen wir zuerst eine Transformation von Wörtern in Zahlen. Auch hierfür benutzen wir eine Darstellung bezüglicher einer Basis (die aber im Allgemeinen nicht die Dezimaldarstellung sein wird).
Sei $M = (X, Z, z_0, Q, \delta)$ eine TURING-Maschine. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass X und Z disjunkt sind. Es sei

$$X \cup Z \cup \{*\} = \{a_1, a_2, \ldots, a_p\}.$$

Dann definieren wir die Funktion $\psi : (X \cup Z \cup \{*\})^* \to \mathbf{N}$ durch

$$\psi(a_{i_1}a_{i_2}\ldots a_{i_n}) = \sum_{j=0}^{n} i_j(p+1)^{n-j},\ a_{i_j} \in X \cup Z \cup \{*\}$$

($i_1 i_2 \ldots i_n$ ist die $(p+1)$-adische Darstellung von $\psi(a_{i_1}a_{i_2}\ldots a_{i_n})$). Ist umgekehrt x eine beliebige natürliche Zahl, in deren $(p+1)$-adischer Darstellung $i_1 i_2 \ldots i_n$ keine 0 vorkommt, so gilt $\psi^{-1}(x) = a_{i_1}a_{i_2}\ldots a_{i_n}$. Daher ist ψ eine eineindeutige Abbildung der Menge der

nichtleeren Wörter über $X \cup Z \cup \{*\}$ auf die Menge aller natürlichen Zahlen, in deren $(p+1)$-adischer Darstellung keine 0 vorkommt. (Bei Benutzung einer p-adischen Darstellung müsste ein Element aus $X \cup Z \cup \{*\}$ der Null zugeordnet werden, wodurch Darstellungen mit führenden Nullen möglich sind, was ausgeschlossen sein soll.)
Es seien $w = b_1 b_2 \dots b_n$ mit $b_i \in X \cup Z$ für $1 \leq i \leq n$ und $w' \in (X \cup Z)^*$. Dann gelten – wie man leicht nachrechnet – folgende Beziehungen für die folgenden Funktionen:

$$\begin{aligned}
Lg(\psi(w)) &= |w| = min\{m : (p+1)^m > \psi(w)\},\\
Prod(\psi(w), \psi(w')) &= \psi(ww') = \psi(w)(p+1)^{Lg(\psi(w'))} + \psi(w'),\\
Anfang(\psi(w), i) &= \psi(b_1 b_2 \dots b_i) = \psi(w)\ div\ (p+1)^{n-1}\ (ganzzahligeDivision),\\
Ende(\psi(w), i) &= \psi(b_i b_{i+1} \dots b_n) = \psi(w)\ mod\ (p+1)^{n-i+1},\\
Elem(\psi(w), i) &= \psi(b_i) = Ende(Anfang(\psi(w), i), 1)
\end{aligned}$$

(für die Definition von *div* und *mod* siehe Übungsaufgabe 11), d.h. aus der Kenntnis von $\psi(w)$ und $\psi(w')$ können wir den Wert von ψ auf Anfangsstücken und Endstücken von w sowie ww' berechnen. Man erkennt aus den angegebenen Funktionen, den Sätzen 1.6, 1.7 und 1.8 und Übungsaufgaben 1, 5 und 11, dass die Berechnung des Wertes auf Anfangsstücken, Endstücken und Produkten mittels **LOOP/WHILE**-berechenbarer Funktionen möglich ist.
Zuküftig schreiben wir $Prod(x, y, z)$ für $Prod(Prod(x, y), z)$ (dies ist möglich, da $Prod$ assoziativ ist, wie man leicht nachrechnet.)
Wegen $X \cap Z = \emptyset$ können wir eine Konfiguration (u, z, v) auch als Wort $K = uzv$ angeben, da der Zustand in K eindeutig bestimmt ist. Wir zeigen nun, dass wir die Stelle, an der z steht, mittels **LOOP/WHILE**-berechenbarer Funktionen auch aus $\psi(K)$ berechnen können.
Es sei $f : \mathbf{N}_0 \to \mathbf{N}_0$ die Funktion

$$f(t) = \begin{cases} 0 & \psi^{-1}(t) \in Z \\ 1 & \text{sonst} \end{cases}.$$

f ist **LOOP/WHILE**-berechenbar (siehe Übungsaufgabe 5). Nun definieren wir die Funktion $g : \mathbf{N} \to \mathbf{N}$ durch

$$g(\psi(K)) = \min\{i \mid f(Elem(\psi(K), i)) = 0\},$$

d.h. für ein Wort $w = x_1 x_2 \dots x_n$ wird der minimale (bei Konfigurationen sogar einzige) Index i mit $x_i \in Z$ bestimmt. Damit ist gezeigt, dass die Stelle in einer Konfiguration w, an der der Zustand z steht, mittels **LOOP/WHILE**-berechenbarer Funktionen ermittelt werden kann.
Wir definieren die folgenden **LOOP/WHILE**-berechenbaren Funktionen:

$$\begin{aligned}
r(x) &= Anfang(x, g(x) \ominus 2),\\
s(x) &= Prod(Elem(x, g(x) \ominus 1), Elem(x, g(x)), Elem(x, g(x)+1)),\\
t(x) &= Ende(x, g(x)+2).
\end{aligned}$$

Für ein Wort $K = u'azbv'$ mit $a, b \in X$, $u', v' \in X^*$ und $z \in Z$, das eine Konfiguration beschreibt, ergeben sich folgende Werte:

$$r(\psi(K)) = \psi(u'), \quad s(\psi(K)) = \psi(azb), \quad t(\psi(K)) = \psi(v').$$

Ferner gilt

$$\psi(K) = Prod(r(\psi(K)), s(\psi(K)), t(\psi(K)).$$

Wir definieren Δ auf der Menge der Kodierungen von Konfigurationen durch

$$\Delta(\psi(K_1)) = \begin{cases} \psi(K_2) & K_1 = azb, a, b \in X, z \in Z, K_1 \models K_2 \\ nicht\ definiert & \text{sonst} \end{cases}.$$

Nach Übungsaufgabe 5 ist Δ **LOOP/WHILE**-berechenbar.
Damit ist die Konfiguration K', in die $K = u'azbv'$ mittels δ überführt wird wie folgt kodiert:

$$\begin{aligned} \psi(K') &= Prod(\psi(u'), \Delta(\psi(azb)), \psi(v')), \\ &= Prod(r(K), \Delta(s(K)), t(K)). \end{aligned}$$

Entsprechende Relationen lassen sich auch herleiten, wenn die Konfiguration nicht durch ein Wort der Form $K = u'azbv'$ beschrieben wird. Insgesamt ergibt sich dann, dass die Funktion $\overline{\Delta}$ mit $\overline{\Delta}(\psi(K)) = \psi(K')$ **LOOP/WHILE**-berechenbar ist. Damit haben wir gezeigt, dass die Relation $\models$ mittels der **LOOP/WHILE**-berechenbaren Funktion $\overline{\Delta}$ simuliert werden kann.
Wir erweitern dies auf die Iteration von $\models$, indem wir die Funktion D mittels Methode aus Satz 1.7 durch

$$\begin{aligned} D(x, 0) &= x, \\ D(x, n+1) &= \overline{\Delta}(D(x, n)) \end{aligned}$$

definieren. Damit gilt $D(\psi(K), n) = \psi(K'')$, wobei K'' die Konfiguration ist, die aus K mittels n-facher direkter Überführung entsteht.

Entsprechend der Definition der TURING-Berechenbarkeit wird einem Wort w das Wort w' zugeordnet, das bei Erreichen einer Endkonfiguration auf dem Band steht. Intuitiv werden also folgende Schritte unternommen:

1. Aus w ergibt sich die Anfangskonfiguration $K_0 = z_0 w$.
2. Aus K_0 wird die Folge der Konfigurationen berechnet bis eine Endkonfiguration $\overline{K}$ vorliegt.
3. Aus $\overline{K}$ ermitteln wir das Wort auf dem Band.

Offenbar ist der Schritt 1 durch eine **LOOP/WHILE**-berechenbare Funktion, die $\psi(w)$ auf $\psi(K_0) = \psi(z_0 w)$ abbildet, simulierbar. Die Folge der Kodierungen der Konfigurationen ist nach obigem ebenfalls mittels **LOOP/WHILE**-berechenbarer Funktionen berechenbar. Da die Maschine bei Erreichen der ersten Endkonfiguration stoppt, kann die Endkonfiguration mittels der Funktionen

$$\begin{aligned} Stop_1(\psi(K)) &= \begin{cases} 0 & K \text{ ist Endkonfiguration} \\ 1 & \text{sonst} \end{cases}, \\ Stop_2(\psi(K)) &= \min\{i \mid Stop_1(D(K, i)) = 0\}, \\ Stop_3(\psi(K)) &= D(K, Stop_2(K)) \end{aligned}$$

berechnet werden ($Stop_1$ stellt fest, ob $\psi(K)$ eine Kodierung einer Endkonfiguration ist, $Stop_2$ berechnet die Anzahl der Schritte bis zum Erreichen einer Endkonfiguration bei

Start mit K, und $Stop_3$ berechnet dann die Kodierung der zu K gehörigen Endkonfiguration). Unter Verwendung der obigen Ideen ist leicht zu zeigen, dass $Stop_1$ **LOOP/WHILE**-berechenbar ist (wir bestimmen zuerst den Zustand in K und testen dann, ob er in Q liegt). Dann sind nach Konstruktion auch $Stop_2$ und $Stop_3$ **LOOP/WHILE**-berechenbar. Wenn wir nun noch beachten, dass auch der obige dritte Schritt mittels **LOOP/WHILE**-berechenbarer Funktionen simuliert werden kann (wir können ausgehend von $\psi(v_1qv_2)$ sowohl $\psi(v_1), \psi(v_2)$ als auch $\psi(v_1v_2)$ und damit $\psi(v)$ berechnen), ist damit gezeigt, dass die Funktion p, die jeder Zahl $\psi(w)$, $w \in X^*$, den Wert $\psi(f_M(w))$ zuordnet, **LOOP/WHILE**-berechenbar ist.

Aus diesen Überlegungen resultiert der folgende Satz.

Satz 1.24 *Seien M eine* TURING*-Maschine und ψ die zugehörige Kodierung. Dann ist die Funktion $f : \mathbf{N}_0 \to \mathbf{N}_0$ mit $f(\psi(w)) = \psi(f_M(w))$* **LOOP/WHILE***-berechenbar.* □

Fassen wir unsere Ergebnisse über Beziehungen zwischen Berechenbarkeitsbegriffen zusammen, so ergibt sich folgender Satz.

Satz 1.25 *Für eine Funktion f sind die folgenden Aussagen gleichwertig:*

- *f ist durch ein* **LOOP/WHILE***-Programm berechenbar.*
- *f ist bis auf Konvertierung der Zahlendarstellung durch eine* TURING*-Maschine berechenbar.* □

Damit erhalten wir auch die folgende Folgerung.

Folgerung 1.26 *Es gibt Funktionen, die nicht* TURING*-berechenbar sind.* □

Der amerikanische Logiker A. CHURCH (1903–1995) hat nun die nach ihm benannte These aufgestellt, dass eine Funktion, die berechenbar in irgendeinem Sinn (der den eingangs formulierten intuitiven Bedingungen genügt) ist, auch TURING-berechenbar (und damit **LOOP/WHILE**-berechenbar) ist, d.h. dass die von uns hier eingeführten Berechenbarkeitsbegriffe die allgemeinsten sind. Auch alle anderen bisher betrachteten Berechenbarkeiten lieferten tatsächlich nur TURING-berechenbare Funktionen. Daher wird die CHURCHsche These heute allgemein akzeptiert. (Die These kann nicht bewiesen werden, da eine allgemeine Formalisierung des intuitiven Algorithmenbegriffs nicht möglich ist; sie ließe sich aber widerlegen, indem man zeigt, dass bei einer speziellen Formalisierung Funktionen als berechenbar gelten, die nicht TURING-berechenbar sind.)

1.2 Entscheidbarkeit von Problemen

Unter einem Problem (genauer einem Entscheidungsproblem) P verstehen wir im Folgenden stets eine Aussageform, d.h. einen Ausdruck $A(x_1, x_2, \ldots, x_n)$, der eine oder mehrere Variable x_i, $1 \leq i \leq n$, enthält und der bei Ersetzen der Variablen x_i durch Elemente a_i aus dem zugehörigen Grundbereich X_i, $1 \leq i \leq n$, in eine Aussage $A(a_1, a_2, \ldots, a_n)$ überführt wird, die den Wahrheitswert „wahr“, repräsentiert durch 1, oder den Wahrheitswert „falsch“, repräsentiert durch 0, annimmt. Wir beschreiben ein Problem im Folgenden daher

- durch ein *"Gegeben:"*, das eine Belegung $a_1, a_2, \ldots, a_n$ der Variablen angibt, und
- durch die *"Frage:"* nach der Gültigkeit der Aussage $A(a_1, a_2, \ldots, a_n)$.

Beim *Halteproblem für* TURING-*Maschinen* wird die Aussageform

x stoppt bei Abarbeitung von y.

mit den Variablen x und y betrachtet. Dabei ist x mit einer TURING-Maschine und y mit einem Wort zu belegen. Damit können wir das Halteproblem durch

Gegeben: TURING-Maschine M, Wort w
Frage: Gilt „M stoppt bei Abarbeitung von w“ ?

beschreiben. Offenbar ist die folgende Beschreibung dazu gleichwertig, da bei ihr nur die hinter dem Problem stehende Aussage schon als Frage formuliert wird.

Gegeben: TURING-Maschine M, Wort w
Frage: Stoppt M bei Abarbeitung von w ?

Eine andere Beschreibung des Halteproblems ist durch

Gegeben: TURING-Maschine M, Wort w
Frage: Ist $f_M(w)$ definiert?

gegeben, bei der nur die obige Frage durch eine gleichwertige ersetzt wurde.
Betrachten wir den Ausdruck

x ist eine Primzahl.

so ergeben sich

Gegeben: natürliche Zahl n
Frage: Gilt „n ist eine Primzahl“ ?

und

Gegeben: natürliche Zahl n
Frage: Ist n eine Primzahl?

als mögliche Beschreibungen des Problems.

Diese Beschreibung eines Problems ist intuitiv meist die verständlichste, und daher werden wir sie in diesem Abschnitt bevorzugen. Aber sie gestattet kaum eine präzise Fassung des Begriffs der (algorithmischen) Entscheidbarkeit. Daher geben wir noch weitere Formen zur Beschreibung von Problemen an, die dies gestatten.
Eine Menge M lässt sich in der Regel durch eine Eigenschaft angeben, die (genau) ihren Elementen zukommt. Formal wird dies durch

$$M = \{x : x \in X \text{ und } A(x)\} \tag{1.1}$$

ausgedrückt, wobei X der Grundbereich ist, dem die Elemente x zu entnehmen sind, und A ein Ausdruck ist, der die Eigenschaft beschreibt. Unter Verwendung dieser Schreibweise lässt sich ein Problem P, das durch den Ausdruck A beschrieben ist, auch als Menge

$$M = \{(a_1, a_2, \ldots, a_n) : a_i \in X_i \text{ für } 1 \leq i \leq n \text{ und } A(a_1, a_2, \ldots, a_n)\}$$

angeben. Wenn die Grundbereiche aus dem Kontext klar sind, lassen wir diese fort.
Für unsere beiden obigen Beispiele ergeben sich die Mengen

$$M_{halt} = \{(M, w) : f_M(w) \text{ ist definiert}\}$$

und

$$P = \{n : n \text{ ist prim}\}.$$

Für die Grundbereiche ist im Fall der Menge der Primzahlen die Menge $\mathbf{N}$ der natürlichen Zahlen zu wählen. Beim Halteproblem gehen wir zur Bestimmung des Grundbereichs wie folgt vor: Für ein Alphabet X sei M_X die Menge aller TURING-Maschinen mit Eingabealphabet X. Dann muss

$$M_{halt} \subseteq \bigcup_X M_X \times X^*$$

gefordert werden.
Eine weitere Beschreibung von Problemen kann durch Funktionen vorgenommen werden. Dabei gehen wir von der Mengendarstellung (1.1) aus und definieren die Funktion

$$\varphi_M(x) = \begin{cases} 1 & x \in M \\ 0 & \text{sonst} \end{cases},$$

die charakteristische Funktion der Menge M genannt wird. Für unsere beiden Beispiele ergeben sich (bei Fortlassung der Grundbereiche), über denen die Funktion definiert ist,

$$\varphi_1(M, w) = \begin{cases} 1 & M \text{ stoppt auf } w \\ 0 & \text{sonst} \end{cases}$$

und

$$\varphi_2(n) = \begin{cases} 1 & n \text{ ist Primzahl} \\ 0 & \text{sonst} \end{cases}.$$

Auf diese Weise gehören zu jedem Problem P ein Ausdruck A_P, eine Menge M_P und eine Funktion φ_P mit

$$M_P = \{(a_1, a_2, \ldots, a_n) : A_P(a_1, a_2, \ldots, a_n)\} \quad \text{und} \quad \varphi_P = \begin{cases} 1 & A_P(a_1, a_2, \ldots, a_n) \\ 0 & \text{sonst} \end{cases}.$$

Offenbar beschreiben umgekehrt jede Menge und jede Funktion mit Wertevorrat $\{0, 1\}$ auch ein Problem.
Wir werden in den folgenden Ausführungen stets die Beschreibung des Problems so wählen, wie wir es für günstig in dem Zusammenhang halten.

Definition 1.27 *Wir sagen, dass ein Problem P algorithmisch entscheidbar (oder kurz nur entscheidbar) ist, wenn die entsprechend dieser Konstruktion zum Problem gehörende charakteristische Funktion φ_P* TURING*-berechenbar ist. Anderenfalls heißt P (algorithmisch) unentscheidbar.*

Natürlich ist dies gleichwertig zu der Forderung, dass ϕ_P **LOOP**/**WHILE**-berechenbar ist.
Da Probleme als Mengen interpretiert werden können, ist klar, dass wir anstelle der Entscheidbarkeit von Problemen auch die Entscheidbarkeit von Mengen definieren können,

wofür auch der Begriff der Rekursivität der Menge benutzt wird.

Definition 1.27' *Wir sagen, dass eine Menge M (algorithmisch) entscheidbar (oder rekursiv) ist, wenn die zugehörige charakteristische Funktion φ_M* TURING*-berechenbar ist. Anderenfalls heißt M (algorithmisch) unentscheidbar.*

Offenbar gilt, dass *ein Problem P genau dann entscheidbar ist, wenn die zugehörige Menge M_P entscheidbar ist,* da die zugehörigen charakteristischen Funktionen identisch sind.

Bisher haben wir Entscheidungsprobleme behandelt, bei denen der Ausdruck nach Belegung nur einen der beiden Wahrheitswerte annehmen kann. Daneben gibt es natürlich auch noch Berechnungsprobleme, bei denen eine Funktion $f : X \to Y$ gegeben ist und nach dem Wert $f(x)$ für ein gegebenes x gefragt wird. Wir wollen dabei hier annehmen, dass die Funktion f für jedes $x \in X$ definiert ist. (Die einfache Erweiterung auf den Fall partieller Funktionen bleibt dem Leser überlassen.) Formal wird eine Funktion als Menge definiert, und zwar als

$$M_f = \{(x, y) : f(x) = y\}.$$

Dies ist offensichtlich die Mengenbeschreibung des Entscheidungsproblems

Gegeben:	$x \in X$ und $y \in Y$
Frage:	Nimmt f an der Stelle x den Wert y an?

dessen charakteristische Funktion durch

$$\varphi(x, y) = \begin{cases} 1 & f(x) = y \\ 0 & \text{sonst} \end{cases}$$

gegeben ist. Somit reicht es, im Folgenden nur Entscheidungsprobleme zu betrachten, da Berechnungsprobleme in solche umformuliert werden können.
Als ein Beispiel für ein entscheidbares Problem geben wir das folgende an:

Gegeben:	$w \in \{0, 1\}^*$
Frage:	Hat w ungerade Länge?

Mittels einer leichtem Modifikation der TURING-Maschine aus Beispiel 1.19 b) lässt sich die TURING-Berechenbarkeit von

$$\varphi_P(w) = \begin{cases} 1 & w \text{ hat ungerade Länge} \\ 0 & \text{sonst} \end{cases}$$

zeigen.
Andererseits folgt aus Obigem und Folgerung 1.11 sofort, dass es ein unentscheidbares Problem gibt. Jedoch scheint das aus Folgerung 1.11 resultierende Problem relativ künstlich zu sein, da die im Beweis von Satz 1.3 konstruierte Funktion auf den ersten Blick keinen praktischen Sinn hat. Daher wollen wir nun ein weiteres unentscheidbares Problem angeben, das (zumindest in gewissen Grenzen) eine Interpretation für Programmiersprachen besitzt.

Satz 1.28 *Das Halteproblem für* TURING*-Maschinen ist unentscheidbar.*

Beweis: Sei $M = (X, Z, z_0, Q, \delta)$ eine TURING-Maschine. Zur vollständigen Angabe von M reicht es offenbar aus, alle Elemente aus X, alle Elemente aus $Z \setminus Q$ und alle Elemente aus der Menge $\delta \subseteq (Z \setminus Q) \times (X \cup \{*\}) \times Z \times (X \cup \{*\}) \times \{R, L, N\}$ (jede Funktion $f : X \to Y$ kann als Relation $R \subseteq X \times Y$ aufgefasst werden) anzugeben, wenn wir ohne Beschränkung der Allgemeinheit vereinbaren, bei der Angabe der Elemente aus Z mit z_0 anzufangen. (Q lässt sich dann als die Menge der Zustände ermitteln, die in der dritten Komponente von δ, aber nicht in $Z \setminus Q$ vorkommen.) Seien $X = \{x_1, x_2, \ldots, x_n\}$, $Z = \{z_0, z_1, \ldots, z_m\}$ und $Q = \{z_{k+1}, z_{k+2}, \ldots, z_m\}$. Wir setzen $x_0 = *$. Für $0 \leq i \leq m$ und $0 \leq j \leq n$ setzen wir ferner $\delta_{ij} = (z_i, x_j, z_{ij}, x_{ij}, r_{ij})$, falls $\delta(z_i, x_j) = (z_{ij}, x_{ij}, r_{ij})$ gilt. Damit lässt sich M durch

$$x_1, x_2, \ldots, x_n, z_0, z_1, \ldots, z_k, \delta_{00}, \delta_{01}, \ldots, \delta_{0n}, \delta_{10}, \delta_{11}, \ldots, \delta_{1n}, \ldots, \delta_{kn}$$

beschreiben. Um aus dieser Beschreibung ein Wort zu erhalten, betrachten wir die Kodierung, die durch folgende eineindeutige Zuordnung gegeben ist:

$$\begin{aligned}
&x_j \to 01^{j+1}0 \quad \text{für } 0 \leq j \leq n,\\
&z_i \to 01^{i+1}0^2 \quad \text{für } 0 \leq i \leq k,\\
&R \to 010^3, \quad L \to 01^20^3, \quad N \to 01^30^3,\\
&(\to 010^4, \quad) \to 01^20^4,\\
&, \to 010^5.
\end{aligned}$$

Man beachte, dass durch die letzten drei Zuordnungen den zur Beschreibung eines Quintupels δ_{ij} notwendigen Zeichen „Klammer auf“, „Klammer zu“ und „Komma“ jeweils ein Wort zugeordnet wird. Durch diese Kodierung wird M durch ein Wort über $\{0, 1\}$ beschrieben.

Wir illustrieren diese Konstruktion anhand der TURING-Maschine aus Beispiel 1.19 b). Zuerst erhalten wir (mit $x_1 = a, x_2 = b, z_2 = q$) die Beschreibung

$$\begin{aligned}
&a, b, z_0, z_1, (z_0, *, z_0, *, N), (z_0, a, z_1, a, R), (z_0, b, z_1, b, R), (z_1, *, q, *, N),\\
&(z_1, a, z_0, a, R), (z_1, b, z_0, b, R)
\end{aligned}$$

und nach der Kodierung mittels

$$\begin{aligned}
&* \to 010, a \to 01^20, b \to 01^30, z_0 \to 010^2, z_1 \to 01^20^2, q \to 01^30^2,\\
&R \to 010^3, L \to 01^20^3, N \to 01^30^3, (\to 010^4,) \to 01^20^4, , \to 010^5
\end{aligned}$$

die Beschreibung durch das Wort

$$\begin{aligned}
&01^20010^501^30010^5010^2010^501^20^2010^5010^4010^2010^5010010^5010^2010^5\\
&010010^501^30^301^20^4010^5010^4010^2010^501^20010^501^20^2010^501^20010^5\\
&010^301^20^4010^5010^4010^2010^501^30010^501^20^2010^501^30010^5010^301^20^4010^5\\
&010^401^20^2010^5010010^501^30^2010^5010010^501^30^301^20^4010^5010^401^20^2010^5\\
&01^20010^5010^2010^501^20010^5010^301^20^4010^5010^401^20^2010^501^30010^5\\
&010^2010^501^30010^5010^301^20^4.
\end{aligned}$$

Mit S bezeichnen wir die Menge aller TURING-Maschinen $M = (X, Z, z_0, Q, \delta)$ mit $X = \{0, 1\}$, $Z = \{z_0, z_1, \ldots, z_m\}$ und $Q = \{z_m\}$ für ein $m \geq 1$ (wegen Lemma 1.21 können wir ohne Beschränkung der Allgemeinheit annehmen, dass Q einelementig ist). Für eine TURING-Maschine $M \in \mathcal{S}$ sei w_M das Wort, das M nach obiger Kodierung beschreibt. M bestimmt w_M eindeutig, und ist umgekehrt $w \in \{0, 1\}^*$ die Beschreibung einer TURING-Maschine aus S, so ist die TURING-Maschine $M \in \mathcal{S}$ mit $w = w_M$ eindeutig bestimmt. Ferner ist die Kodierung w_M von $M \in \mathcal{S}$ eine mögliche Eingabe für die TURING-Maschine M.

Hilfssatz 1. Das Problem

Gegeben: $w \in \{0, 1\}^*$
Frage: Ist w Kodierung einer TURING-Maschine aus S ?

ist entscheidbar.

Wir geben nur die Idee des Beweises, die Realisierung der Idee durch eine formale TURING-Maschine ist aufwendig und bleibt dem Leser überlasssen.
Um festzustellen, ob das Eingabealphabet der TURING-Maschine $\{0, 1\}$ ist und Z mindestens zwei Zustände enthält, ist nur zu testen, ob w mit $010010^5 01^2 0010^5 010^2 010^5 01^2 0^2$ beginnt. Dann wird getestet, ob nach diesem Anfang (von der Kodierung der Kommas abgesehen) Kodierungen von Zuständen und Quadrupeln δ_{ij} folgen und ob die in den Quadrupeln auftauchenden Eingabesymbole und Zustände auch in X bzw. Z vorhanden sind. Abschließend wird getestet, ob für jedes Paar $(z_i, x_j), z_i \in Z \setminus Q, x_j \in X \cup \{*\}$, ein Quintupel δ_{ij} existiert.
Wir betrachten nun die Funktion $f : \{0, 1\}^* \to \{0, 1\}$, die durch

$$f(w) = \begin{cases} 0 & w = w_M \text{ für ein } M \in \mathcal{S},\ f_M(w_M) \text{ ist nicht definiert} \\ \textit{nicht definiert} & \text{sonst} \end{cases}$$

gegeben ist.

Hilfssatz 2. f ist nicht TURING*-berechenbar.*

Beweis: Wir führen den Beweis indirekt, d.h. wir nehmen an, dass f TURING-berechenbar ist und leiten einen Widerspruch her.
Wenn f TURING-berechenbar ist, so gibt es nach Definition eine TURING-Maschine N mit $f_N = f$. Da offensichtlich $N \in \mathcal{S}$ gilt, existiert eine Kodierung w_N von N.
Wenn für die Kodierung w_N von N der Wert $f(w_N)$ definiert ist, so folgt aus der Definition von f, dass $f_N(w_N)$ nicht definiert ist. Dies widerspricht aber $f = f_N$.
Ist dagegen $f(w_N)$ nicht definiert, so besagt die Definition von f gerade, dass $f_N(w_N)$ definiert sein muss. Damit erhalten wir erneut einen Widerspruch zu $f = f_N$.

Da es nach Hilfssatz 1 entscheidbar ist, ob ein Wort $w \in \{0, 1\}^*$ eine Kodierung einer TURING-Maschine ist, kann es nicht entscheidbar sein, ob $f_M(w_M)$ definiert ist oder nicht, da sonst die Funktion aus Hilfssatz 2 TURING-berechenbar wäre. Damit ist die Behauptung von Satz 1.28 bewiesen (es ist sogar noch mehr gezeigt worden, da nicht beliebige Wörter x sondern nur Kodierungen von TURING-Maschinen betrachtet wurden). □

Satz 1.29 *Das Problem*

Gegeben: **LOOP/WHILE**-*Programm* Π, $n \in \mathbf{N}$
Frage: *Ist* $\Phi_{\Pi,1}(n)$ *definiert?*

ist unentscheidbar.

Beweis: Zu jeder TURING-Maschine M können wir entsprechend den Beweisen von Satz 1.24 und Satz 4.6 ein **LOOP/WHILE**-Programm Π mit $\psi(f_M(w)) = \Phi_{\Pi,1}(\psi(w))$ konstruieren. Die Funktion ψ aus dem Beweis von Satz 1.24 und ihre Umkehrung sind TURING-berechenbar. Wäre nun das Problem aus Satz 1.29 entscheidbar, so wäre auch entscheidbar, ob $\psi(f_M(w))$ und damit $f_M(w)$ definiert ist oder nicht. Dies widerspricht aber Satz 1.28. □

Wir merken an, dass die Aussage von Satz 1.29 wie folgt gedeutet werden kann: Sind in einer Programmiersprache Konstrukte vorhanden, die der **LOOP**- bzw. **WHILE**-Anweisung entsprechen, so kann für ein beliebiges Programm nicht entschieden werden, ob es bei einer beliebigen Eingabe ein Resultat liefert. Um dieser katastrophalen Situation zu entgehen, werden bei der Definition von Programmiersprachen zusätzlichen Bedingungen eingebaut, die eine uneingeschränkte Anwendung der Konstrukte nicht zulassen.

Definition 1.30 *i) Zwei* TURING*-Maschinen M_1 und M_2 heißen äquivalent, wenn $f_{M_1} = f_{M_2}$ gilt.*
ii) Zwei **LOOP/WHILE***-Programme Π_1 und Π_2 heißen äquivalent, wenn $\Phi_{\Pi_1,1} = \Phi_{\Pi_2,1}$ gilt.*

Es ist leicht zu sehen, dass diese beiden Äquivalenzen die Eigenschaften einer Äquivalenzrelation erfüllen.

Sei M eine Menge, in der eine Äquivalenz erklärt ist. Das *Äquivalenzproblem* für Elemente aus M ist durch

Gegeben:	zwei Elemente A_1 und A_2 aus M
Frage:	Sind A_1 und A_2 äquivalent?

gegeben.

Satz 1.31 *Das Äquivalenzproblem für* TURING*-Maschinen bzw.* **LOOP/WHILE***-Programme ist unentscheidbar.*

Beweis: Wir geben den Beweis nur für TURING-Maschinen. Die Übertragung auf den Fall der **LOOP/WHILE**-Programme erfolgt analog zum Beweis von Satz 1.29.
Seien eine TURING-Maschine M und ein Wort w über dem Eingabealphabet von M gegeben. Wir konstruieren zunächst in Abhängigkeit von M und w die TURING-Maschinen M_1 und N, deren induzierte Funktionen f_{M_1} und f_N durch

$$f_{M_1}(v) = \begin{cases} 1 & v = w \\ nicht\ definiert & \text{sonst} \end{cases}$$

und

$$f_N(v) = \begin{cases} v & f_M(v) \text{ ist definiert} \\ nicht\ definiert & \text{sonst} \end{cases}$$

gegeben sind. (f_{M_1} ist nach Übungsaufgabe 5 **LOOPWHILE**-berechenbar, und wegen Satz 1.25 gibt es daher eine solche TURING-Maschine M_1; N ergibt sich aus M durch folgende Modifikation: zuerst kopiert N das Eingabewort v auf das Band, arbeitet auf v

wie M, und falls ein Endzustand erreicht wird, wird das erhaltene Resultat $f_M(v)$ gelöscht, womit auf dem Band nur noch die Kopie von v steht.)
Ferner können wir nun eine TURING-Maschine M_2 konstruieren, die die Komposition von f_N und f_{M_1} ist (siehe Lemma 1.22). Offenbar gilt

$$f_{M_2}(v) = f_{M_1}(f_N(v)) = \begin{cases} 1 & v = w \text{ und } f_M(w) \text{ ist definiert} \\ \textit{nicht definiert} & \text{sonst} \end{cases} .$$

Damit gilt $f_{M_1} = f_{M_2}$ genau dann, wenn $f_M(w)$ definiert ist. Wenn die Äquivalenz von M_1 und M_2 entscheidbar wäre, so wäre auch entscheidbar, ob $f_M(w)$ definiert ist. Wegen Satz 1.28 ist daher das Äquivalenzproblem für TURING-Maschinen unentscheidbar. □

Auch hier ist wieder festzustellen, dass eine Interpretation von Satz 1.31 dahingehend möglich ist, dass es unentscheidbar ist, ob zwei Programme die gleichen Abbildung der Eingaben in Ausgaben realisieren.

Bei den Beweisen von Satz 1.29 und 1.31 wurde die gleiche Methode benutzt. Es erfolgte eine Reduktion des zu betrachtenden Problems auf ein Problem, dessen Unentscheidbarkeit bereits gezeigt wurde, in der Weise, dass aus der Entscheidbarkeit des betrachteten Problems auch die des unentscheidbaren Problems folgen würde. Diese Methode ist die am meisten benutzte, um Unentscheidbarkeiten zu zeigen.

Im Folgenden wollen wir zuerst zwei weitere Probleme angeben, die unentscheidbar sind und in natürlicher Weise entstehen. Hierbei werden wir auif die Beweise der Unentscheidbarkeit aus Platzgründen verzichten. Unter Verwendung des zweiten dieser Probleme zeigen wir dann die Unentscheidbarkeit von Problemen der Prädikatenlogik.

Im Jahre 1900 hielt der deutsche Mathematiker DAVID HILBERT (1862–1943) auf dem Internationalen Mathematikerkongress einen Hauptvortrag, in dem er 23 Probleme vorstellte, die nach seiner Meinung von besonders großer Bedeutung für die Mathematik waren. Das 10. Problem lautet:

Gegeben:	eine natürliche Zahl $n \geq 1$, eine endliche Menge $F \subset \mathbf{N}_0^n$, $c_{i_1,i_2,\ldots,i_n} \in \mathbf{Z}$, ein Polynom $p(x_1, x_2, \ldots, x_n) = \sum_{(i_1,i_2,\ldots,i_n)\in F} c_{i_1 i_2 \ldots i_n} x_1^{i_1} x_2^{i_2} \ldots x_n^{i_n}$
Frage:	Gibt es eine Lösung von $p(x_1, x_2, \ldots, x_n) = 0$ in $\mathbf{Z}^n$?

Zum Beispiel hat

$$p(x, y, z) = 3xyz^2 + 5xy^2 - 4x^2yz = 0$$

die Lösung $x = 2, y = 1, z = 1$, während

$$p(x, y, z) = 2x^4y^2 + 3x^2z^2 + 2y^2z^6 - 1 = 0$$

keine ganzzahlige Lösung besitzt (da geradezahlige Potenzen von ganzen Zahlen stets nichtnegative ganze Zahlen und somit die ersten drei Summanden 0 oder ≥ 2 sind).
Genauer gesagt, fragte HILBERT nach einem Algorithmus zur Lösung des eben genannten Problems. In unserer Terminologie stellte er die Frage nach der Entscheidbarkeit des Problems. Die Lösung des Problems wurde nach Vorarbeiten von ROBINSON im Jahre 1960 vom JU.V.MATIJASEVIC gegeben.

Satz 1.32 *Das 10.* HILBERT*sche Problem ist unentscheidbar.* □

Entsprechend diesem Ergebnis gibt es keinen Algorithmus, der für alle Polynome die richtige Antwort gibt. Auf der anderen Seite gibt es natürlich Teilmengen, die nur spezielle Polynome enthalten, für die es dann Algorithmen gibt. Wir erwähnen hier zwei solche Fälle.

- Wir beschränken uns auf die Menge der linearen Polynome, d.h. die Polynome sind von der Form

  $$p(x_1, x_2, \ldots, x_n) = a_0 + a_1x_1 + a_2x_2 + \ldots + a_nx_n.$$

  Dann gibt es genau dann eine ganzzahlige Lösung, wenn der größte gemeinsame Teiler d der Koeffizienten $a_1, a_2, \ldots, a_n$ ein Teiler von a_0 ist. (Sei zuerst $(b_1, b_2, \ldots, b_n) \in \mathbf{Z}^n$ eine Lösung. Dann ist d ein Teiler von $a_1b_1 + a_2b_2 + \ldots + a_nb_n$. Wegen $-a_0 = a_1b_1 + a_2b_2 + \ldots + a_nb_n$ ist d damit ein Teiler von a_0. Umgekehrt gibt es für den größten gemeinsamen Teiler d von $a_1, a_2, \ldots, a_n$ eine Darstellung der Form $d = a_1c_1 + a_2c_2 + \ldots + a_nc_n$ mit gewissen ganzen Zahlen $c_1, c_2, \ldots, c_n$. Falls $a_0 = kd$, dann ist $(kc_1, kc_2, \ldots, kc_n)$ eine Lösung.) Da der größte gemeinsame Teiler nach dem EUKLIDischen Algorithmus bestimmt werden kann und es entscheidbar ist, ob eine ganze Zahl Teiler einer anderen ganzen Zahl ist, ist es auch entscheidbar, ob ein Polynom der obigen speziellen Art eine Nulstelle in $\mathbf{Z}^n$ hat.

- Wir betrachten nur Polynome in einer Variablen, d.h. Polynome der Form

  $$p(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n.$$

  Wenn $-a_0 = a_1x + a_2x^2 + \ldots + a_nx^n$ gelten soll, muss offenbar x ein Teiler von a_0 sein. Da es nur endlich viele Teiler von a_0 gibt, lässt sich mittels Durchtesten aller dieser Teiler feststellen, ob einer von ihnen Nullstelle von p ist. Dies liefert offensichtlich einen Algorithmus zur Beantwortung, ob p eine ganzzahlige Nullstelle hat.

Wir betrachten nun das POST*sche Korrespondenzproblem*:

Gegeben:	Alphabet X mit mindestens zwei Buchstaben, $n \geq 1$, Menge $\{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$ von Paaren mit $u_i, v_i \in X^+$ für $1 \leq i \leq n$
Frage:	Gibt es eine Folge $i_1i_2 \ldots i_m$ mit $1 \leq i_j \leq n$ für $1 \leq j \leq m$ derart, dass $u_{i_1}u_{i_2} \ldots u_{i_m} = v_{i_1}v_{i_2} \ldots v_{i_m}$ gilt?

Beispiel 1.33 a) Seien $n = 3$, $X = \{a, b, c\}$ und die Menge $\{(aa, a), (bc, ab), (c, cca)\}$ von Paaren gegeben. Dann ist $i_1i_2i_3i_4 = 1231$ eine Folge der gesuchten Art, denn es gilt

$$u_1u_2u_3u_1 = aa \cdot bc \cdot c \cdot aa = a \cdot ab \cdot cca \cdot a = v_1v_2v_3v_1.$$

b) Für $n = 2$, $X = \{a, b\}$ und die Menge $\{(aab, aa), (ab, ba)\}$ von Paaren gibt es dagegen keine derartige Folge. Dies ist wie folgt zu sehen: Wegen $|u_1| > |v_1|$ und $|u_2| = |v_2|$ kann die Folge keine 1 enthalten, und die Folge kann nicht nur aus dem Symbol 2 bestehen, da u_2 mit a und v_2 mit b anfängt.

$$\begin{array}{ccccc} & \phi_1 & & \phi_2 & \\ u_1 & \leftarrow & 1 & \rightarrow & v_1 \\ u_2 & \leftarrow & 2 & \rightarrow & v_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ u_n & \leftarrow & n & \rightarrow & v_n \end{array}$$

Abbildung 1.3:

Zur Motivation des POSTschen Korrespondenzproblems betrachten wir zwei Kodierungen ϕ_1 und ϕ_2 der Menge $\{1, 2, \ldots, n\}$, die in der Abbildung 1.3 gegeben sind.
Dann gelten

$$\phi_1(i_1 i_2 \ldots i_m) = u_{i_1} u_{i_2} \ldots u_{i_m} \quad \text{und} \quad \phi_2(i_1 i_2 \ldots i_m) = v_{i_1} v_{i_2} \ldots v_{i_m}.$$

Folglich ist die Frage des POSTschen Korrespondenzproblems damit gleichwertig, zu fragen, ob eine Folge von Elementen aus $\{1, 2, \ldots, n\}$ existiert, die bei beiden Kodierungen ϕ_1 und ϕ_2 auf das gleiche Wort abgebildet wird.

Satz 1.34 *Das* POST*sche Korrespondenzproblem ist unentscheidbar.* □

Für die Diskussion von Spezialfällen verweisen wir auf die Übungsaufgaben 15 und 17.
Wir werden nun die Unentscheidbarkeit zweier Probleme der Prädikatenlogik durch Reduktion auf das Postsche Korrespondenzproblem zeigen.[2]

Satz 1.35 *i) Es ist unentscheidbar, ob ein prädikatenlogischer Ausdruck eine Tautologie ist.*
ii) Es ist unentscheidbar, ob ein prädikatenlogischer Ausdruck erfüllbar ist.

Beweis. i) Wir geben eine Reduktion auf das POSTsche Korrespondenzproblem an. Sei dazu eine Menge $V = \{(u_1, v_1), (u_2, v_2), \ldots, u_n, v_n)\}$ von Paaren nichtleerer Wörter über dem Alphabet $\{0, 1\}$ gegeben. Dieser ordnen wir nun eine Signatur $\mathcal{S}_V$ und einen Ausdruck A_V so zu, dass A_V genau dann eine Tautologie ist, wenn das zu V gehörende POSTsche Korrespondenzproblem eine Lösung hat.
Wir definieren zuerst die Signatur $\mathcal{S}_V$ durch

$$\begin{aligned} &F_1 = \{f_0, f_1\},\ F_i = \emptyset \text{ für } i \geq 2, \\ &R_2 = \{r\},\ R_j = \emptyset \text{ für } j = 1 \text{ und } j \geq 3, \\ &K = \{a\}\,. \end{aligned}$$

die Funktion f_w für ein nichtleeres Wort $w = i_1 i_2 \ldots i_m$, $i_k \in \{0, 1\}$ für $1 \leq k \leq m$ durch

$$f_w(x) = f_{i_1}(f_{i_2}(\ldots f_{i_m}(x)\ldots))\,.$$

Offenbar gilt dann $f_{wi}(x) = f_w(f_i(x))$ für $w \in \{0, 1\}^+$ und $i \in \{0, 1\}$ und damit auch $f_{w_1 w_2}(x) = f_{w_1}(f_{w_2}(x))$ für $w_1, w_2 \in \{0, 1\}^*$.

[2]Wir nehmen dabei an, dass der Leser mit den Grundbegriffen der Prädikatenlogik vertraut ist. Wir verwenden hier die Terminologie von J. DASSOW, Logik für Informatiker, Teubner-Verlag, 2005.

Weiterhin setzen wir

$$\begin{aligned} A_1 &= (r(f_{u_1}(a), f_{v_1}(a)) \wedge r(f_{u_2}(a), f_{v_2}(a)) \wedge \ldots \wedge r(f_{u_n}(a), f_{v_n}(a)))\,, \\ A_2 &= \forall x \forall y (r(x,y) \rightarrow (r(f_{u_1}(x), f_{v_1}(y)) \wedge r(f_{u_2}(x), f_{v_2}(y)) \wedge \ldots \wedge r(f_{u_n}(x), f_{v_n}(y))))\,, \\ A_3 &= \exists z\; r(z,z)\,, \\ A_V &= ((A_1 \wedge A_2) \rightarrow A_3)\,. \end{aligned}$$

Wir nehmen nun zuerst an, dass A_V eine Tautologie ist. Sei $I = (U, \tau)$ die Interpretation mit

- $U = \{0,1\}^*$,
- $\tau(a) = \lambda$,
- für $i \in \{0,1\}$ ist die Funktion $\tau(f_i)$ durch $\tau(f_i)(w) = iw$ definiert,
- $\tau(r)$ ist die Menge aller Paare $(w, w') \in (\{0,1\}^*)^2$, für die es eine Folge $i_1 i_2 \ldots i_r$ mit $i_j \in \{1, 2, \ldots, n\}$ für $1 \leq j \leq r$, $w = u_{i_1} u_{i_2} \ldots u_{i_r}$ und $w' = v_{i_1} v_{i_2} \ldots v_{i_r}$ gibt (d.h. dass w und w' werden also durch Konkatenation der ersten bzw. zweiten Komponente von Elementen aus V gebildet).

Man sieht nun sofort, dass $\tau(f_w)(w') = ww'$ gilt. Damit gilt für jede Belegung α die Beziehung $w^I_\alpha(A_1) = 1$, da $(\tau(f_{u_k})(\lambda), \tau(f_{v_k})(\lambda)) = (u_k, v_k) \in \tau(r)$ für $1 \leq k \leq n$ gültig ist. Gilt nun $(w, w') \in \tau(r)$, also $w = u_{i_1} u_{i_2} \ldots u_{i_r}$ und $w' = v_{i_1} v_{i_2} \ldots v_{i_r}$, so ergibt sich

$$(\tau(f_{u_k})(w), \tau(f_{v_k})(w')) = (u_k w, v_k w') = (u_k u_{i_1} u_{i_2} \ldots u_{i_r}, v_k v_{i_1} v_{i_2} \ldots v_{i_r}) \in \tau(r)$$

für $1 \leq k \leq n$ und damit auch $w^I_\alpha(A_2) = 1$. Außerdem gilt $w^I_\alpha(A_3) = 1$ genau dann, wenn es eine Lösung des POSTschen Korrespondenzproblems bez. V gibt.
Da A_V eine Tautologie ist, folglich $w^I_\alpha(A_V) = 1$ ist, ergibt sich auch $w^I_\alpha(A_3) = 1$. Somit hat das POSTsche Korrespondenzproblem bez. V eine Lösung.

Habe jetzt umgekehrt das POSTsche Korrespondenzproblem bez. V eine Lösung $j_1 j_2 \ldots j_s$. Wir setzen

$$u = u_{j_1} u_{j_2} \ldots u_{j_s} = v_{j_1} v_{j_2} \ldots v_{j_s}\,.$$

Ferner sei $J = (U', \tau')$ eine beliebige Interpretation von $\mathcal{S}_V$ und α eine beliebige Belegung bez. J. Dann definieren wir die Abbildung $\mu : \{0,1\}^* \rightarrow U'$ induktiv durch

$$\begin{aligned} \mu(\lambda) &= \tau'(a), \\ \mu(w0) &= \tau'(f_0)(\mu(w)), \\ \mu(w1) &= \tau'(f_1)(\mu(w)). \end{aligned}$$

Damit ergibt sich durch einen Induktionsbeweis

$$\mu(x) = \tau'(f_x)(\tau'(a))\,.$$

Falls $w^J_\alpha(A_1) = 0$ oder $w^J_\alpha(A_2) = 0$ gelten, so ist $w^J_\alpha(A_V) = 1$. Sei daher $w^J_\alpha(A_1) = 1$ und $w^J_\alpha(A_2) = 1$. Für $1 \leq k \leq n$ folgt aus ersterem

$$(\tau'(f_{u_k})(\tau'(a)), \tau'(f_{v_k}(\tau'(a))) = (\mu(u_k), \mu(v_k)) \in \tau'(r)\,,$$

und aus letzterem folgt, dass $(\mu(w), \mu(w') \in \tau'(r)$ die Beziehung $\mu(wx_k), \mu(w'v_k) \in \tau'(r)$ impliziert. Hieraus erhalten wir durch Induktion

$$(\mu(u_{i_1}u_{i_2}\dots u_{i_t}), \mu(v_{i_1}v_{i_2}\dots v_{i_t})) \in \tau'(r)$$

für beliebige $t \geq 1$, $i_l \in \{1, 2, \dots, n\}$, $1 \leq l \leq t$. Insbesondere erhalten wir

$$(\mu(u), \mu(u)) = (\mu(u_{j_1}u_{j_2}\dots u_{j_s}), \mu(v_{j_1}v_{j_2}\dots v_{j_s})) \in \tau'(r)\,.$$

Dies bedeutet aber $w_\alpha^J(A_3) = 1$ und somit $w_\alpha^J(A_V) = 1$.
Folglich ist A_V eine Tautologie.

ii) Offenbar ist A genau dann erfüllbar, wenn $\neg A$ keine Tautologie ist. Die Entscheidbarkeit der Erfüllbarkeit von A würde daher die Entscheidbarkeit der Frage, ob $\neg A$ eine Tautologie ist, nach sich ziehen. Dies führt zu einem Widerspruch zu i). □

Bei der Behandlung von Entscheidbarkeitsfragen für formale Grammatiken und Sprachen werden wir weitere Anwendungen des POSTschen Korrespondenzproblems behandeln.

Übungsaufgaben

1. Konstruieren Sie **LOOP**/**WHILE**-Programme für folgende Funktionen:
   a) $f(x_1) = 2^{x_1}$,
   b) $f(x_1) = x_1^a$, wobei a eine feste natürliche Zahl ist.

2. Geben Sie **LOOP**/**WHILE**-Programme für folgende Konstrukte aus Programmiersprachen an:
   a) **IF** $x_2 > 2$ **THEN** $x_1 := x_1 + x_2$ **ELSE** $x_1 := 0$,
   b) **FOR** $i = 10$ **TO** 20 **DO** $x_1 := i * x_1$.

3. Welche Funktionen werden durch die nachfolgenden Programme berechnet?
   a) $x_2 := P(x_2);\ x_2 := P(x_2);\ x_2 := P(x_2);$
   WHILE $x_2 \neq 0$ **BEGIN**
   LOOP x_1 **BEGIN** $x_3 := S(x_3)$ **END**;
   $x_2 := P(x_2)$
   END;
   $x_1 := x_3$
   b) $x_2 := x_1;$
   LOOP x_1 **BEGIN** $x_3 := P(x_3)$ **END**;
   WHILE $x_3 \neq 0$ **BEGIN** $x_1 := x_3; x_3 := 0$ **END**

4. Berechnen Sie, welchen Wert die Variable x_1 nach Abarbeitung des folgenden Programms bei gegebener Eingabe x_1 annimmt.
   $x_2 := S(x_1);\ x_3 := 0;\ x_4 := x_1;$
   WHILE $x_1 \neq 0$ **BEGIN**
   $x_1 := P(x_1); x_1 := P(x_1); x_2 := P(x_2); x_2 := P(x_2)$
   END;

WHILE $x_2 \neq 0$ **BEGIN** $x_3 := S(x_3); x_2 := P(x_2)$ **END**;
WHILE $x_3 \neq 0$ **BEGIN**
LOOP x_4 **BEGIN** $x_4 := S(x_4)$ **END**;
$x_3 := P(x_3)$
END;
$x_1 := x_4$

5. Beweisen Sie folgende Aussagen:
a) Eine totale Funktion, die nur an endlich vielen Stellen einen von 0 verschiedenen Wert annimmt, ist **LOOP**/**WHILE**-berechenbar.
b) Seien N eine endliche Menge von $\mathbf{N}_0$ und $f : N \to N'$ eine totale Funktion. Dann sind die Funktionen f' und f'' mit

$$f'(x) = \begin{cases} 0 & x \in N \\ 1 & \text{sonst} \end{cases}$$

und

$$f''(x) = \begin{cases} f(x) & x \in N \\ nicht\ definiert & \text{sonst} \end{cases}$$

LOOP/**WHILE**-berechenbar.

6. Durch die beiden folgenden Tabellen sei jeweils eine TURING-Maschine beschrieben:

a)

	z_0	z_1
$*$	$(z_0, *, N)$	$(q, *, N)$
a	(z_1, a, R)	(z_0, a, R)
b	(z_1, b, R)	(z_0, b, R)

b)

	z_0	z_a^1	z_b^1	z_a^2	z_b^2	z_1	z_2	z_3
$*$	$(q, *, N)$	$(z_a^2, *, R)$	$(z_b^2, *, R)$	(z_1, a, L)	(z_1, b, L)	$(z_2, *, L)$	$(z_3, *, R)$	$(q, *, N)$
a	$(z_a^1, *, R)$	(z_a^1, a, R)	(z_b^1, a, R)	(z_a^2, a, R)	(z_b^2, a, R)	(z_1, a, L)	(z_2, a, L)	$(z_0, *, R)$
b	$(z_b^1, *, R)$	(z_a^1, b, R)	(z_b^1, b, R)	(z_a^2, b, R)	(z_b^2, b, R)	(z_1, b, L)	(z_2, b, L)	$(z_0, *, R)$

Berechnen Sie die von diesen TURING-Maschinen indizierten Funktionen $\{a, b\}^* \to \{a, b\}^*$.

7. Es sei

$$M = (\{a, b\}, \{z_0, z_1, z_2, z_3, q\}, z_0, \{q\}, \delta)$$

eine TURING-Maschine, bei der die Funktion δ durch folgende Tabelle gegeben ist:

	z_0	z_1	z_2	z_3
$*$	$(z_2, *, L)$	$(q, *, N)$	$(q, *, N)$	$(q, *, N)$
a	(z_1, a, R)	(z_0, a, R)	(z_3, a, L)	(z_2, b, L)
b	(z_1, a, R)	(z_0, b, R)	(z_3, b, L)	(z_2, b, L)

i) Bestimmen Sie $f_M(abaabb)$.
ii) Bestimmen Sie die induzierte Funktion $f_M : \{a, b\}^* \longrightarrow \{a, b\}^*$.

8. Geben Sie eine TURING-Maschine M an, deren induzierte Funktion
   a) die Funktion P ist,
   b) die Funktion $f_M : \{a, b\}^* \to \{a, b\}^*$ mit

$$f_M(x_1x_2 \dots x_n) = x_1x_1x_2x_2 \dots x_nx_n = x_1^2x_2^2 \dots x_n^2$$

   ist.

9. Beweisen Sie, daß es zu jeder TURING-Maschine M eine TURING-Maschine M' mit

$$f_{M'}(x) = \begin{cases} 1 & f_M(x) \text{ ist definiert} \\ \textit{nicht definiert} & \text{sonst} \end{cases}$$

   gibt.

10. Geben Sie eine TURING-Maschine an, die 1 bei einem Palindrom und sonst 0 ausgibt.

11. Mit *div* bzw. *mod* seien die ganzzahlige Division bzw. der dabei auftretende Rest bezeichnet. Ferner sei die Funktion $\ominus : \mathbf{N}^2 \to \mathbf{N}$ durch

$$x \ominus y = \begin{cases} x - y & \text{für } x \geq y \\ 0 & \text{sonst} \end{cases}$$

    gegeben. Beweisen Sie jeweils mittels der Definitionen (d.h. ohne Benutzung von Aussagen mittels derer eine Berechenbarkeit in eine andere überführt wird), dass diese drei Funktionen
    a) **LOOP**-berechenbar,
    b) TURING-berechenbar
    sind.

12. Eine Menge $M \subseteq X^*$ heißt genau dann *rekursiv-aufzählbar*, wenn es eine TURING-berechenbare Funktion $\mathbf{N} \to X^*$ gibt, deren Wertebereich M ist. (Anstelle der TURING-Berechenbarkeit können wir auch einen anderen der gleichwertigen Berechenbarkeitsbegriffe zugrundelegen, wobei dann aber statt einer Menge von Wörtern eine Menge natürlicher Zahlen zu nehmen ist.)
    Beweisen Sie, dass die Menge aller Wörter über dem Alphabet $\{a, b\}$, die genau zwei Vorkommen des Buchstaben a enthalten, und die Menge der Primzahlen rekursiv-aufzählbar sind.

13. Beweisen Sie, dass eine Menge M genau dann rekursiv-aufzählbar (siehe Übungsaufgabe 12) ist, wenn M Definitionsbereich einer TURING-berechenbaren Funktion ist.

14. Beweisen Sie, dass eine Menge M genau dann entscheidbar ist, wenn M und $X^* \setminus M$ rekursiv-aufzählbar (siehe Übungsaufgaben 12 und 13) sind.

15. Beweisen Sie, dass das Problem

Gegeben:	Alphabet X, $n \geq 1$, $m \geq 1$, $\{(u_1, v_1), (u_2, v_2), \dots, (u_n, v_n)\}$, $u_i, v_i \in X^+$ und $\lvert u_i \rvert = \lvert v_i \rvert = m$ für $1 \leq i \leq n$
Frage:	Gibt es eine Folge $i_1i_2 \dots i_k$ mit $u_{i_1}u_{i_2} \dots u_{i_k} = v_{i_1}v_{i_2} \dots v_{i_k}$

entscheidbar ist.

16. Beweisen Sie, dass das POSTsche Korrespondenzproblem für einelementige Alphabete X entscheidbar ist.

17. Untersuchen Sie, ob das 10. Hilbertsche Problem für folgende Fälle eine Lösung besitzt:
    a) $x^3 - 3x^2 - 6x + 18 = 0$,
    b) $2x^3y + 4xz^2 - 2y + 1 = 0$,
    c) $x^4 - 2x^2y^2 + 2y^4 - 3 = 0$.

Kapitel 2

Formale Sprachen und Automaten

2.1 Die Sprachfamilien der Chomsky-Hierarchie

2.1.1 Definition der Sprachfamilien

Im Kapitel 1 haben wir mehrere gleichwertige Definitionen für Algorithmen behandelt. Als Grundlage dienten dabei einmal eine spezielle einfache Programmiersprache, die **LOOP**/ **WHILE**-Programme erzeugt, und ein anderes Mal ein spezieller Typ von Maschinen, die TURING-Maschinen. In diesem Kapitel werden wir uns direkt dem Studium von formalen Sprachen bzw. Automaten als Abstraktionen von Programmier- und natürlichen Sprachen bzw. von Computern und Rechenmaschinen zuwenden.
Wir beginnen dabei mit der Definition eines allgemeinen Typs von formalen Grammatiken und Sprachen und geben dann einige wichtige und interessante Spezialfälle an.

Jede natürliche Sprache basiert auf einer Grammatik, in der die Regeln zusammengestellt sind, nach denen sich syntaktisch richtige Sätze der Sprache bilden lassen. Eine ähnliche Rolle spielen die Handbücher für Programmiersprachen; auch sie enthalten verschiedene Anweisungen und Kommandos, durch deren Anwendung korrekte Programme erzeugt werden.

Die Syntax einer natürlichen Sprachen gibt an, wie ein Satz bzw. Teile eines Satzes aus grammatischen Einheiten aufgebaut werden kann. Wir erwähnen hier beispielhaft die folgenden Konstruktionen.

(Satz) → (Substantivphrase)(Verbphrase)
(Satz) → (Substantivphrase)(Verbphrase)(Objektphrase)
(Substantivphrase) → (Artikel)(Substantiv)
(Verbphrase) → (Verb)(Adverb)

Das erste Konstrukt besagt, dass ein Satz aus einem Substantiv und einem Verb bestehen kann, das zweite entspricht dem vom Englischunterricht her bekannten Aufbau eines Satzes aus Subjekt, Prädikat und Objekt (man sieht, dass für einen Satz verschiedene Zerlegungen in grammatikalische Teile möglich sind). Die beiden letzten Vorschriften sagen, wie eine Substantivphrase bzw. eine Verbphrase weiter zergliedert bzw. wie diese aufgebaut werden können. Weiterhin gibt es eine Zuordnung der Wörter der deutschen Sprache zu Wortarten. Dies kann durch die folgenden Konstruktionen beschrieben werden.

(Substantiv) → Hund
(Substantiv) → Banane
(Artikel) → der
(Artikel) → ein
(Verb) → geht
(Verb) → singt
(Adverb) → langsam

Durch Nacheinanderanwendung der obigen Vorschriften können u. a.

$$\begin{aligned}(\text{Satz}) &\Longrightarrow (\text{Substantivphrase})(\text{Verbphrase})\\ &\Longrightarrow (\text{Substantivphrase})(\text{Verb})(\text{Adverb})\\ &\Longrightarrow (\text{Substantivphrase})\ \text{geht}\ (\text{Adverb})\\ &\Longrightarrow (\text{Substantivphrase})\ \text{geht langsam}\\ &\Longrightarrow (\text{Artikel})(\text{Substantiv})\ \text{geht langsam}\\ &\Longrightarrow \text{der}\ (\text{Substantiv})\ \text{geht langsam}\\ &\Longrightarrow \text{der Hund geht langsam}\end{aligned}$$

und in analoger Weise kann auch

$$(\text{Satz}) \Longrightarrow \ldots \Longrightarrow \text{ein Banane singt langsam}$$

hergeleitet werden. Wir machen darauf aufmerksam, dass der letzte Satz zwar inhaltlich falsch, aber syntaktisch korrekt ist.
Kommen wir nun zu den Programmiersprachen. Hier legt das Programmierhandbuch fest, in welcher Weise das Programm selbst bzw. seine Teilstücke aufgebaut sein können. Als Beispiel geben wir nachfolgend einige Regeln, die sagen, wie Zahlen in einem PASCAL-Programm aussehen können.

(unsigned integer) → (digit) | (digit){digit}
(unsigned real) → (unsigned integer).(digit){digit} | (unsigned integer)E(scale factor)
(scale factor) → (unsigned integer) | (sign) (unsigned integer)
(digit) → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
(sign) → + | –

Hieraus erhalten wir die folgende Sequenz

$$\begin{aligned}(\text{unsigned real}) &\Longrightarrow (\text{unsigned integer})\text{E}(\text{scale factor})\\ &\Longrightarrow (\text{digit})\{\text{digit}\}\text{E}(\text{scale factor})\\ &\Longrightarrow 3\{\text{digit}\}\text{E}(\text{scale factor})\\ &\Longrightarrow 314\text{E}(\text{scale factor})\\ &\Longrightarrow 314\text{E}(\text{sign})(\text{unsigned integer})\\ &\Longrightarrow 314\text{E}–(\text{unsigned integer})\\ &\Longrightarrow 314\text{E}–(\text{digit})\\ &\Longrightarrow 314\text{E}–2\end{aligned}$$

aus der hervorgeht, dass (die Näherung) 3,14 (für π) eine reelle Zahl ist.
Wir stellen folgende Gemeinsamkeiten fest:

- Eigentlich handelt es sich bei den Vorschriften um Ersetzungsregeln. Gewisse Objekte werden durch andere ersetzt.
- Es gibt Objekte, die ersetzt werden (z. B. (Substantivphrase), (unsigned real)), und andere Objekte, die durch die Ersetzungen nicht verändert werden, sondern endgültigen Charakter haben (wie die Wörter der Sprache selbst oder die Ziffern 0,1,2,...,9 und die Zeichen + und –).
- Die Erzeugungen beginnen mit festgelegten Objekten (wie (Satz) oder (program)) und enden, wenn nur noch unveränderliche Objekte vorhanden sind.

Wir werden auf dieser Basis im Folgenden formale Grammatiken und Sprachen definieren. Dabei wollen wir Objekte als Buchstaben eines Alphabets auffassen, und die erzeugten Sätze bzw. Programme bzw. Programmstücke sind dann Wörter über dem Alphabet, das z. B. als Buchstaben alle deutschen Wörter bzw. die Elemente `if`, `while`, Ziffern usw. enthält.
Um die Möglichkeiten zur Wahl von Alphabeten nicht ausufern zu lassen, wollen wir im Folgenden immer annehmen, dass die betrachteten Alphabete (endliche) Teilmengen einer festen abzählbar-unendlichen Menge sind.
Unter einer Sprache über dem Alphabet V verstehen wir im Folgenden stets eine beliebige Teilmenge von V^*. In den folgenden Abschnitten werden verschiedene Möglichkeiten der Beschreibung von (unendlichen) Sprachen durch endliche Objekte untersucht.

Definition 2.1 *Eine Regelgrammatik (oder kurz Grammatik) ist ein Quadrupel*

$$G = (N, T, P, S),$$

wobei

- *N und T endliche, disjunkte Alphabete sind, deren Vereinigung wir mit V bezeichnen,*
- *P eine endliche Teilmenge von $(V^* \setminus T^*) \times V^*$ ist, und*
- *$S \in N$ gilt.*

Dabei ist N das Alphabet der Nichtterminale oder Hilfssymbole (wie (Substantivphrase) oder (unsigned real)) und T das Alphabet der Terminale. Im Folgenden werden wir meist große lateinische Buchstaben zur Bezeichnung der Nichtterminale und kleine lateinische Buchstaben für die Terminale verwenden. Die Elemente aus P heißen Regeln. Meistens werden wir das Paar (α, β) aus P in der Form $\alpha \longrightarrow \beta$ schreiben, da diese Notation der Anwendung von Regeln (in der nächsten Definition) als Ersetzung entspricht. S heißt Axiom oder Startwort (und entspricht (Satz) bzw. (program)).

Definition 2.2 *Es sei $G = (N, T, P, S)$ eine Regelgrammatik wie in Definition 2.1 beschrieben. Wir sagen, dass aus dem Wort $\gamma \in V^+$ das Wort $\gamma' \in V^*$ erzeugt wird, wenn*

$$\gamma = \gamma_1 \alpha \gamma_2,\ \gamma' = \gamma_1 \beta \gamma_2,\ \alpha \longrightarrow \beta \in P$$

für gewisse $\gamma_1, \gamma_2 \in V^*$ gelten. Wir schreiben dann

$$\gamma \Longrightarrow \gamma'.$$

Entsprechend Definition 2.2 entsteht γ' aus γ, indem ein Teilwort α in γ durch β ersetzt wird, wenn eine Regel $\alpha \longrightarrow \beta$ in P existiert. Die Regeln geben also an, welche lokalen Ersetzungen ausgeführt werden können, um aus einem Wort ein neues zu erzeugen.
Die Anwendung einer Regel nennen wir auch einen Ableitungsschritt oder sagen, dass γ' aus γ direkt abgeleitet oder generiert wird. Falls die bei der Erzeugung verwendete Regel $p = \alpha \longrightarrow \beta$ betont werden soll, so schreiben wir $\gamma \Longrightarrow_p \gamma'$. Durch $\Longrightarrow$ wird offenbar eine Relation, d.h. eine Teilmenge von $V^+ \times V^*$, definiert. Wie üblich kann hiervon der reflexive und transitive Abschluss $\Longrightarrow^*$ gebildet werden, d.h. es gilt

$$\gamma \Longrightarrow^* \gamma'$$

genau dann, wenn es eine natürliche Zahl $n \geq 0$ und Wörter $\delta_0, \delta_1, \delta_2, \ldots, \delta_{n-1}, \delta_n$ mit

$$\gamma = \delta_0 \Longrightarrow \delta_1 \Longrightarrow \delta_2 \Longrightarrow \ldots \Longrightarrow \delta_{n-1} \Longrightarrow \delta_n = \gamma'$$

gibt (im Fall $n = 0$ gilt $\gamma = \gamma'$, und im Fall $n = 1$ haben wir $\gamma \Longrightarrow \gamma'$). Somit gilt $\gamma \Longrightarrow^* \gamma'$ genau dann, wenn γ' durch iterierte Anwendung von (nicht notwendigerweise gleichen) Regeln aus γ entsteht. Gilt $\gamma \Longrightarrow^* \gamma'$, so sagen wir auch γ' ist aus γ (in mehreren Schritten) ableitbar oder erzeugbar.
Ein Wort $w \in V^*$ heißt *Satzform* von G, wenn $S \Longrightarrow^* w$ gilt, d.h. wenn w aus S erzeugt werden kann.

Definition 2.3 *Für eine Grammatik $G = (N, T, P, S)$ aus Definition 2.1 ist die von G erzeugte Sprache $L(G)$ durch*

$$L(G) = \{w : w \in T^* \text{ und } S \Longrightarrow^* w\}$$

definiert.

Entsprechend dieser Definition besteht die von G erzeugte Sprache also aus allen Satzformen von G, die nur Terminale enthalten. Ferner zeigt diese Definition die Notwendigkeit der Angabe von S in der Definition 2.1, da nur die aus S in mehreren Schritten ableitbaren Wörter über T die Sprache bilden.
Diese Definition macht auch deutlich, warum die Elemente aus N bzw. T Nichtterminale oder Hilfssymbole bzw. Terminale heißen. Die Elemente aus N werden für die Sprache selbst nicht benötigt, sie erscheinen nur in Zwischenschritten der Ableitung, haben daher Hilfscharakter. Die Terminale dagegen bilden das Alphabet, über dem die Endwörter definiert werden, wobei Endwort so zu verstehen ist, dass aus diesen Wörtern keine weiteren mehr abgeleitet werden können.
Wir betrachten nun einige Beispiele.

Beispiel 2.4 Wir betrachten die Regelgrammatik

$$G_1 = (\{S, A, B\}, \{a, b\}, \{p_1, p_2, p_3, p_4, p_5\}, S)$$

mit

$$p_1 = S \longrightarrow AB,\ p_2 = A \longrightarrow aA,\ p_3 = A \longrightarrow \lambda,\ p_4 = B \longrightarrow Bb,\ p_5 = B \longrightarrow \lambda.$$

Wir zeigen zuerst, dass jede Satzform von G_1 eine der folgenden Formen hat, wobei n und m beliebige natürliche Zahlen sind:

$$S,\ a^nABb^m,\ a^nAb^m,\ a^nBb^m,\ a^nb^m. \qquad (*)$$

Dies gilt offensichtlich für das Startwort S und das einzige daraus in einem Schritt ableitbare Wort AB ($n = m = 0$). Wir betrachten nun ein Wort der Form a^nABb^m. Hierfür ergeben sich nur die folgenden direkten Ableitungen

$$a^nABb^m \Longrightarrow_{p_2} a^naABb^m, \quad a^nABb^m \Longrightarrow_{p_3} a^n\lambda Bb^m,$$
$$a^nABb^m \Longrightarrow_{p_4} a^nABbb^m, \quad a^nABb^m \Longrightarrow_{p_5} a^nA\lambda b^m.$$

Folglich sind aus a^nABb^m nur die Wörter

$$a^{n+1}ABb^m,\ a^nBb^m,\ a^nABb^{m+1},\ a^nAb^m$$

in einem Schritt ableitbar, die alle von der gewünschten Form sind. Analog kann man leicht nachweisen, dass auch alle in einem Schritt aus a^nAb^m bzw. a^nBb^m ableitbaren Wörter von einer der Formen aus $(*)$ sind. Da aus a^nb^m keine Wörter ableitbar sind, ist damit die obige Aussage bewiesen.
Wir beweisen nun, dass sogar jedes Wort der in $(*)$ genannten Form eine Satzform von G_1 ist. Mit Ausnahme von a^nAb^m folgt dies aus der folgenden Ableitung:

$$\begin{aligned} S \ &\Longrightarrow_{p_1} \ \underbrace{AB \Longrightarrow aAB \Longrightarrow aaAB \Longrightarrow \ldots \Longrightarrow a^{n-1}AB}_{(n-1)-malige\ Anwendung\ von\ p_2} \\ &\Longrightarrow_{p_2} \ \underbrace{a^nAB \Longrightarrow a^nABb \Longrightarrow a^nABb^2 \Longrightarrow \ldots \Longrightarrow a^nAB^m}_{m-malige\ Anwendung\ von\ p_4} \\ &\Longrightarrow_{p_3} \ a^nBb^m \Longrightarrow_{p_5} a^nb^m. \end{aligned}$$

Da die von G_1 erzeugte Sprache nur Wörter über $\{a, b\}$ enthält, besteht $L(G_1)$ aus allen Wörtern der Form $(*)$ in $\{a, b\}^*$. Somit gilt

$$L(G_1) = \{a^nb^m : n \geq 0, m \geq 0\}.$$

Beispiel 2.5 Es sei

$$G_2 = (\{S\}, \{a, b\}, \{S \longrightarrow aSb, S \longrightarrow ab\}, S).$$

Mittels vollständiger Induktion zeigen wir nun, dass durch $n \geq 1$ Ableitungsschritten genau die Wörter a^nSb^n und a^nb^n aus S erzeugt werden können.
Dies gilt offenbar für $n = 1$, denn aus dem Axiom S werden durch Anwendung der beiden Regel $S \longrightarrow aSb$ bzw. $S \longrightarrow ab$ die Wörter aSb bzw. ab abgeleitet.
Sei nun w ein Wort, das durch n Ableitungsschritte aus S erzeugt wird. Nach Definition muss w dann durch Anwendung einer Regel auf ein Wort v entstehen, wobei sich v in $n - 1$ Schritten erzeugen lässt. Nach Induktionsannahme muss also $v = a^{n-1}Sb^{n-1}$ oder $v = a^{n-1}b^{n-1}$ gelten. Im ersten Fall sind durch Ersetzung von S entsprechend den beiden Regeln die Wörter a^nSb^n und a^nb^n ableitbar; im zweiten Fall enthält v nur Terminale,

womit aus v kein Wort mehr abgeleitet werden kann. Somit sind in n Schritten nur a^nSb^n und a^nb^n erzeugbar. Dies beweist aber gerade die Induktionsbehauptung.
Da die Wörter aus $L(G_2)$ in einer endlichen Anzahl von Schritten abgeleitet werden müssen und nur Terminale enthalten dürfen, folgt

$$L(G_2) = \{a^nb^n : n \geq 1\}.$$

Beispiel 2.6 Wir betrachten die Regelgrammatik

$$G_3 = (\{S, A\}, \{a, b\}, \{S \longrightarrow \lambda, S \longrightarrow aS, S \longrightarrow Sb\}, S).$$

Wie in Beispiel 2.4 können wir zeigen, dass die Menge der Satzformen aus allen Wörtern der Form a^nSb^m oder a^nb^m mit $n \geq 0$ und $m \geq 0$ besteht, oder wir beweisen in Analogie zu Beispiel 2.5, dass in $k \geq 1$ Schritten genau die Wörter $a^nSb^m, a^{n-1}b^m, a^nb^{m-1}$ mit $n + m = k$ erzeugt werden können. Daraus ergibt sich

$$L(G_3) = \{a^nb^m : n \geq 0, m \geq 0\}.$$

Beispiel 2.7 Es sei

$$G_4 = (\{S, A\}, \{a, b\}, \{S \longrightarrow \lambda, S \longrightarrow aS, S \longrightarrow a, S \longrightarrow A, A \longrightarrow bA, A \longrightarrow b\}, S).$$

In Abbildung 2.1 sind – bis auf $S \Longrightarrow \lambda$ – im Wesentlichen alle möglichen Ableitungen dargestellt, wobei die nach rechts gerichten Pfeile der Anwendung von $S \longrightarrow aS$ bzw. $A \longrightarrow bA$, die nach oben der von $S \longrightarrow a$ und die nach unten der von $A \longrightarrow b$ entsprechen; die durch die Regel $S \longrightarrow A$ hervorgebrachten Ableitungen sind noch zusätzlich einzutragen (jeweils senkrecht bis zum nächsten Wort). Daraus ist leicht zu ersehen, dass sich erneut

$$L(G_4) = \{a^nb^m : n \geq 0, m \geq 0\}$$

ergibt. Ein formaler Beweis wie in den vorangegangenen Beispielen bleibt dem Leser überlassen.

Beispiel 2.8 Es sei die Regelgrammatik

$$G_5 = (\{S, A, B, B', B''\}, \{a, b, c\}, \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}, S)$$

mit

$$p_1 = S \longrightarrow ABA,\ p_2 = AB \longrightarrow aAbB',\ p_3 = AB \longrightarrow abB'',\ p_4 = B'b \longrightarrow bB',$$
$$p_5 = B''b \longrightarrow bB'',\ p_6 = B'A \longrightarrow BAc,\ p_7 = B''A \longrightarrow c,\ p_8 = bB \longrightarrow Bb$$

gegeben. Durch eine Analyse aller möglichen Ableitungen wollen wir $L(G_5)$ bestimmen.
Für $n \geq 0$ sei $w_n = a^nABb^nAc^n$.
Wir betrachten zuerst den Fall $n \geq 2$. Die einzigen auf w_n anwendbaren Regeln sind p_2 und p_3.
Fall 1: Anwendung von p_2. Wir erhalten das Wort $a^{n+1}AbB'b^nAc^n$. Nun ist nur p_4 anwendbar, und die Anwendung dieser Regel liefert $a^{n+1}AbbB'b^{n-1}Ac^n$, d.h. wir haben B' um eine Position nach rechts verschoben. Erneut ist nur p_4 anwendbar, und wir können

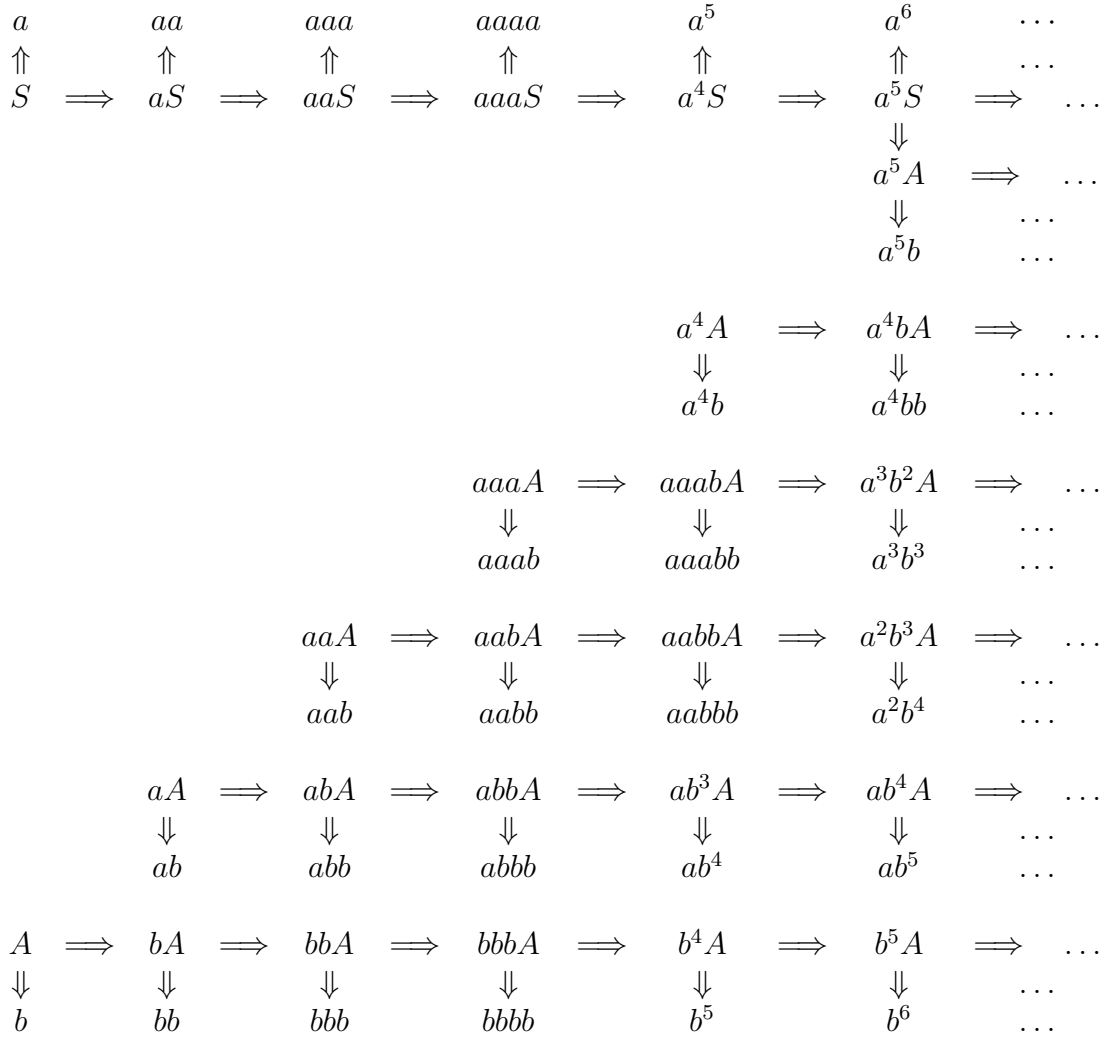


Abbildung 2.1: Ableitungen in Beispiel 2.7

B' um eine Position weiter nach rechts verschieben. Diese Situation hält an, bis wir das Wort $a^{n+1}Ab^{n+1}B'Ac^n$ erzeugt haben. Nun ist nur p_6 anwendbar, durch deren Anwendung $a^{n+1}Ab^{n+1}BAc^{n+1}$ entsteht. Jetzt kann nur p_8 angewendet werden, wodurch eine Verschiebung von B um eine Position nach links bewirkt wird. Erneut ist nur diese Verschiebung möglich, bis wir $w_{n+1} = a^{n+1}ABb^{n+1}Ac^{n+1}$ erhalten.

Fall 2: Anwendung von p_3. Wir erhalten das Wort $a^{n+1}bB''b^nAc^n$. Nun ist nur p_5 anwendbar, d.h. B'' wird um eine Position nach rechts verschoben. Diese Situation bleibt erhalten, bis wir das Wort $a^{n+1}b^{n+1}B''Ac^n$ erzeugt haben. Nun ist nur p_7 anwendbar, durch deren Anwendung $a^{n+1}b^{n+1}c^{n+1}$ entsteht.

Somit wird aus w_n entweder w_{n+1}, womit der eben beschriebene Prozess erneut gestartet werden kann, oder $a^{n+1}b^{n+1}c^{n+1}$ abgeleitet.

Analog kann man sich überlegen, dass w_0 und w_1 nur die Ableitungen

$$w_0 \Longrightarrow^* w_1, w_0 \Longrightarrow^* abc, w_1 \Longrightarrow^* w_2, w_1 \Longrightarrow^* a^2b^2c^2$$

gestatten. Wegen $S \Longrightarrow w_0$ gilt folglich

$$L(G_5) = \{a^n b^n c^n : n \geq 1\}.$$

Beispiel 2.9 Wir betrachten die Regelgrammatik

$$G_6 = (\{S, A, B, B', B''\}, \{a, b, c\}, \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}, S)$$

mit

$$\begin{aligned} &p_0 = S \to abc,\ p_1 = S \longrightarrow aABbA,\ p_2 = AB \longrightarrow aAbB',\\ &p_3 = AB \longrightarrow abB'',\ p_4 = B'b \longrightarrow bB',\ p_5 = B''b \longrightarrow bB'',\\ &p_6 = B'A \longrightarrow BAc,\ p_7 = B''A \longrightarrow cc,\ p_8 = bB \longrightarrow Bb. \end{aligned}$$

Wie im vorhergehenden Beispiel können wir

$$L(G_6) = \{a^n b^n c^n \mid n \geq 1\}$$

zeigen.

Beispiel 2.10 Wir betrachten die Regelgrammatik $G_7 = (N, T, P, S)$ mit

$$\begin{aligned} N &= \{S\},\\ T &= \{x, y, z, +, -, \cdot, :, (,)\},\\ P &= \{S \longrightarrow (S + S), S \longrightarrow (S - S), S \longrightarrow (S \cdot S), S \longrightarrow (S : S),\\ &\qquad S \longrightarrow x, S \longrightarrow y, S \longrightarrow z\}. \end{aligned}$$

Wir wollen beweisen, dass $L(G_7)$ aus allen exakt geklammerten arithmetischen Ausdrücken mit den Variablen x, y, z (wobei keine Vorrangregeln für die Operationen beachtet werden und auch äußere Klammern mitgeführt werden) besteht.
Hierfür zeigen wir erst, dass jede Satzform, die aus S erzeugt werden kann, ein exakt geklammerter Ausdruck in den Variablen S, x, y, z ist. Dies folgt aber sofort daraus, dass das Axiom ein solcher Ausdruck ist und aus exakt geklammerten Ausdrücken wieder nur solche entstehen, denn die Ersetzung von S durch x, y, z oder $(S \circ S)$ mit $\circ \in \{+, -, \cdot, :\}$ bewahrt exakte Klammerungen.
Wir zeigen nun mittels Induktion über die Anzahl der Schritte in der Konstruktion eines exakt geklammerten Ausdrucks, dass *alle* exakt geklammerten Ausdrücke in $L(G_7)$ sind. Für $n = 0$ erhalten wir nur Variable, und x, y, z sind aus S mittels der Anwendung der Regeln $S \longrightarrow x, S \longrightarrow y, S \longrightarrow z$ direkt erzeugbar. Seien nun $n \geq 1$ und w ein durch n Schritte erzeugter exakt geklammerter Ausdruck. Dann gilt $w = (w_1 \circ w_2)$ für eine Operation $\circ \in \{+, -, \cdot, :\}$ und exakt geklammerte Ausdrücke w_1 und w_2, von denen jeder durch höchstens $n - 1$ Konstruktionsschritte gewonnen wird. Nach Induktionsannahme gelten damit

$$S \Longrightarrow^* w_1 \quad \text{und} \quad S \Longrightarrow^* w_2.$$

Somit gibt es auch die Ableitung

$$S \Longrightarrow (S \circ S) \Longrightarrow^* (w_1 \circ S) \Longrightarrow^* (w_1 \circ w_2) = w.$$

Damit ist $w \in L(G_7)$ gezeigt.

Beispiel 2.11 In diesem Beispiel wollen eine Regelgrammatik angeben, die alle **LOOP**/-**WHILE**-Programme aus Abschnitt 1.1 erzeugt.
Entsprechend den Definitionen müssen sich alle **LOOP**/**WHILE**-Programme aus dem Startsymbol herleiten lassen. Die Regeln, mittels derer **LOOP**/**WHILE**-Programme erzeugt werden können, sind im Wesentlichen bei der Definition von **LOOP**/**WHILE**-Programmen angegeben worden; es handelt sich um die Grundanweisungen, das Hintereinanderausführen und den **LOOP**- bzw. **WHILE**-Befehl. Wir müssen diesen Prozess nur formal als Grammatik aufschreiben. Dafür verwenden wir das Nichtterminal A als Bezeichnung für ein beliebiges Programm und ersetzen es jeweils durch die zugelassen Befehle; wir haben also A für die Bezeichnungen Π, Π_1 und Π_2 von Programmen zu ersetzen. A ist dann natürlich auch das Axiom, da wir Programme erzeugen wollen. (Wir wählen die Bezeichnung A, da S bereits für die Nachfolgerfunktion vergeben ist.)
Ein Problem bereiten noch die Variablen, da wir davon unendlich viele benötigen, unsere Alphabete der Terminale und Nichtterminale aber endlich sein müssen. Deshalb gehen wir wie folgt vor. Anstelle von x_i verwenden wir die Notation $x[i]$ (wie in Programmiersprachen üblich). Nun muss i eine natürliche Zahl sein, und kann daher durch eine Folge von Ziffern repräsentiert werden. Wir gehen daher von $x[I]$ aus, wobei I ein zusätzliches Nichtterminal ist, aus dem wir alle Ziffernfolgen (ohne führende Nullen) ableiten.
Aus diesen Bemerkungen ergibt sich formal die Regelgrammatik

$$G_8 = (\{A, I, J\}, T, P, A)$$

mit dem Terminalalphabet

$$\begin{aligned} T \;=\; \{&S, P, \mathbf{LOOP}, \mathbf{WHILE}, \mathbf{BEGIN}, \mathbf{END}, :=, \neq, ;, , (,) \\ &0, 1, 2, 3, 4, 5, 6, 7, 8, 9, x, [,] \,\} \end{aligned}$$

(man beachte, dass das Semikolon ein Element von T ist, während die Kommata beim Aufschreiben von T als Trennzeichen zwischen den Elementen aus T fungieren) und der Regelmenge

$$\begin{aligned} P \;=\; \{&A \to x[I] := 0,\ A \to x[I] := x[I],\ A \to x[I] := S(x[I]),\ A \to x[I] := P(x[I]), \\ &A \to A; A,\ A \to \mathbf{LOOP}\ x[I]\ \mathbf{BEGIN}\ A\ \mathbf{END}, \\ &A \to \mathbf{WHILE}\ x[I] \neq 0\ \mathbf{BEGIN}\ A\ \mathbf{END}\} \\ &\cup \{I \to z,\ I \to Jz,\ J \to Jz \mid z \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}\} \\ &\cup \{J \to z \mid z \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}\} \end{aligned}$$

(zuerst erzeugen wir aus I die letzte Ziffer mittels $I \to z$ oder $I \to Jz$, wobei z eine beliebige Ziffer ist; nun werden aus J analog die davor stehenden Ziffern erzeugt; bei der abschließenden Terminierung durch $J \to z$ darf dann z nicht 0 sein, da sonst eine führende Null entstehen würde).

Wir führen nun einige spezielle Typen von Regelgrammatiken ein.

Definition 2.12 *Es sei $G = (N, T, P, S)$ eine Regelgrammatik wie in Definition 2.1. Wir sagen,*

- *G ist monoton, wenn für alle Regeln $\alpha \longrightarrow \beta \in P$ die Bedingung $|\alpha| \leq |\beta|$ erfüllt ist, wobei als Ausnahme $S \longrightarrow \lambda$ zugelassen ist, wenn $|\beta'|_S = 0$ für alle Regeln $\alpha' \longrightarrow \beta' \in P$ gilt,*

- *G ist kontextabhängig, wenn alle Regeln in P von der Form $uAv \longrightarrow uwv$ mit $u, v \in V^*$, $A \in N$ und $w \in V^+$ sind, wobei als Ausnahme $S \longrightarrow \lambda$ zugelassen ist, wenn $|\beta'|_S = 0$ für alle Regeln $\alpha' \longrightarrow \beta' \in P$ gilt,*

- *G ist kontextfrei, wenn alle Regeln in P von der Form $A \longrightarrow w$ mit $A \in N$ und $w \in V^*$ sind,*

- *G ist regulär, wenn alle Regeln in P von der Form $A \longrightarrow wB$ oder $A \longrightarrow w$ mit $A, B \in N$ und $w \in T^*$ sind.*

Die monotonen Grammatiken haben – abgesehen von der Ausnahmeregelung – die Eigenschaft, dass bei Anwendung einer Regel die Länge des abgeleiteten Wortes nicht kleiner ist als die des Ausgangswortes, d.h. $\longrightarrow$ ist bezüglich der Wortlänge eine monotone Relation. Bei kontextabhängigen Grammatiken wird bei Anwendung einer Regel $uAv \longrightarrow uwv$ eigentlich nur das Nichtterminal A durch das Wort w ersetzt; aber diese Ersetzung ist nur erlaubt, wenn links bzw. rechts von A das Wort u bzw. v stehen, d.h. es wird die Existenz eines lokalen Kontextes von A für die Ersetzung gefordert. Genau dieser Kontext wird bei kontextfreien Grammatiken nicht gefordert (daher wäre der Begriff „kontextunabhängig“ eigentlich besser, denn A steht in einem Kontext, der aber für die Ersetzung unerheblich ist; es hat sich aber „kontextfrei“ eingebürgert und durchgesetzt).
Reguläre Grammatiken sind entsprechend der Definition 2.12 ein Spezialfall kontextfreier Grammatiken, die durch zusätzliche strukturelle Forderungen an die rechten Seiten der Regeln gekennzeichnet sind.
Da das Leerwort als rechte Seite bei Regeln von Regelgrammatiken, kontextfreien und regulären Grammatiken zugelassen ist, ist klar, dass das Leerwort auch in der erzeugten Sprache liegen kann. Die Ausnahmeregelungen in der Definition monotoner und kontextabhängiger Grammatiken dienen dazu, diese Eigenschaft auch für diese Typen von Grammatiken abzusichern.
Außer den in Definition 2.12 eingeführten Bezeichnungen wird vielfach auch Typ 0 für beliebige Regelgrammatiken, Typ 1 für kontextabhängige, Typ 2 für kontextfreie und Typ 3 für reguläre Grammatiken benutzt.

Wir klassifizieren nun die Grammatiken aus den obigen Beispielen hinsichtlich der Eigenschaften von Definition 2.12.
G_1 ist wegen der Regel $p_3 = A \longrightarrow \lambda$ nicht monoton und nicht kontextabhängig. G_1 ist auch nicht regulär, da die Regel $p_4 = B \longrightarrow Bb$ in der Regelmenge von G_1 existiert. G_1 ist aber offensichtlich kontextfrei.
G_2 ist monoton, kontextabhängig (für alle Regeln gilt $u = v = \lambda$) und kontextfrei, aber nicht regulär.
G_3 ist nicht monoton und nicht kontextabhängig (wegen der gleichzeitigen Existenz der Regeln $S \longrightarrow \lambda$ und $S \longrightarrow aS$) und nicht regulär, aber kontextfrei.
G_4 ist regulär und damit auch kontextfrei, aber nicht monoton und nicht kontextabhängig.
G_5 hat keine der in Definition 2.12 gegebenen Eigenschaften. G_6 ist monoton, aber

weder kontextabhängig noch kontextfrei noch regulär. G_7 und G_8 sind monoton, kontextabhängig und kontextfrei, jedoch nicht regulär.

Definition 2.13 *Eine Sprache L heißt monoton (bzw. kontextabhängig, kontextfrei oder regulär), wenn es eine monotone (bzw. kontextabhängige, kontextfreie oder reguläre) Grammatik G mit $L = L(G)$ gibt.*

Nach dieser Definition ist $L = \{a^n b^m : n \geq 0, m \geq 0\}$ eine kontextfreie Sprache, denn es gilt $L = L(G_3)$, und G_3 ist eine kontextfreie Grammatik. Jedoch lässt sich aus der Tatsache, dass G_3 keine reguläre Grammatik ist, nicht schließen, dass L keine reguläre Sprache ist. Da nämlich G_4 ebenfalls die Sprache L erzeugt und G_4 eine reguläre Grammatik ist, ist L regulär.

Mit $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$, $\mathcal{L}(MON)$ und $\mathcal{L}(RE)$ bezeichnen wir die Menge aller Sprachen, die von regulären, kontextfreien, kontextabhängigen, monotonen und beliebigen Regelgrammatiken erzeugt werden.[1]
Wir bemerken zuerst, dass für zwei Typen X und Y von Grammatiken aus dem Fakt, dass jede Grammatik vom Typ X auch eine vom Typ Y ist, sich die Aussage $\mathcal{L}(X) \subseteq \mathcal{L}(Y)$ ergibt. Hieraus folgt sofort das folgende Lemma.

Lemma 2.14 *$\mathcal{L}(CS) \subseteq \mathcal{L}(MON) \subseteq \mathcal{L}(RE)$ und $\mathcal{L}(REG) \subseteq \mathcal{L}(CF) \subseteq \mathcal{L}(RE)$.* □

Im nächsten Abschnitt werden weitere Beziehungen zwischen den eingeführten Mengen hergeleitet und festgestellt, ob die Inklusionen in Lemma 2.14 echt oder Gleichheiten sind.

2.1.2 Normalformen und Schleifensätze

Wir werden in diesem Abschnitt zuerst zeigen, dass für die im vorangegangenen Abschnitt eingeführten Typen von Grammatiken jeweils Normalformen existieren, d.h. Grammatiken dieses Typs mit weiteren Einschränkungen an die Regeln, die es aber trotzdem gestatten, jede Sprache dieses Typs von einer Grammatik in Normalform zu erzeugen. Wir benutzen diese Normalformen vor allem als beweistechnische Hilfsmittel und zur Herleitung von Eigenschaften, die uns den Nachweis gestatten, dass gewisse Sprachen nicht durch Grammatiken eines gegebenen Typs erzeugt werden können.
Wir beweisen jeweils nicht nur die Existenz der Normalform, sondern zeigen auch, dass eine Grammatik in Normalform konstruktiv gewonnen werden kann.

Wir beginnen mit Normalformen für monotone Grammatiken.

Lemma 2.15 *Zu jeder Regelgrammatik $G = (N, T, P, S)$ kann eine Regelgrammatik $G' = (N', T, P', S)$ so konstruiert werden, dass alle Regeln aus P' von der Form $\alpha \longrightarrow \beta$ mit $\alpha, \beta \in (N')^*$ oder $A \longrightarrow a$ mit $A \in N', a \in T$ sind und $L(G) = L(G')$ gilt. Ist außerdem G eine monotone, kontextabhängige bzw. kontextfreie Grammatik, so ist auch G' monoton, kontextabhängig bzw. kontextfrei.*

[1] Die hierbei verwendeten Bezeichnungen *REG, CF, CS, MON, RE* sind Abkürzungen der entsprechenden englischen Wörter regular, context-free, context-sensitive, monotone, recursively enumerable.

Beweis. Für jedes Terminal a sei a' ein neues Symbol (das also weder in N noch in T liegt). Ferner sei für $a \neq b, a, b \in T$ auch $a' \neq b'$. Wir setzen

$$N' = N \cup \{a' : a \in T\}.$$

Ist $w = x_1 x_2 \dots x_n$ ein Wort aus V^*, so sei $w' = y_1 y_2 \dots y_n$ das Wort aus $(N')^*$ mit

$$y_i = \begin{cases} x_i & \text{für } x_i \in N \\ x_i' & \text{für } x_i \in T \end{cases}$$

für $1 \leq i \leq n$. Wir definieren nun die Regelmenge von G' durch

$$P' = \{\alpha' \longrightarrow \beta' : \alpha \longrightarrow \beta \in P\} \cup \{a' \longrightarrow a : a \in T\}.$$

Wir beweisen nun $L(G') = L(G)$.
Sei dazu zuerst $w \in L(G)$. Dann gibt es in G eine Ableitung

$$S = w_0 \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \dots \Longrightarrow w_n = w.$$

Entsprechend der Konstruktion von P' gibt es dann in G' die Ableitung

$$S = w_0' \Longrightarrow w_1' \Longrightarrow w_2' \Longrightarrow \dots w_n' = w' = v_0 \Longrightarrow v_1 \Longrightarrow v_2 \Longrightarrow \dots \Longrightarrow v_m = w,$$

bei der wir für den Übergang von w_i' zu w_{i+1}' stets die Regel $\alpha' \longrightarrow \beta' \in P'$ anwenden, wenn w_{i+1} aus w_i durch Anwendung der Regel $\alpha \longrightarrow \beta \in P$ entstanden ist und die direkten Ableitungen $v_j \Longrightarrow v_{j+1}$ durch Anwendung einer Regel der Form $a' \longrightarrow a$ geschehen. Daher gilt auch $w \in L(G')$, womit $L(G) \subseteq L(G')$ gezeigt ist.
Sei nun $x \in L(G')$. Dann gibt es für x eine Ableitung der Form

$$S = x_0' \Longrightarrow x_1' \Longrightarrow x_2' \Longrightarrow \dots \Longrightarrow x_n' = x' = y_0 \Longrightarrow y_1 \Longrightarrow y_2 \Longrightarrow \dots \Longrightarrow y_m = x$$

(eine Ableitung dieser Form entsteht aus einer beliebigen Ableitung von w, indem man die Reihenfolge der angewendeten Regeln so vertauscht, dass im ersten Teil nur Regeln der Form $\alpha' \longrightarrow \beta'$ und im zweiten Teil nur Regeln der Form $a' \longrightarrow a$ angewendet werden, wodurch auch abgesichert ist, dass die im ersten Teil der Ableitung entstehenden Satzformen sämtlich nur Symbole aus N' enthalten). Wenn wir nun die Reihenfolge der Regelanwendung nicht ändern, aber stets statt $\alpha' \longrightarrow \beta' \in P'$ die Regel $\alpha \longrightarrow \beta \in P$ benutzen, so erhalten wir die Ableitung

$$S = x_0 \Longrightarrow x_1 \Longrightarrow x_2 \Longrightarrow \dots \Longrightarrow x_n = x$$

in G. Dies beweist $x \in L(G)$ und damit $L(G') \subseteq L(G)$.
Aus den beiden nachgewiesenen Inklusionen folgt $L(G) = L(G')$.
Bei der Konstruktion von P' wird eine Regel $\alpha \longrightarrow \beta$ mit $|\alpha| \leq |\beta|$ in eine Regel $\alpha' \longrightarrow \beta'$ mit $|\alpha'| \leq |\beta'|$ überführt, da $|\alpha| = |\alpha'|$ und $|\beta| = |\beta'|$ gelten. Damit ist G' monoton, wenn G monoton ist. Analog ist sofort zu sehen, dass Regeln der Form $uAv \longrightarrow uwv$ bzw. $A \longrightarrow w$ wieder in Regeln dieser Form übergehen. Hieraus folgt sofort die Aussage über die Kontextabhängigkeit und Kontextfreiheit. □

Satz 2.16 *Zu jeder monotonen Grammatik $G = (N, T, P, S)$ kann eine monotone Grammatik $G' = (N', T, P', S)$ so konstruiert werden, dass jede Regel aus P' von einer der Formen*

$$A \longrightarrow BC,\ A \longrightarrow B,\ AB \longrightarrow CB,\ AB \longrightarrow AC \text{ oder } A \longrightarrow a$$

mit $A, B, C \in N', a \in T$ oder $S \longrightarrow \lambda$ ist und $L(G) = L(G')$ gilt.

Beweis. Wegen Lemma 2.15 können wir annehmen, dass alle Regeln von P von der Form $\alpha \longrightarrow \beta$ oder $A \longrightarrow a$ mit $\alpha, \beta \in N^+, A \in N, a \in T$ (oder $S \longrightarrow \lambda$) sind.
Jeder Regel aus P werden wir nun eine Menge von Regeln und Nichtterminalen so zuordnen, dass die Mengen P' und N' mit den gewünschten Eigenschaften als Vereinigung aller dieser Mengen von Regeln bzw. aller dieser Mengen von Nichtterminalen und N entstehen. Die dabei neu eingeführten Symbole sollen stets paarweise verschieden sein und nicht in $N \cup T$ liegen.
Sei $p = X_1 X_2 \dots X_n \longrightarrow Y_1 Y_2 \dots Y_m$ eine Regel aus P.
Fall 1. $n = 1$ *und* $m \leq 2$. Dann setzen wir

$$P_p = \{p\} \quad \text{und} \quad N_p = \emptyset,$$

d.h. wir übernehmen die Regel p in P' und führen keine neue Hilfssymbole ein.
Fall 2. $n = 1$ *und* $m \geq 3$. Dann setzen wir

$$N_p = \{C_{p,1}, C_{p,2}, \dots, C_{p,m-2}\}$$

und

$$P_p = \{X_1 \longrightarrow Y_1 C_{p,1}, C_{p,1} \longrightarrow Y_2 C_{p,2}, \dots, C_{p,m-3} \longrightarrow Y_{m-2} C_{p,m-2}, C_{p,m-2} \longrightarrow Y_{m-1} Y_m\}.$$

Fall 3. $n \geq 2$. Dann gilt auch $m \geq 2$. Wir setzen nun

$$N'_p = \{C_{p,1}, C_{p,2}, \dots, C_{p,n}, D\}$$

und

$$\begin{aligned} P'_p \;=\; \{ & X_1 X_2 \longrightarrow C_{p,1} X_2, C_{p,1} X_2 \longrightarrow C_{p,1} C_{p,2}, C_{p,2} X_3 \longrightarrow C_{p,2} C_{p,3}, \\ & \dots, C_{p,n-2} X_{n-1} \longrightarrow C_{p,n-2} C_{p,n-1}, C_{p,n-1} X_n \longrightarrow C_{p,n-1} C_{p,n}, \\ & C_{p,1} C_{p,2} \longrightarrow Y_1 C_{p,2}, C_{p,2} C_{p,3} \longrightarrow Y_2 C_{p,3}, \\ & \dots, C_{p,n-2} C_{p,n-1} \longrightarrow Y_{n-2} C_{p,n-1}, C_{p,n-1} C_{p,n} \longrightarrow Y_{n-1} C_{p,n}, \\ & Y_{n-1} C_{p,n} \longrightarrow Y_{n-1} D, D \longrightarrow Y_n Y_{n+1} \dots Y_m \}. \end{aligned}$$

Die Mengen N_p und P_p entstehen nun aus N'_p und P'_p indem wir $D \in N'_p$ und $D \longrightarrow Y_n Y_{n+1} \dots Y_m \in P'_p$ entsprechend Fall 2 durch Nichtterminale und Regeln mit einer rechten Seite der Länge ≤ 2 ersetzen.
Wir konstruieren $G' = (N', T, P', S)$ durch

$$N' = N \cup \bigcup_{p \in P} N_p \qquad \text{und} \qquad P' = \bigcup_{p \in P} P_p.$$

Aus der Konstruktion ist sofort zu sehen, dass alle Regeln von P' von der geforderten Form sind.

Sei nun $v = w_1X_1X_2 \dots X_nw_2$ mit $w_1, w_2 \in V^*$ und $n \geq 2$ eine Satzform von G. Durch Anwendung von p entsteht $v' = w_1Y_1Y_2 \dots Y_mw_2$. In G' haben wir dann die folgende Ableitung

$$\begin{aligned}
v &\Longrightarrow w_1C_{p,1}X_2X_3 \dots X_nw_2 \Longrightarrow w_1C_{p,1}C_{p,2}X_3 \dots X_nw_2 \\
&\Longrightarrow \dots \Longrightarrow w_1C_{p,1}C_{p,2} \dots C_{p,n-1}X_nw_2 \Longrightarrow w_1C_{p,1}C_{p,2} \dots C_{p,n-1}C_{p,n}w_2 \\
&\Longrightarrow w_1Y_1C_{p,2} \dots C_{p,n-1}C_{p,n}w_2 \Longrightarrow w_1Y_1Y_2 \dots C_{p,n-1}C_{p,n}w_2 \\
&\Longrightarrow \dots \Longrightarrow w_1Y_1Y_2 \dots Y_{n-1}C_{p,n}w_2 \\
&\Longrightarrow w_1Y_1Y_2 \dots Y_{n-1}D_{p,n}w_2 \Longrightarrow w_1Y_1Y_2 \dots Y_{n-1}Y_nY_nD_{p,n+1}w_2 \\
&\Longrightarrow w_1Y_1Y_2 \dots Y_{n-1}Y_nY_{n+1}D_{p,n+2}w_2 \Longrightarrow \dots \\
&\Longrightarrow w_1Y_1Y_2 \dots Y_{n-1}Y_nY_{n+1} \dots Y_{m-1}D_{p,m}w_2 \\
&\Longrightarrow w_1Y_1Y_2 \dots Y_{n-1}Y_nY_{n+1} \dots Y_{m-1}Y_mw_2 = v',
\end{aligned}$$

wobei wir die Regeln aus P_p genau in der in Fall 3 angegebenen Reihenfolge anwenden. Damit ist gezeigt, dass wir die Anwendung von p in G durch Anwendung der Regeln aus P_p in G' simulieren können. Analoges gilt auch in den Fällen 1 und 2. Damit kann jede Ableitung in G in G' simuliert werden.
Wir zeigen nun, dass bis auf die Reihenfolge in der Anwendung von Regeln in G' nur derartige Simulationen möglich sind. Dies sieht man wie folgt ein: Wenden wir auf v die Regel $X_1X_2 \longrightarrow C_{p,1}X_2$ an, so können wir auf die entstehende Satzform $v_1 = w_1C_{p,1}X_2 \dots X_nw_2$ nur die Regel $C_{p,1}X_2 \longrightarrow C_{p,1}C_{p,2}$ aus P_p anwenden. Wir setzen dann die Ableitung mittels Regeln aus P_p wie oben fort oder durch Anwendung von $C_{p,1}C_{p,2} \longrightarrow Y_1C_{p,2}$ fort, wodurch $w_1Y_1C_{p,2}X_3 \dots X_nw_2$ entsteht. Auf letztere Satzform ist nur $C_{p,2}X_3 \longrightarrow C_{p,2}C_{p,3}$ anwendbar, wodurch $w_1Y_1C_{p,2}C_{p,3}X_4 \dots X_nw_2$ generiert wird. Auch nun gibt es die Möglichkeit durch Regeln aus P_p das Symbol $C_{p,2}$ durch Y_2 oder X_4 durch $C_{p,4}$ zu ersetzen. Man erkennt also, dass bis auf die Reihenfolge der Regeln schließlich $w_1Y_1 \dots Y_{n-1}D_{p,n}w_2$ erzeugt wird. Nun sind die folgenden anwendbaren Regeln stets eindeutig bestimmt, und wie oben wird v' abgeleitet.
Wir haben noch zu diskutieren, was passiert, wenn auf eine Satzform, die während dieser Simulation entsteht, eine Regel angewendet wird, die nicht zu P_p gehört und mindestens eines der Symbole $X_1, X_2, X_3, \dots, X_n$ verändert. Wir diskutieren dies nur für v_1; die Überlegungen bei den anderen Satzformen sind ähnlich. Es ist leicht zu sehen, dass die Regeln zur Änderung von Symbolen aus $N_p \setminus \{D_{p,m}\}$ (und mindestens das in v_1 vorkommende $C_{p,1} \in N_p$ ist zu ändern, damit die Ableitung auf ein Wort über T führt) ein weiteres Symbol aus N_p einführt. Damit kann v_1 nur dann in ein terminales Wort überführt werden, wenn nach einigen Schritten nur noch $D_{p,m}$ in der Satzform ist und $D_{p,m} \longrightarrow Y_m$ angewendet wird. Dies erfordert aber, dass alle Regeln aus P_p angewendet wurden und damit die Anwendung von p in G simuliert wurde.
Da somit in G' alle direkten Ableitungen in G simuliert werden können und nur Simulationen von Ableitungen in G möglich sind, gilt für Wörter w, w' über $N \cup T$, dass $w \Longrightarrow_G^* w'$ genau dann gilt, wenn auch $w \Longrightarrow_{G'}^* w'$ gültig ist. Hieraus folgt $S \Longrightarrow_G^* w$ mit $w \in T^*$ gilt genau dann, wenn $S \Longrightarrow_{G'}^* w$ gültig ist. Dies impliziert $L(G) = L(G')$. □

Folgerung 2.17 $\mathcal{L}(MON) = \mathcal{L}(CS)$.

Beweis. Am Ende von Abschnitt 2.1.1 wurde bereits bemerkt, dass $\mathcal{L}(CS) \subseteq \mathcal{L}(MON)$ gilt.
Wir haben also nur $\mathcal{L}(MON) \subseteq \mathcal{L}(CS)$ zu zeigen, d.h. wir müssen nachweisen, dass jede monotone Sprache auch kontextabhängig ist. Sei L eine monotone Sprache. Dann gibt es eine monotone Grammatik G mit $L = L(G)$. Nach Satz 2.16. gibt es dann eine monotone Grammatik G', deren Regeln alle von kontextabhängiger Form sind, d.h. G' ist kontextabhängig, und die $L = L(G) = L(G')$ erfüllt. Folglich ist L eine kontextabhängige Sprache. □

Entsprechend Satz 2.16 wird jede kontextfreie Sprache durch eine Grammatik erzeugt, die nur Regeln der Form

$$A \to BC, A \to B, A \to \lambda \text{ und } A \to a \text{ mit } A, B, C \in N, a \in T$$

hat. Wir zeigen nun, dass auch die Regeln der Form $A \to \lambda$ eliminiert werden können, wobei wir dann natürlich die gleiche Ausnahmeregelung zulassen müssen, die uns schon von den monotonen oder kontextabhängigen Grammatiken geläufig ist.

Lemma 2.18 *Zu jeder kontextfreien Grammatik $G = (N, T, P, S)$ existiert eine kontextfreie Grammatik $G' = (N', T, P', S)$ derart, dass*
i) P' keine Regel der Form $A \longrightarrow \lambda$ mit $A \neq S$ enthält,
ii) $|w|_S = 0$ für alle Regeln $A \longrightarrow w \in P'$ gilt, und
iii) $L(G) = L(G')$ ist.

Beweis. Wir konstruieren als erstes zu der Grammatik $G = (N, T, P, S)$ eine kontextfreie Grammatik $G'' = (N'', T, P'', S')$, die die Bedingung ii) und $L(G) = L(G'')$ erfüllt. Dazu fügen wir zu N ein neues Nichtterminal S' hinzu, d.h. wir setzen $N'' = N \cup \{S'\}$. Weiterhin erweitern wir die Regelmenge durch $P'' = P \cup \{S' \to S\}$. ii) gilt dann nach Definition. Da alle Ableitungen in G'' von der Form $S'' \Longrightarrow S \Longrightarrow^* w$ sind, haben wir auch $L(G'') = L(G)$.
Es sei

$$M = \{A : A \in N'', A \Longrightarrow^* \lambda\}.$$

Mit jeder Regel

$$q'' = A \longrightarrow v_1 A_1 v_2 A_2 \dots v_m A_m v_{m+1}$$

mit

$$m \geq 0, A_1, A_2, \dots, A_m \in N'', v_1, v_2, \dots, v_{m+1} \in T^*$$

assoziieren wir die Menge $P_{q''}$ aller Regeln der Form

$$A \longrightarrow v_1 X_1 v_2 X_2 \dots v_m X_m v_{m+1} \neq \lambda,$$

für die

$$X_i = A_i \text{ für } A_i \notin M \quad \text{und} \quad X_i \in \{A_i, \lambda\} \text{ für } A_i \in M$$

für $1 \leq i \leq m$ gilt. Aufgrund dieser Definition kann keine Menge $P_{q''}$ eine Regel der Form $Y \longrightarrow \lambda$ enthalten. Damit ist es nicht möglich das Leerwort unter Verwendung von Regeln aus $P_{q''}$ zu erzeugen. Deshalb setzen wir

$$\overline{P} = \begin{cases} \{S' \longrightarrow \lambda\} & \text{falls } S' \in M \\ \emptyset & \text{sonst} \end{cases}.$$

Weiterhin definieren wir $G' = (N', T, P', S')$ durch

$$N' = N'' \quad \text{und} \quad P' = \overline{P} \cup \bigcup_{q'' \in P''} P_{p''}.$$

Wir bemerken, dass bei der Konstruktion von P' aus P'' die Eigenschaft ii) erhalten geblieben ist, und dass P' nach Konstruktion die Eigenschaft i) hat.
Wir zeigen jetzt, dass auch die Bedingung iii) erfüllt ist. Dafür reicht es $L(G'') = L(G')$ zu zeigen.

Zuerst beweisen wir mittels vollständiger Induktion über die Anzahl der Ableitungsschritte, dass für jedes Nichtterminal A und jedes Wort $x \in T^+$ mit $A \Longrightarrow^*_{G''} x$ auch $A \Longrightarrow^*_{G'} x$ gilt.
Sei $n = 1$. Jede direkte Ableitung ist in beiden Grammatiken von der Form $A \Longrightarrow v$, bei der in beiden Fällen die Regel $A \longrightarrow v$ angewendet wird. Somit ist der Induktionsanfang gezeigt.
Sei nun x ein in $n \geq 2$ Schritten aus A ableitbares terminales Wort. Dann gilt

$$A \Longrightarrow_{G''} v_1 A_1 v_2 A_2 \dots v_m A_m v_{m+1} \Longrightarrow^*_{G''} v_1 x_1 v_2 x_2 \dots v_m x_m v_{m+1} = x,$$

wobei die Ableitungen $A_i \Longrightarrow^*_{G''} x_i$ für $1 \leq i \leq m$ sämtlich aus weniger als n Schritten bestehen. Wir unterscheiden nun zwei Fälle:
Fall 1. $x_i \neq \lambda$. Dann setzen wir $X_i = A_i$ und haben nach Induktionsannahme $X_i = A_i \Longrightarrow^*_{G'} x_i$.
Fall 2. $x_i = \lambda$. Dann gilt $A_i \in M$ und wir setzen $X_i = \lambda$.
Nach Konstruktion gibt es in P' die Regel $A \longrightarrow v_1 X_1 v_2 X_2 \dots v_m X_m v_{m+1}$ und wir erhalten in G' die Ableitung

$$A \Longrightarrow_{G'} v_1 X_1 v_2 X_2 \dots v_m X_m v_{m+1} \Longrightarrow^*_{G'} v_1 x_1 v_2 x_2 \dots v_m x_m v_{m+1},$$

wobei wir für $x_i = \lambda$ einfach $X_i = x_i = \lambda$ und für $x_i \neq \lambda$ die Ableitungen $X_i \Longrightarrow^*_{G'} x_i$ benutzen.
Betrachten wir die gerade bewiesene Aussage für $A = S$, so ist jedes vom Leerwort verschiedene Wort aus $L(G'')$ auch in G' ableitbar. Damit gilt $L(G'') \setminus \{\lambda\} \subseteq L(G') \setminus \{\lambda\}$. Da durch $\overline{P}$ gesichert ist, dass $\lambda \in L(G'')$ genau dann gilt, wenn $\lambda \in L(G')$ ist, ist sogar $L(G'') \subseteq L(G')$ gültig.

Wir zeigen nun wiederum mittels vollständiger Induktion die Umkehrung, d.h., dass jede Ableitung $A \Longrightarrow^*_{G'} y$ eines terminalen Wortes y auch eine Entsprechung $A \Longrightarrow^*_{G''} y$ findet.
Der Induktionsanfang ergibt sich wie oben.
Sei daher $A \Longrightarrow^*_{G'} y$ eine Ableitung aus $n \geq 2$ Schritten. Dann gilt

$$A \Longrightarrow v_1 X_1 v_2 X_2 \dots v_m X_m v_{m+1} \Longrightarrow^*_{G'} v_1 x_1 v_2 x_2 \dots v_m x_m v_{m+1},$$

wobei für $X_i = \lambda$ auch $x_i = \lambda$ ist, und für $X_i \neq \lambda$ ist $X_i \Longrightarrow^*_{G'} x_i$ eine Ableitung mit weniger als n Schritten. Nach Konstruktion der Regel $A \longrightarrow v_1X_1v_2X_2 \ldots v_mX_mv_{m+1}$ aus P' gibt es dann eine Ableitung $A_i \Longrightarrow^* \lambda = x_i$, falls $X_i = \lambda$ ist, und nach Induktionsvoraussetzung gilt auch $A_i \Longrightarrow^*_{G''} x_i$ für $X_i \neq \lambda$. Deshalb existiert in G'' die Ableitung

$$A \Longrightarrow_{G''} v_1A_1v_2A_2 \ldots v_mA_mv_{m+1} \Longrightarrow^*_{G'} v_1x_2v_2x_2 \ldots v_mx_mv_{m+1}.$$

Hiervon ausgehend zeigt man wie oben $L(G') \subseteq L(G'')$. □

Um die Grammatik G' aus dem vorstehenden Beweis wirklich konstruieren zu können, benötigen wir einen Algorithmus, der die Menge M bestimmt. Wir setzen

$$\begin{aligned}
M_0 &= \emptyset, \\
P_0 &= P, \\
M_i &= M_{i-1} \cup \{A : A \in N'', A \to \lambda \in P_{i-1}\}, \\
P_i &= \{A \to w_1w_2 \ldots w_{n+1} : A \to w_1A_1w_2A_2 \ldots w_nA_nw_{n+1} \in P_{i-1} \\
&\quad n \geq 0, w_j \in (N'' \setminus M_i)^* \text{ für } 1 \leq j \leq n+1, A_j \in M_i \text{ für } 1 \leq j \leq n\}
\end{aligned}$$

für $i \geq 1$. Für $i \geq 1$ erfordert die Konstruktion von M_i das Durchmustern aller Regeln von P_{i-1}, ob sie von der Form $A \to \lambda$ sind, und die Konstruktion von P_i das Ersetzen aller Symbole aus M_i durch das Leerwort in allen Regeln von P.
Wir zeigen zuerst mittels Induktion $M_i \subseteq M$ für $i \geq 0$. Für $i = 0$ und $i = 1$ ist dies nach Konstruktion klar. Für $A \in M_i$, $i \geq 2$, gibt es nach Definition von M_i eine Regel $A \to A_1A_2 \ldots A_n$ mit $A_j \in M_{i-1}$ für $1 \leq j \leq n$. Da nach Induktionsvoraussetzung $A_j \in M$ für $1 \leq j \leq n$ gilt, gibt es die Ableitung

$$A \Longrightarrow A_1A_2 \ldots A_n \Longrightarrow^* \lambda A_2A_3 \ldots A_n \Longrightarrow^* \lambda\lambda A_3 \ldots A_n \Longrightarrow^* \lambda^n = \lambda,$$

woraus $A \in M$ folgt.
Sei nun $A \in M$. Wir betrachten eine Ableitung $A \Longrightarrow^* \lambda$. In keiner Satzform dieser Ableitung kann ein Terminal vorkommen, die Satzformen sind also alle Wörter über N''. Durch Umordnen der Ableitungsschritte können wir eine Ableitung

$$A = w_0 \Longrightarrow^* w_1 \Longrightarrow^* w_2 \Longrightarrow^* \ldots \Longrightarrow^* w_m = \lambda$$

erreichen, bei der $w_{i-1} \Longrightarrow^* w_i$ dadurch entsteht, dass alle Nichtterminale aus w_{i-1} entsprechend einer Regel ersetzt werden. Offenbar gilt dann $w_{m-1} \in M_1^*$, da die darin enthaltenen Nichtterminale in einem Ableitungsschritt durch das Leerwort ersetzt werden. Für ein Nichtterminal B aus w_{m-2} gilt daher $B \to \lambda$ oder $B \to w \in M_1^+$, womit sich $B \in M_1$ oder $B \in M_2$ und damit sicher $B \in M_2$ ergibt. So fortfahrend erhalten wir $w_{m-3} \in M_3^*$, $w_{m-4} \in M_4^*$ und schließlich $A = w_0 = w_{m-m} \in M_m$.
Aus dem bisher Bewiesenem folgt

$$M = \bigcup_{i \geq 0} M_i.$$

Entsprechend den obigen Definitionen impliziert $M_i = M_{i+1}$ sofort $P_i = P_{i+1}$ und dann

$$M_i = M_{i+1} = M_{i+2} = \ldots \quad \text{und} \quad P_i = P_{i+1} = P_{i+2} = \ldots$$

Da außerdem stets $M_i \subseteq M_{i+1}$ gilt, tritt die Gleichheit spätestens bei M_t ein. Somit ergibt sich

$$M_t = \bigcup_{i \geq 0} M_i = M.$$

Beispiel 2.19 Wir illustrieren die eben beschriebene Konstruktion anhand der Grammatik

$$G = (\{S, A, B\}, \{a, b\}, \{S \to SA, S \to \lambda, A \to aAb, A \to B, B \to \lambda\}, S)\,.$$

Wir bemerken, dass

$$L(G) = \{a^{n_1} b^{n_1} a^{n_2} b^{n_2} \ldots a^{n_k} b^{n_k} : k \geq 0, n_i \geq 0, 1 \leq i \leq k\}$$

gilt, da durch die ersten beiden Regeln eine beliebige Anzahl von A's erzeugt wird, von denen jedes eine Sprache der Form $\{a^n b^n : n \geq 0\}$ erzeugt.
Es ergeben sich dann

$$\begin{aligned}
& N'' = N \cup \{S'\} = \{S, A, B, S'\}, \\
& P'' = \{S' \to S, S \to SA, S \to \lambda, A \to aAb, A \to B, B \to \lambda\} \\
& M_0 = \emptyset \text{ und } P_0 = P''\,, \\
& M_1 = \{S, B\} \text{ und } P_1 = \{S' \to \lambda, S \to A, S \to \lambda, A \to aAb, A \to \lambda, B \to \lambda\}\,, \\
& M_2 = \{S, B, S', A\} = N'' \\
& N' = N'' = \{S', S, A, B\}, \\
& \overline{P} = \{S \longrightarrow \lambda\}, \\
P' \;=\; & \overline{P} \cup \{S' \to S, S \to SA, S \to A, S \to S, A \to aAb, A \to ab\}, A \to B\}\,.
\end{aligned}$$

Man sieht sofort, dass P' offenbar überflüssige Regeln enthält. Dies trifft auf $S \to S$ zu, da diese Regel keine Änderung bei ihrer Anwendung bewirkt, und auf $A \to B$ zu, da P' keine Regeln enthält, die B auf der rechten Seite haben. Wir werden diese Regeln aber hier nicht streichen, da dies der Algorithmus im Beweis von Lemma 2.18 nicht vorsieht.

Es ist offenbar, dass – mit Ausnahme der eventuell existierenden Regel $S \longrightarrow \lambda$ – für alle anderen Regel $A \longrightarrow w \in P'$ bei der in Lemma 2.18 konstruierten Grammatik G' die Beziehung $w \in (N' \cup T)^+$ und damit $|w| \geq 1 = |A|$ gilt. Dies bedeutet, dass G' eine monotone Grammatik ist. Somit erhalten wir das folgende Resultat.

Folgerung 2.20 $\mathcal{L}(CF) \subseteq \mathcal{L}(MON)$. □

Wir zeigen nun, dass die in Satz 2.16 zugelassenen Regeln der Form $A \longrightarrow B$ mit $A, B \in N$ ebenfalls eliminiert werden können.

Lemma 2.21 *Zu jeder kontextfreien Grammatik $G = (N, T, P, S)$ kann eine kontextfreie Grammatik $G' = (N, T, P', S)$ so konstruiert werden, dass P' keine Regel der Form $A \longrightarrow B$ mit $A, B \in N$ enthält und $L(G) = L(G')$ gilt.*

Beweis. Nach Lemma 2.18 können wir ohne Beschränkung der Allgemeinheit annehmen, dass G – mit Ausnahme des möglichen Sonderfalles $S \to \lambda$ keine Regeln der Form $A \to \lambda$ enthält.
Für ein Nichtterminal A definieren wir

$$M_A = \{B : B \Longrightarrow_G^* A, B \in N\}$$

(man beachte, dass nach Definition stets $A \in M_A$ gilt). Für eine Regel $p = A \to w$ mit $w \notin N$ setzen wir

$$P_p = \{B \to w : B \in M_A\}$$

(d.h. wir ersetzen eine Ableitung

$$B \Longrightarrow B_1 \Longrightarrow B_2 \Longrightarrow \ldots \Longrightarrow B_k = A \Longrightarrow w$$

durch eine Regel $B \to w$). Wir setzen nun

$$P' = \bigcup_{p \in P} P_p.$$

Offensichtlich erfüllt P' nach Konstruktion die geforderte Bedingung. Die Gültigkeit von $L(G) = L(G')$ lässt sich nun in Analogie zum Beweis von Lemma 2.18. zeigen. □

Beispiel 2.22 Wenden wir die im Beweis von Lemma 2.21 gegebene Konstruktion auf Beispiel 2.19 an, so erhalten wir

$$M_B = \{B, A, S, S'\},\ M_A = \{A, S, S'\},\ M_S = \{S, S'\} \text{ und } M_{S'} = \{S'\}$$

und daher

$$\begin{aligned} P_{S' \to \lambda} &= \{S' \to \lambda\}, \\ P_{S \to SA} &= \{S \to SA, S' \to SA\}, \\ P_{A \to aAb} &= \{A \to aAb, S \to aAb, S' \to aAb\}, \\ P_{A \to ab} &= \{A \to ab, S \to ab, S' \to ab\} \end{aligned}$$

und die gesamte Regelmenge ergibt sich als Vereinigung der vier vorstehenden Mengen.

Wir geben nun die Normalform an, die auf N. Chomsky zurückgeht und durch Kombination der vorstehenden Normalform gewonnen werden kann.

Satz 2.23 *Zu jeder kontextfreien Grammatik $G = (N, T, P, S)$ kann eine kontextfreie Grammatik $G' = (N', T, P', S)$ so konstruiert werden, dass P' nur Regeln der Form*

$$A \longrightarrow BC \quad \textit{und} \quad A \longrightarrow a \quad \textit{mit} \quad A, B, C \in N',\ a \in T$$

enthält, wobei $S \longrightarrow \lambda$ als Ausnahme zugelassen ist, falls S in keiner rechten Seite einer Regel aus P' vorkommt, und $L(G) = L(G')$ gilt.

Beweis. Durch Nacheinanderausführung der Konstruktionen in den Beweisen von Lemma 2.15, Satz 2.16, Lemma 2.18 und Lemma 2.21 erreichen wir eine Grammatik, die keine Regeln der Form $A \longrightarrow w$ mit $|w| > 2$ oder $w = \lambda$ bei $A \neq S$ und keine der Form $A \longrightarrow B$ mit Nichtterminalen A und B enthält. □

Wir geben nun noch eine Normalform für reguläre Grammatiken.

Satz 2.24 *Zu jeder regulären Grammatik $G = (N, T, P, S)$ kann eine reguläre Grammatik $G' = (N', T, P', S)$ der Größe $O(\#(N) \cdot k(G))$ in der Zeit $O(\#(N) \cdot k(G))$ so konstruiert werden, dass P' nur Regeln der Form*

$$A \longrightarrow aB \quad \textit{und} \quad A \longrightarrow a \quad \textit{mit} \quad A, B \in N',\ a \in T$$

enthält, wobei $S \to \lambda$ als Ausnahme zugelassen ist, falls P' keine Regel der Form $A \to aS$ enthält, und $L(G) = L(G')$ gilt.

Beweis. Entsprechend Lemma 2.18 und 2.21 können wir ohne Beschränkung der Allgemeinheit annehmen, dass die Regelmenge P der gegebenen Grammatik $G = (N, T, P, S)$ unter Beachtung der Ausnahmeregel $S \longrightarrow \lambda$ und den damit verbundenen Bedingungen nur Regeln der Form $A \longrightarrow wB$ und $A \longrightarrow w$ mit $A, B \in N$, $w \in T^+$ enthält.
Mit der Regel

$$p = A \longrightarrow a_1 a_2 \dots a_n B \text{ mit } a_1, a_2, \dots, a_n \in T$$

assozieren wir nun die Menge

$$N_p = \{B_{p,1}, B_{p,2}, \dots, B_{p,n-1}\}$$

zusätzlicher Nichtterminale und die Menge

$$\begin{aligned} P_p \;=\; \{&A \longrightarrow a_1 B_{p,1}, B_{p,1} \longrightarrow a_2 B_{p,2}, B_{p,2} \longrightarrow a_3 B_{p,3}, \dots \\ &\dots, B_{p,n-2} \longrightarrow a_{n-1} B_{p,n-1}, B_{p,n-1} \longrightarrow a_n B\} \end{aligned}$$

von Regeln. Für die Regel

$$q = A \longrightarrow a_1 a_2 \dots a_n \text{ mit } a_1, a_2, \dots, a_n \in T$$

setzen wir ebenfalls

$$N_q = \{B_{q,1}, B_{q,2}, \dots, B_{q,n-1}\}$$

und

$$\begin{aligned} P_q \;=\; \{&A \longrightarrow a_1 B_{q,1}, B_{q,1} \longrightarrow a_2 B_{q,2}, B_{q,2} \longrightarrow a_3 B_{q,3}, \dots \\ &\dots, B_{q,n-2} \longrightarrow a_{n-1} B_{q,n-1}, B_{q,n-1} \longrightarrow a_n\}. \end{aligned}$$

Hierbei seien alle neu eingeführten Symbole wieder paarweise voneinander verschieden. Wir definieren dann $G' = (N', T, P', S)$ durch

$$N' = N \cup \bigcup_{r \in P} N_r \quad \text{und} \quad P' = \bigcup_{r \in P} P_r \cup \overline{P},$$

wobei $\overline{P}$ erneut genau dann die leere Menge ist, wenn $S \longrightarrow \lambda$ nicht in P liegt und sonst nur aus dieser Regel besteht. Es ist leicht zu sehen, dass durch die Anwendung der Regeln aus P_r in der in der Definition angegebenen Reihenfolge zu einer Simulation der Anwendung von r führt, und umgekehrt jede Anwendung einer Regel $A \longrightarrow a_1 B_{r,1}$ die Simulation von r zur Folge hat. Daher gilt $L(G) = L(G')$.
Die Aussagen zur Komplexität können in Analogie zu den entsprechenden Aussagen über kontextfreie Grammatiken bewiesen werden. Wir überlassen die Details dem Leser. □

Wir geben nun zwei Folgerungen aus den in Satz 2.23 und Satz 2.24 gegebenen Normalformen an, die es uns dann gestatten, zu beweisen, dass gewisse Sprachen nicht kontextfrei bzw. nicht regulär sind.
Für reguläre Sprachen leistet der folgende Satz das Gewünschte.

Satz 2.25 *Sei L eine reguläre Sprache. Dann gibt es eine (von L abhängige) Konstante k derart, dass es zu jedem Wort $z \in L$ mit $|z| \geq k$ Wörter u, v, w gibt, die den folgenden Eigenschaften genügen:*
i) $z = uvw$,
ii) $|uv| \leq k$, $|v| > 0$, und
iii) $uv^i w \in L$ für alle $i \geq 0$.

Beweis. Wegen Satz 2.24 können wir annehmen, dass $L = L(G)$ für eine reguläre Grammatik $G = (N, T, P, S)$ gibt, deren Regelmenge P (mit Ausnahme von vielleicht $S \longrightarrow \lambda$) nur Regeln der Form $A \longrightarrow aB$ und $A \longrightarrow a$ mit $A, B \in N$ und $a \in T$ enthält. Wir setzen dann $k = |N| + 1$.
Aufgrund der Form der Regeln aus P gibt es für ein Wort

$$z = a_1 a_2 \dots a_n \text{ mit } a_i \in T \text{ für } 1 \leq i \leq n \text{ und } n \geq k$$

eine Ableitung

$$\begin{aligned} S = A_0 \implies & a_1 A_1 \Longrightarrow a_1 a_2 A_2 \Longrightarrow a_1 a_2 a_3 A_3 \Longrightarrow \dots \\ & \Longrightarrow a_1 a_2 \dots a_{n-1} A_{n-1} \Longrightarrow a_1 a_2 \dots a_{n-1} a_n = z. \end{aligned}$$

Dann muss die Menge $\{A_0, A_1, A_2, \dots, A_{n-1}\}$ wegen der Wahl von k ein Nichtterminal doppelt enthalten. Es sei $A = A_i = A_j$ mit $0 \leq i < j \leq n-1$ und für A_t mit $t \leq i$ gelte $A_t \neq A_s$ für $t \neq s$. Wir setzen

$$u = a_1 a_2 \dots a_i,\ v = a_{i+1} a_{i+2} \dots a_j \text{ und } w = a_{j+1} a_{j+2} \dots a_n.$$

Man sieht sofort, dass die Bedingungen i) und ii) erfüllt sind.
Mit den eingeführten Bezeichnungen erhält die obige Ableitung von z die folgende Form

$$S = A_0 \Longrightarrow^* uA \Longrightarrow^* uvA \Longrightarrow^* uvw = z,$$

und wir haben überdies für $i \geq 2$ die Ableitungen

$$S = A_0 \Longrightarrow^* uA \Longrightarrow^* uvA \Longrightarrow^* uvvA \Longrightarrow^* uvvvA \Longrightarrow^* \dots \Longrightarrow^* uv^i A \Longrightarrow^* uv^i w \in T^*$$

und für $i = 0$ die Ableitung $S \Longrightarrow^* uA \Longrightarrow^* uw \in T^*$. Hieraus folgt $uv^i w \in L(G) = L$ für $i \geq 0$, womit auch Bedingung iii) nachgewiesen ist. □

Wir benutzen die Aussage von Satz 2.25, um zu zeigen, dass die kontextfreie Sprache

$$L = \{a^n b^n : n \geq 1\}$$

aus Beispiel 2.5 nicht regulär ist.
Wir zeigen dies indirekt. Sei also angenommen, dass L regulär ist. Ferner sei k die Konstante aus Satz 2.25 und $z = a^k b^k$. Dann gibt es eine Zerlegung $z = uvw$ von z mit

$$|uv| \leq k,\ |v| > 0, \text{ und } uv^i w \in L \text{ für alle } i \geq 1. \qquad (*)$$

Aus den beiden erstgenannten Bedingungen und $z = uvw$ folgen

$$u = a^r,\ v = a^s \text{ und } w = a^{k-r-s}b^k \text{ mit } r \geq 0 \text{ und } s \geq 1.$$

Damit folgt

$$uv^i w = a^r a^{is} a^{k-r-s} b^k = a^{k+(i-1)s} b^k.$$

Wegen der Form der Wörter in L ist daher $uv^i w \notin L$ für $i \geq 2$. Dies widerspricht aber der oben abgeleiteten Aussage in $(*)$.
Somit haben wir das folgende Lemma bewiesen.

Lemma 2.26 $L = \{a^n b^n : n \geq 1\} \in \mathcal{L}(CF) \setminus \mathcal{L}(REG)$. □

Wir wollen nun den Begriff eines *Ableitungsbaumes* t für eine Satzform $w \neq \lambda$ einer kontextfreien Grammatik $G = (N, T, P, S)$ einführen, den wir im Beweis des folgenden Resultats benötigen, der aber auch sonst zur Veranschaulichung von Ableitungen geeignet ist. Wir benutzen dafür vollständige Induktion über die Anzahl n der Schritte zur Ableitung von w und wir setzen voraus, dass G in der Normalform aus Lemma 2.18 ist, also keine Regeln $A \longrightarrow \lambda$ enthält.
Wir werden t als Paar (K, E) beschreiben, wobei K die Menge der Knoten und E die der Kanten bezeichnet. Wir werden die Konstruktion so gestalten, dass S die Wurzel des Baumes sein wird und die Blätter beim Lesen von links nach rechts die Satzform w ergeben.
Sei $n = 0$. Dann handelt es sich um die „Ableitung“ $S \Longrightarrow^* S$ von $w = S$. Der Ableitungsbaum ist für $n = 0$ der Baum, der keine Kanten enthält und dessen einziger Knoten S ist. S ist dann sowohl Wurzel als auch Blatt.
Sei $n = 1$. Dann hat die Ableitung die Form $S \Longrightarrow w$, wobei die Regel $S \longrightarrow w$ angewendet wird. Sei $w = x_1 x_2 \ldots x_m$ mit $x_i \in N \cup T$. Dann wird die Menge K der Knoten von t durch die Symbole $S, x_1, x_2, \ldots, x_m$ gebildet, und die Menge E besteht aus allen Kanten (S, x_i), $1 \leq i \leq m$. S ist dabei die Wurzel des Baumes, und $x_1, x_2, \ldots, x_m$ sind die Blätter von t. Dabei ordnen wir die Kanten so an, dass wir $w = x_1 x_2 \ldots x_m$ erhalten, wenn wir die Blätter von links nach rechts lesen.
Sei $n \geq 2$. Dann gibt es eine Ableitung

$$S \Longrightarrow^* u = y_1 y_2 \ldots y_s A z_1 z_2 \ldots z_r \Longrightarrow y_1 y_2 \ldots y_s x_1 x_2 \ldots x_m z_1 z_2 \ldots z_r = w,$$

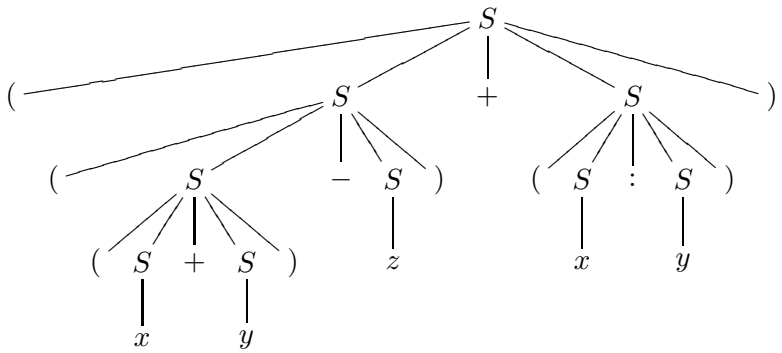
wobei $x_i, y_j, z_k \in N \cup T$ für $1 \leq i \leq m, 0 \leq j \leq s, 0 \leq k \leq r$ gilt. Die Ableitung $S \Longrightarrow^* u$ besteht dabei aus $n-1$ Schritten, und der zu ihr gehörende Ableitungsbaum $t' = (K', E')$

hat daher die Wurzel S und die Blätter ergeben von links nach rechts gelesen u. Wir konstruieren nun $t = (K, E)$ durch die Setzungen

$$K = K' \cup \{x_1, x_2, \ldots, x_m\} \quad \text{und} \quad E = E' \cup \{(A, x_i) : 1 \leq i \leq m\},$$

wobei wir die neuen Kanten wieder so anordnen, dass die Blätter von links nach rechts gelesen gerade w ergeben.

Zur Illustration geben wir den Ableitungsbaum für das Wort $(((x+y)-z)+(x:y))$, das von der in Beispiel 2.9 gegebenen Grammatik G_6 erzeugt wird. Dabei schreiben wir unter den Baum noch einmal die Blätter, um zu dokumentieren, dass sie von links nach rechts gelesen die zur Diskussion stehende Satzform ergeben.

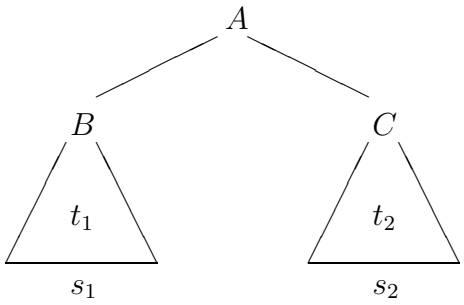


(((x + y) − z) + (x : y))

Wir geben jetzt ein Analogon zu Satz 2.25 für kontextfreie Sprachen.

Satz 2.27 *Sei L eine kontextfreie Sprache. Dann gibt es eine (von L abhängige) Konstante k derart, dass es zu jedem Wort $z \in L$ mit $|z| \geq k$ Wörter u, v, w, x, y gibt, die folgenden Eigenschaften genügen:*
i) $z = uvwxy$,
ii) $|vwx| \leq k$, $|v| + |x| > 0$, und
iii) $uv^iwx^iy \in L$ für alle $i \geq 0$.

Beweis. Wegen Satz 2.23 können wir annehmen, dass $L = L(G)$ für eine kontextfreie Grammatik $G = (N, T, P, S)$ in CHOMSKY-Normalform gilt. Es sei $n = |N|$. Dann setzen wir $k = 2^n$.
Sei $A \Longrightarrow^* s \in T^*$ eine Ableitung, deren zugehöriger Ableitungsbaum die Tiefe m hat. Wir zeigen zuerst mittels vollständiger Induktion über die Tiefe m des Ableitungsbaumes, dass dann $|s| < 2^m$ gilt.
$m = 1$. Die Ableitung kann nur aus einem Schritt bestehen und ist folglich aufgrund der CHOMSKY-Normalform $A \Longrightarrow \lambda$ oder $A \Longrightarrow a$ für ein $a \in T$. Dies bedeutet aber $s = \lambda$ oder $s = a$, woraus sofort $|s| \leq 1 < 2 = 2^1$ folgt. Damit ist der Induktionsanfang gezeigt.
Sei nun $A \Longrightarrow w$ eine Ableitung mit einem Ableitungsbaum t der Tiefe $m \geq 2$. Dann hat t wegen der CHOMSKY-Normalform die Form



wobei t_1 und t_2 Ableitungsbäume mit einer maximalen Tiefe $m-1$ sind und $s_1 s_2 = s$ gilt. Nach Induktionsannahme gilt dann

$$|s| = |s_1| + |s_2| < 2^{m-1} + 2^{m-1} = 2^m,$$

womit auch die Induktionsbehauptung gezeigt ist.
Wir benutzen die gerade bewiesene Aussage für Wörter $z \in L$ mit $|z| \geq k$. Sie liefert, dass der zu z gehörige Ableitungsbaum t' entsprechend der obigen Wahl von k eine Tiefe $m \geq n+1$ hat. Damit hat t' die Form gemäß Abbildung 2.2.

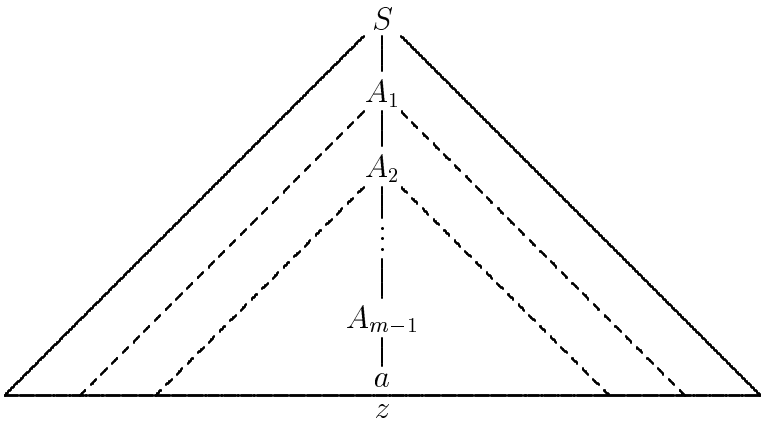


Abbildung 2.2:

Nun müssen wegen $m-1 \geq n$ mindestens zwei Elemente aus $\{S, A_1, A_2, \ldots, A_{m-1}\}$ identisch sein. Sei A dieses Nichtterminal. Damit ergibt sich für t' dann die Form gemäß Abbildung 2.3.
Dabei gilt $vx \neq \lambda$, da G eine Grammatik in CHOMSKY-Normalform ist. Weiterhin können wir ohne Beschränkung der Allgemeinheit annehmen, dass $|vwx| \leq k$ ist, da sonst im Ableitungsbaum zu $A \Longrightarrow^* vwx$ ein Weg existiert, auf dem ein Nichtterminal A' doppelt auftritt, und wir könnten dann mit A' anstelle von A argumentieren. Damit sind die Bedingungen i) und ii) nachgewiesen.
Ferner entnehmen wir dem Ableitungsbaum t' auch die Existenz der folgenden Ableitungen:

$$S \Longrightarrow^* uAy, \qquad A \Longrightarrow^* vAx, \qquad A \Longrightarrow^* w.$$

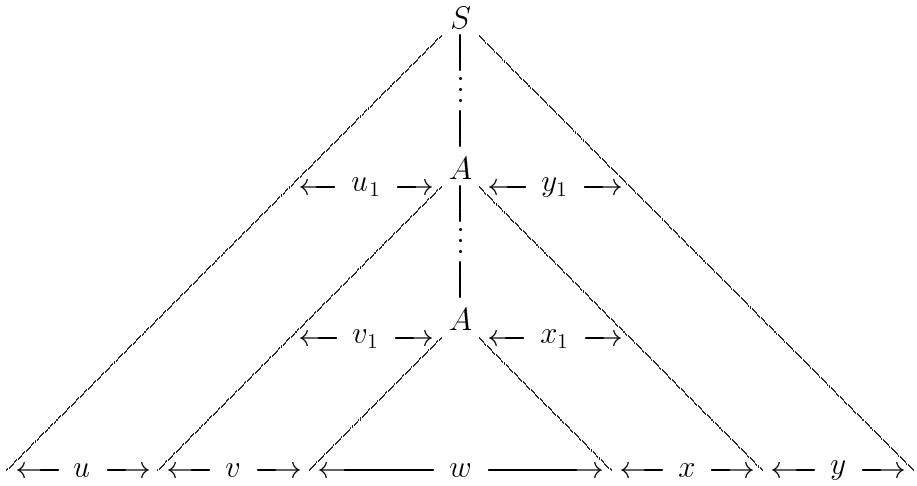


Abbildung 2.3:

Damit gibt es auch die Ableitung

$$S \Longrightarrow^* uAy \Longrightarrow^* uvAxy \Longrightarrow^* uvvAxxy \Longrightarrow^* uvvvAxxxy \Longrightarrow^* \ldots$$
$$\ldots \Longrightarrow^* uv^iAx^iy \Longrightarrow^* uv^iwx^iy$$

für $i \geq 0$ (für $i = 1$ entsteht gerade z). Da diese Ableitung zu einem Wort aus T^* führt, gilt $uv^iwx^iy \in L(G) = L$ für $i \geq 0$, womit auch Bedingung iii) nachgewiesen ist. □

Die in Satz 2.25 und 2.27 gegebenen Aussagen heißen *Schleifensätze* (oder auch *Pumping-Sätze*), da sie im Wesentlichen besagen, dass gewisse Ableitungen wie eine Schleife beliebig oft hintereinander ausgeführt werden können (oder einige Teilwörter „aufgepumpt“ werden können, z.B. v zu v^i).

Wir benutzen nun Satz 2.27, um ein zu Lemma 2.26 analoges Resultat herzuleiten.

Lemma 2.28 $L = \{a^nb^nc^n : n \geq 1\} \in \mathcal{L}(MON) \setminus \mathcal{L}(CF)$.

Beweis. Die Aussage $L \in \mathcal{L}(MON)$ folgt aus Beispiel 2.9.
Wir nehmen nun an, dass L kontextfrei ist. Nach Satz 2.27 gibt es dann eine Konstante k und für $z = a^kb^kc^k$ eine Zerlegung $z = uvwxy$ mit den in Satz 2.27 genannten Eigenschaften. Wir betrachten im Folgenden nur den Fall $v \neq \lambda$; die Überlegungen für $v = \lambda, x \neq \lambda$ verlaufen analog.
Wir unterscheiden die folgenden Fälle (wegen $|vwx| \leq k$ ist diese Fallunterscheidung vollständig):
Fall 1. $v = a^rb^s$ mit $r \geq 1, s \geq 0$. Wegen $|vwx| \leq k$ enthält vwx kein Vorkommen von c. Damit enthält uv^2wx^2y mindestens $k + r > k$ Vorkommen des Buchstaben a, aber nur k Vorkommen von c. Aufgrund der Form der Wörter in L, ergibt sich daraus $uv^2wx^2y \notin L$ im Widerspruch zur Eigenschaft iii) aus Satz 2.27.
Fall 2. $v = b^sc^t$ mit $s \geq 1, t \geq 0$. Dann enthält vwx kein Vorkommen von a, und daher gelten $|uv^2wx^2y|_a = k$ und $|uv^2wx^2y|_b \geq k + s > k$, womit sich in Analogie zum Fall 1 ein Widerspruch ergibt.
Fall 3. $v = c^t$ mit $t \geq 1$. Erneut enthält vwx kein Vorkommen von a, und daher gelten $|uv^2wx^2y|_a = k$ und $|uv^2wx^2y|_c \geq k + t > k$, womit sich in Analogie zum Fall 1 ein Widerspruch ergibt. □

Wir kombinieren nun die Aussagen der Lemmata 2.14, 2.26 und 2.28 und der Folgerungen 2.17 und 2.20 und erhalten den folgenden Satz. Man entnimmt ihm, dass die in Definition 2.12 eingeführten Sprachmengen eine Hierarchie bilden, die nach N. CHOMSKY benannt wird.

Satz 2.29 $\mathcal{L}(REG) \subset \mathcal{L}(CF) \subset \mathcal{L}(CS) = \mathcal{L}(MON) \subseteq \mathcal{L}(RE)$. □

Entsprechend Satz 2.29 ist also nur noch die Bestimmung der genauen Relation zwischen $\mathcal{L}(RE)$ und $\mathcal{L}(MON)$ offen. Wir werden die Klärung dieses Problems erst im Kapitel 2.4 herbeiführen.

2.2 Sprachen als akzeptierte Wortmengen

Zur Beschreibung von Sprachen haben wir im vorangehenden Abschnitt Grammatiken benutzt; bei diesen werden die Worte der Sprache mittels eines Ableitungsprozesses aus einem Startwort generiert. Ein grundsätzlich anderes Vorgehen liegt der Beschreibung von Sprachen durch Automaten zugrunde. Hier wird ein Wort als Eingabe verwendet und der Automat sagt „ja“, falls das Wort zu der Sprache gehört, und „nein“, falls das Wort nicht zu der Sprache gehört. Dies erinnert an die Funktionen, die mit Entscheidungsproblemen verbunden sind und im ersten Kapitel (insbesondere in Abschnitt 1.2) untersucht wurden. Wir werden hier eine Modifikation des dortigen Vorgehens betrachten, die sich von der in Kapitel 1 dadurch unterscheidet, dass die Antwort „ja“ oder „nein“ nicht der Ausgabe des Automaten entnommen wird, sondern mittels der Zustände gegeben wird, da die Ausgabe in diesem Zusammenhang nicht von Bedeutung ist.

2.2.1 TURING-Maschinen als Akzeptoren

Wir formalisieren den oben beschriebenen Ansatz.

Definition 2.30 *Eine akzeptierende* TURING-*Maschine M ist ein Sechstupel*

$$M = (X, Z, z_0, Q, \delta, F)\,,$$

wobei X, Z, z_0, Q und δ wie in Definition 1.15 gegeben sind und $F \subseteq Q$ gilt. Die von M akzeptierte Sprache $T(M)$ wird durch

$$T(M) = \{w : w \in X^*,\ (\lambda, z_0, w) \models^* (v_1, q, v_2) \text{ für ein } q \in F\}$$

definiert.

Liegt ein Wort w in $T(M)$ für eine TURING-Maschine M, so sagen wir, dass w von M akzeptiert wird. F heißt die Menge der akzeptierenden Zustände.
Wir bemerken, dass es zwei Möglichkeiten gibt, die dazu führen, dass ein Wort w nicht akzeptiert wird: entweder die TURING-Maschine stoppt bei Eingabe von w nicht, oder sie stoppt in einem Zustand $q \in Q \setminus F$.

Beispiel 2.31 Wir betrachten Modifikationen M_1' und M_2' der TURING-Maschinen M_1 und M_2 aus Beispiel 1.19.
M_1 merkt sich den ersten Buchstaben, löscht diesen und fügt ihn ans Ende des Wortes an. Bei M_1' betrachten wir statt eines Stopzustandes q in M_1 zwei Stopzustände q_a und q_b in Abhängigkeit davon, ob sich a oder b gemerkt und an das Wort angefügt wurde. Formal ergibt dies die TURING-Maschine

$$M_1' = (\{a,b\}, \{z_0, z_a, z_b, q_a, q_b\}, z_0, \{q_a, q_b\}, \delta', \{q_a\})$$

mit δ aus Abb. 2.4. M_1' erreicht wie M_1 aus Beispiel 1.19 stets einen Stopzustand, akzep-

δ	z_0	z_a	z_b
$*$	$(q, *, N)$	(q_a, a, N)	(q_b, b, N)
a	$(z_a, *, R)$	(z_a, a, R)	(z_b, a, R)
b	$(z_b, *, R)$	(z_a, b, R)	(z_b, b, R)

Abbildung 2.4: Überführungsfunktion von M_1'

tiert aber nur die Wörter, bei denen sich die Maschine a gemerkt (und schließlich auf das Band geschrieben) hat. Folglich erhalten wir

$$T(M_1') = \{aw \mid w \in \{a,b\}^*\}\,.$$

Um M_2' aus M_2 zu erhalten legen wir nur die Menge der akzeptierenden Zustände fest. Wir wollen in jedem Stoppzustand akzeptieren, d.h. wir setzen

$$M_2' = (\{a,b\}, \{z_0, z_1, q\}, z_0, \{q\}, \delta, \{q\}),$$

wobei δ wie bei M_2 aus Beispiel 1.19 gegeben sei. Entsprechend den Betrachtungen für M_2 in Beispiel 1.19 erhalten wir

$$T(M_2') = \{w : w \in \{a,b\}^*, |w| \; \textit{ungerade}\}.$$

Bei der Behandlung von TURING-Machinen im Abschnitt 1.1.2 haben wir eine Normalform hergeleitet, bei der nur ein Stoppzustand benutzt wurde. Wir wollen nun ein analoges Resultat für akzeptierende TURING-Maschinen angeben.

Lemma 2.32 *Zu jeder akzeptierenden* TURING*-Maschine M gibt es eine akzeptierende* TURING*-Maschine M', deren Menge der Stopzustände mit der Menge der akzeptierenden Zustände übereinstimmt und für die $T(M) = T(M')$ gilt. Dabei kann die Menge der Stoppzustände von M' einelementig gewählt werden.*

Beweis. Sei $M = (X, Z, z_0, Q, \delta, F)$ eine TURING-Maschine. Wir konstruieren aus M die TURING-Maschine $M' = (X, Z', z_0, \{q\}, \delta', \{q\})$ mit

$$\begin{aligned}
Z' &= Z \cup \{q\} \text{ wobei } q \notin Z,\\
\delta'(z,x) &= \delta(z,x) \text{ für } z \in Z \setminus Q,\\
\delta'(z,x) &= (z,x,N) \text{ für } z \in Q \setminus F,\ x \in X \cup \{*\},\\
\delta'(z,x) &= (q,x,N) \text{ für } z \in F.
\end{aligned}$$

Entsprechend diesen Setzungen
- verhält sich M' wie M solange M keinen seiner Stopzustände erreicht hat,
- geht M' in eine Schleife, wenn M einen nichtakzeptierenden Stopzustand erreicht hat,
- stoppt M' nach einem weiteren Schritt, wenn M einen akzeptierenden Stopzustand erreicht hat.
Hieraus folgt $T(M') = T(M)$ sofort. □

Aus Lemma 2.32 folgt sofort der folgende Satz, der die Verbindung zu den Betrachtungen aus Abschnitt 1.1.2 herstellt.

Satz 2.33 *Eine Sprache wird genau dann von einer* TURING*-Maschine akzeptiert, wenn sie Definitionsbereich einer* TURING*-berechenbaren Funktion ist.* □

Lemma 2.32 legt die Frage nahe, warum in der Definition 2.30 der akzeptierenden TURING-Maschine die Menge der akzeptieenden Zustände eingeführt wurde. Beim Beweis der Normalform wurden die nichtakzeptierenden Stoppzustände einfach in Zustände überführt, in denen die Maschine nicht stoppt. Dies verbietet sich aber dann, wenn – wie bei anderen Typen von Automaten – stets eine Stoppsituation eintritt oder man für jede Eingabe eine Antwort braucht. In diesen Fällen muss dann die Akzeptanz bzw. Nichtakzeptanz allein mittels der Zustände geschehen können. Die akzeptierenden Zustände entsprechen dem „ja“ und die nichtakzeptierenden dem „nein“.
Fordert man stets ein Erreichen eines Stoppzustandes bei TURING-Maschinen kommt man zum Begriff der rekursiven Sprache.

Definition 2.34 *Eine Sprache $L \subseteq X^*$ heißt rekursiv, falls es eine akzeptierende* TURING*-Maschine $M = (X, Z, z_0, Q, \delta, F)$ gibt, die auf jeder Eingabe stoppt und L akzeptiert.*

Satz 2.35 *Eine Sprache $L \subseteq X^*$ ist genau dann rekursiv, wenn sowohl L als auch $X^* \setminus L$ von* TURING*-Maschinen akzeptiert werden.*

Beweis. Es sei zuerst L eine rekursive Sprache. Dann gibt es eine akzeptierende TURING-Maschine $M = (X, Z, z_0, Q, \delta, F)$, die L akzeptiert und auf jeder Eingabe stoppt. Die akzeptierende TURING-Maschine $M' = (X, Z, z_0, Q, \delta, Q \setminus F)$ akzeptiert dann offenbar genau die Eingaben, die von M verworfen werden. Damit gilt $T(M') = X^* \setminus L$.
Es seien nun L und $X^* \setminus L$ von den akzeptierenden TURING-Maschinen N und N' akzeptiert. Wir nehmen ohne Beschränkung der Allgemeinheit an, dass N und N' in der Normalform aus Lemma 2.32 gegeben sind. Wir betrachten dann die akzeptierende TURING-Maschine N'', die wie folgt arbeitet: Zuerst schreibt N'' eine Kopie des Eingabewortes hinter die Eingabe auf das Band. Im Folgenden wird auf dem ersten Wort N und auf dem zweiten Wort N' simuliert. N'' führt diese Simulationsschritte abwechselnd aus und stoppt, falls ein Stoppzustand von N bzw. N' erreicht wird. Dabei fungieren die Stoppzustände von N als akzeptierende Stoppzustände und die von N' als ablehnende Stoppzustände. Da entweder $w \in X$ oder $w \in X^* \setminus L$ gilt, erreicht N'' bei Eingabe von w im ersten Fall einen Stoppzustand von N und im zweiten Fall einen Stoppzustand aus N'. Damit wird in jedem Fall ein Stoppzustand erreicht und L akzeptiert. Dies bedeutet, dass L rekursiv ist. □

Satz 2.36 *Für eine rekursive Sprache L ist die charakteristische Funktion*

$$\varphi_L(x) = \begin{cases} 0 & x \notin L \\ 1 & x \in L \end{cases}$$

von L algorithmisch berechenbar.

Beweis. Es seien L eine rekursive Menge und M die akzeptierende TURING-Maschine, die auf jeder Eingabe stoppt und L akzeptiert. Wir betrachten, die TURING-Maschine M', die zuerst M simuliert und bei Erreichen eines akzeptierenden Stoppzustandes den gesamten Bandinhalt durch eine 1 ersetzt bzw. bei Erreichen eines ablehnenden Stoppzustandes den gesamten Bandinhalt durch eine 0 ersetzt. Offenbar berechnet M' die charakteristische Funktion von L. □

Satz 2.37 *Die Menge der rekursiven Sprachen ist echt in der Menge der von* TURING-*Maschinen akzeptierbaren Sprachen enthalten.*

Beweis. Mit den Bezeichnungen aus dem Beweis von Satz 1.28 definieren wir die Menge

$$L_{halt} = \{w : w \in \{0,1\}^*, w = w_M \text{ für eine TURING-Maschine } M, f_M(w_M) \text{ ist definiert}\},$$

die im Wesentlichen dem Halteproblem von TURING-Maschinen entspricht, denn sie besteht aus allen Beschreibungen von TURING-Maschinen, die auf ihrer Beschreibung stoppen. Da wegen der Unentscheidbarkeit des Halteproblems die charakteristische Funktion von L_{halt} nicht berechenbar ist, ist L_{halt} nach Satz 2.36 nicht rekursiv.
L_{halt} wird aber von der folgenden TURING-Maschine N akzeptiert, deren Arbeitsweise wir nur informell beschreiben (die exakte Beschreibung von N bleibt dem Leser überlassen). N stellt zuerst fest, ob das Wort w auf dem Band die Kodierung w_M einer TURING-Maschine M ist. Bei negativer Antwort geht N in eine Schleife und stoppt nicht. Bei positiver Antwort kopiert N das Eingabewort $w = w_M$ ein zweites Mal auf das Band. Nun simuliert N auf dem ersten Wort $w = w_M$ die Arbeit von M, wobei N die erforderlichen Informationen über M aus der zweiten Kopie von $w = w_M$ auf dem Band bezieht. N stoppt, falls M stoppt. Daher stoppt N genau dann, wenn die Eingabe w Kodierung w_M einer TURING-Maschine M ist und $f_M(w_M)$ definiert ist. Somit gilt $T(N) = L_{halt}$. □

Die nächsten Aussagen geben das Verhältnis der von TURING-Maschinen akzeptierten Sprachen zu den im vorhergehenden Abschnitt untersuchten Sprachen, die von Grammatiken erzeugt werden, an.

Lemma 2.38 *Zu jeder akzeptierenden* TURING*-Maschine M gibt es eine Regelgrammatik G mit $L(G) = T(M)$.*

Beweis. Es sei die TURING-Maschine $M = (X, Z, z_0, Q, \delta, F)$ gegeben. Wir konstruieren zuerst die TURING-Maschine $M'' = (X, Z'', z_0, \{q''\}, \delta'', \{q''\})$ entsprechend dem Beweis von Lemma 2.32 und aus M'' die TURING-Maschine $M' = (X \cup \{\S, \#\}, Z', z_0', \{q'\}, \delta', \{q'\})$ entsprechend dem Beweis von Lemma 1.21. Jede Folge von Konfigurationen von M' hat die folgende Form:

$$(*) \quad K_0 = (\lambda, z_0', w) \models^* K_1 = (\S, z_0, w\#) \models^* K_2 = (\S v_1, q, v_2\#) \models^* K_3 = (\lambda, q', v),$$

wobei $v_1v_2 = *^r v *^s$ gilt. Außerdem ist sofort zu sehen, dass $T(M) = T(M')$ gilt. Ferner nehmen wir ohne Beschränkung der Allgemeinheit an, dass $X \cap Z' = \emptyset$ und $\S, \#, * \notin Z'$ gelten. Daher ist es möglich, eine Konfiguration (w_1, z, w_2) auch als w_1zw_2 zu beschreiben. Wir werden jetzt eine Regelgrammatik $G = (N, T, P, S)$ so konstruieren, dass im Wesentlichen $w_1zw_2 \models w_1'z'w_2'$ genau dann gilt, wenn $w_1'z'w_2' \Longrightarrow w_1zw_2$ gilt, d.h. wir werden die Überführungen in M' schrittweise in umgekehrter Reihenfolge simulieren. Dadurch wird erreicht, dass die Grammatik in einer Ableitung eine „Endkonfiguration“ in eine „Anfangskonfiguration“ überführt, aus der durch „Streichen“ des Zustands das akzeptierte Wort entsteht.
Wir geben nun die formale Definition von G. Dazu setzen wir

$$\begin{aligned} N &= Z' \cup \{\S, \#, *, S, S', A\}, \\ T &= X \end{aligned}$$

und definieren P als die Menge aller Regeln der folgenden Formen:

$$\begin{aligned} (i) \quad & S \longrightarrow \S S'\#, \\ & S' \longrightarrow xS', S' \longrightarrow S'x \quad \text{für } x \in X \cup \{*\}, \\ & S' \longrightarrow q \quad \text{für } q \in Q \end{aligned}$$

(mittels dieser Regeln wird eine Ableitung $S \Longrightarrow^* \S v_1 q v_2 \#$ realisiert und damit die Beschreibung der Konfiguration K_2 aus $(*)$ erreicht),

$$\begin{aligned} (ii) \quad & z'ab' \longrightarrow azb \quad \text{für } z, z' \in Z', a, b, b' \in X \cup \{*\}, \delta'(z, b) = (z', b', L), \\ & z'b' \longrightarrow zb \quad \text{für } z, z' \in Z', b, b' \in X \cup \{*\}, \delta'(z, b) = (z', b', N), \\ & b'z' \longrightarrow zb \quad \text{für } z, z' \in Z, b, b' \in X \cup \{*\}, \delta'(z, b) = (z', b', R) \end{aligned}$$

(dies ist eine direkte Simulation einer inversen Überführung, bei der die TURING-Maschine über einem Element aus $X \cup \{*\}$ stand),

$$\begin{aligned} (iv) \quad & \S z_0 \longrightarrow A, \\ & Aa \longrightarrow aA \quad \text{für } a \in X, \\ & A\# \longrightarrow \lambda \end{aligned}$$

(durch diese Regeln wird aus einem Wort der Form $\S z_0 w\#$ mit $w \in X^*$ das Wort w abgeleitet).
Aufgrund der im Anschluss an die Regeln gegebenen Ausführungen ist klar, dass jede Ableitung in G die Form

$$(**) \quad S \Longrightarrow^* u_1 = \S v_1 q v_2 \# \Longrightarrow^* u_2 = \S z_0 w \# \Longrightarrow^* w$$

hat, wobei die Ableitung $u_1 \Longrightarrow^* u_2$ durch schrittweise Simulation der Überführungsschritte von $K_1 \models^* K_2$ in umgekehrter Reihenfolge erhalten wird.
Damit ist gezeigt, dass es eine Ableitung $(**)$ in G genau dann gibt, wenn es auch eine Überführung $(*)$ gibt. Hieraus folgt sofort, dass $w \in L(G)$ genau dann gilt, wenn auch $w \in T(M)$ gilt. Somit erhalten wir $L(G) = T(M)$. □

Die Idee des Beweises von Satz 2.38 besteht im Wesentlichen darin, dass die Überführungen $w_1zw_2 \models w_1'z'w_2'$ der TURING-Maschine in umgekehrter Reihenfolge durch einen direkten Ableitungsschritt $w_1'z'w_2' \Longrightarrow w_1zw_2$ simuliert werden. Es ist naheliegend, diesen Gedanken auch dafür zu verwenden, um zu zeigen, dass jede von einer Regelgrammatik erzeugte Sprache von einer TURING-Maschine akzeptiert wird. Dabei tritt aber die Schwierigkeit, dass eine Satzform einer Grammatik durch Anwendung verschiedener Regeln auf verschiedene Satzformen entstanden sein kann. Bei der Umkehrung erfordert dies, dass aus einer Konfiguration mehrere verschiedene Konfigurationen entstehen können müssen. Um diese Schwierigkeit zu überwinden, definieren wir daher eine nichtdeterministische Variante der TURING-Maschine, bei der auf eine Konfiguration dann mehrere Konfigurationen folgen können.

Definition 2.39 *Eine nichtdeterministische* TURING-*Maschine M ist ein Sechstupel*

$$M = (X, Z, z_0, Q, \tau, F),$$

wobei X, Z, z_0, Q und F wie bei einer (akzeptierenden deterministischen) TURING-*Maschine definiert sind und τ eine Funktion*

$$\tau : (Z \setminus Q) \times (X \cup \{*\}) \to 2^{Z \times (X \cup \{*\}) \times \{R,N,L\}}$$

ist.

Entsprechend dieser Definition besteht $\tau(z, x)$ aus einer Menge von Elementen der Form (z', x', r) mit $z' \in Z, x' \in (X \cup \{*\}), r \in \{R, L, N\}$.
Die in Definition 1.15 angegebenene TURING-Maschine ist der Spezialfall, dass die Menge $\tau(z, x)$ nur aus dem Element $\delta(z, x)$ besteht.
Wir definieren nun die Konfiguration einer nichtdeterministischen TURING-Maschine wie bei einer (deterministischen) TURING-Maschine (Definition 1.16) und die Relation $K_1 \models K_2$ wie in Definition 1.17, wobei wir nur $\delta(z, x) = (z', x', r)$ durch die Forderung $(z', x', r) \in \tau(z, x)$ ersetzen.
Hieraus folgt offensichtlich, dass aus einer Konfiguration K_1 mehrere Konfigurationen K_2 erzeugt werden können, wenn $\tau(z, x)$ mehrere Elemente enthält. Wir definieren die von einer nichtdeterministischen TURING-Maschine akzeptierte Wortmenge in Analogie zu Definition 2.30.

Definition 2.40 *Es sei $M = (X, Z, z_0, Q, \tau, F)$ eine nichtdeterministische* TURING-*Maschine wie in Definition 2.39. Die von M akzeptierte Sprache $T(M)$ wird durch*

$$T(M) = \{w : w \in X^*,\ (\lambda, z_0, w) \models^* (v_1, q, v_2) \text{ für ein } q \in F\}$$

definiert.

Wir geben nun ein Beispiel.

Beispiel 2.41 Wir betrachten die nichtdeterministische TURING-Maschine

$$M = (\{a, b\}, \{z_0, z_0', z_0'', z_{0,2}z_{1,2}, z_2, z_2', z_2'', z_{0,3}z_{1,3}, z_{2,3}, z_3, z_3', z_3'', q\}, z_0, \{q\}, \tau, \{q\})$$

mit

$$\begin{aligned}
\tau(z_0, a) &= \{(z_0, a, R)\},\\
\tau(z_0, b) &= \{(z_0, b, N)\},\\
\tau(z_0, *) &= \{(z_0', *, L)\},\\
\tau(z_0', a) &= \{(z_0', a, L)\},\\
\tau(z_0', *) &= \{(z_0'', *, R)\}
\end{aligned}$$

(die Maschine entscheidet, ob auf dem Band nur as stehen; ist dies nicht der Fall, so geht sie in eine Schleife),

$$\begin{aligned}
\tau(z_0'', *) &= \{(z_o'', *, N)\},\\
\tau(z_0'', x) &= \{(z_2, x, N), (z_3, x, N)\} \quad \text{für } x \in \{a, b\}
\end{aligned}$$

(steht kein Buchstabe auf dem Band, so geht die Maschine in eine Schleife; ansonsten entscheidet sich die Maschine nichtdeterministisch zwischen zwei Varianten, die im Index 2 bzw. 3 festgelegt sind),

$$\begin{aligned}
\tau(z_i, a) &= \{(z_i', a, R)\} \quad \text{für } i \in \{2, 3\},\\
\tau(z_i, b) &= \{(z_i, b, R)\} \quad \text{für } i \in \{2, 3\},\\
\tau(z_i', a) &= \{(z_i'', a, R)\} \quad \text{für } i \in \{2, 3\},\\
\tau(z_i', b) &= \{(z_i', b, R)\} \quad \text{für } i \in \{2, 3\},\\
\tau(z_i'', x) &= \{(z_i'', x, R)\} \quad \text{für } x \in \{a, b\},\\
\tau(z_i, *) &= \{(z_i, *, N)\} \quad \text{für } i \in \{2, 3\},\\
\tau(z_i', *) &= \{(q, *, N)\} \quad \text{für } i \in \{2, 3\},\\
\tau(z_i'', *) &= \{(z_{0,i}, *, L)\} \quad \text{für } i \in \{2, 3\}
\end{aligned}$$

(die Maschine liest von links nach rechts das Wort auf dem Band und prüft, ob es kein a, genau ein a oder mindestens zwei a enthält, wozu die Zustände z_i, z_i' und z_i'' dienen; ist kein a vorhanden, so geht die Maschine in eine Schleife und stoppt nicht; ist genau ein a vorhanden, so akzeptiert die Maschine die Eingabe; sind mindestens zwei a vorhanden, so wird die nächste Phase eingeleitet),

$$\begin{aligned}
\tau(z_{0,2}, a) &= \{(z_{1,2}, b, L)\},\\
\tau(z_{1,2}, a) &= \{(z_{0,2}, a, L)\},\\
\tau(z_{j,2}, b) &= \{(z_{i,2}, b, L)\} \quad \text{für } j \in \{0, 1\}
\end{aligned}$$

(die Maschine liest das Wort auf dem Band von rechts nach links; abwechselnd wird dabei ein a durch ein b ersetzt bzw. stehengelassen; somit erfolgt eine Halbierung der Anzahl der Vorkommen von a; im Zustand $z_{j,2}$ gibt j die Anzahl der gelesenen a modulo 2 an),

$$\begin{aligned}
\tau(z_{0,3}, a) &= \{(z_{1,2}, b, L)\},\\
\tau(z_{1,3}, a) &= \{(z_{2,3}, b, L)\},\\
\tau(z_{2,3}, a) &= \{(z_{0,3}, a, L)\},\\
\tau(z_{j,3}, b) &= \{(z_{j,3}, b, L)\} \quad \text{für } j \in \{0, 1, 2\}
\end{aligned}$$

(analog erfolgt eine Drittelung der Anzahl der Vorkommen von a; in $z_{j,3}$ gibt j die Anzahl der gelesenen a modulo 3 an),

$$\begin{aligned}\tau(z_{0,i},*) &= \{(z_i,*,R)\} \quad \text{für } i \in \{2,3\},\\ \tau(z_{j,i},*) &= \{(z_{j,i},*,N)\} \quad \text{für } j \in \{1,2\}\ , i \in \{2,3\}\end{aligned}$$

(ist die Halbierung bzw. Drittelung ganzzahlig möglich, d.h. $z_{0,2}$ bzw. $z_{0,3}$ liegt vor, so wird der Gesamtprozess iteriert, anderenfalls geht die Maschine in eine Schleife und akzeptiert daher nicht).
Nach diesen Erklärungen ist klar, dass die TURING-Maschine nur solche Wörter akzeptiert bei denen iterierte Halbierung bzw. Drittelung der Anzahl der Vorkommen von a zu einem Wort auf dem Band führt, dass genau ein a enthält und akzeptiert dann. Somit ergibt sich als akzeptierte Sprache

$$T(M) = \{w : \#_a(w) = 2^n \text{ oder } \#_a(w) = 3^n \text{ für ein } n \geq 0\}.$$

Mit diesem Typ von TURING-Maschinen sind wir nun in der Lage, die Umkehrung von Lemma 2.38 zu beweisen.

Lemma 2.42 *Zu jeder Regelgrammatik G gibt es eine nichtdeterministische* TURING-*Maschine M mit $T(M) = L(G)$.*

Beweis. Wir geben hier keinen detaillierten vollständigen Beweis, sondern erläutern nur die wesentliche Idee der Konstruktion.
Es sei die Grammatik $G = (N, T, P, S)$ gegeben. Wir konstruieren nun eine nichtdeterministische TURING-Maschine M mit dem Eingabealphabet $N \cup T \cup \{\S\}$ und folgender Arbeitsweise auf einer Eingabe w.

1. Da nur Wörter über T akzeptiert werden sollen, testet M als erstes, ob w in T^* liegt. Ist dies nicht der Fall, so geht M in eine Schleife (und akzeptiert daher w nicht); gilt dagegen $w \in T^*$, so erreicht M die Konfiguration (λ, z_1, w), bei der der Zustand z_1 den Beginn der zweiten Phase andeutet.

2. In dieser Phase testet M, ob auf dem Band nur S steht. Ist dies der Fall, so stoppt M; steht nicht nur S auf dem Band, erreicht die Maschine die Konfiguration (λ, z_2, w), bei der z_2 den Beginn der Phase 3 markiert.

3. Diese Phase dient der Simulation eines Ableitungsschrittes, wobei wir wie bereits im Beweis von Lemma 2.38 die Richtung umkehren, d.h. wir simulieren die Anwendung einer Regel $u \longrightarrow u'$ und damit die Ableitung

   $$xuy \Longrightarrow xu'y = w$$

   durch den Übergang

   $$(\lambda, z_2, w) = (\lambda, z_2, xu'y) \models^* (\lambda, z_1, xuy).$$

   Hierzu bestimmt M zuerst nichtdeterministisch eine Stelle, an der die Anwendung der Regel $p = u \longrightarrow u'$ simuliert werden soll, d.h. M erreicht die Konfiguration

(x, z_p, x'), bei der z_p den Beginn der Simulation von p markiert. M testet nun, ob das Wort u' hinter x auf dem Band steht. Ist dies nicht der Fall, so geht M in eine Schleife. Ist dies aber der Fall arbeitet M wie folgt. Falls $|u'| - |u| = m \geq 0$ ist, ersetzt M das Wort u' durch $u\S^m$, wodurch $(xu\S^m, z'_p, y)$ entsteht, verschiebt y um m Zellen nach links und kehrt an den Wortanfang und in den Zustand z_1 zurück. Falls $|u| - |u'| = m' > 0$ ist, verschiebt M zuerst das hinter u' stehende Wort y um m' Zellen nach rechts, schreibt in die entstehende Lücke $\S^{m'}$, ersetzt dann $u'\S^{m'}$ durch u und kehrt dann an den Wortanfang und in den Zustand z_1 zurück. Damit entsteht jeweils die Konfiguration (λ, z_1, xuy) aus $(\lambda, z_2, xu'y)$, womit die Simulation abgeschlossen ist.

Danach wird erneut Phase 2 gestartet.

Entsprechend dieser Arbeitsweise wird jede Ableitung

$$S \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \ldots \Longrightarrow w_{n-1} \Longrightarrow w_n = w$$

durch

$$\begin{aligned}(\lambda, z_0, w_n) \quad &\models^* \quad (\lambda, z_1, w_n) \models^* (\lambda, z_2, w_n)\\ &\models^* \quad (\lambda, z_1, w_{n-1}) \models^* (\lambda, z_2, w_{n-1})\\ &\models^* \quad \ldots \models^* (\lambda, z_1, w_2) \models^* (\lambda, z_2, w_2)\\ &\models^* \quad (\lambda, z_1, w_1) \models^* (\lambda, z_2, w_1)\\ &\models^* \quad (\lambda, z_1, S) \models^* (\lambda, q, S)\end{aligned}$$

simuliert. Weiterhin erreicht M nur einen Endzustand, wenn M eine Ableitung simuliert, da M sonst in eine Schleife geht. Setzen wir nun noch die Menge der akzeptierenden Zustände als die Menge aller Stoppzustände, so gilt $T(M) = L(G)$. □

Satz 2.43 *Die folgenden Aussagen sind äquivalent:*
i) L wird von einer Regelgrammatik erzeugt.
ii) L wird von einer deterministischen TURING*-Maschine akzeptiert.*
iii) L wird von einer nichtdeterministischen TURING*-Maschine akzeptiert.*

Beweis. Wegen Lemma 2.38 und 2.42 reicht es zu zeigen, dass jede Sprache, die von einer nichtdeterministischen TURING-Sprache akzeptiert wird auch von einer (deterministischen TURING-Maschine akzeptiert wird.
Wir geben hier erneut keinen vollständigen formalen Beweis sondern nur die Beweisidee. Es seien $M = (X, Z, z_0, Q, \tau, F)$ eine nichtdeterministische TURING-Maschine und

$$n = max\{\#(\tau(z, x)) : z \in Z, x \in X \cup \{*\}\} \quad \text{und} \quad N = \{1, 2, ..., n\}.$$

In der Menge der Folgen über N führen wir eine Ordnung ein, bei der zuerst nach der Länge und bei gleicher Länge lexikographisch sortiert wird. $NFOLGE$ sei die Funktion, bei der der Folge x die auf x entsprechend der Ordnung folgende Folge $NFOLGE(x)$ zugeordnet wird. Wir sagen, dass die Folge $d_1 d_2 ... d_r$ durch M abgearbeitet wird, wenn bei Vorliegen des Zustandes z und Lesen von x nach $i - 1$ Schritten im i-ten Schritt das d_i-te Element aus $\tau(z, x)$ benutzt wird, soweit es vorhanden ist.

Wir betrachten nun die TURING-Maschine M', die wie folgt auf der Eingabe w arbeitet (dabei ist \$ ein gesondertes Trennzeichen):

	Programm	Bandinhalt
	BEGIN	w
	$f := \lambda$	
	Schreibe f hinter das Wort auf dem Band	$w\$f$
A:	Schreibe hinter das Wort auf dem Band eine Kopie von w und zwei Kopien von $f' = NFOLGE(f)$	$w\$f\$w\$f'\$f'$
	Lösche $f\$$	$w\$w\$f'\$f'$
	Arbeite f' auf erstem w ab (dabei wird $f'\$$ gelöscht)	$w'\$w\$f'$
	(falls das d_i-te Element nicht vorhanden ist, lösche $w\$$ und $f'\$$ und **GOTO** A)	$(w\$f')$
	IF erreichter Zustand $z \notin Q$ **THEN** lösche $w'\$$ und **GOTO** A	$w\$f'$
	END	

Jeder einzelne dieser Schritte ist deterministisch realisierbar, und durch spezielle Komponenten in den Zuständen kann deterministisch der Schritt, in dem sich M' befindet, gespeichert werden.
Verwenden wir F auch als Menge der akzeptierenden Zustände von M', so akzeptiert M' entsprechend ihrer Arbeitsweise genau dann ein Wort w, wenn es eine Folge $d_1 d_2 \dots d_r$ gibt, bei deren Abarbeitung M das Wort w akzeptiert. Daher gilt $T(M') = T(M)$. □

Satz 2.43 kann auch wie folgt formuliert werden: *Eine Sprache L wird genau dann von einer (nichtdeterministischen)* TURING*-Maschine akzeptiert, wenn $L \in \mathcal{L}(RE)$ gilt.*

Durch Kombination dieser Formulierung mit Satz 2.33 und den Übungsaufgaben 12und 13 zu Abschnitt 1 wird die Bezeichnung RE als Abkürzung von rekursiv-aufzählbar (engl. recursively enumerable) als sinnvoll nachgewiesen.

Wir wollen nun ein Analogon zu Satz 2.43 für kontextabhängige Sprachen geben. Wegen Folgerung 2.17 können wir eine monotone Grammatik zur Erzeugung der Sprache verwenden. Wir werfen nun einen Blick auf den Beweis von Lemma 2.42. Da bei monotonen Grammatiken für alle Regeln $u \to u'$ die Beziehung $|u| \leq |u'|$ gilt, wird bei der von der TURING-Maschine in umgekehrter Richtung durchgeführten Simulation der Übergang $w_1 u' w_2 \models w_1 u w_2$ zu einer Verkürzung des Bandinhaltes führen. Folglich stehen auf dem Band nur Wörter, deren Länge höchstens die Länge des Wortes ist, das zu Beginn auf dem Band steht. Außerdem bewegt sich der Kopf der Maschine immer nur auf Zellen, in denen ein Buchstabe des Eingabealphabetes steht, oder auf den mit $*$ gefüllten Zellen direkt vor oder direkt hinter dem Wort (dies ist nötig, um den Wortanfang oder das Wortende zu finden). Damit ist durch die um 2 erhöhte Länge des Wortes, das zu Beginn auf dem Band steht, eine obere Schranke für die Anzahl der Zellen der TURING-Maschine, über denen sich während der Arbeit der Kopf befinden kann. Dies führt zur folgenden Definition.

Definition 2.44 *Ein linear beschränkter Automat ist eine nichtdeterministische* TURING*-Maschine $M = (X, Z, z_0, Q, \delta, F)$, deren Kopf sich während der Abarbeitung der Eingabe $w \in X^*$ höchstens über $|w| + 2$ verschiedenen Zellen befindet.*

Aus den vorstehend gemachten Ausführungen folgt sofort, dass jede von einer monotonen oder kontextabhängigen Grammatik erzeugte Sprache von einem linear beschränkten Automaten akzeptiert wird. Ohne Beweis geben wir an, dass auch die Umkehrung gilt. Damit erhalten wir den folgenden Satz.

Satz 2.45 *Eine Sprache ist genau dann kontextabhängig, wenn sie von einem linear beschränkten Automaten akzeptiert werden kann.* □

Für TURING-Maschinen haben wir gezeigt, dass deterministische und nichtdeterministische Varianten die gleiche Menge von Sprachen akzeptieren. Die analoge Frage ist für deterministische linear beschränkte Automaten noch offen, d.h. es ist weder ein Beweis gegeben worden, dass jede kontextabhängige Sprache von einem deterministischen linear beschränkten Automaten akzeptiert werden kann, noch ein Beispiel einer kontextabhängigen Sprache bekannt, die nicht von einem deterministischen linear beschränkten Automaten akzeptiert werden kann.

2.2.2 Endliche Automaten

Im vorangehenden Abschnitt haben wir Charakterisierungen der Sprachfamilien $\mathcal{L}(RE)$ und $\mathcal{L}(CS)$ mittels TURING-Maschinen bzw. linear beschränkten Automaten angegeben. Wir wollen nun eine analoge Charakterisierung für die Familie der regulären Sprachen herleiten. Zuerst definieren dazu den hierfür geeigneten Automatentyp.

Definition 2.46 *i) Ein endlicher Automat ist ein Quintupel*

$$\mathcal{A} = (X, Z, z_0, F, \delta),$$

wobei
- X und Z Alphabete sind,
- $z_0 \in Z$ und $F \subseteq Z$ gelten,
- δ eine Funktion von $Z \times X$ in Z ist.
ii) Die Erweiterung δ^* von δ auf $Z \times X^*$ ist durch

$$\delta^*(z, \lambda) = z,$$
$$\delta^*(z, wx) = \delta(\delta^*(z, w), x) \text{ für } w \in X^*, x \in X$$

definiert.
iii) Die durch $\mathcal{A}$ akzeptierte Wortmenge ist durch

$$T(\mathcal{A}) = \{w : w \in X^*, \delta^*(z_0, w) \in F\}$$

definiert.

Wie bei TURING-Maschinen nennen wir die Elemente von X erneut Eingabesymbole und die von Z Zustände; z_0 ist der Anfangszustand, und F ist die Menge der akzeptierenden Zustände; δ heißt erneut Überführungsfunktion. Im Folgenden werden wir meistens zwischen der Funktion δ und ihrer Erweiterung δ^* nicht unterscheiden und beide mit δ

bezeichnen, zumal aus der Definition sofort $\delta^*(z,x) = \delta(\delta^*(z,\lambda),x) = \delta(z,x)$ für $x \in X$ und $z \in Z$ folgt.
Die Arbeitsweise eines endlichen Automaten können wir uns wie folgt vorstellen: Der Automat liest von links nach rechts die Buchstaben des Eingabewortes und ändert bei jedem Lesevorgang seinen Zustand entsprechend δ, wobei er im Zustand z_0 beginnt. Ein Wort wird genau dann akzeptiert, wenn er nach Lesen des gesamten Wortes in einen akzeptierenden Zustand gelangt ist.
Entsprechend dieser Interpretation kann ein endlicher Automat als TURING-Maschine aufgefasst werden, bei der sich der Kopf nur nach rechts bewegt und beim Lesen des $*$ hinter dem Wort ein Stoppzustand erreicht wird. Das Schreiben auf das Band ist bei endlichen Automaten – wie wir sie definiert haben – nicht von Interesse, da die Zellen, in die geschrieben werden kann, wegen der ständigen Rechtsbewegung nicht mehr gelesen werden können, so dass das Schreiben keinen Einfluss auf die Akzeptanz hat. Wir merken aber an, dass dann, wenn man sich nicht nur für das Akzeptanzverhalten von endlichen Automaten interessiert, auch eine entsprechende Modifikation des Begriffs zum endlichen Automaten mit Ausgabe möglich ist.

Um einen endlichen Automaten zu beschreiben, ist es nach Definition notwendig, die einzelnen Komponenten X, Z, z_0, F, δ von $\mathcal{A}$ anzugeben. Vielfach wird aber eine Beschreibung von $\mathcal{A}$ durch einen gerichteten Graphen $G = (V, E)$, dessen Kanten bewertet (oder markiert) sind, bevorzugt. Als Knotenmenge V verwenden wir die Zustandsmenge Z, und es gibt genau dann eine Kante von z nach z', die mit x bewertet ist, falls $\delta(z,x) = z'$ gilt. Zur Auszeichnung des Anfangszustandes bzw. der akzeptierenden Zustände benutzen wir einen auf den Knoten gerichteten Pfeil bzw. einen doppelten Kreis. In dieser Beschreibung wird $\delta^*(z, x_1x_2\ldots x_n) = z'$ durch die Existenz eines Weges von z nach z' widergespiegelt, bei dem die Folge der Bewertungen durch $x_1x_2\ldots x_n$ gegeben ist.

Beispiel 2.47 Der endliche Automat $\mathcal{A} = (X, Z, z_0, F, \delta)$ sei durch

$$\begin{aligned} X &= \{a,b,c\} \\ Z &= \{z_0,z_1,z_2,z_3\}, \\ F &= \{z_2\}, \\ \delta(z,x) &= \begin{cases} z_1 & \text{für } z = z_0, x = a \\ z_2 & \text{für } z = z_1, x = a \\ z_0 & \text{für } z \in \{z_0,z_2\}, x = c \\ z_3 & \text{sonst} \end{cases} \end{aligned}$$

gegeben. Die Darstellung von $\mathcal{A}$ durch einen Graphen wird in Abb. 2.5 gezeigt.
Wir bestimmen nun die von $\mathcal{A}$ akzeptierte Wortmenge. Wir geben die Erläuterungen dabei immer durch Bezug auf die Überführungsfunktion; der Leser möge sie jedoch auch anhand des Graphen zu verfolgen.
Wir stellen dazu erst einmal fest, dass wegen $\delta(z_3,x) = z_3$ für alle $x \in X$ der Zustand z_3 nicht mehr verlassen werden kann, womit eine Akzeptanz ausgeschlossen ist. Da außerdem $\delta(z,b) = z_3$ für alle $z \in Z$ gilt, wird z_3 erreicht, wenn im Wort ein b vorkommt. Hieraus folgt, dass ein Wort nur dann akzeptiert werden kann, wenn es kein b enthält. Wir bemerken noch, dass die einzige Möglichkeit des Übergangs von z_0 zu z_2 durch $\delta(z_0,aa) = z_2$ gegeben ist. Da $\delta(z_2,c^n) = \delta(z_0,c^{n-1}) = z_0$ für beliebige natürliche Zahlen $n \geq 1$ gelten,

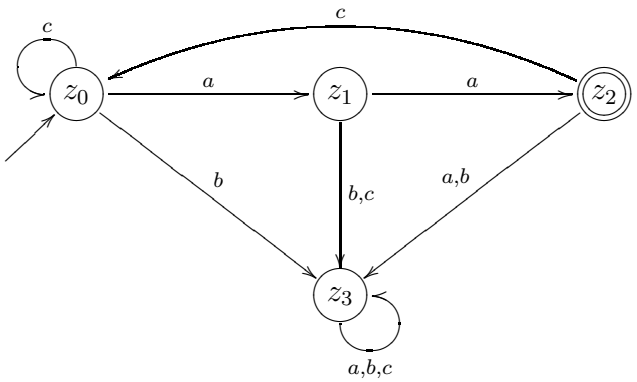


Abbildung 2.5:

erhalten wir, dass $T(\mathcal{A})$ aus allen Wörtern besteht, bei denen einer beliebigen Anzahl von Vorkommen von c stets aa folgt und die kein b enthalten, d.h.

$$T(\mathcal{A}) = \{c^{n_1}aac^{n_2}aa \ldots c^{n_k}aa : \ k \geq 1, \ n_1 \geq 0, \ n_i \geq 1 \text{ für } 1 \leq i \leq k\}.$$

Beispiel 2.48 Wir wollen einen endlichen Automaten $\mathcal{A}$ so bestimmen, dass

$$T(\mathcal{A}) = \{a^n b^m : n \geq 1, m \geq 2\}$$

gilt.[2] Offensichtlich können wir $X = \{a, b\}$ annehmen. Ferner benutzen wir Zustände, um zu zählen, wieviele Buchstaben a bzw. b bereits im gelesenen Teil des Wortes enthalten sind. Folgende Zustände entsprechen folgenden Situationen:
z_1 – es ist mindestens ein a und kein b gelesen worden,
z_2 – es sind mindestens ein a und genau ein b gelesen worden,
z_3 – es sind mindestens ein a und mindestens zwei b gelesen worden.
Ferner haben wir zu beachten, dass bei zu akzeptierenden Wörtern kein a auf ein b folgen darf. Ein endlicher Automat mit dieser Eigenschaft ist offenbar durch Abb. 2.6 gegeben.

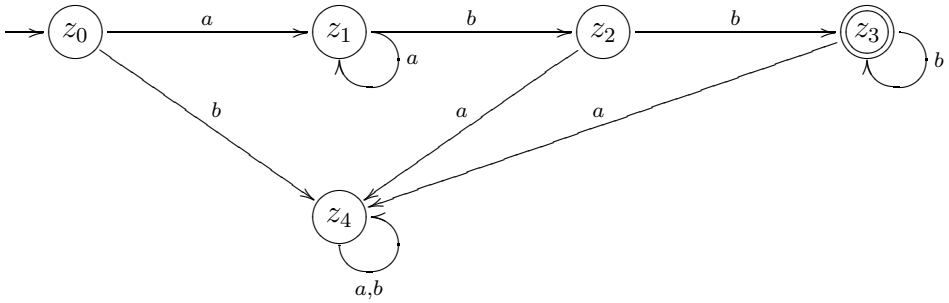


Abbildung 2.6:

Wir definieren nun eine nichtdeterministische Variante des endlichen Automaten. Dabei gehen wir analog zu TURING-Maschinen vor, d.h. die Bilder bei δ werden Mengen von Zuständen anstelle von einzelnen Zuständen sein.

[2]Wir diskutieren hier nicht die Frage, ob diese Sprache überhaupt von einem endlichen Automaten akzeptiert werden kann, da wir in diesem Abschnitt diese Frage generell klären werden.

Definition 2.49 *i) Ein nichtdeterministischer endlicher Automat ist ein Quintupel* $\mathcal{A} = (X, Z, z_0, F, \delta)$*, wobei für* X, Z, z_0, F *die gleichen Bedingungen wie in Definition 2.46 gelten und* δ *eine Funktion von* $Z \times X$ *in die Menge der Teilmengen von* Z *ist.*
ii) Wir definieren $\delta^*(z, \lambda) = \{z\}$ *für* $z \in Z$*, und für* $w \in X^*$*,* $x \in X$ *und* $z \in Z$ *gelte* $z' \in \delta^*(z, wx)$ *genau dann, wenn es einen Zustand* $z'' \in \delta^*(z, w)$ *mit* $z' \in \delta(z'', x)$ *gibt.*
iii) Die von $\mathcal{A}$ *akzeptierte Wortmenge ist durch*

$$T(\mathcal{A}) = \{w : \delta^*(z_0, w) \cap F \neq \emptyset\}$$

definiert.

Erneut ist damit der (deterministische) endliche Automat der Spezialfall des nichtdeterministischen Automaten, bei dem jede Menge $\delta(z, x)$ und damit dann auch jede Menge $\delta^*(z, w)$ einelementig ist. Daher stimmt dann auch die vom so interpretierten nichtdeterministischen Automaten akzeptierte Sprache mit der des deterministischen überein.

Wir beweisen nun die Äquivalenz von deterministischen und nichtdeterministischen endlichen Automaten bezüglich ihrer Akzeptierfähigkeit, die wir für TURING-Maschinen schon in Satz 2.43 gezeigt haben.

Satz 2.50 *Die beiden folgenden Aussagen sind für eine Sprache* L *äquivalent:*
i) L *wird von einem (deterministischen) endlichen Automaten akzeptiert.*
ii) L *wird von einem nichtdeterministischen endlichen Automaten akzeptiert.*

Beweis. i) $\Rightarrow$ ii) folgt sofort aus den obigen Bemerkungen, dass (deterministische) endliche Automaten als spezielle nichtdeterministische aufgefasst werden können.
ii) $\Rightarrow$ i). Es sei $\mathcal{A} = (X, Z, z_0, F, \delta)$ ein nichtdeterministischer Automat. Wir konstruieren den (deterministischen) endlichen Automaten $\mathcal{A}' = (X, Z', z_0', F', \delta')$ mit

$$\begin{aligned} Z' &= \{U : U \subseteq Z\}, \\ z_0' &= \{z_0\}, \\ F' &= \{U : U \in Z',\ U \cap F \neq \emptyset\}, \\ \delta'(U, x) &= \cup_{z \in U} \delta(z, x). \end{aligned}$$

Mittels vollständiger Induktion über die Wortlänge zeigen wir nun

$$(*) \quad (\delta')^*(\{z_0\}, w) = \delta^*(z_0, w)$$

für alle Wörter $w \in X^*$.
Für $w = \lambda$ folgt dies direkt aus den Definitionen der Erweiterungen. Damit ist der Induktionsanfang bewiesen.
Sei nun $w = w'x$ und die Aussage bereits für w' gültig. Dann gilt

$$(\delta')^*(\{z_0\}, w'x) = \delta'((\delta')^*(\{z_0\}, w'), x) = \delta'(\delta^*(z_0, w'), x) = \cup_{z \in \delta^*(z_0, w')} \delta(z, x) = \delta^*(z_0, w'x).$$

Damit gilt $(\delta')^*(\{z_0\}, w) = \delta^*(z_0, w)$, womit auch der Induktionschritt nachgewiesen ist.

Sei nun $w \in T(\mathcal{A})$. Dann gilt $\delta^*(z_0, w) \cap F \neq \emptyset$. Nach Definition heißt dies $\delta^*(z_0, w) \in F'$. Wegen $(*)$ gilt auch $(\delta')^*(\{z_0\}, w) \in F'$, womit $w \in T(\mathcal{A}')$ gezeigt ist.
Durch Umkehrung der eben durchgeführten Schlüsse können wir zeigen, dass aus $w \in T(\mathcal{A}')$ auch $w \in T(\mathcal{A})$ folgt. Damit ist dann $T(\mathcal{A}) = T(\mathcal{A}')$ gezeigt. □

Wir kommen nun zum Hauptresultat dieses Abschnittes, in dem wir zeigen, dass die von (nichtdeterministischen) endlichen Automaten akzeptierten Sprachen mit den regulären Sprachen übereinstimmen.

Satz 2.51 *Für eine Sprache L sind die folgenden Aussagen äquivalent.*

i) L ist regulär.

ii) L wird von einem nichtdeterministischen endlichen Automaten akzeptiert.
iii) L wird von einem (deterministischen) endlichen Automaten akzeptiert.

Beweis. i) ⇒ ii). Wir geben den Beweis zuerst für den Fall, dass L das Leerwort nicht enthält.
Es sei $G = (N, T, P, S)$ eine reguläre Grammatik mit $L(G) = L$. Entsprechend Satz 2.24 können wir ohne Beschränkung der Allgemeinheit annehmen, dass alle Regeln in P von der Form $A \to xB,\ A \to x$ mit $A, B \in N,\ x \in T$ sind. Wir konstruieren nun zuerst die reguläre Grammatik $G' = (N', T, P', S)$ mit

$$\begin{aligned} N' &= N \cup \{\$\}, \\ P' &= \{A \to xB : A \to xB \in P\} \cup \{A \to x\$: A \to x \in P\} \cup \{\$ \to \lambda\}, \end{aligned}$$

wobei \$ ein zusätzliches Symbol ist ($\$ \notin N \cup T$). Da die terminierenden Ableitungen in G bzw. G' die Form

$$S \Longrightarrow^* wA \Longrightarrow wa$$

bzw.

$$S \Longrightarrow^* wA \Longrightarrow wa\$ \Longrightarrow wa$$

haben, ist leicht zu sehen, dass $L(G) = L(G') = L$ gilt.
Wir konstruieren nun einen nichtdeterministischen endlichen Automaten $\mathcal{A}$, für den $T(\mathcal{A}) = L$ gilt. Damit ist dann die Behauptung gezeigt.
Wir setzen dazu $\mathcal{A} = (T, N', S, \{\$\}, \delta)$, wobei die Überführungsfunktion durch

$$\delta(A, x) = \{B : A \to xB \in P\}$$

gegeben ist.
Wir zeigen zuerst mittels vollständiger Induktion über die Wortlänge, dass eine Ableitung $A \Longrightarrow^* x_1 x_2 \dots x_n B$ genau dann in G' existiert, wenn $B \in \delta(A, x_1 x_2 \dots x_n)$ gilt.
Der Induktionsanfang, d.h. diese Aussage für $n = 1$, gilt nach der Definition von δ.
Es sei nun eine Ableitung $A \Longrightarrow x_1 x_2 \dots x_{n-1} B' \Longrightarrow x_1 x_2 \dots x_{n-1} x_n B$ in G' gegeben. Nach Induktionsvoraussetzung und Definition von δ gelten dann $B' \in \delta(A, x_1 x_2 \dots x_{n-1})$ und $B \in \delta(B', x_n)$. Folglich ist $B \in \delta(A, x_1 x_2 \dots x_{n-1} x_n)$.

Gilt umgekehrt $B \in \delta(A, x_1x_2 \dots x_n)$. Dann gibt es einen Zustand B' (d.h. ein Nichtterminal B') derart, dass $B \in \delta(B', x_n)$ und $B' \in \delta(A, x_1x_2 \dots x_{n-1})$ gelten. Nach Induktionsvoraussetzung gibt es damit eine Ableitung $A \Longrightarrow^* x_1x_2 \dots x_{n-1}B'$ in G', und aus der Definition von δ folgt $B' \Longrightarrow x_nB$. Somit existiert eine Ableitung $A \Longrightarrow^* x_1x_2 \dots x_{n-1}B' \Longrightarrow x_1x_2 \dots x_{n-1}x_nB$.
Damit ist auch der Induktionsschritt vollzogen.
Wir betrachten jetzt ein Wort $w \in L(G')$. Dann gibt es eine Ableitung $S \Longrightarrow^* w\$ \Longrightarrow w$ in G'. Entsprechend der oben bewiesenen Aussage gilt dann $\$ \in \delta(S, w)$ und damit $w \in T(\mathcal{A})$.
Umgekehrt folgt aus $w \in T(\mathcal{A})$, also $\$ \in \delta(S, w)$, mittels der obigen Aussage die Existenz einer Ableitung $S \Longrightarrow^* w\$$ in G' und damit wegen $\$ \to \lambda \in P'$ auch $S \Longrightarrow^* w$.
Aus den beiden letzten Bemerkungen folgt $T(\mathcal{A}) = L(G')$, womit wegen $L = L(G) = L(G')$ auch $T(\mathcal{A}) = L$ bewiesen ist.

Gilt $\lambda \in L$, so modifizieren wir die Konstruktion wie folgt. Die Grammatik in der Normalform aus Satz 2.24 enthält dann zusätzlich die Regel $S \to \lambda$ und S kommt in keiner rechten Seite von Regeln aus P vor. Wir nehmen diese zusätzliche Regel auch in P' auf, und da diese nur die direkte Ableitung des Leerwortes bewirkt, muss auch S in die Menge der akzeptierenden Zustände von $\mathcal{A}$ aufgenommen werden. Nun laufen die Argumentationen für $L(G) = T(\mathcal{A})$ wie oben ab.

ii) $\Rightarrow$ iii) ist durch Satz 2.50 gegeben.

iii) $\Rightarrow$ i). Sei ein deterministischer endlicher Automat $\mathcal{A} = (X, Z, z_0, F, \delta)$ gegeben. Wir konstruieren dazu die reguläre Grammatik $G = (Z, X, P, z_0)$ mit

$$P = \{z \to az' : z' \in \delta(z, a)\} \cup \{z \to \lambda : z \in F\}.$$

Wie im ersten Teil dieses Beweises können wir zeigen, dass $z \in \delta(z_0, w)$ für ein $z \in F$ genau dann gilt, wenn es eine Ableitung $z_0 \Longrightarrow^* wz \Longrightarrow w$ gibt, woraus $T(\mathcal{A}) = L(G)$ folgt. □

Wir illustrieren die beiden Konstruktionen im Beweis von Satz 2.51 durch jeweils ein Beispiel.

Beispiel 2.52 Wir betrachten die Grammatik

$$G = (\{S, A, B\}, \{a, b\}, P, S),$$

bei der P aus den Regeln

$$S \to \lambda, S \to aA, S \to a, S \to b, S \to bB, A \to a,$$
$$A \to b, A \to aA, A \to bB, B \to bB, B \to bB, B \to b$$

besteht. Die im Beweis zuerst vorgenommene Umformung liefert dann

$$G' = (\{S, A, B, \$\}, \{a, b\}, P', S)$$

mit

$$\begin{aligned} P' \;=\; & \{S \to \lambda, S \to aA, S \to a\$, S \to b\$, S \to bB, A \to a\$, \\ & A \to b\$, A \to aA, A \to bB, B \to bB, B \to b\$, \$ \to \lambda\}. \end{aligned}$$

Der gesuchte Automat $\mathcal{B}$ ergibt sich dann durch

$$\mathcal{B} = (\{a, b\}, \{S, A, B, \$\}, S, \{S, \$\}, \delta)$$

mit

$$\begin{aligned}
&\delta(S, a) = \delta(A, a) = \{A, \$\},\\
&\delta(S, b) = \delta(A, b) = \delta(B, b) = \{B, \$\},\\
&\delta(B, a) = \delta(\$, a) = \delta(\$, b) = \emptyset.
\end{aligned}$$

Weiterhin konstruieren wir noch einen (deterministischen) endlichen Automaten $\mathcal{B}'$ an, der die gleiche Menge wie $\mathcal{B}$ akzeptiert. Dazu gehen wir wie im Beweis von Satz 2.50 vor. Die Menge Z der Zustände von $\mathcal{B}'$ wird dann von allen Teilmengen von $\{S, A, B, \$\}$ und die Menge F der akzeptierenden Zustände von allen den Teilmengen, die S oder $\$$ enthalten, gebildet. Wir erhalten dann

$$\mathcal{B}' = (\{a, b\}, Z, \{S\}, F, \delta),$$

wobei

$$\begin{aligned}
&\delta'(\{S\}, a) = \delta'(\{A\}, a) = \delta'(\{S, A\}, a) = \delta'(\{S, B\}, a) = \delta'(\{A, B\}, a)\\
&\qquad = \delta'(\{S, A, B\}, a) = \{A, \$\},\\
&\delta'(\{B\}, a) = \delta'(\emptyset, a) = \delta'(\emptyset, b) = \emptyset,\\
&\delta'(\{S\}, b) = \delta'(\{A\}, b) = \delta'(\{B\}, b) = \delta'(\{S, A\}, b) = \delta'(\{S, B\}, b)\\
&\qquad = \delta'(\{A, B\}, b) = \delta'(\{S, A, B\}, b) = \{B, \$\},\\
&\delta'(U \cup \{\$\}, x) = \delta'(U, x) \cup \{\S\} \quad \text{für } U \subseteq \{S, A, B\}, x \in \{a, b\}
\end{aligned}$$

gesetzt wird.

Beispiel 2.53 Haben wir eben zu einer Grammatik G einen (nichtdeterministischen) endlichen Automaten angegeben, der $L(G)$ akzeptiert, so konstruieren wir nun umgekehrt zu dem Automaten $\mathcal{A}$ aus Beispiel 2.47 eine reguläre Grammatik G mit $L(G) = T(\mathcal{A})$. Entsprechend dem Beweis von Satz 2.51 ergibt sich

$$G = (\{z_0, z_1, z_2, z_3\}, \{a, b, c\}, P, z_0)$$

mit

$$\begin{aligned}
P \;=\; &\{z_0 \to az_1, z_0 \to bz_3, z_0 \to cz_0, z_1 \to az_2, z_1 \to bz_3, z_1 \to cz_3,\\
&\quad z_2 \to az_3, z_2 \to bz_3, z_2 \to cz_0, z_3 \to az_3, z_3 \to bz_3, z_3 \to cz_3\}.
\end{aligned}$$

2.2.3 Kellerautomaten

Die im vorhergehenden Abschnitt angegebenen Charakterisierungen von Sprachen mittels Maschinen oder Automaten ergänzen wir in diesem Abschnitt durch eine solche Charakterisierung der kontextfreien Sprachen. Endliche Automaten können kontextfreie, aber nicht reguläre Sprachen wie $\{a^n b^n : n \geq 1\}$ oder $\{wcw^R : w \in \{a, b\}^*\}$ im Wesentlichen deshalb

nicht akzeptieren, weil eine endliche Menge von Zuständen nicht ausreicht, um sich die Länge bzw. die Struktur des schon gelesenen Wortanfangs zu merken. Um kontextfreie Sprachen zu akzeptieren, müssen wir den Automaten daher mit einer Möglichkeit zum Speichern dieser Information versehen. Hierfür werden wir ein zusätzliches Arbeitsband benutzen.
Wenn wir keine Restriktionen an das Arbeiten auf dem Arbeitsband stellen, so könnten wir die Eingabe von links nach rechts lesen und dabei auf das Arbeitsband übertragen und anschließend das Arbeitsband wie bei einer TURING-Maschine nutzen. Dann könnten offenbar wie bei TURING-Maschinen alle rekursiv-aufzählbaren Sprachen akzeptiert werden. Da wir einen Automatentyp suchen, der nur die kontextfreien Sprachen akzeptiert, müssen wir eine Einschränkung der Arbeit auf dem Arbeitsband vornehmen.
Wir nehmen folgende Einschränkungen vor:

- Die Symbole des Eingabebandes können nur von links nach rechts gelesen werden, d.h. Kopfbewegungen nach links sind verboten (im Gegensatz zum endlichen Automaten gestatten wir aber das Verharren des Lesekopfes an einer Stelle, damit Veränderungen des Arbeitsbandes ohne gleichzeitiges Lesen vorgenommen werden können).

- Eine Zelle des Arbeitsbandes ist mit einem speziellen Symbol # markiert, das nicht gelöscht und überschrieben werden kann. Der Lese-/Schreibkopf des Arbeitsbandes bewegt sich nicht auf die Zellen rechts von der mit # markierten Zelle. Dadurch entsteht im Prinzip ein einseitig begrenztes Arbeitsband.

- Die Arbeit auf dem Arbeitsband erfolgt wie bei der Datenstruktur *Keller*. Dies bedeutet, dass jeweils nur das am weitesten links stehende Symbol verändert werden kann und nur Anfügungen nach links vorgenommen werden können. Damit werden nach einem Symbol γ rechts erzeugte Symbole nicht bearbeitet, bevor γ bearbeitet wird. Daher bezeichnet man diese Arbeitsweise auch als *zuletzt hinein - zuerst hinaus* (engl. *last in - first out* oder abgekürzt LIFO). [3]

  Wir werden das Arbeitsband auch als Keller bezeichnen.

Anschaulich ist diese Interpretation in der Abbildung 2.7 dargestellt.
Wir geben nun die formale Definition des entsprechenden Automatentyps.

Definition 2.54 *Ein Kellerautomat ist ein Sechstupel*

$$\mathcal{M} = (X, Z, \Gamma, z_0, F, \delta),$$

wobei
- X das Eingabealphabet ist,
- Z die endliche Menge von Zuständen ist,
- Γ das Bandalphabet ist,
- $z_0 \in Z$ und $F \subseteq Z$ gelten,
- δ eine Funktion von $Z \times X \times (\Gamma \cup \{\#\})$ in die Menge der endlichen Teilmengen von $Z \times \{R, N\} \times \Gamma^*$ ist, wobei $\# \notin \Gamma$, R und N zusätzliche Symbole sind.

[3] Wir bemerken, dass dieses Prinzip auch bei einem üblichen Keller gilt; was als letztes in den Keller getragen wird, ist auch als erstes herauszuholen, sonst kann auf vorher Eingelagertes nicht zugegriffen werden. Hierin liegt der Grund für die Bezeichnung des Automatentyps.

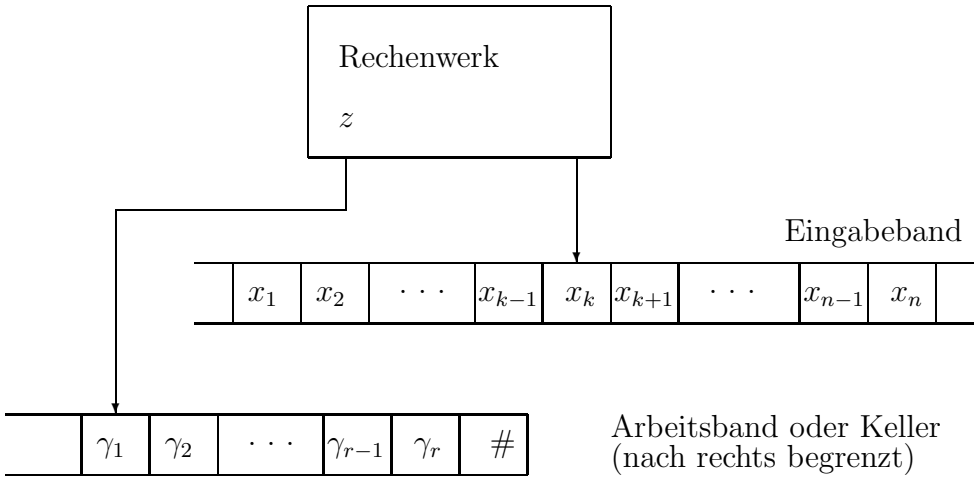


Abbildung 2.7: Schematische Darstellung eines Kellerautomaten

Die Arbeitsweise des Kellerautomaten wird wie folgt festgelegt.

Definition 2.55 *Es sei $\mathcal{M} = (X, Z, \Gamma, z_0, F, \delta)$ ein Kellerautomat wie in Definition 2.54. Eine Konfiguration K des Kellerautomaten $\mathcal{M}$ ist ein Tripel $(w, z, \alpha\#)$ mit $w \in X^*$, $z \in Z$ und $\alpha \in \Gamma^*$.*
Der Übergang von einer Konfiguration K_1 in die nachfolgende Konfiguration K_2 (den wir erneut mit $K_1 \models K_2$ bezeichnen) wird wie folgt beschreiben: Für $x \in X, v \in X^*, z \in Z, z' \in Z, \gamma \in \Gamma, \beta \in \Gamma^*, \alpha \in \Gamma^*$ gilt

$$\begin{array}{lll}
(xv, z, \gamma\alpha\#) \models (v, z', \beta\alpha\#), & \textit{falls} & (z', R, \beta) \in \delta(z, x, \gamma), \\
(xv, z, \gamma\alpha\#) \models (xv, z', \beta\alpha\#), & \textit{falls} & (z', N, \beta) \in \delta(z, x, \gamma), \\
(xv, z, \#) \models (v, z', \beta\#), & \textit{falls} & (z', R, \beta) \in \delta(z, x, \#), \\
(xv, z, \#) \models (xv, z', \beta\#), & \textit{falls} & (z', N, \beta) \in \delta(z, x, \#).
\end{array}$$

Eine Konfiguration $K = (w, z, \alpha\#)$ gibt den noch nicht gelesenen Teil w des Eingabewortes, den Zustand z des Automaten und das Wort auf dem Arbeitsband (im Keller) an.
Anschaulich wird bei einer Konfigurationsüberführung ausgehend vom Zustand, in dem sich das Rechenwerk befindet, dem gelesenen Symbol und dem ersten Kellersymbol ein neuer Zustand ermittelt und der Inhalt des Kellers verändert, indem das erste Symbol durch ein Wort ersetzt wird oder ein Wort vorangestellt wird. Genauer heißt dies:

- $(z', R, \beta) \in \delta(z, x, \gamma)$ bedeutet, dass der Kellerautomat, der sich im Zustand z befindet, das Symbol x liest und γ als erstes Symbol im Keller hat, in den Zustand z' geht, den Lesekopf des Eingabebandes nach rechts bewegt und das Symbol γ durch das Wort β ersetzt,

- $(z', N, \beta) \in \delta(z, x, \gamma)$ bedeutet, dass der Kellerautomat, der sich im Zustand z befindet, das Symbol x liest und γ als erstes Symbol im Keller hat, in den Zustand z'

geht, den Lesekopf nicht bewegt[4] und das Symbol γ durch das Wort β ersetzt,

- $(z', R, \beta) \in \delta(z, x, \#)$ bedeutet, dass der Kellerautomat, der sich im Zustand z befindet, das Symbol x liest und nur # im Keller hat, in den Zustand z' geht, den Lesekopf des Eingabebandes nach rechts bewegt und das Wort β vor # in den Keller schreibt,

- $(z', N, \beta) \in \delta(z, x, \#)$ bedeutet, dass der Kellerautomat, der sich im Zustand z befindet, das Symbol x x liest und nur # im Keller hat, in den Zustand z' geht, keine Kopfbewegung ausführt und β vor # in den Keller schreibt.

Der Lese-/Schreibkopf des Kellers (Arbeitsbandes) wird stets auf das erste Symbol im Keller gesetzt.

Mit $\models^*$ bezeichnen wir erneut den reflexiven und transitiven Abschluss der Relation $\models$.

Definition 2.56 *Es sei $\mathcal{M}$ ein Kellerautomat wie in Definition 2.54. Die von $\mathcal{M}$ akzeptierte Sprache ist durch*

$$\mathcal{M} = \{w : (w, z_0, \#) \models^* (\lambda, q, \#) \textit{ für ein } q \in F\}$$

definiert.

Ein Wort wird entsprechend dieser Definition akzeptiert, wenn ausgehend vom Anfangszustand und einem leeren Keller (d.h. der Keller enthält nur das Markierungssymbol #) nach dem vollständigen Lesen des Eingabewortes der Keller wieder leer ist und sich der Automat in einem akzeptierenden Zustand befindet. (Es lassen sich noch andere Varianten des Akzeptierens denken, so z. B. durch die Forderung, dass unabhängig vom Kellerinhalt nur ein akzeptierender Zustand erreicht wird oder unabhängig vom Zustand der Keller leer ist. Man kann beweisen, dass diese Varianten die Menge der akzeptierbaren Sprachen jedoch nicht verändern.)

Beispiel 2.57 Wir betrachten den Kellerautomaten

$$\mathcal{M} = (X, Z, \Gamma, z_0, F, \delta)$$

mit

$$\begin{aligned} &X = \{a, b\}, \quad \Gamma = \{a\}, \quad Z = \{z_0, z_1, z_2\}, \quad F = \{z_1\}, \\ &\delta(z_0, a, \#) = \{(z_0, R, aa)\}, \quad \delta(z_0, a, a) = \{(z_0, R, aaa)\}, \\ &\delta(z_0, b, a) = \{(z_1, R, \lambda)\}, \qquad \delta(q, b, a) = \{(z_1, R, \lambda)\} \end{aligned}$$

und

$$\delta(z, x, \gamma) = \{(z_2, R, \gamma)\}$$

[4]In manchen Lehrbüchern wird das Bewegen des Kopfes anders interpretiert. Eine Bewegung nach rechts entspricht dem Lesen des Symbols, während die Nichtbewegung als Nichtlesen gedeutet wird.

in allen sonstigen Fällen. Dann ergeben sich für die Eingaben $aabbbb$ und aba die folgenden Folgen von Konfigurationen:

$$\begin{aligned}(aabbbb, z_0, \#) &\models (abbbb, z_0, aa\#) \models (bbbb, z_0, aaaa\#) \models (bbb, z_1, aaa\#) \\ &\models (bb, z_1, aa\#) \models (b, z_1, a\#) \models (\lambda, z_1, \#)\end{aligned}$$

und

$$(aba, z_0, \#) \models (ba, z_0, aa\#) \models (a, z_1, a\#) \models (\lambda, z_2, \#).$$

Damit gelten $aabbbb \in T(\mathcal{M})$ und $aba \notin T(\mathcal{M})$.
Es ist leicht zu sehen, dass im Zustand z_0 beim Lesen eines a auf dem Band und einem a oder # an der Spitze des Kellers jeweils zwei a zusätzlich in den Keller geschrieben werden. Beim Lesen des ersten b wird in den Zustand z_1 gewechselt, und es beginnt der Prozeß des Kürzens des Kellers um ein a. Dies wird solange fortgesetzt, wie b gelesen werden und a im Keller sind. In allen anderen Situationen wird der Zustand z_2 erreicht, den der Kellerautomat nicht mehr verlassen kann, womit eine Akzeptanz des Wortes verhindert ist.
Da doppelt soviele a in den Keller geschrieben werden wie gelesen werden, müssen doppelt soviele b wie a gelesen werden, um den Keller wieder zu leeren. Folglich gilt

$$T(\mathcal{M}) = \{a^n b^{2n} : n \geq 1\}.$$

Die Idee hinter $\mathcal{M}$ ist im Wesentlichen folgende: Im Keller wird die Struktur des gelesenen Teilwortes gespeichert und dann mit dem noch nicht gelesenen Teilwort verglichen, wobei nur bei positiven Ausgang des Vergleichs das Eingabewort akzeptiert wird.
Eine grundsätzlich andere Idee für eine Konstruktion eines Kellerautomaten, der $L = \{a^n b^{2n} : n \geq 1\}$ akzeptiert, besteht darin, im Keller durch Speicherung der Satzformen im Wesentlichen die Ableitung der Wörter aus L zu simulieren, und dann das terminale Wort mit dem Eingabewort zu vergleichen. So einfach ist diese Idee aber nicht zu realisieren, da bei der Ableitung auch Nichtterminale zu ersetzen sind, die nicht am Wortanfang stehen, was nach der Arbeitsweise des Kellers nicht möglich ist. Gelöst wird dies Problem dadurch, dass immer bereits die Satzform und das Eingabewort soweit verglichen werden, wie dies zu dem Zeitpunkt möglich ist.
Wir formalisieren nun die eben beschriebenen Vorgehensweise, in dem wir den zugehörigen Kellerautomaten angeben. Dafür benötigen wir eine L erzeugende Grammatik. Dies ist

$$G = (\{S\}, \{a, b\}, \{S \to aSbb, S \to abb\}, S).$$

Wir definieren nun

$$\mathcal{M}' = (\{a, b\}, \{z_0', z_1', z_2'\}, \{S, a, b\}, z_0', \{z_1'\}, \delta')$$

mit

$$\delta'(z_0', x, \#) = \{(z_1', N, S)\} \quad \text{für } x \in \{a, b\}$$

(wir initialisieren den Keller mit dem Startsymbol S, das die zu Beginn vorliegende Satzform ist),

$$\delta'(z_1', x, S) = \{(z_1', N, aSbb), (z_1', N, abb)\} \quad \text{für } x \in \{a, b\}$$

(wir simulieren die Anwendung einer Regel für S im Keller, d.h. wir ersetzen S im Keller durch die rechte Seite einer Regel),

$$\delta'(z_1', x, x) = \{(z_1', R, \lambda)\} \quad \text{für } x \in \{a, b\}$$

(wir vergleichen das erste Symbol des Kellers mit dem gerade gelesenen Buchstaben auf dem Band) und

$$\delta'(z, x, \gamma) = \{(z_2', R, \lambda)\}$$

in allen weiteren Fällen. Für das obige Eingabewort $aabbbb$ erhalten wir

$$\begin{aligned}(aabbbb, z_0', \#) &\models (aabbbb, z_1', S\#) \models (aabbbb, z_1', aSbb\#) \models (abbbb, z_1', Sbb\#) \\ &\models (abbbb, z_1', abbbb\#) \models (bbbb, z_1', bbbb\#) \models (bbb, z_1', bbb\#) \\ &\models (bb, z_1', bb\#) \models (b, z_1', b\#) \models (\lambda, z_1', \#)\end{aligned}$$

als Konfigurationsfolge. Dagegen ist

$$(aba, z_0', \#) \models (aba, z_1', S\#) \models (aba, z_1', abb\#) \models (ba, z_1', bb\#) \models (a, z_1', b\#) \models (\lambda, z_2', \#)$$

nur eine mögliche Folge von Konfigurationen für die Eingabe aba, bei der eine Konfiguration vorliegt, die nicht mehr verändert werden kann, jedoch kann man sich leicht überlegen, dass wir bei jeder anderen Konfigurationsfolge auch $(\lambda, z_2', \#)$ erreichen.
Im Keller sei $Sb^{2n}\#$ enthalten (aus der Anfangskonfiguration $(w, z_0, \#)$ erhalten wir diese Situation mit $n = 0$ im ersten Schritt der Arbeit von $\mathcal{M}$'). Nun wird eine der Regeln $S \to aSbb$ oder $S \to abb$ simuliert, wodurch im Keller $aSb^{2(n+1)}\#$ oder $ab^{2(n+1)}\#$ entsteht. Im ersten Fall lesen wir einen Buchstaben, vergleichen diesen mit dem Spitzensymbol a des Kellers und erhalten $Sb^{2(n+1)}\#$, d.h. ein Wort der Form wie zu Beginn der Betrachtungen, oder erreichen den Zustand z_2', wodurch Akzeptanz ausgeschlossen wird. Im zweiten Fall vergleichen wir das noch nicht gelesene Wort mit dem Kellerinhalt und kommen bei Übereinstimmung zur Akzeptanz oder bei Nichtübereinstimmung in den Zustand z_2'.
Hieraus folgt, dass wir das Eingabewort dann akzeptieren, wenn es mit einem bei der Simulation erzeugten terminalen Satzform übereinstimmt. Somit gilt

$$T(\mathcal{M}) = L(G) = L.$$

Wir verallgemeinern nun die Idee der zweiten Konstruktion, um zu zeigen, dass durch Simulation von Ableitungen in kontextfreien Grammatiken die Akzeptierbarkeit der zugehörigen Sprache bewiesen werden kann.
Im Beispiel haben wir nach partiellem Vergleich immer das erste Nichtterminal der Satzform erreicht und dann nachfolgend keine Schwierigkeiten bekommen, da dies auch das einzige Nichtterminal der Satzform ist. Für die Verallgemeinerung ist es daher notwendig, zu zeigen, dass wir die erzeugte Sprache nicht verändern, wenn wir stets das am weitesten links stehende Nichtterminal ersetzen. Derartige Ableitungen nennen wir *Linksableitungen.*
Seien nun eine Satzform $w_1Aw_2Bw_3$ und zwei Regeln $A \to v_1$ und $B \to v_2$ gegeben. Dann bestehen die Ableitungen

$$w_1Aw_2Bw_3 \Longrightarrow w_1v_1w_2Bw_3 \Longrightarrow w_1v_1w_2v_2w_3$$

und

$$w_1Aw_2Bw_3 \Longrightarrow w_1Aw_2v_2w_3 \Longrightarrow w_1v_1w_2v_2w_3,$$

die beide zum gleichen Ergebnis führen. Somit ist eine derartige Vertauschung der Reihenfolge der Regelanwendungen möglich, ohne das erzeugte Wort zu verändern. Fortgesetzte derartige Änderung der Reihenfolge führt dazu, dass wir eine Linksableitung erhalten und das gleiche Wort erzeugen. Daraus folgt, dass wir bei Beschränkung auf Linksableitungen die gleiche Sprache erzeugen wie mittels beliebiger Ableitungen.

Lemma 2.58 *Für jede kontextfreie Sprache L gibt es einen Kellerautomaten $\mathcal{M}$ mit $T(\mathcal{M}) = L$.*

Beweis. Es sei $G = (N', T, P, S)$ eine kontextfreie Grammatik mit $L(G) = L$. Wir konstruieren nun zu G den Kellerautomaten

$$\mathcal{M} = (T, \{z_0, z_1, z_2\}, N' \cup T, z_0, \{z_1\}, \delta)$$

mit

$$\begin{aligned}
&\delta(z_0, x, \#) = \{(z_1, N, S)\} \quad \text{für } x \in T,\\
&\delta(z_1, x, A) = \{(z_1, N, v) : A \to v \in P\} \quad \text{für } x \in T,\\
&\delta(z_1, x, x) = \{(z_1, R, \lambda)\} \quad \text{für } x \in T
\end{aligned}$$

und

$$\delta(z, x, \gamma) = \{(z_2, R, \lambda)\}$$

in allen weiteren Fällen.

Zuerst bemerken wir, dass - abgesehen von der Anfangskonfiguration - nur Konfigurationen entstehen, die den Zustand z_1 oder z_2 enthalten. Ferner wird bei Erreichen des Zustands z_2 dieser nicht mehr verändert, womit eine Akzeptanz von w nicht mehr möglich ist. Wir untersuchen daher jetzt, welche Konfigurationen mit dem Zustand z_1 erreicht werden können. Wir zeigen, dass

$$(*) \qquad (w_1w_2, z_0, \#) \models^* (w_2, z_1, v\#)$$

genau dann gilt, wenn es eine Linksableitung

$$(**) \qquad S \Longrightarrow^* w_1v$$

in G gibt. Hieraus folgt dann mit $w = w_1, \lambda = w_2, v = \lambda$, dass $(\lambda, z_1, \#)$ genau dann erreicht wird, wenn es eine Linksableitung $S \Longrightarrow^* w$ gibt. Somit wird ein Wort genau dann akzeptiert, wenn es durch eine Linksableitung erzeugt werden kann. Nach den Bemerkungen vor diesem Lemma gilt folglich $T(\mathcal{M}) = L(G) = L$, womit das Lemma bewiesen ist.

$(*) \to (**)$. Wir benutzen absteigende Induktion über die Länge des noch nicht gelesenen Wortes. Für $w_1 = \lambda, w_2 = w, v = S$ gilt die Behauptung, da ausgehend von $(w, z_0, \#)$ in einem Schritt nur $(w, z_1, S\#)$ erreicht werden kann und $S \Longrightarrow^* S$ eine Linksableitung (mit null Ableitungsschritten) ist.

Sei nun $(w_1w_2, z_0, \#) \models^* (w_2, z_1, v\#)$ eine Überführung, bei der $w_2 \neq \lambda$ ist und für die eine Linksableitung $S \Longrightarrow^* w_1v$ existiert. Wir unterscheiden drei Fälle:
a) $v = av'$ für ein $a \in T$. Gilt auch $w_2 = aw_2'$, so gelten

$$(w_1aw_2', z_0, \#) \models^* (aw_2', z_1, av'\#) \models (w_2', z_1, v'\#)$$

und

$$S \Longrightarrow^* w_1v = w_1av',$$

womit die Aussage für das kürzere Wort w_2' gilt. Gilt dagegen $w_2 = bw_2'$ mit $a \neq b$, so kann nur in den Zustand z_2 übergegangen werden.
b) $v = Av'$ für ein $A \in N$. Ferner sei $A \to Xx$ eine Regel aus P. Dann erhalten wir durch Simulation dieser Regel

$$(w_2, z_1, Av'\#) \models (w_2, z_1, Xxv'\#).$$

Ist $X \in T$, so erreichen wir damit die unter a) diskutierte Situation, ist $X \in N$ fahren wir wie beschrieben fort, bis wir zu einer Anwendung einer Regel kommen, deren rechte Seite mit einem Terminal beginnt.
c) $v = \lambda$. Wegen $w_2 \neq \lambda$ gehen wir in den Zustand z_2.
$(**) \to (*)$. Wir führen den Beweis mittels vollständiger Induktion über die Anzahl der Ableitungsschritte in $(**)$ und zeigen sogar die schärfere Aussage: Ist nach n Ableitungsschritten in einer Linksableitung das Wort w_1Au mit $w_1 \in T^*$ erzeugt worden, so gibt es die Überführung $(w_1w_2, z_0, \#) \models^* (w_2, z_1, Au)$.
Für $n = 0$ folgt die Aussage mit $w_1 = \lambda, w_2 = w$ direkt aus der nach Definition existierenden Überführung $(w_2, z_0, \#) \models (w_2, z_1, S\#)$ und dem Fakt, dass in null Schritten nur S erzeugbar ist.
Sei die Aussage nun schon für n bewiesen. Ferner sei

$$S \Longrightarrow^* w_1Au \Longrightarrow w_1v_1Bv_2u$$

eine Linksableitung aus $n+1$ Ableitungsschritten, wobei der letzte Schritt in der Anwendung der $A \to v_1Bv_2$ mit $v_1 \in T^*$ besteht. Da es eine Linksableitung ist, gilt $w_1 \in T^*$. Nach Induktionsvoraussetzung gilt daher

$$(w_1w_2, z_0, \#) \models^* (w_2, z_1, Au\#).$$

Aufgrund der Definition von $\mathcal{M}$ gilt dann weiterhin

$$(w_2, z_1, Au\#) \models (w_2, z_1, v_1Bv_2u\#).$$

Falls $w_2 = v_1w_3$ erhalten wir außerdem

$$(v_1w_3, z_1, v_1Bv_2u\#) \models^* (w_3, z_1, Bv_2u\#)$$

und durch Kombination dieser Relation

$$(w_1v_1w_3, z_0, \#) \models^* (w_3, z_1, Bv_2u\#),$$

womit die Aussage auch für die Ableitung aus $n+1$ Schritten gilt. Ist aber $w_2 \neq v_1w_3$ für alle $w_3 \in T^*$, so erreichen wir ausgehend von $(w_2, z_1, v_1Bv_2u\#)$ den Zustand z_2. □

Ohne Beweis geben wir das folgende Lemma.

Lemma 2.59 *Zu jedem Kellerautomaten $\mathcal{M}$ gibt es eine kontextfreie Grammatik G mit $L(G) = T(\mathcal{M})$.* □

Durch Kombination der beiden vorstehenden Lemmata erhalten wir das Hauptresultat dieses Abschnittes.

Satz 2.60 *Die beiden folgenden Aussagen sind für eine Sprache L äquivalent:*
i) L ist eine kontextfreie Sprache.
ii) $L = T(\mathcal{M})$ gilt für einen Kellerautomaten $\mathcal{M}$. □

Der Kellerautomat ist nach Definition nichtdeterministisch. Auch hier kann eine deterministische Variante eingeführt werden, bei der es zu jeder Konfiguration genau eine Folgekonfiguration gibt. Dafür reicht es zu fordern, dass alle Mengen $\delta(z, x, \gamma)$ einelementig sind. Der in Beispiel 2.57 angegebene Kellerautomat $\mathcal{M}$ ist deterministisch. Damit ist klar, dass deterministische Kellerautomaten nichtreguläre Sprachen akzeptieren können. Andererseits kann gezeigt werden, dass deterministische Kellerautomaten nicht in der Lage sind, die Sprache $\{ww^R : w \in \{a, b\}\}$ zu akzeptieren. Somit liegt die Menge der von deterministischen Kellerautomaten akzeptierten Sprachen echt zwischen der der regulären Sprachen und der der kontextfreien Sprachen.

2.3 Sprachen und algebraische Operationen

Nachdem wir im Abschnitt 2.1 verschiedene Typen formaler Sprachen mittels erzeugender Grammatiken definiert haben, gelang uns im Abschnitt 2.2 eine Charakterisierung der zugehörigen Sprachmengen mittels verschiedener Typen von Automaten. Ziel dieses Abschnittes ist es, eine weitere Charakterisierung der Menge der regulären Sprachen anzugeben, indem wir zeigen, dass sie sich als spezielle (universelle) Algebra beschreiben lassen.

Da Sprachen Mengen sind, können wir auf diese problemlos die mengentheoretischen Operationen Vereinigung und Durchschnitt anwenden. Eine weitere wichtige mengentheoretische Operation ist die Komplementbildung, bei der aber erst zu klären ist, bezüglich welcher Gesamtheit das Komplement zu bilden ist. Sei zum Beispiel $L \subseteq X^*$. Dann ist sicher $X^* \setminus L$ eine mögliche Definition des Komplements. Jedoch gilt natürlich für jedes Symbol $a \notin X$ auch $L \subseteq (X \cup \{a\})^*$, womit auch $(X \cup \{a\})^* \setminus L$ als Komplement möglich wäre. Wir wollen uns hier auf den Fall beschränken, dass das zugrunde liegende Alphabet minimal gewählt wird.
Für eine Sprache L definieren wir $alph(L)$ als die Menge aller Buchstaben, die in mindestens einem Wort von L vorkommen und das Komplement von L als

$$\overline{L} = (alph(L))^* \setminus L.$$

Wir definieren nun einige der Algebra entlehnten Operationen.

Definition 2.61 *Es seien L, L_1, L_2 Sprachen über einem Alphabet X. Wir definieren dann das Produkt von L_1 und L_2 durch*

$$L_1 \cdot L_2 = \{w_1 w_2 : w_1 \in L_1, w_2 \in L_2\}.$$

Weiterhin setzen wir

$$\begin{aligned} L^0 &= \{\lambda\}, \\ L^{n+1} &= L^n \cdot L \quad \textit{für} \quad n \geq 0 \end{aligned}$$

und definieren den KLEENE-*Abschluss (oder* KLEENE-$*$*) von L durch*

$$L^* = \bigcup_{n \geq 0} L^n$$

und den positiven KLEENE-*Abschluss (oder* KLEENE-$+$*) von L durch*

$$L^+ = \bigcup_{n \geq 1} L^n.$$

Falls keine Missdeutungen möglich sind, lassen wir wie üblich den Punkt als Operationszeichen beim Produkt fort.

Beispiel 2.62 Seien

$$L = \{ab, ac\} \quad \text{und} \quad L' = \{ab^n a : n \geq 1\}$$

gegeben. Dann ergeben sich:

$$\begin{aligned} L \cdot L = L^2 &= \{abab, abac, acab, acac\}, \\ L \cdot L' &= \{abab^n a : n \geq 1\} \cup \{acab^n a : n \geq 1\}, \\ (L')^3 &= \{ab^i aab^j aab^k a : i \geq 1, j \geq 1, k \geq 1\}, \\ L^* &= \{ax_1 ax_2 \ldots ax_r : r \geq 1, x_i \in \{b, c\}, 1 \leq i \leq r\} \cup \{\lambda\}, \\ (L')^+ &= \{ab^{s_1} aab^{s_2} a \ldots ab^{s_t} a : t \geq 1, s_j \geq 1, 1 \leq j \leq t\}. \end{aligned}$$

Vom algebraischen Standpunkt aus ist das Produkt das übliche Komplexprodukt in der (freien) Halbgruppe der Wörter über X. L^* ist dann die kleinste Halbgruppe mit neutralem Element, die L enthält, und L^+ ist entsprechend die kleinste Halbgruppe, die L enthält.
Wir bemerken, dass nach Definition stets

$$L^* = L^+ \cup L^0 = L^+ \cup \{\lambda\}$$

gilt, während $L^+ = L^* \setminus \{\lambda\}$ nur dann gilt, wenn $\lambda \notin L$ gilt.
Weiterhin merken wir an, dass im Spezialfall $L = X$ die Menge L^n aus genau allen Wörtern der Länge n über X besteht. Somit ist dann L^* die Menge aller Wörter über X, d.h. $L^* = X^*$, womit auch die Rechtfertigung für die Bezeichnung X^* in diesem Zusammenhang nachgewiesen ist.
Mit Hilfe der mengentheoretischen und den eben eingeführten Operationen lassen sich einige Sprachen sehr einfach beschreiben, für die wir bisher „relativ umständliche“ Definitionen gegeben haben. Wir wollen dies an einigen Beispielen demonstrieren.
Da offensichtlich nach Definition für jedes Symbol x

$$\{x\}^* = \{x^n : n \geq 0\} \quad \text{und} \quad \{x\}^+ = \{x^n : n \geq 1\} = \{x\}\{x\}^*$$

gelten, können wir die in den Beispielen 2.47 bzw. 2.48 akzeptierten (regulären) Sprachen wie folgt beschrieben:

$$\begin{aligned}\{c^{n_1}aac^{n_2}aa\ldots c^{n_k}aa : k \geq 1, n_1 \geq 0, n_i \geq 1, 2 \leq i \leq k\} &= \{c\}^*\{a\}\{a\}(\{c\}^+\{a\}\{a\})^* \\ &= \{c\}^*\{a\}\{a\}(\{c\}\{c\}^*\{a\}\{a\})^*\end{aligned}$$

und

$$\{a^nb^m : n \geq 1, m \geq 2\} = \{a\}^+\{b\}\{b\}^+.$$

Die Sprache R bestehe aus allen Wörtern über dem Alphabet X, die mindestens einen Buchstaben aus der Menge $Y \subseteq X$ enthalten. Hierfür ergibt sich

$$R = \bigcup_{x \in Y} X^*\{x\}X^*.$$

Satz 2.63 *Wenn L und L' reguläre Sprachen sind, so sind auch die Sprachen*
i) $L \cup L'$,
ii) $L \cap L'$,
iii) $V^* \setminus L$ (wobei $L \subseteq V^*$ gilt),
iv) $L \cdot L'$,
v) L^+ und L^*
regulär.

Beweis. i) Es seien L_1 und L_2 zwei reguläre Sprachen über dem Alphabet T. Wir haben zu zeigen, dass auch $L_1 \cup L_2$ eine reguläre Sprache (über T) ist. Dazu seien

$$G_1 = (N_1, T_1, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T_2, P_2, S_2)$$

zwei reguläre Grammatiken mit

$$L(G_1) = L_1 \quad \text{und} \quad L(G_2) = L_2.$$

Offenbar können wir ohne Beschränkung der Allgemeinheit annehmen, dass

$$T_1 = T_2 = T \quad \text{und} \quad N_1 \cap N_2 = \emptyset$$

gelten (notfalls sind die Nichtterminale umzubenennen). Ferner sei S ein Symbol, das nicht in $N_1 \cup N_2 \cup T$ liegt. Wir betrachten nun die reguläre Grammatik

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S).$$

Offenbar hat jede Ableitung in G die Form

$$S \Longrightarrow S_i \Longrightarrow^* w, \tag{2.1}$$

wobei $i \in \{1, 2\}$ gilt und $S_i \Longrightarrow^* w$ eine Ableitung in G_i ist (da wegen $N_1 \cap N_2 = \emptyset$ keine Symbole aus N_j, $j \neq i$ entstehen können und damit keine Regeln aus P_j anwendbar sind). Folglich gilt $w \in L(G_i)$. Hieraus folgt sofort

$$L(G) \subseteq L(G_1) \cup L(G_2) = L_1 \cup L_2.$$

Man sieht aber auch aus (2.1) sofort, dass jedes Element aus $L(G_i)$, $i \in \{1,2\}$, erzeugt werden kann, womit auch die umgekehrte Inklusion

$$L(G) \supseteq L(G_1) \cup L(G_2) = L_1 \cup L_2$$

gezeigt ist.

ii) Wir haben zu zeigen, dass für zwei reguläre Sprachen L_1 und L_2 auch ihr Durchschnitt $L_1 \cap L_2$ regulär ist. Wir führen den Beweis nur für den Fall dass $\lambda \notin L_1 \cap L_2$ liegt und überlassen dem Leser die Modifikationen für die allgemeine Situation.
Es seien dazu wieder

$$G_1 = (N_1, T_1, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T_2, P_2, S_2)$$

reguläre Grammatiken mit

$$L(G_1) = L_1 \quad \text{und} \quad L(G_2) = L_2.$$

Diesmal können wir ohne Beschränkung der Allgemeinheit neben $T = T_1 = T_2$ noch annehmen, dass G_1 und G_2 den in Satz 2.24 gegebenen Bedingungen genügen. Wir betrachten diesmal die reguläre Grammatik

$$G = (N_1 \times N_2, T, P, (S_1, S_2))$$

mit

$$\begin{aligned} P \;=\; & \{(A_1, A_2) \to a(B_1, B_2) : A_1 \to aB_1 \in P_1, A_2 \to aB_2 \in P_2\} \\ & \cup \{(A_1, A_2) \to a : A_1 \to a \in P_1, A_2 \to a \in P_2\}. \end{aligned}$$

Es ist leicht zu sehen, dass

$$(S_1, S_2) \Longrightarrow^* w'(A_1, A_2) \Longrightarrow^* w$$

genau dann gilt, wenn es in G_1 und G_2 Ableitungen

$$S_1 \Longrightarrow^* w'A_1 \Longrightarrow^* w \quad \text{und} \quad S_2 \Longrightarrow^* w'A_2 \Longrightarrow^* w$$

gibt. Folglich gilt $w \in L(G)$ genau dann, wenn auch $w \in L(G_1)$ und $w \in L(G_2)$ erfüllt sind. Somit ergibt sich

$$L(G) = L(G_1) \cap L(G_2) = L_1 \cap L_2.$$

Damit ist der Durchschnitt von L_1 und L_2 als regulär nachgewiesen.

iii) Es sei L eine reguläre Sprache. Dann gibt es einen endlichen Automaten

$$\mathcal{A} = (alph(L), Z, z_0, F, \delta)$$

mit $T(\mathcal{A}) = L$, der also L akzeptiert. Offenbar gilt daher genau dann $w \in \overline{L}$ oder gleichwertig $w \notin T(\mathcal{A})$, wenn $\delta(z_0, w) \notin F$, d.h. $\delta(z_0, w) \in Z \setminus F$ ist. Somit akzeptiert der endliche Automat

$$\mathcal{A}' = (alph(L), Z, z_0, Z \setminus F, \delta)$$

das Komplement von L, welches damit als regulär nachgewiesen ist.
iv) Es seien wieder

$$G_1 = (N_1, T_1, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T_2, P_2, S_2)$$

reguläre Grammatiken mit

$$L(G_1) = L_1 \quad \text{und} \quad L(G_2) = L_2$$

und $N_1 \cap N_2 = \emptyset$. Wir konstruieren aus G_1 und G_2 die reguläre Grammatik

$$G = (N_1 \cup N_2, T, P_1' \cup P_2, S_1)$$

mit

$$P_1' = \{A \to wB : A \to wB \in P_1, B \in N_1\} \cup \{A \to wS_2 : A \to w \in P_1, w \in T^*\}.$$

Entsprechend dieser Konstruktion sind die Ableitungen in G von der Form

$$S_1 \Longrightarrow^* w'A \Longrightarrow w'wS_2 \Longrightarrow^* w'ww_2,$$

wobei $S_1 \Longrightarrow^* w'A \Longrightarrow w'w = w_1$ eine Ableitung in G_1 und $S_2 \Longrightarrow^* w_2$ eine Ableitung in G_2 sind. Damit ergibt sich

$$L(G) = \{w_1w_2 : w_1 \in L(G_1), w_2 \in L(G_2)\} = L(G_1) \cdot L(G_2).$$

v) Wir beweisen die Aussage zuerst für L^+.
Es sei $G = (N, T, P, S)$ eine reguläre Grammatik mit $L(G) = L$. Wir konstruieren die reguläre GRammatik $G' = (N, T, P', S)$, wobei P' aus P entsteht, indem wir zu P die Regeln

$$A \to wS \text{ für } A \to w \in P, w \in T^*$$

hinzufügen. Die Ableitungen sind dann (bis auf die Reihenfolge der Anwendung der Regeln) von der Form

$$\begin{aligned} S &\Longrightarrow w_1'A_1 \Longrightarrow w_1'w_1''S \Longrightarrow^* w_1'w_2''w_2'A_2 \Longrightarrow w_1'w_1''w_2'w_2''S \\ &\Longrightarrow^* w_1'w_1'' \dots w_{n-1}'w_{n-1}''S \Longrightarrow^* w_1'w_1'' \dots w_{n-1}'w_{n-1}''w_n, \end{aligned}$$

wobei $w_i'w_i'' \in L(G)$ for $1 \le i \le n-1$ und $w_n \in L(G)$ gelten. Hieraus folgt leicht die zu beweisende Aussage.
Wir geben nun die Modifikationen für den Kleene-$*$. Gilt $\lambda \in L$, so können wir wegen der dann gegebenen Gültigkeit von $L^* = L^+$ wie oben vorgehen. Ist $\lambda \notin L$, so haben wir $L^* = L^+ \cup \{\lambda\}$. Da $\{\lambda\}$ eine reguläre Sprache ist (erzeugt von der Grammatik mit der einzigen Regel $S \to \lambda$), folgt die Regularität von L^* aus Teil i) dieses Satzes. □

Wir haben oben Beispiele betrachtet, bei denen (reguläre) Sprachen erzeugt werden konnten, indem die Operationen Vereinigung, Produkt und (positiver) Kleene-Abschluss auf einelementige Mengen iteriert angewandt wurden. Wir wollen nun das auf S. C. Kleene zurückgehende Resultat zeigen, dass auf diese Weise genau die regulären Sprachen beschrieben werden können. Dafür verwenden wir reguläre Ausdrücke, die auch an anderer Stelle in der Informatik zur Beschreibung von Mengen eingesetzt werden.

Definition 2.64 *Reguläre Ausdrücke über einem Alphabet X sind induktiv wie folgt definiert:*

1. *$\emptyset$, λ und x mit $x \in X$ sind reguläre Ausdrücke.*
2. *Sind R_1, R_2 und R reguläre Ausdrücke, so sind auch $(R_1 + R_2)$, $(R_1 \cdot R_2)$ und R^* reguläre Ausdrücke.*
3. *Ein Ausdruck ist nur dann regulär, wenn dies aufgrund von 1. oder 2. der Fall ist.*

Wir ordnen nun jedem regulären Ausdruck über X eine Sprache über X zu.

Definition 2.65 *Die einem regulären Ausdruck U über dem Alphabet X zugeordnete Menge $M(U)$ ist induktiv durch die folgenden Festlegungen definiert:*

- *$M(\emptyset) = \emptyset$, $M(\lambda) = \{\lambda\}$ und $M(x) = \{x\}$ für $x \in X$,*
- *Sind R_1, R_2 und R reguläre Ausdrücke, so gelten*

$$\begin{aligned} M((R_1 + R_2)) &= M(R_1) \cup M(R_2), \\ M((R_1 \cdot R_2)) &= M(R_1) \cdot M(R_2), \\ M(R^*) &= (M(R))^* . \end{aligned}$$

Beispiel 2.66 Sei $X = \{a, b, c\}$. Dann sind nach 1. aus Definition 2.64

$$R_0 = \lambda,\ R_1 = a,\ R_2 = b,\ R_3 = c$$

reguläre Ausdrücke über X. Nach 2. aus Definition 2.64 sind dann auch die folgenden Konstrukte reguläre Ausdrücke:

$$\begin{aligned} R_1' &= (R_1 \cdot R_1) = (a \cdot a), \\ R_1'' &= (R_1' \cdot R_1) = ((a \cdot a) \cdot a), \\ R_2' &= R_2^* = b^*, \\ R_2'' &= (R_2' + R_1'') = (b^* + ((a \cdot a) \cdot a))), \\ R_3' &= R_3^* = c^*, \\ R_3'' &= (R_3 \cdot R_3') = (c \cdot c^*), \\ R_4 &= (R_2'' \cdot R_3'') = ((b^* + ((a \cdot a) \cdot a))) \cdot (c \cdot c^*)), \\ R_5 &= (R_0 + R_4) = (\lambda + ((b^* + ((a \cdot a) \cdot a))) \cdot (c \cdot c^*))) . \end{aligned}$$

Entsprechend Definition 2.65 erhalten wir die folgenden zugeordneten Mengen (wobei wir offensichtliche Vereinfachungen stets vornehmen):

$$\begin{aligned} M(R_0) &= \{\lambda\},\ M(R_1) = \{a\},\ M(R_2) = \{b\},\ M(R_3) = \{c\}, \\ M(R_1') &= = M((R_1 \cdot R_1)) = \{a\} \cdot \{a\} = \{a^2\}, \\ M(R_1'') &= M((R_1' \cdot R_1)) = \{a^2\} \cdot \{a\} = \{a^3\}, \\ M(R_2') &= M(R_2^*) = \{b\}^* = \{b^m : m \geq 0\}, \end{aligned}$$

$$
\begin{aligned}
M(R_2'') &= M((R_2' + R_1'')) = \{b^m : m \geq 0\} \cup \{a^3\}\,, \\
M(R_3') &= M(R_3^*) = \{c\}^* = \{c^n : n \geq 0\}\,, \\
M(R_3'') &= M((R_3 \cdot R_3')) = \{c\}\{c^n : n \geq 0\} = \{c^n : c \geq 1\}\,, \\
M(R_4) &= M((R_2'' \cdot R_3'')) = (\{b^m : m \geq 0\} \cup \{a^3\}) \cdot \{c^n : n \geq 1\} \\
&= \{b^m c^n : m \geq 0, n \geq 1\} \cup \{a^3 c^n : n \geq 3\}\,, \\
M(R_5) &= M((R_0 + R_4)) = \{\lambda\} \cup (\{b^m c^n : m \geq 0, n \geq 1\} \cup \{a^3 c^n : n \geq 3\}) \\
&= \{\lambda\} \cup \{b^m c^n : m \geq 0, n \geq 1\} \cup \{a^3 c^n : n \geq 3\}.
\end{aligned}
$$

Ist $U = ((\ldots((R_1 + R_2) + R_3) + \ldots) + R_n)$, so schreiben wir dafür kurz

$$U = \sum_{i=1}^{n} R_i\,.$$

Offenbar ist

$$M(U) = \bigcup_{i=1}^{n} M(R_i)\,.$$

In analoger Weise benutzen wir Summen bzw. Vereinigungen über gewisse Indexbereiche.

Satz 2.67 *Eine Sprache L ist genau dann regulär, wenn es einen regulären Ausdruck R mit $M(R) = L$ gibt.*

Beweis. $\Longrightarrow$) Sei L eine reguläre Sprache. Dann gibt es einen endlichen deterministischen Automaten

$$\mathcal{A} = (X, Z, z_0, F, \delta)$$

mit $T(\mathcal{A}) = L$. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass

$$Z = \{0, 1, 2, \ldots r\} \quad \text{und} \quad z_0 = 0$$

für ein gewisses $k \geq 0$ gelten. Für $i, j, k \in Z$ bezeichnen wir mit $L_{i,j}^k$ die Menge aller Wörter w mit den beiden folgenden Eigenschaften:

- $\delta(i, w) = j$,
- für jedes $u \neq \lambda$ mit $w = uu'$ und $|u| < |w|$ gilt $\delta(i, u) < k$.

Offenbar gilt dann

$$L = T(\mathcal{A}) = \bigcup_{j \in F} L_{0,j}^{r+1}. \tag{2.2}$$

Wir beweisen nun, dass es für jede Menge $L_{i,j}^k$ einen regulären Ausdruck $R_{i,j}^k$ mit $M(R_{i,j}^k) = L_{i,j}^k$ gibt. Der Beweis hierfür wird nun mittels Induktion über k gezeigt.
Sei zuerst $k = 0$. Für $i \neq j$ besteht $L_{i,j}^0$ nach Definition aus allen Wörtern w, die den Zustand i direkt in den Zustand j überführen, da aufgrund der zweiten Bedingung keine Zwischenzustände auftreten können. Damit muss w ein Wort der Länge 1 sein, und es gilt

$$L_{i,j}^0 = \{x : x \in X, \delta(i, x) = j\}.$$

Wir schreiben dies als

$$L^0_{i,j} = \bigcup_{\substack{x\in X\\ \delta(i,x)=j}} \{x\}.$$

Damit gilt auch

$$L^0_{i,j} = M(\sum_{\substack{x\in X\\ \delta(i,x)=j}} x).$$

womit die Aussage bewiesen ist. Gilt $i = j$, so kommt zu den Wörtern der Länge 1, die i in i transformieren, noch das leere Wort hinzu. Daher ist auch in diesem Fall

$$L^0_{i,j} = M(\lambda + \sum_{\substack{x\in X\\ \delta(i,x)=i}} x).$$

Sei nun $k \geq 1$ und für alle Mengen der Form $L^s_{i,j}$ mit $s < k$ existiere ein regulärer Ausdruck $R^s_{i,j}$ mit $L^s_{i,j} = M(R^s_{i,j})$. Wir zeigen zuerst

$$L^k_{i,j} = L^{k-1}_{i,k-1}(L^{k-1}_{k-1,k-1})^* L^{k-1}_{k-1,j} \cup L^{k-1}_{i,j}\,. \tag{2.3}$$

Sei $w = x_1x_2\ldots x_n$ ein Wort aus $L^k_{i,j}$. Für $1 \leq p \leq n-1$ setzen wir

$$z_p = \delta(i, x_1x_2\ldots x_p).$$

Gilt $z_p < k-1$ für $1 \leq p \leq n-1$, so ist w auch in $L^{k-1}_{i,j}$. Folglich erhalten wir $w \in R$. Deshalb sei nun für gewisse $t \geq 1$ und $1 \leq p_1 \leq p_2 \leq \ldots \leq p_t \leq n-1$

$$z_{p_1} = z_{p_2} = \ldots = z_{p_t} = k-1 \quad \text{und} \quad z_p < k-1 \text{ für } p \notin \{p_1, p_2, \ldots, p_t\}.$$

Dann gelten

$$\begin{aligned}
&\delta(i, x_1x_2\ldots x_{p_1}) = k-1,\\
&\delta(k-1, x_{p_q+1}x_{p_q+2}\ldots x_{p_{q+1}}) = k-1 \quad \text{für} \quad 1 \leq q \leq t-1,\\
&\delta(k-1, x_{p_t}x_{p_t+1}\ldots x_n) = j.
\end{aligned}$$

Weiterhin wird bei keiner dieser Überführungen als Zwischenschritt der Zustand $k-1$ erreicht. Daher erhalten wir

$$\begin{aligned}
&x_1x_2\ldots x_{p_1} \in L^{k-1}_{i,k-1},\\
&x_{p_q}x_{p_q+1}x_{p_q+2}\ldots x_{p_{q+1}} \in L^{k-1}_{k-1,k-1} \quad \text{für} \quad 1 \leq q \leq t-1,\\
&x_{p_t}x_{p_t+1}x_{p_t+2}\ldots x_n \in R^{k-1}_{k-1,j}.
\end{aligned}$$

und

$$w = x_1\ldots x_{p_1}\ldots x_{p_2}\ldots x_{p_t}\ldots x_n \in L^{k-1}_{i,k-1}(L^{k-1}_{k-1,k-1})^* L^{k-1}_{k-1,j}.$$

Folglich ist

$$L^k_{i,j} \subseteq L^{k-1}_{i,k-1}(L^{k-1}_{k-1,k-1})^* L^{k-1}_{k-1,j} \cup L^{k-1}_{i,j}\,.$$

Die umgekehrte Inklusion und damit die Gleichheit aus (2.3) folgt durch analoge Schlüsse.

(2.3) liefert nun sofort

$$\begin{aligned} L_{i,j}^k &= M(R_{i,k-1}^{k-1})M(R_{k-1,k-1}^{k-1})^* M(R_{k-1,j}^{k-1}) \cup M(L_{i,j}^{k-1}) \\ &= M((((R_{i,k-1}^{k-1} \cdot [R_{k-1,k-1}^{k-1}]^*) \cdot R_{k-1,j}^{k-1}) + R_{i,j}^{k-1}))\,, \end{aligned}$$

womit gezeigt ist, dass jede Menge $L_{i,j}^k$ durch einen regulären Ausdruck $R_{i,j}^k$ beschrieben werden kann.
Beachten wir nun noch die aus (2.2) herrührende Relation

$$L = \bigcup_{j \in F} L_{0,j}^{r+1} = M(\sum_{j \in F} R_{0,j}^{r+1})$$

so ist diese Richtung des Satzes von KLEENE gezeigt.

$\Longleftarrow$) Wir zeigen induktiv, dass für jeden regulären Ausdruck U die zugehörige Menge $M(U)$ regulär ist.
Ist U ein regulärer nach Ausdruck nach 1. aus Definition 2.64, so sind die zugehörigen Mengen $M(\emptyset) = \emptyset$, $M(\lambda) = \{\lambda\}$ und $M(x) = \{x\}$ mit $x \in X$ alle endlich und folglich auch regulär (siehe auch Übungsaufgabe 5).
Sei nun U ein regulärer Ausdruck, der aus den regulären Ausdrücken R_1, R_2 und R entsprechend 2. aus Definition 2.64 gebildet wurde, wobei die Mengen $M(R_1)$, $M(R_2)$ und $M(R)$ nach Induktionsvoraussetzung regulär sind. Falls $U = (R_1 + R_2)$ gilt, so erhalten wir $M(U) = M(R_1) \cup M(R_2)$. Nach Satz 2.63 i) ist $M(U)$ regulär. Gelten $U = (R_1 \cdot R_2)$ bzw. $U = R^*$, so sind nach den Saztz 2.63 die zugehörigen Mengen $M(U) = M(R_1) \cdot M(R_2)$ bzw. $M(U) = (M(R))^*$ ebenfalls regulär. □

Wir geben noch eine andere Formulierung des Satzes von KLEENE an, bei der wir statt der regulären Ausdrücke eine direkte Beschreibung durch die Mengenoperationen angeben, die bei der Interpretation der Ausdrücke durch Mengen auftreten.

Satz 2.67' *Eine Sprache $L \subseteq X$ ist genau dann regulär, wenn sie in endlich vielen Schritten mittels der Operationen Vereinigung, Produkt und* KLEENE*-Abschluss aus den Mengen $\emptyset$, $\{\lambda\}$ und $\{x\}$ für $x \in X$ erzeugt werden kann.* □

Das folgende Beispiel verdeutlicht die in den Beweisen der vorstehenden Lemmata angegebenen Konstruktionen.

Beispiel 2.68 Wir betrachten den endlichen Automaten $\mathcal{A}$ aus Beispiel 2.47 und konstruieren zu der durch ihn akzeptierten Sprache die Darstellung durch Vereinigung, Produkt und KLEENE-Abschluss. Zur Vereinfachung der Schreibweisen werden wir dabei statt z_i die Bezeichnung i verwenden. Es ergibt sich

$$\begin{aligned} T(\mathcal{A}) &= L_{0,2}^4 \\ &= L_{0,3}^3 (L_{3,3}^3)^* L_{3,2}^3 \cup L_{0,2}^3 \\ &= L_{0,2}^3 (\text{wegen } L_{3,2}^3 = \emptyset) \\ &= L_{0,2}^2 (L_{2,2}^2)^* L_{2,2}^2 \cup L_{0,2}^2 \\ &= L_{0,2}^2 (L_{2,2}^2)^* (\text{wegen } \lambda \in L_{0,2}^2) \\ &= (L_{0,1}^1 (L_{1,1}^1)^* L_{1,2}^1 \cup L_{0,2}^1)(L_{2,1}^1 (L_{1,1}^1)^* L_{1,2}^1 \cup L_{2,2}^1)^* \end{aligned}$$

$$
\begin{aligned}
&= L^1_{0,1}\{a\} \cdot (L^1_{2,1}\{a\})^* \text{wegen } L^1_{1,2} = \{a\}, L^1_{1,1} = L^1_{0,2} = L^1_{2,2} = \emptyset) \\
&= (L^0_{0,0}(L^0_{0,0})^* L^0_{0,1} \cup L^0_{0,1})\{a\} \cdot ((L^0_{2,0}(L^0_{0,0})^* L^0_{0,1} \cup L^0_{2,1})\{a\})^* \\
&= (\{\lambda, c\}\{\lambda, c\}^*\{a\} \cup \{a\})\{a\} \cdot ((\{c\}\{\lambda, c\}^*\{a\})\{a\})^*,
\end{aligned}
$$

woraus die abschließende Darstellung

$$T(\mathcal{A}) = ((((((\lambda + c) \cdot (\lambda + c)^*) \cdot a) + a) \cdot a) \cdot (((c \cdot (\lambda + c)^*) \cdot a) \cdot a^*))$$

gewonnen wird.
Wir bemerken, dass diese Darstellung nicht mit der auf auf Seite 98 gegebenen Darstellung

$$T(\mathcal{A}) = \{c\}^*\{a\}\{a\}(\{c\}\{c\}^*\{a\}\{a\})^*$$

identisch ist. Daher zeigt dieses Beispiel auch noch, dass es mehrere Beschreibungen durch Operationen für eine reguläre Menge geben kann.

Wir setzen das Beispiel jetzt fort, indem wir ausgehend von der Beschreibung von $T(\mathcal{A})$ eine Grammatik konstruieren, die $T(\mathcal{A})$ erzeugt. Zur Abkürzung des Prozesses starten wir mit der letzten oben gegebenen Darstellung für $T(\mathcal{A})$.
Offenbar ist für alle nachfolgenden Grammatiken die Menge T der Terminale durch die Eingabemenge $\{a, b, c\}$ von $\mathcal{A}$ gegeben.
Wir konstruieren nun zuerst Grammatiken, die die notwendigen (einelementigen) Mengen erzeugen. Ferner sichern wir dabei die Disjunktheit aller Mengen von Nichtterminalen, da diese in den Beweisen der Abgeschlossenheit unter Vereinigung, Produkt und KLEENE-Abschluss teilweise vorausgesetzt wurde. Wir gehen daher von

$$
\begin{aligned}
G_i &= (\{S_i\}, T, \{S_i \to c\}, S_i) \quad \text{für } i \in \{1, 4, 5\} \\
G_j &= (\{S_j\}, T, \{S_j \to a\}, S_j) \quad \text{für } i \in \{2, 3, 6, 7\}
\end{aligned}
$$

aus, für die

$$L(G_i) = \{c\} \quad \text{und} \quad L(G_j) = \{a\}$$

und damit auch

$$T(\mathcal{A}) = L(G_1)^* L(G_2) L(G_3)(L(G_4) L(G_5)^* L(G_6) L(G_7))^*$$

gelten. Wir gehen nun entsprechend den Konstruktionen des Satzes 2.63 vor. In der folgenden Tabelle geben wir stets die erzeugte Sprache, die Regeln und das Axiom an (die Nichtterminale können aus den Regeln abgelesen werden).

$L(G_1)^* = \{a\}^*$	$S_1' \to \lambda, S_1' \to S_1, S_1 \to cS_1, S_1 \to c$	S_1'
$L(G_1)^*L(G_2)$	$S_1' \to S_2, S_1' \to S_1, S_1 \to cS_1, S_1 \to cS_2,$ $S_2 \to a$	S_1'
$L(G_1)^*L(G_2)L(G_3)$	$S_1' \to S_2, S_1' \to S_1, S_1 \to cS_1, S_1 \to cS_2,$ $S_2 \to cS_3, S_3 \to c$	S_1'
$L(G_5)^*$	$S_5' \to \lambda, S_5' \to S_5, S_5 \to cS_5, S_5 \to c$	S_5'
$L(G_4)L(G_5)^*$	$S_4 \to cS_5', S_5' \to \lambda, S_5' \to S_5, S_5 \to cS_5,$ $S_5 \to c$	S_4
$L(G_4)L(G_5)^*L(G_6)L(G_7)$	$S_4 \to cS_5', S_5' \to S_6, S_5' \to S_5, S_5 \to cS_5,$ $S_5 \to cS_6, S_6 \to aS_7, S_7 \to a$	S_4
$(L(G_4)L(G_5)^*L(G_6)L(G_7))^*$	$S_4' \to \lambda, S_4' \to S_4, S_4 \to cS_5', S_5' \to S_6,$ $S_5' \to S_5, S_5 \to cS_5, S_5 \to cS_6, S_6 \to aS_7,$ $S_7 \to a$	S_4'
$T(\mathcal{A})$	$S_1' \to S_2, S_1' \to S_1, S_1 \to cS_1, S_1 \to cS_2,$ $S_2 \to cS_3, S_3 \to cS_4', S_4' \to \lambda, S_4' \to S_4,$ $S_4 \to cS_5', S_5' \to S_6, S_5' \to S_5, S_5 \to cS_5,$ $S_5 \to cS_6, S_6 \to aS_7, S_7 \to a$	S_1'

2.4 Entscheidbarkeitsprobleme bei formalen Sprachen

Formale Sprachen sind für uns ein Modell, das als theoretische Grundlage der Untersuchung von Programmiersprachen, der Syntaxanalyse und der Compilerkonstruktion benutzt werden kann. In diesem Zusammenhang ist das folgende natürliche Entscheidungsprobleme von besonderem Interesse.

Das *Mitgliedsproblem* ist die Frage, ob eine gegebene Grammatik ein gegebenes Wort erzeugt. Hierbei ist aber wichtig, wie die Sprache gegeben ist. Entsprechend den vorhergehenden Abschnitten kann dies sowohl durch eine Grammatik als auch durch einen akzeptierenden Automaten (und im Fall einer regulären Sprache auch durch einen regulären Ausdruck) geschehen. Daraus resultieren mindestens die zwei folgenden Varianten des Mitgliedsproblems für kontextfreie Sprachen:

Gegeben: Grammatik $G = (N, T, P, S)$ und Wort $w \in T^*$
Frage : Ist w in $L(G)$ enthalten ?

oder

Gegeben: Kellerautomat $\mathcal{M} = (X, Z, \Gamma, z_0, F, \delta)$ und Wort $w \in X^*$
Frage: Ist W in $T(\mathcal{M})$ enthalten ?

Wir haben das Problem nur für kontextfreie Grammatiken bzw. Kellerautomaten angeben. Natürlich kann die gleiche Frage auch für andere Typen von Grammatiken gestellt werden, für beliebige Regelgrammatiken (bzw. Turing-Maschinen) oder kontextsensitive Grammatiken oder monotone Grammatiken (bzw. linear beschränkte Automaten) oder reguläre Grammatiken.
Im Folgenden interessieren wir uns zuerst dafür, ob das Problem entscheidbar ist oder nicht, d.h. wir untersuchen, ob es einen Algorithmus gibt, der die Frage beantwortet. Die

Antwort ist dann unabhängig von der Formulierung des Problems, da sowohl der Übergang von einer kontextfreien Grammatik G zu einem Kellerautomaten $\mathcal{M}$ mit $L(G) = T(\mathcal{M})$ als auch der umgekehrte Übergang von einem Kellerautomaten zu einer kontextfreien Grammatik konstruktiv - also mittels eines Algorithmus - erfolgen. Folglich haben beide Formulierungen stets die gleiche Antwort.
Eine analoge Situation ist auch hinsichtlich der anderen Typen von Grammatiken und zugehörigen Automaten gegeben.
Im Fall der Existenz eines Algorithmus zur Beantwortung des Problems ist natürlich auch die Komplexität des Algorithmus von großem Interesse. Hier ist eine Abhängigkeit vom Problem gegeben, da schon die Größe der Eingabe Grammatik bzw. Automat (Maschine) unterschiedlich sind. Wir geben hier stets nur die Komplexität des Algorithmus bezogen auf die Größe der Grammatik an. Ist man an der Komplexität bezogen auf die (hier noch nicht definierte) Größe des Automaten interessiert, so lässt sich diese meist leicht dadurch ermitteln, dass man den Aufwand für den Übergang vom Automaten zur Grammatik noch hinzufügt. Letzterer Aufwand kann aus den Konstruktionen in Abschnitt 2.2 relativ einfach ermittelt werden.
Wir bestimmen nun den Entscheidbarkeitsstatus und die Komplexität des Mitgliedsproblems für die Grammatiken der CHOMSKY-Hierarchie.

Satz 2.69 *Das Mitgliedsproblem ist für (beliebige) Regelgrammatiken unentscheidbar.*

Beweis. Aus den Sätzen 2.33 und 2.43 ergibt sich sofort, dass $w \in L(G)$ genau dann gilt, wenn die zugehörige TURING-Maschine auf w stoppt. Die Entscheidbarkeit des Mitgliedsproblems würde daher die Entscheidbarkeit der Frage, ob eine TURING-Maschine auf einem Wort stoppt, zur Folge haben. Das widerspricht aber Satz 1.28. □

Satz 2.70 *Das Mitgliedsproblem ist für monotone (oder kontextsensitive) Grammatiken entscheidbar.*

Beweis. Es seien die monotone Grammatik $G = (N, T, P, S)$ und das Wort $w \in T^*$ gegeben.
Entsprechend der Definition von monotonen Grammatiken kann $\lambda \in L(G)$ nur gelten, wenn P die Regel $S \to \lambda$ enthält. Daher ist das Mitgliedsproblem für $w = \lambda$ entscheidbar, und wir können von nun ab voraussetzen, dass $w \in T^+$ gilt.
Es sei

$$S = w_0 \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow ... \Longrightarrow w_n = w$$

eine Ableitung von w in G. Falls $w_i = w_j$ für $i < j$ gilt, so ist auch

$$S = w_0 \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow ... \Longrightarrow w_i \Longrightarrow w_{j+1} \Longrightarrow w_{j+2} \Longrightarrow ... \Longrightarrow w_n = w$$

eine Ableitung von w in G. Daher können wir ohne Beschränkung der Allgemeinheit annehmen, dass bei $w \in L(G)$ eine Ableitung von w in G existiert, in der keine Satzform mehrfach auftritt. Da bei monotonen Grammatiken $|w_{i-1}| > |w_i|$ ausgeschlossen ist und nur $\#(V)^k$ Wörter der Länge k über $V = N \cup T$ existieren, tritt innerhalb einer Ableitung von w stets nach höchstens $\#(V)^{|w|}$ Schritten eine Verlängerung der Satzform ein. Daher muss es, falls $w \in L(G)$ gilt, eine Ableitung von w in G geben, die höchstens $|w|\#(V)^{|w|+1}$

Schritte hat. Da es höchstens $\#(P)^{|w|\#(V)^{|w|+1}}$ Ableitungen dieser Länge gibt, besteht die Möglichkeit diese durchzutesten und damit festzustellen, ob $w \in L(G)$ gilt. □

Der eben beschriebene Algorithmus zur Lösung des Mitgliedsproblems für monotone (kontextsensitive) Grammatiken hat exponentielle Komplexität bez. der Länge von w, da $\#(P)^{|w|\#(V)^{|w|+1}}$ mögliche Ableitungen zu testen sind.
Aus Satz 2.70 folgt sofort, dass die monotonen Sprachen rekursiv sind. Damit ergibt sich unter Beachtung von Satz 2.37 die folgende Aussage, die dann die verbliebene Lücke bei der Behandlung der CHOMSKY-Hierarchie in Abschnitt 2.1 schließt.

Satz 2.71 $\mathcal{L}(MON) \subset \mathcal{L}(RE)$ □

Aus Satz 2.70 folgt natürlich sofort, dass das Mitgliedsproblem für kontextfreie und reguläre Grammatiken ebenfalls entscheidbar ist. Wir sind aber in der Lage für diese Grammatiktypen die Komplexität näher zu bestimmen. Zur Formulierung der Aussage benötigen wir den Begriff der Größe $k(G)$ einer Grammatik $G = (N, T, P, S)$, der durch

$$k(G) = \sum_{\alpha \to \beta \in P} |\alpha| + |\beta| + 1$$

definiert ist (wir fassen Eine Regel als Wort auf und addieren die Längen aller Regeln).

Satz 2.72 *i) Das Mitgliedsproblem ist für kontextfreie Grammatiken $G = (N, T, P, S)$ in* CHOMSKY*-Normalform in der Zeit $O(\#(P) \cdot |w|^3\})$ entscheidbar.*
ii) Das Mitgliedsproblem ist für kontextfreie Grammatiken $G = (N, T, P, S)$ in der Zeit $O(k(G) \cdot \#(N) \cdot \#(P) \cdot |w|^3)$ entscheidbar.

Beweis. i) Es seien die kontextfreie Grammatik $G = (N, T, P, S)$ in CHOMSKY-Normalform und ein Wort $w = a_1 a_2 \dots a_n$ der Länge n gegeben. Wir konstruieren schrittweise die Mengen $V_{i,j}$ mit $0 \leq i < j \leq n$ wie folgt: Zuerst setzen wir

$$V_{i-1,i} = \{A \mid A \in N, A \to a_i \in P\}.$$

Sind dann für $i < k < j$ die Mengen $V_{i,k}$ und $V_{k,j}$ bereits definiert, so setzen wir

$$V_{i,j} = \{A \mid A \in N, A \to BC \in P, B \in V_{i,k}, C \in V_{k,j} i < k < j\}.$$

Da es höchstens n Möglichkeiten für k gibt und für jedes k alle Regeln von P durchzumustern sind, kann jede Menge $V_{i,j}$ in $\#(P) \cdot n$ Schritten konstruiert werden. Da insgesamt $\frac{n(n+1)}{2}$ Mengen zu konstruieren sind, ergibt sich damit ein durch $\frac{\#(P)n^2(n+1)}{2}$ nach oben beschränkter Gesamtaufwand für die Konstruktion der Mengen.
Wir beweisen nun mittels Induktion über die Differenz $j - i$, dass

$$V_{i,j} = \{A \mid A \in N, A \Longrightarrow^* a_{i+1} a_{i+2} \dots a_j\} \tag{2.4}$$

ist.
Für $j - i = 1$ gilt dies nach Konstruktion.

Es sei nun $A \in V_{i,j}$. Dann gibt es nach Konstruktion Nichtterminale $B \in V_{i,k}$ und $C \in V_{k,j}$ mit $A \to BC \in P$. Nach Induktionsvoraussetzung gelten dann

$$B \Longrightarrow^* a_{i+1}a_{i+2}\dots a_k \quad \text{und} \quad C \Longrightarrow^* a_{k+1}a_{k+2}\dots a_j.$$

Folglich ergibt sich

$$A \Longrightarrow BC \Longrightarrow^* a_{i+1}a_{i+2}\dots a_kC \Longrightarrow^* a_{i+1}a_{i+2}\dots a_ka_{k+1}a_{k+2}\dots a_j.$$

Gilt umgekehrt $A \Longrightarrow^* a_{i+1}a_{i+2}\dots a_j$, so muss es wegen der CHOMSKY-Normalform Nichtterminale B und C und ein k mit $i < k < j$ und

$$A \to BC \in P, \quad B \Longrightarrow^* a_{i+1}a_{i+2}\dots a_k, \quad C \Longrightarrow^* a_{k+1}a_{k+2}\dots a_j$$

geben. Nach Induktionsvoraussetzung haben wir $B \in V_{i,k}$ und $C \in V_{k,j}$, woraus wir nach Konstruktion von $V_{i,j}$ dann $A \in V_{i,j}$ erhalten.
Somit ist (2.4) bewiesen.
Aus (2.4) ergibt sich aber genau dann $S \Longrightarrow^* a_1a_2\dots a_n = w$, wenn $S \in V_{0,n}$ gilt. Damit sind $w \in L(G)$ und $S \in V_{0,n}$ gleichwertig. Um $w \in L(G)$ zu entscheiden, reicht es also die Mengen $V_{i,j}$ mit $0 \le i < j \le n$ zu konstruieren und $S \in V_{0,n}$ zu überprüfen. Nach obigem ist daher die Entscheidung des Mitgliedproblems für G und w in $\theta(\#(P) \cdot |w|^3)$ Schritten möglich.

ii) folgt aus i) sofort, wenn wir beachten dass bei der Umwandlung einer beliebigen kontextfreien Grammatik $G = (N, T, P, S)$ in eine Grammatik $G' = (N', T, P', S')$ in CHOMSKY-Normalform entsprechend den Konstruktionen aus Abschnitt 2.1.2 die Beziehung $\#(P') = O(k(G) \cdot \#(N) \cdot \#(P))$ gilt. □

Beispiel 2.73 Wir illustrieren den eben beschriebenen Algorithmus, den sogenannten Cocke-Younger-Kasami-Algorithmus, anhand der Grammatik

$$G = (\{S, T, U\}, \{a, b\}, P, S)$$

mit den Regeln

$$S \to ST, T \to TU, T \to TT, U \to TS, S \to a, T \to a, U \to b$$

in P. Wir wollen zuerst untersuchen, ob das Wort $w = aabaa$ in $L(G)$ liegt. Wir müssen also die zugehörigen Mengen $V_{i,j}$ mit $0 \le i < j \le 5$ konstruieren. Es ergeben sich

$$\begin{aligned}
V_{0,1} &= \{A \mid A \to a \in P\} = \{S, T\},\\
V_{1,2} &= \{A \mid A \to a \in P\} = \{S, T\},\\
V_{2,3} &= \{A \mid A \to b \in P\} = \{U\},\\
V_{0,2} &= \{A \mid A \to BC \in P, B \in V_{0,1}, C \in V_{1,2}\} = \{S, T, U\},\\
V_{1,3} &= \{A \mid A \to BC \in P, B \in V_{1,2}, C \in V_{2,3}\} = \{T\},\\
V_{0,3} &= \{A \mid A \to BC \in P, B \in V_{0,1}, C \in V_{1,3}\}\\
&\quad \cup \{A' \mid A' \to B'C' \in P, B' \in V_{0,2}, C' \in V_{2,3}\}\\
&= \{S, T\} \cup \{T\} = \{S, T\}.
\end{aligned}$$

Die weiteren Mengen können der nachfolgenden Tabelle entnommen werden, wobei das i-te Symbol des Wortes w im Schnittpunkt der Zeile i und Spalte i und die Menge $V_{i,j}$ im Schnittpunkt der Zeile i und Spalte j eingetragen und die Mengenklammern fortgelassen wurden.

	0	1	2	3	4	5
0		S,T	S,T,U	S,T	S,T,U	S,T,U
1		a	S,T	T	T,U	T,U
2			a	U	$\emptyset$	$\emptyset$
3				b	S,T	S,T,U
4					a	S,T
5						a

Wegen $S \in V_{0,5}$ folgt $w = aabaa \in L(G)$.
Für $v = abaaa$ ergibt sich die Tabelle

	0	1	2	3	4	5
0		S,T	T	T,U	T,U	T,U
1		a	U	$\emptyset$	$\emptyset$	$\emptyset$
2			b	S,T	S,T,U	S,T,U
3				a	S,T	S,T,U
4					a	S,T
5						a

und damit $v \notin L(G)$ wegen $S \notin V_{0,5}$.

Eine genaue Analyse des Cocke-Younger-Kasami-Algorithmus ergibt, dass die Bestimmung der Mengen $V_{i,j}$ eine Analogie zur Matrizenmultiplikation aufweist. Hierdurch ist bei fester Grammatik G (und damit festem P) eine Verbesserung möglich, da Algorithmen für die Matrizenmultiplikation bekannt sind, die $O(n^\alpha)$ mit $\alpha < 3$ erfordern. So erfordert z.B. die Multiplikation von Matrizen nach STRASSEN nur $O(n^{\log_2(7)})$.

Für reguläre Sprachen läßt sich die folgende Verschärfung von Satz 2.72 angeben.

Satz 2.74 *Für eine reguläre Grammatik $G = (N, T, P, S)$ und ein Wort w ist in der Zeit $O(k(G) \cdot \#(N) \cdot |w|)$ entscheidbar, ob $w \in L(G)$ gilt.*

Beweis. Zuerst konstruieren wir entsprechend Satz 2.24 in der Zeit $O(\#(N)k(G))$ die reguläre Grammatik $G' = (N', T, P', S')$ zu G, die nur Regeln der Form $A \to aB$ oder $A \to a$ mit $A, B \in N', a \in T$ besitzt (vielleicht mit Ausnahme der Regel $S' \to \lambda$) und $L(G') = L(G)$ erfüllt. Für G' gelten außerdem $\#(N') = \theta(k(G))$ und $\#(P') \leq 4 \cdot k(G') = O(\#(N)k(G))$ nach dem Beweis von Satz 2.24.
Es sei $w = a_1 a_2 \dots a_n$. Dann setzen wir $M_0 = \{S\}$ und

$$M_i = \{A \mid B \to a_i A \text{ für ein } B \in M_{i-1}\}$$

für $1 \leq i \leq n-1$. Die Bestimmung von M_i, $1 \leq i \leq n$, aus M_{i-1} kann in der Zeit $O(\#(P'))$ erfolgen, da einmal die Regeln aus P' durchzumustern sind. Aus der Konstruktion der Mengen folgt sofort, dass $A \in M_i$ genau dann gilt, wenn es die Ableitung

$S \Longrightarrow^* a_1 a_2 \dots a_i A$ gibt. Nun überprüfen wir, ob es ein Nichtterminal A in M_{n-1} gibt, für das eine Regel $A \to a_n$ in P vorhanden ist. Gibt es ein solches Nichtterminal, so existiert die Ableitung

$$S \Longrightarrow^* a_1 a_2 \dots a_{n-1} A \Longrightarrow a_1 a_2 \dots a_{n-1} a_n = w,$$

womit $w \in L(G') = L(G)$ gilt. Ist dagegen kein solches Nichtterminal vorhanden, so kann es keine nach Erzeugung von a_n terminierende Ableitung geben, woraus $w \notin L(G') = L(G)$ folgt. Da die Existenz eines solchen Nichtterminals erneut in der Zeit $O(\#(P'))$ getestet werden kann, erhalten wir als gesamten Zeitbedarf

$$O(\#(P')|w|) = O(k(G) \cdot \#(N) \cdot |w|).$$

□

Übungsaufgaben

1. Bestimmen Sie die von der Grammatik

   $$G = (\{S, X_1, X_2, X_3\}, \{a, b, c\}, P, S)$$

   mit

   a) $P = \{S \to X_1 S X_2, S \to X_3, X_1 \to a X_1 b, X_1 \to \lambda, X_2 \to b X_2 a, X_2 \to \lambda, X_3 \to c\}$

   b) $P = \{S \to a X_1 X_2, a X_1 \to a a X_1 b, X_1 b \to b X_1 X_3, X_3 b \to b X_3, X_3 X_2 \to X_2 c,$
   $X_1 X_2 \to bc\}$

   c) $P = \{S \to a S X_1, S \to a X_2, X_2 X_1 \to b X_2 c, c X_1 \to X_1 c, X_2 \to bc\}$

   erzeugte Sprache.

2. Geben sei die Grammatik

   $$G = (\{S\}, \{a, b\}, \{S \to SS, S \to aaSb, S \to bSaa, S \to \lambda\}, S).$$

   Gilt

   $$L(G) = \{w : w \in T^*, |w|_a = 2 \cdot |w|_b\} \ ?$$

3. Geben Sie für die folgenden Sprachen kontextfreie Grammatiken an:

   a) $\{a^n b^n c^m : n \geq 1, m \geq 3\}$,

   b) $\{a_1^{n_1} a_2^{n_2} \dots a_k^{n_k} b_k^{n_k} b_{k-1}^{n_{k-1}} \dots b_2^{n_2} b_1^{n_1} : n_i \geq 1 \text{ für } 1 \leq i \leq k\}$.

4. Geben Sie eine reguläre Grammatik an, die die Menge aller Wörter $w \in \{a, b, c\}^*$, die genau drei Vorkommen von a und höchstens zwei Vorkommen von c haben, erzeugt.

5. Beweisen Sie, dass jede endliche Sprache regulär ist.

6. Es sei
$$L = \{x_1 x_2 \dots x_n x_n x_{n-1} \dots x_1 : n \geq 1, x_i \in T, 1 \leq i \leq n\}.$$
Beweisen Sie, dass
i) L eine kontextfreie Sprache ist,
ii) L keine reguläre Sprache ist.

7. Es sei
$$L = \{a^{2^n} : n \geq 1\}.$$
Beweisen Sie, dass
i) L eine monotone Sprache ist,
ii) L keine kontextfreie Sprache ist.

8. Beweisen Sie, dass
$$\{ww : w \in T^+\} \in \mathcal{L}(CS)$$
und
$$\{ww : w \in T^+\} \notin \mathcal{L}(CF)$$
gelten.

9. Geben Sie für die Grammatik $G = (N, T, P, S)$ mit
$$\begin{aligned} N &= \{S, A, B\}, \\ T &= \{a, b, c\}, \\ P &= \{S \to cSc, S \to AB, A \to aAb, B \to cBb, A \to ab, B \to \lambda\} \end{aligned}$$
eine Grammatik G' in Chomsky-Normalform mit $L(G') = L(G)$.

10. Eine Grammatik $G = (N, T, P, S)$ heißt *linear*, falls P nur Regeln der Form
$$A \longrightarrow w_1 B w_2 \text{ und } A \longrightarrow w \quad \text{mit} \quad A, B \in N \text{ und } w_1, w_2, w \in T^*$$
enthält.
a) Beweisen Sie, daß es eine lineare Sprache gibt, die nicht regulär ist.
b) Beweisen Sie, daß es eine kontextfreie Sprache gibt, die nicht linear ist.

11. Für eine Sprache L über dem Alphabet V mit $a \in V$ sei
$$L_a = \{w : aw \in L\} \quad \text{und} \quad L^a = \{v : va \in L\}.$$
Zeigen Sie, dass für reguläres L auch L_a und L^a regulär sind.

12. Für eine Sprache L sei L_{ger} die Menge der in L enthaltenen Wörter gerader Länge. Beweisen Sie, dass für reguläres L auch L_{ger} regulär ist.

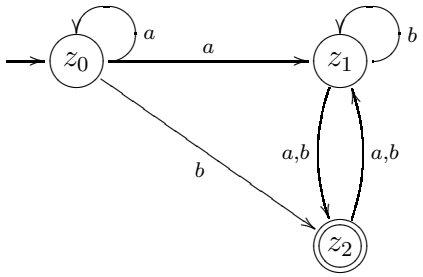
13. Gegeben sei der endliche Automat
$$\mathcal{A} = (\{a, b\}, \{z_0, z_1, z_2\}, z_0, \{z_2\}, \delta)$$

mit

$$\delta(z_0, a) = \delta(z_2, b) = z_0,$$
$$\delta(z_0, b) = \delta(z_1, b) = z_1,$$
$$\delta(z_1, a) = \delta(z_2, a) = z_2.$$

a) Beschreiben Sie $\mathcal{A}$ durch einen Graphen.
b) Welche der Wörter *abaa*, *bbbabb*, *bababa* und *bbbaa* werden von $\mathcal{A}$ akzeptiert?
c) Bestimmen Sie die von $\mathcal{A}$ akzeptierte Sprache.

14. Gegeben sei der Graph



der den Automaten $\mathcal{A}$ beschreibt.
a) Geben Sie alle möglichen Überführungen bei Eingabe von *aaabb* an.
b) Wird *aaabb* von $\mathcal{A}$ akzeptiert.
c) Bestimmen Sie die von $\mathcal{A}$ akzeptierte Sprache.
d) Geben Sie einen deterministischen endlichen Automaten $\mathcal{B}$ mit $T(\mathcal{A}) = T(\mathcal{B})$ an.

15. Konstruieren Sie einen (nichtdeterministischen) endlichen Automaten, der die Sprache aller Wörter über $\{1, 2, 3\}$ akzeptiert, bei denen die Quersumme durch 6 teilbar ist.

16. Konstruieren Sie einen (nichtdeterministischen) endlichen Automaten, der die Sprache aller Wörter über $\{a, b, c\}$ akzeptiert, bei denen jedes Teilwort der Länge 3 mindestens ein a enthält.

17. Gegeben sei ein endlicher Automat $\mathcal{A}$ mit n Zuständen. Beweisen Sie, dass
a) $T(\mathcal{A})$ genau dann nicht leer ist, wenn $T(\mathcal{A})$ ein Wort der Länge m mit $m < n$ enthält,
b) $T(\mathcal{A})$ genau dann unendlich ist, wenn $T(\mathcal{A})$ ein Wort der Länge m mit $n \leq m < 2n$ enthält.

18. Geben Sie für die regulären Sprachen aus den Aufgaben 4, 13, 14 und 16 eine Darstellung mittels Vereinigung, Produkt und KLEENE-Abschluss.

19. Gegeben sei der Kellerautomat

$$\mathcal{M} = (\{z_0, z_1, z_2\}, \{a, b\}, \{X\}, z_0, \{z_0\}, \delta)$$

mit

$$\begin{aligned}
&\delta(z_0, b, \#) = \{(z_0, X)\}, \quad \delta(z_0, b, X) = \{(z_0, XX)\},\\
&\delta(z_0, a, X) = \{(z_1, \lambda)\}, \quad \delta(z_1, a, X) = \{(z_1, \lambda)\},\\
&\delta(z_1, a, \#) = \{(z_0, \lambda)\}
\end{aligned}$$

und $\delta(z, x, \gamma) = \{(z_2, \lambda)\}$ in allen anderen Fällen. a) Untersuchen Sie, ob *baabaa*, *babaaaa* und *baaabaa* von M akzeptiert werden.
b) Bestimmen Sie die von $\mathcal{M}$ akzeptierte Wortmenge.

20. Bestimmen Sie für die nachfolgend genannten Sprachen jeweils einen Kellerautomaten, der sie akzeptiert.
a) $\{wdw^R : w \in \{a, b\}^*\}$,
b) die Menge aller Palindrome über $\{a, b\}$,
c) die Menge aller Wörter über $\{a, b\}$, bei denen die Anzahl der Vorkommen von a und b gleich sind,
d) die Menge aller Wörter über $\{(,)\}$, die einer korrekten Klammerung entsprechen.

21. Konstruieren Sie zu der Grammatik

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

mit

$$P = \{S \to aABA, S \to aBB, A \to bA, A \to b, B \to cB, B \to c\}$$

einen Kellerautomaten $\mathcal{M}$ mit $T(\mathcal{M}) = L(G)$.

22. Bestimmen Sie zu regulären Grammatiken G, G_1 und G_2 reguläre Grammatiken H und H' mit $L(H) = \overline{L(G)}$ und $L(H') = L(G_1) \cap L(G_2)$.

23. Beweisen Sie, dass zu kontextfreien Grammatiken G_1 und G_2 kontextfreie Grammatiken G und G' mit $L(G) = L(G_1) \cup L(G_2)$, $L(G') = L(G_1) \cdot L(G_2)$ und $k(G) = \theta(max\{k(G_1), k(G_2)\})$ und $k(G') = \theta(max\{k(G_1), k(G_2)\})$ gibt.

24. Gegeben seien die Grammatik $G = (\{S, A, B, C\}, \{a, b\}, P, S)$ mit

$$P = \{S \to AB, S \to BC, A \to BA, A \to a, B \to CC, B \to b, C \to AB, C \to a\}$$

und die Wörter $w_1 = abbba$, $w_2 = baaba$ und $w_3 = bbbaaa$. Stellen Sie mittels des Cocke-Younger-Kasami-Algorithmus fest, welche der Wörter w_1, w_2, w_3 in $L(G)$ liegen.

Kapitel 3

Elemente der Komplexitätstheorie

3.1 Definitionen und ein Beispiel

Im ersten Abschnitt haben wir gesehen, dass es Probleme gibt, die durch keinen Algorithmus gelöst werden können, zu deren Lösung es also kein für alle Eingaben korrekt arbeitendes Programm gibt. Neben dieser generellen Schranke für Algorithmen gibt es aber auch der Praxis entstammende Grenzen. Stellen wir uns vor, dass durch ein Programm entschieden wird, ob ein Element zu einer Menge gehört[1] und dass bei einem Element der Größe[2] n für die Entscheidung vom Computer $f(n)$ Operationen ausgeführt werden müssen. In der folgenden Tabelle sind einige Zeiten zusammengestellt, die sich bei verschiedenen Funktionen f ergeben, wobei wir annehmen, dass der Computer eine Million Operationen in der Sekunde ausführt. (Wenn wir eine andere Geschwindigkeit des Computers annehmen, z.B. 10^9 Operationen je Sekunde, so ändern sich die Tabellenwerte nur um einen konstanten Faktor. Wir merken aber an, dass aus physikalischen Gründen eine Schranke für die Geschwindigkeit existiert.)

n / f	5	10	50	100
n^2	0,000025 s	0,0001 s	0,0025 s	0,01 s
n^5	0,003125 s	0,1 s	312,5 s	ca. 3 Std.
2^n	0,000032 s	0,001024 s	ca. 36 Jahre	ca. 10^{17} Jahre
n^n	0,003125 s	ca. 3Std.	$> 10^{71}$ Jahre	

Abb. 2.1.

Man sieht an den Werten deutlich, dass bei den Funktionen $f(n) = 2^n$ und $f(n) = n^n$ der Zeitaufwand bei praktisch relevanten Fällen mit Eingaben einer Größe ≥ 50 so groß ist, dass sie nicht in erträglicher Zeit zu einem Ergebnis führen. Hieraus resultiert, dass nur solche Algorithmen von Interesse sind, die nicht zu zeitaufwendig sind. Gleiches gilt für die zur Lösung notwendige Speicherkapazität. Wir formalisieren nun diesen Ansatz zur Messung der Effizienz eines Algorithmus.

[1] Wir erinnern daran, dass jedes entscheidbare Problem auf eine solche Frage zurückgeführt werden kann.

[2] Wir gehen hier nicht näher auf den Begriff der Größe ein. Dies kann z. B. bei Wörtern die Länge, bei Matrizen die Anzahl der Zeilen, bei Polynomen der Grad sein.

Dabei verwenden wir TURING-Maschinen als Beschreibung der Algorithmen. Um eine einheitliche Definition zu erreichen, werden wir TURING-Maschinen mit k-Arbeitsbändern verwenden. Wir beginnen daher mit der Definition solcher Maschinen.

Definition 3.1 *Eine akzeptierende k-Band-*TURING*-Maschine ist ein 6-Tupel*

$$M = (k, X, Z, z_0, Q, F, \delta),$$

wobei $k \geq 1$ eine natürliche Zahl ist, X, Z, z_0, Q und F wie bei einer TURING*-Maschine definiert sind, δ eine Funktion*

$$(Z \setminus Q) \times (X \cup \{*\})^{k+1} \longrightarrow Z \times (X \cup \{*\})^k \times \{R, L, N\}^{k+1}$$

ist und $* \notin X$ gilt.

Die k-Band-TURING-Maschine verfügt über ein Eingabeband, auf dem nur gelesen werden darf, und k Arbeitsbänder mit jeweils einem Lese-Schreibkopf. Wir interpretieren X, Z, z_0, Q und die Elemente aus $\{R, L, N\}$ wie bei einer TURING-Maschine. Falls

$$\delta(z, x_e, x_1, x_2, \ldots, x_k) = (z', y_1, y_2, \ldots, y_k, r_e, r_1, r_2, \ldots, r_k)$$

gilt, so interpretieren wir dies wie folgt: Die Maschine liest im Zustand z auf dem Eingabeband den Buchstaben x_e, auf dem i-ten Arbeitsband den Buchstaben x_i, $1 \leq i \leq k$, geht in den Zustand z' über, schreibt den Buchstaben y_i auf das i-te Arbeitsband, $1 \leq i \leq k$, der Lesekopf des Eingabebandes bewegt sich nach $r_e \in \{R, N, L\}$ und der Kopf des i-ten Arbeitsbandes nach $r_i \in \{R, L, N\}$, $1 \leq i \leq k$.

Definition 3.2 *Sei M eine k-Band-*TURING*-Maschine wie in Definition 3.1.*
Eine Konfiguration von M ist ein $2k + 5$-Tupel

$$(z, w_e, w'_e, w_1, w'_1, w_2, w'_2, \ldots, w_k, w'_k, w_a, w'_a), \tag{3.1}$$

wobei $z \in Z$, $w_e, w'_e \in (X \cup \{*\})^*$ und $w_i, w'_i \in (X \cup \{*\})^*$ für $1 \leq i \leq k$ gelten.
Eine Konfiguration heißt Anfangskonfiguration, falls $z = z_0$, $w_e = w_1 = w_2 = \ldots = w_k = \lambda$ und $w'_a = w'_1 = w'_2 = \ldots = w'_k = *$ gelten.
Eine Konfiguration heißt Endkonfiguration, falls z in Q liegt.

Wir interpretieren eine Konfiguration (3.1) wie folgt: Die Maschine befindet sich im Zustand z, auf dem Eingabeband steht $w_e w'_e$ und der Lesekopf steht über dem ersten Buchstaben von w'_e, für $1 \leq i \leq k$ steht auf dem i-ten Arbeitsband $w_i w'_i$ und steht der Kopf über dem ersten Buchstaben von w'_i.
Bei einer Anfangskonfiguration ist die Maschine im Zustand z_0, auf dem Eingabeband steht w'_e, der Lesekopf befindet sich über dem ersten Buchstaben von w'_e, alle Arbeitsbänder sind leer, aber durch ein angegebenes $*$ wird die Position des Kopfes des Bandes angegeben.
Wir überlassen dem Leser eine formale Definition der Änderung der Konfiguration K_1 in die Konfiguration K_2, die sich aus dem bisher gesagtem in Analogie zu Definition 1.17 ergibt und die wir wieder mit $K_1 \vdash K_2$ bezeichnen.

Definition 3.3 *Sei M eine k-Band-*TURING*-Maschine wie in Definition 3.1. Die von M akzeptierte Sprache besteht aus allen Wörtern $w \in X^*$, die die Anfangskonfiguration*

$$K = (z_0, \lambda, w, \lambda, *, \lambda, *, \ldots, \lambda, *)$$

eine Endkonfiguration

$$K' = (q, w_e, w'_e, w_1, w'_1 w_2, w'_2, \ldots, w_k, w'_k) \text{ mit } q \in F$$

überführen.

Wir definieren nun die Grundbegriffe der Komplexitätstheorie.

Definition 3.4 *Sei $M = (k, X, Z, z_0, Q, \delta, F)$ eine deterministische akzeptierende k-Band-*TURING*-Maschine, die bei jeder Eingabe einen Stoppzustand erreicht. Ferner sei $r = \#(X)$.*
i) Mit $t_M(w)$, $w \in X^*$, bezeichnen wir die Anzahl der (direkten) Überführungsschritte, die M ausführt, um die Anfangskonfiguration $(z_0, \lambda, w, \lambda, *, \lambda, *, \ldots, \lambda, *)$ in die zugehörige Endkonfiguration zu transformieren, und nennen $t_M(w)$ die Zeitkomplexität von w bezüglich M.
ii) Für eine natürliche Zahl n setzen wir

$$t_M(n) = max\{t_M(w) : |w| = n\}$$

und

$$\overline{t_M}(n) = \frac{\sum_{|w|=n} t_M(w)}{r^n}.$$

Die Funktionen t_M und $\overline{t_M}$ von $\mathbf{N}$ in $\mathbf{N}$ heißen Zeitkomplexität des ungünstigsten Falles (worst-case time complexity) und durchschnittliche Zeitkomplexität (average time complexity) von M.

Bei $t_M(n)$ wird die Zeitkomplexität $t_M(w)$ des Wortes w der Länge n genommen, für das M am meisten Schritte benötigt, d.h. die Komplexität des ungünstigsten Wortes wird benutzt. Bei der durchschnittlichen Zeitkomplexität wird zuerst die Summe der Zeitkomplexitäten aller Wörter der Länge n gebildet und dann - wie bei Durchschnittsbildungen üblich - durch die Anzahl r^n aller Wörter der Länge n dividiert.
Wir betrachten die Funktionen t_M und $\overline{t_M}$ als Maße für die Effizienz des durch M realisierten Algorithmus.

Neben dem Zeitaufwand zur Lösung eines Problems ist auch noch der Speicherbedarf eine wesentliche Kenngröße.

Definition 3.5 *Seien M und r wie in Definition 3.4 gegeben.*
i) Mit $s_M(w)$, $w \in X^*$, bezeichnen wir die Anzahl der Zellen auf den Arbeitsbändern, über denen während der Überführung der Anfangskonfiguration $(z_0, \lambda, w, \lambda, *, \lambda, *, \ldots, \lambda, *)$ in die zugehörige Endkonfiguration mindestens einmal der Lese-/Schreibkopf stand. $s_M(w)$ heißt die Raumkomplexität von w auf M.

ii) Für $n \in \mathbf{N}$ setzen wir

$$s_M(n) = max\{s_M(w) : |w| = n\}$$

und

$$\overline{s_M}(n) = \frac{\sum_{|w|=n} s_M(w)}{r^n}.$$

s_M und $\overline{s_M}$ heißen Raumkomplexität des ungünstigsten Falles bzw. durchschnittliche Raumkomplexität von M.

Wir illustrieren die Begriffe nun an einem Beispiel.

Beispiel 3.6 Wir betrachten die Sprache

$$L = \{a^n b^n : n \geq 1\}$$

und die 1-Band-TURING-Maschine M, die wie folgt arbeitet:

- Ist M im Anfangszustand z_0 und liest ein a, so geht sie in den Zustand z_a und schreibt ein a auf das Arbeitsband.
- Ist M im Zustand z_a, so bleibt sie in z_a, solange sie ein a liest und schreibt jedes Mal beim Lesen eines a auch ein a zusätzlich auf das Arbeitsband. Beim Lesen des ersten b in z_a geht M in z_b und löscht ein a auf dem Arbeitsband.
- Den Zustand z_b verändert M nicht, solange ein b gelesen wird, und bei jedem Lesen eines b wird ein a gelöscht. Wird dann ein $*$ gelesen und ist das Arbeitsband leer, so geht M in den akzeptierenden Stopzustand z_{akz}.
- In allen Fällen wechselt M in den ablehnenden Stopzustand z_{abl}.

Hieraus ergeben sich folgende Aussagen zur Komplexität von $w \in \{a, b\}^*$:

w	$t_M(w)$	$s_M(w)$
$a^r b^s$, $r \geq s \geq 1$	$r + s + 1$	r
$a^r b^s$, $s \geq r \geq 1$	$2r + 1$	r
bw', $w' \in \{a, b\}^*$	1	0
$a^r b^s a w''$, $r \geq 1, s \geq 1, w'' \in \{a, b\}^*$	$\min\{r + s + 1, 2r + 1\}$	r

Daher gelten stets $|w|+1 \geq \min\{r+s+1, 2r+1\}$ und $|w| \geq r$. Ferner gelten $t_M(a^n) = n+1$ und $s_M(a^n) = n$. Somit erhalten wir

$$t_M(n) = n + 1 \qquad \text{und} \qquad s_M(n) = n$$

als Zeit- bzw. Raumkomplexität des schlechtesten Falles. In beiden Komplexitätsmaßen erhalten wir lineare Funktionen als Komplexitäten des schlechtesten Falles.
Wir betrachten nun die durchschnittliche Raumkomplexität. Dazu bemerken wir zunächst, dass jedes Wort der Länge n entweder a^n oder von der Form $a^r b w''$ mit $r \geq 0$ und $w'' \in \{a, b\}^*$ ist. Dann gelten

$$s_M(a^n) = n \quad \text{und} \qquad s_M(a^r b w'') = \begin{cases} r & r \geq 1 \\ 0 & r = 0 \end{cases}.$$

Ferner gibt es genau 2^{n-r-1} verschiedene Wörter w'' der Länge $n-r-1$, womit sich

$$\overline{s_M}(n) = \frac{n + \sum_{r=1}^{n-1} r 2^{n-r-1}}{2^n} = \frac{n + (2^n - n - 1)}{2^n} = 1 - \frac{1}{2^n}$$

ergibt. Die durchschnittliche Raumkomplexität von M ist also durch eine Konstante beschränkt.
Ohne Beweis merken wir an, dass dies auch für die durchschnittliche Zeitkomplexität von M gilt.

Wir können zur Entscheidung von L aber auch die 1-Band-TURING-Maschine M' benutzen, die sich von M nur dadurch unterscheidet, dass sie zuerst 0 auf das Arbeitsband schreibt und dann bei Lesen von a in z_a bzw. z_0 die Zahl auf dem Arbeitsband um Eins erhöht und bei Lesen von b in z_b um Eins erniedrigt wird.
Die Komplexitäten ändern sich dann wie folgt: Die Länge des Wortes auf dem Eingabeband ist bei binärer Zahlendarstellung dann durch $\log_2(n)$ beschränkt (und bei Verwendung einer anderen Basis zur Darstellung wird dieser Wert nur um einen konstanten Faktor verkleinert). Somit gilt unter Verwendung der Landau-Symbole

$$s_{M'}(n) = O(\log_2(n)).$$

Da die Addition von 1 unter Umständen mehrere Schritte erfordert, ist die Betrachtung der Zeit im schlechtesten Fall etwas komplizierter. Wenn wir bei der Addition wie in Beispiel 1.19 vorgehen, so ergibt sich für die Anzahl der Schritte bei der Addition von 2^k Einsen zu 0 die Rekursion

$$t_{M'}(2^k) = 2 \cdot t_{M'}(2^{k-1}) + 2\log_2(n),$$

woraus letztlich

$$t_{M'}(n) = O(n)$$

resultiert.
Während wir hinsichtlich der Raumkomplexität im schlechtesten Fall also bei M' gegenüber M eine deutliche größenordnungsmäßige Verbesserung konstatieren können, ist für die Zeitkomplexität im schlechtesten Fall in beiden Fällen Linearität vorhanden (jedoch ist der konstante Koeffizient bei n bei M kleiner).
Es erhebt sich nun die Frage, ob es eine noch bessere k-Band-TURING-Maschine zur Berechnung von M gibt. Wir wollen nun zeigen, dass dies hinsichtlich der Raumkomplexität im schlechtesten Fall nicht der Fall ist, genauer gesagt wir beweisen die folgende Aussage:
*Für jede k-Band-*TURING-*Maschine M'', die L entscheidet, gilt $s_{M''}(n) = O(\log_2(n))$.*
Wir bemerken zuerst, dass wir uns auf 1-Band-TURING-Maschinen beschränken können. Dies folgt daraus, dass wir statt k Bändern ein Band mit k Spuren betrachten können und jeweils der Reihe nach die Spuren in Analogie zu den Bändern ändern. Dies erfordert jeweils ein Suchen des (markierten) Symbols der Spur über dem der Kopf gerade steht und damit einen zusätzlichen Zeitaufwand, aber der Raumbedarf wird dadurch nicht größer. Vielmehr ist nun der Raumbedarf durch den maximalen Raum auf einem der Bänder gegeben, der aber (da die Zahl der Bänder für eine Maschine fest ist) nur um einen konstanten Faktor kleiner ist, als der Platzbedarf auf allen Bändern.

Wir nehmen erst einmal an, dass sich der Lesekopf des Eingabebandes stets über einer Zelle steht, in der sich ein Buchstabe des Eingabeworts befindet.
Wir bezeichnen mit $U(n)$ die Menge der möglichen Teilkonfigurationen, die aus dem Tripel (z, k, w) bestehen, wobei z den Zustand, w das Wort auf dem Arbeitsband und k die Position des Kopfes auf dem Arbeitsband (d.h. der Kopf steht über dem k-ten Buchstaben von w) angeben, und die bei Eingabe eines Wortes der Länge n erreicht werden können. Dann gibt es höchstens $s_{M''}(n)$ Positionen für den Kopf und höchstens $2^{s_{M''}(n)}$ verschiedene Wörter auf dem Band bei einer Eingabe der Länge n. Damit gilt

$$\#(U(n)) \leq \#(Z) \cdot s_{M''}(n) \cdot 2^{s_{M''}(n)}.$$

Durch Logarithmieren gewinnen wir

$$\log_2(\#(U(n))) \leq \log_2(\#(Z)) + \log_2(s_{M''}(n)) + s_{M''}(n).$$

Wir nehmen nun an, dass

$$s_{M''}(n) = o(\log_2(n))$$

gilt, womit aus der vorstehenden Ungleichung

$$\log_2(\#(U(n))) \leq s_{M''}(n) = o(\log_2(n)) < \log_2\left(\frac{n}{2}\right)$$

für hinreichend großes n folgt. Dies impliziert, dass $U(n)$ für hinreichend großes n weniger als $n/2$ Elemente enthält.
Wir betrachten die Arbeit von M'' auf $a^n b^n$ mit hinreichend großem n.
Falls M'' das Wort $a^n b^n$ bereits akzeptiert, ohne ein b zu lesen, so wird auch $a^n b^{n+1}$ akzeptiert. Dies ist aber ein Widerspruch zur Definition von L.
Daher muss M'' also mindestens ein b von der Eingabe $a^n b^n$ lesen und somit mindestens jedes a. Mit u_i, $1 \leq i \leq 2n$, bezeichnen wir das Element von $U(2n)$, das vorliegt, wenn das erste Mal der i-te Buchstabe von $a^n b^n$ gelesen wird. Da $U(2n)$ weniger als n Elemente enthält, muss es Zahlen i und j mit $1 \leq i < j \leq n$ derart geben, dass $u_i = u_j$ gilt.
Wir betrachten nun die Eingabe $a^{n+n!}b^n$. Sei v_s, $1 \leq s \leq n + n!$, das Element von $U(2n + n!)$, das beim erstmaligen Lesen des s-ten Buchstaben von $a^{n+n!}b^n$ vorliegt. Dann gilt $u_i = v_i$ und $u_j = v_j$, da in beiden Fällen ausgehend von der gleichen Ausgangssituation die gleichen Elemente auf dem Eingabeband gelesen werden. Ferner folgt aus $v_k = v_t$ auch $v_{k+1} = v_{t+1}$ für $k, t \leq n + n!$, da ausgehend von gleichen Konfiguration auf dem Arbeitsband nur a's gelesen werden. Damit gilt

$$u_i = v_i = u_j = v_j = v_{i+(j-i)} = v_{i+2(j-i)} = \ldots = v_{i+r(j-i)}$$

mit $r = \frac{n!}{j-i}$. Wegen $i + r(j - i) = i + n!$ erhalten wir $u_i = v_{i+n!}$. Daraus ergibt sich

$$u_i = v_{i+n!}, u_{i+1} = v_{i+1+n!}, \ldots, u_n = v_{n+n!}, \ldots u_{2n} = v_{2n+n!}\,.$$

Dies impliziert, dass M'' auch die Eingabe $a^{n+n!}b^n$ akzeptiert, womit erneut ein Widerspruch zur Definition von L gegeben ist.
Daher muss unsere (einzige) Annahme, nämlich dass die Raumkomplexität von M'' größenordnungsmäßig kleiner als $\log_2(n)$ ist, falsch sein.

Sollte sich der Eingabekopf nicht immer über einer Zelle befinden, in der ein Buchstabe des Eingabeworts steht, so gibt es eine natürliche Zahl h so, dass sich M'' für jedes $i \geq 1$ nach dem Lesen des i-ten Buchstaben von $a^n b^n$ nur noch maximal h Zellen nach links bewegt. Wäre dies nämlich nicht der Fall, so würde es öfter als $\#(U(2n))$ mal das erste a lesen. Dies würde implizieren, dass zweimal das gleiche Element von $U(2n)$ beim Lesen des ersten Buchstaben vorliegt. Damit würde sich eine Schleife ergeben, die dazu führt, dass $a^n b^n$ nicht akzeptiert wird.
Nun können wir obigen Beweis dahingehend modifizieren, dass wir jeweils die Anzahl der Buchstaben um h erhöhen, da nach dem Lesen des $h + i$-ten Buchstaben nur Zellen betreten werden, in denen Buchstaben des Eingabewortes stehen.

Wir haben die Komplexitäten bisher anhand der k-Band-TURING-Maschine eingeführt. Für die TURING-Maschine (ohne Arbeitsbänder und ohne separatem Ausgabeband) lässt sich die Zeitkomplexität in völliger Analogie definieren. Dagegen ist die Definition der Raumkomplexität $s_M(w)$ für eine TURING-Maschine M und ein Eingabewort w etwas problematisch, da zum einen auf dem Band stets schon $|w|$ Zellen beschriftet sind und zum anderen in der Regel die Eingabe vollständig gelesen werden muss, wodurch $s_M(w) \geq |w|$ als notwendig erscheint. Dies würde logarithmische Komplexität wie im obigen Beispiel unmöglich machen. Wir diskutieren daher für TURING-Maschinen nur die Zeitkomplexität.
Der folgende Satz gibt einen Zusammenhang zwischen den Komplexitäten der verschiedenen Varianten von Maschinen.

Satz 3.7 *Zu jeder deterministischen akzeptierenden k-Band-*TURING*-Maschine M, die auf jeder Eingabe stoppt, gibt es eine deterministische akzeptierende* TURING*-Maschine M' derart, dass*

$$T(M') = T(M) \quad \textit{und} \quad t_{M'}(n) = O((t_M(n))^2)$$

gelten und M' auf jeder Eingabe stoppt.

Beweis. Wir verwenden die Simulation von M durch M' in der Weise, dass wir alle Bänder zu einem Band zusammenfassen, in dessen Zellen dann $(k+1)$-Tupel von Bandsymbolen stehen. Die Simulation wird wie folgt vorgenommen. Durch das Koppeln von Lesesymbol und Zustand auf einem Band von M, d.h. statt x steht (x, z) in der Komponente des entsprechendes Bandes, wird die Stelle, wo sich der Kopf des Bandes befindet markiert. Die Simulation eines Schrittes von M besteht nun darin, dass der Kopf von M' von Beginn des beschriebenen Teils mehrfach über das Eingabeband von M' läuft und dabei der Reihe die Position des Lesekopfes über dem Eingabeband, die die Inhalte der Arbeitsbänder entsprechend der Arbeitsweise von M ändert. Jede Änderung bei M', die einer Änderung eines Bandinhaltes entspricht erfordert nur eine endliche Anzahl von Schritten. Ferner kann bei jedem Schritt von M der Inhalt eines Arbeitsbandes höchstens um 1 vergrößert werden, so dass jedes Band von M höchstens ein Wort der Länge $t_M(n)$ enthält. Damit sind von M' in jedem Simulationsschritt höchstens $2(k+1)t_M(n)+ck$ Schritte erforderlich, wobei c eine Konstante ist. Hieraus folgt die Behauptung, da $t_M(n)$ Schritte zu simulieren sind. □

Ohne Beweis geben wir das folgende Resultat, das zeigt, dass es Funktionen gibt, für deren Berechnung ein beliebig großer vorgegebener Zeitaufwand erforderlich ist.

Satz 3.8 *Zu jeder Funktion g von $\mathbf{N}$ in $\mathbf{N}$ gibt es eine Sprache L derart, dass für jede* TURING*-Maschine M, die L entscheidet,*

$$t_M(n) \geq g(n)$$

gilt. □

Um zu verdeutlichen, wie katastrophal die Aussage des Satzes 3.8 ist, betrachten wir die durch

$$g(0) = 2 \quad \text{und} \quad g(n+1) = g(n)^{g(n)}$$

gegebene Funktion g. Wir erhalten

$$f(1) = 4, \quad f(2) = 256, \quad f(3) = 256^{256} \approx 3 \cdot 10^{616},$$

d.h. es gibt eine Funktion, deren Berechnung auf einer beliebigen Maschine bereits auf Eingaben der Länge 3 mindestens $3 \cdot 10^{616}$ Schritte erfordert und damit praktisch unlösbar ist.

Da in den meisten Fällen von praktischer Bedeutung die Funktion $t_M(n)$ nicht genau bestimmt werden kann und man sich daher mit Abschätzungen zufrieden geben muss, führen wir folgende Sprechweisen ein.

Definition 3.9 *Es seien $t : \mathbf{N} \to \mathbf{N}$ eine Funktion, $f : X^* \to X^*$ eine* TURING*-berechenbare Funktion und $M = (X', Z, z_0, Q, \delta)$ eine deterministische* TURING*-Maschine mit $X \subseteq X'$ und $f_M = f$. Wir sagen, dass M die Funktion f in der Zeit t berechnet, wenn M für jedes Wort w aus dem Definitionsbereich von f nach höchstens $t(|w|)$ Überführungsschritten einen Stopzustand erreicht.*

Definition 3.10 *Es seien $t : \mathbf{N} \to \mathbf{N}$ eine Funktion und $L \subset X^*$ eine rekursive Sprache und $M = (X', Z, z_0, Q, \delta, F)$ eine akzeptierende deterministische* TURING*-Maschine mit $X \subset X'$ und $L = T(M)$. Wir sagen, dass M die Sprache L in der Zeit t entscheidet, wenn M für jedes Wort $w \in X^*$ nach höchstens $t(|w|)$ Überführungsschritten einen Stopzustand erreicht.*

Bisher haben wir nur deterministische TURING-Maschinen betrachtet. Auf nichtdeterministische TURING-Maschinen lassen sich die Begriffe nicht so einfach übertragen. Zuerst erinnern wir daran, dass bei Akzeptanz von w durch die nichtdeterministische TURING-Maschine M mindestens einmal bei Abarbeitung von w auf M ein akzeptierenden Zustand erreicht wird, bei anderen Abarbeitungen aber sowohl ablehnende als auch akzeptierende Zustände erreicht werden können. Daher ist $t_M(w)$ nicht eindeutig definierbar. Dies legt es nahe, nur eine Übertragung von Definition 3.10 vorzunehmen. Da auch bei Erreichen eines Stopzustandes bei einer Abarbeitung, bei einer nichtdeterministischen TURING-Maschine die Möglichkeit besteht, dass bei einer anderen Abarbeitung kein Stopzustand erreicht wird, ist es naheliegend, statt der Entscheidbarkeit einer Menge nur die Akzeptanz der Menge zu verlangen. Dies führt zu folgender Definition.

Definition 3.11 *Es seien $t : \mathbf{N} \to \mathbf{N}$ eine Funktion und $L \subset X^*$ eine rekursiv-aufzählbare Sprache und $M = (X', Z, z_0, Q, \delta, F)$ eine akzeptierende (deterministische oder nichtdeterministische)* TURING*-Maschine mit $X \subset X'$ und $L = T(M)$. Wir sagen, dass M die Sprache L in der Zeit t akzeptiert, wenn M für jedes Wort $w \in L$ nach höchstens $t(|w|)$ Überführungsschritten einen akzeptierenden Stopzustand erreicht.*

3.2 Nichtdeterminismus und das P-NP-Problem

Wir betrachten einführend das Erfüllungsproblem SAT, das der Illustration der Problematik dieses Abschnitts dienen soll, aber auch von großer theoretischer Bedeutung dafür ist.
Unter einer Disjunktion oder Alternative in n Booleschen Variablen (die nur mit den Wahrheitswerten 1 für *wahr* und 0 für *falsch* belegt werden können) verstehen wir einen logischen Ausdruck $E(x_1, x_2, ..., x_n)$ der Form

$$E(x_1, x_2, ..., x_n) = x_{i_1}^{\sigma_{i_1}} \vee x_{i_2}^{\sigma_{i_2}} \vee ... \vee x_{i_r}^{\sigma_{i_r}},$$

wobei $r \geq 1$, $i_j \in \{1, 2, \ldots, n\}$ und $\sigma_{i_j} \in \{0, 1\}$ für $1 \leq j \leq r$ gelten, x^1 die Identität und x^0 die Negation sind.

Problem:	SAT
Gegeben:	n Boolesche Variable $x_1, x_2, ..., x_n$ und m Alternativen $E_i(x_1, x_2, ..., x_n)$, $1 \leq i \leq m$.
Frage:	Gibt es eine Belegung $b : x_i \to a_i \in \{0, 1\}$ der Variablen derart, dass $E_j(a_1, a_2, ..., a_n)$ den Wert 1 für $1 \leq j \leq m$ annimmt, d.h. dass alle Alternativen bei b wahr werden.

Zur Lösung von SAT gibt es offenbar folgenden naheliegenden Algorithmus. Wir erzeugen alle 2^n möglichen Belegungen der logischen Variablen $x_1, x_2, ..., x_n$ und testen für jede Belegung, ob alle Alternativen auf dieser Belegung den Wert 1 annehmen. Da das Testen einer Belegung auf einer Disjunktion höchstens die Berechnung von n Negationen und $n - 1$ zweistelligen Alternativen erfordert, ergibt sich für den Gesamtaufwand die obere Schranke $(2n - 1) \cdot m \cdot 2^n$. Andererseits ist für diesen Algorithmus eine untere Schranke durch die Zahl 2^n der möglichen Belegungen gegeben. Damit gilt für diesen naheliegenden Algorithmus A

$$2^n \leq t_A(n) \leq (2n - 1) \cdot m \cdot 2^n.$$

Das exponentielle Wachstum der Zeitkomplexität von A zeigt, dass dieser Algorithmus praktisch für große Werte von n nicht verwendbar ist. (Wir werden im Folgenden zeigen, dass alle bisher bekannten Algorithmen zur Lösung von SAT ebenfalls exponentielles Verhalten der Zeitkomplexität aufweisen.) Offenbar ergibt sich der exponentielle Charakter von t_A aus der Tatsache, dass wir der Reihe nach - also sequentiell - die möglichen Belegungen durchtesten. Eine Verbesserung ist daher zu erwarten, wenn wir das Überprüfen der Werte der Belegungen auf den Disjunktionen „gleichzeitig“ („parallel“) durchführen könnten. Beim Algorithmenbegriff auf der Basis von Turing-Maschinen ist dies nicht möglich, weil die Überführungsfunktion δ eine Funktion auf der Menge der Konfigurationen erzeugt.
Daher ist es naheliegend, auch in diesem Zusammenhang nichtdeterministische Maschinen zu betrachten. Diese könnten nichtdeterministisch in n Schritten alle mögliche Belegungen erstellen (wir haben nur nichtdeterministisch für jede Variable die Belegung 1 oder 0 zu wählen) und können dann ebenfalls in $(2n - 1)m$ Schritten die Belegung testen. Damit ergibt sich höchstens die Komplexität $n + (2n - 1)m$.

Aus Satz 2.43 wissen wir, dass nichtdeterministische und deterministische Turing-Maschinen die gleiche Mengen von Sprachen akzeptieren. Aufgrund unserer Betrachtungen

zum Erfüllungsproblem SAT ist aber zu vermuten, dass der Aufwand zur Lösung eines Problems beim Übergang zu nichtdeterministischen Algorithmen sinken kann. Wir wollen dies nun für den polynomialen Fall etwas näher untersuchen. Dazu führen wir die folgenden Mengen von Sprachen ein.

Definition 3.12 **P** *sei die Menge aller Sprachen, die von einer deterministischen akzeptierenden* TURING-*Maschinen in polynomialer Zeit entschieden werden können.*
NP *sei die Menge aller Sprachen, die von einer nichtdeterministischen akzeptierenden* TURING-*Maschine in polynomialer Zeit akzeptiert werden können.*

Eine Sprache L liegt also genau dann in **P**, wenn es eine deterministische akzeptierenden TURING-Maschine M und ein Polynom p derart gibt, dass $T(M) = L$ gilt und M die Sprache L in der Zeit p entscheidet. Analog liegt L genau dann in **NP**, wenn es eine nichtdeterministische akzeptierenden TURING-Maschine M und ein Polynom p derart gibt, dass $T(M) = L$ gilt und M die Sprache L in der Zeit p akzeptiert.

Da deterministische TURING-Maschinen als ein Spezialfall der nichtdeterministischen TURING-Maschinen angesehen werden können, erhalten wir

$$\mathbf{P} \subseteq \mathbf{NP}\,.$$

Um zu zeigen, dass **P** echt in **NP** enthalten ist, reicht es ein Beispiel anzugeben, dass in **NP** aber nicht in **P** enthalten ist. Für den Nachweis der Gleichheit der beiden Mengen ist dagegen zu beweisen, dass jede Sprache aus **P** auch in **NP** liegt. Ziel dieses Abschnittes ist es, zu zeigen, dass auch für den Beweis der Gleichheit ein Beispiel ausreicht, da es Sprachen in **NP** mit folgender Eigenschaft gibt: falls diese Sprache in **P** liegt, so gilt **P**=**NP**.

Definition 3.13 *Seien $L_1 \subseteq X_1^*$ und $L_2 \subseteq X_2^*$ zwei Sprachen. Wir sagen, dass L_1 auf L_2 transformierbar ist, falls es eine Funktion τ gibt, die X_1^* auf X_2^* so abbildet, dass $a \in L_1$ genau dann gilt, wenn $\tau(a) \in L_2$ ist.*

Wir wollen diese Definition auch für Probleme angeben. Dazu erinnern wir zuerst daran, dass jedes Problem P durch eine Funktion $f_P : X_1 \times X_2 \times \ldots \times X_n \to \{0,1\}$ repräsentiert werden kann, wobei $f_P(a_1, a_2, \ldots, a_n) = 1$ genau dann gilt, wenn die Antwort auf die hinter dem Problem stehende Frage bei der Belegung der Variablen mit $a_1, a_2, \ldots, a_n$ „wahr“ ist. Im folgenden schreiben wir immer kurz X_P für das Produkt $X_1 \times X_2 \times \ldots \times X_n$ und $\underline{a}$ für $(a_1, \ldots, a_n)$.
Seien P_1 und P_2 zwei Probleme. Wir sagen, dass P_1 auf P_2 transformierbar ist, falls es eine Funktion τ gibt, die X_{P_1} auf X_{P_2} so abbildet, dass $f_{P_1}(\underline{a}) = 1$ genau dann gilt, wenn $f_{P_2}(\tau(\underline{a})) = 1$ ist.

Beispiel 3.14 Es sei $G = (V, E)$ ein Graph. Eine Teilmenge $V' \subseteq V$ heißt *Clique* in G, falls $(v, v') \in E$ für alle paarweise verschiedenen $v, v' \in V'$ gilt, d.h. die Knoten aus V' sind paarweise durch Kanten verbunden. Wir betrachten das *Cliquenproblem*

Gegeben:	Graph $G = (V, E)$, natürliche Zahl $k \geq 1$,
Frage:	Gibt es eine k-elementige Clique in G ?

und zeigen dass SAT auf das Cliquenproblem transformiert werden kann.
Seien die Alternativen

$$A_i(x_1, x_2, \ldots, x_n) = x_{i,1}^{\sigma_{i,1}} \vee x_{i,2}^{\sigma_{i,2}} \vee \ldots x_{i,r_i}^{\sigma_{i,r_i}}, \quad 1 \leq i \leq m,$$

gegegen. Wir konstruieren nun wie folgt den Graphen $G = (V, E)$. Zuerst setzen wir

$$V = \{(A_i, x_{i,j}^{\sigma_{i,j}}) : 1 \leq i \leq m, 1 \leq j \leq r_i\}.$$

Die Knoten (A, x^{σ}) und $(A', x'^{\sigma'})$ werden genau dann durch eine Kante verbunden, wenn $A \neq A', x \neq x'$ oder $A \neq A', x = x', \sigma = \sigma'$ gelten. E sei die Menge aller so konstruierten Kanten. Ferner setzen wir $k = m$.
Wir illustrieren die eben beschriebene Konstruktion durch ein Beispiel. Wir betrachten die Menge der Alternativen

$$A_1 = x \vee y, \quad A_2 = \overline{x} \vee \overline{y} \vee \overline{z}, \quad A_3 = y \vee z. \tag{3.2}$$

Der zugehörige Graph ist in Abbildung 3.1 dargestellt.

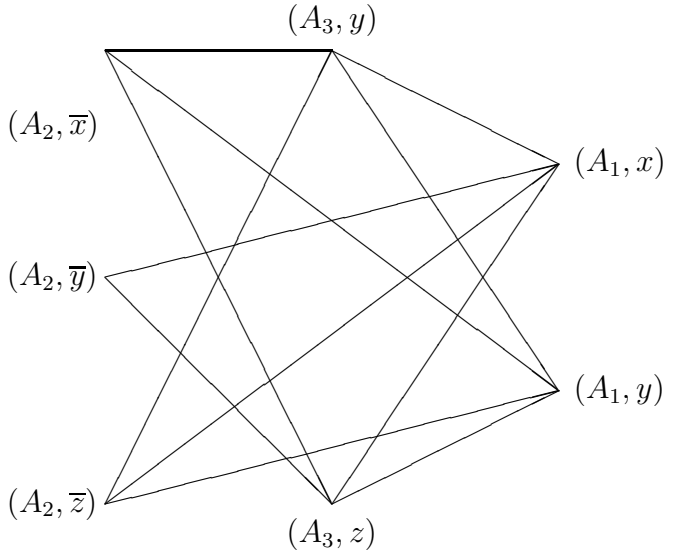


Abbildung 3.1: Graph zu den Alternativen aus (3.1)

Damit haben wir die in Definition 3.13 geforderte Funktion konstruiert, und es bleibt zu zeigen, dass genau dann eine Belegung existiert, für die alle m Alternativen *wahr* werden, wenn es in G eine m-elementige Clique gibt.
Sei zuerst $V' \subseteq V$ eine m-elementige Clique in G. Da nach Konstruktion zwei Knoten, die zur gleichen Alternative A gehören, durch keine Kante verbunden sind, muss V' zu jeder Alternative genau einen Knoten enthalten, d.h.

$$V' = \{(A_1, x_{1,j_1}^{\sigma_{1,j_1}}), (A_2, x_{2,j_2}^{\sigma_{2,j_2}}), \ldots, (A_m, x_{m,j_m}^{\sigma_{m,j_m}}).$$

Gilt für zwei Knoten aus V' die Beziehung $x_{s,j_s} = x_{t,j_t}$, so ist nach Konstruktion von G auch $\sigma_{s,j_s} = \sigma_{t,j_t}$, d.h. jede Variable taucht nur negiert oder nur unnegiert auf. Daher

können wir eine Belegung a_r, $1 \leq r \leq n$, so wählen, dass $a_{i,j_i}^{\sigma_{i,j_i}} = 1$ für $1 \leq i \leq m$ gilt. Damit gilt auch $A_i(a_1, a_2, \ldots, a_n) = 1$ für $1 \leq i \leq m$.
Gilt umgekehrt $A_i(a_1, a_2, \ldots, a_n) = 1$, so gibt es ein j_i, $1 \leq j_i \leq r_i$ mit $x_{i,j_i}^{\sigma_{i,j_i}} = 1$. Es ist nun leicht zu sehen, dass

$$V' = \{(A_1, x_{1,j_1}^{\sigma_{1,j_1}}), (A_2, x_{2,j_2}^{\sigma_{2,j_2}}), \ldots, (A_m, x_{m,j_m}^{\sigma_{m,j_m}})$$

eine m-elementige Clique ist.

In unserem Beispiel entsprechen die Belegungen $(0, 1, 1)$ bzw. $(1, 0, 1)$ den Cliquen $\{(A_1, y), (A_2, \overline{x}), (A_3, z)\}$ bzw. $\{(A_1, x), (A_2, \overline{y}), (A_3, z)\}$.

Beispiel 3.15 Wir betrachten das *Problem des Geschäftsreisenden*

Gegeben:	$n \geq 1$, n Städte $C_1, C_2, ..., C_n$, die Entfernungen $d(C_i, C_j)$ zwischen den Städten C_i und C_j für $1 \leq i, j \leq n$, $B \geq 0$
Frage:	Gibt es eine Rundreise $C_{i_1}, C_{i_2}, ..., C_{i_n}$ durch alle Städte, für die $(\sum_{j=1}^{n-1} d(C_{i_j}, C_{i_{j+1}}) + d(C_{i_n}, C_{i_1}) \leq B$ gilt?

und das *Problem der Existenz von* HAMILTON*-Kreisen*

Gegeben:	Graph $G = (V, E)$ mit $\#(V) = n$
Frage:	Enthält G einen HAMILTON-Kreis, d.h. gibt es eine Folge $v_1, v_2, ..., v_n$ von paarweise verschiedenen Knoten des Graphen G so, dass $(v_i, v_{i+1}) \in E$ für $1 \leq i \leq n$ und $(v_n, v_1) \in E$ gelten?

Wir geben nun eine Transformation des Problems der Existenz eines HAMILTON-Kreises auf das Problem des Geschäftsreisenden.
Sei $G = (V, E)$ ein gegebener Graph mit der Knotenmenge

$$V = \{a_1, a_2, ..., a_n\}.$$

Dann setzen wir

$$\begin{aligned} \tau(a_i) &= C_i \quad \text{für} \quad 1 \leq i \leq n, \\ d(C_i, C_j) &= \begin{cases} 1 & (a_i, a_j) \in E \\ 2 & (a_i, a_j) \notin E \end{cases} \end{aligned}$$

und

$$B = n.$$

Ist nun durch die Folge der Knoten $v_1 = a_{i_1}, v_2 = a_{i_2}, ..., v_n = a_{i_n}$
ein HAMILTON-Kreis gegeben, so definiert die Folge $C_{i_1} = \tau(a_{i_1}), C_{i_2} = \tau(a_{i_2}), ..., C_{i_n} = \tau(a_{i_n})$ eine Rundreise durch alle Städte, bei der

$$(\sum_{j=1}^{n-1} d(C_{i_j}, C_{i_{j+1}})) + d(C_{i_n}, C_{i_1}) = (n-1) + 1 = n = B$$

gilt, womit gezeigt ist, dass das durch $n, C_1, ..., C_n$, die Abstandsfunktion d und B gegebene Problem des Geschäftsreisenden eine Lösung besitzt.
Sei umgekehrt für das durch $n, C_1, C_2, ..., C_n$, die obige Abstandsfunktion d und $B = n$ beschriebene Problem des Geschäftsreisenden die Lösung $C_{i_1}, C_{i_2}, ..., C_{i_n}$ gegeben. Wegen $B = n$ müssen $d(C_{i_j}, C_{i_{j+1}}) = 1$ für $1 \leq j \leq n-1$ und $d(C_{i_n}, C_{i_1}) = 1$ gelten. Das besagt aber gerade, dass $a_{i_1}, a_{i_2}, ..., a_{i_n}$ ein HAMILTON-Kreis in G ist.

Definition 3.16 *Wir sagen, dass die Sprache L_1 polynomial auf die Sprache L_2 transformierbar ist, wenn L_1 durch eine Funktion τ auf L_2 transformiert wird, die mit polynomialer Zeitkomplexität berechnet werden kann, d.h. τ wird von einer deterministischen* TURING-MASCHINE *M mit $t_M(n) = \theta(n^r)$ für ein gewisses $r \in \mathbf{N}$ induziert. Wir bezeichnen dies durch $L_1 \alpha L_2$.*

Die Transformation in Beispiel 3.14 ist offenbar polynomial, denn wenn SAT durch n Variablen und m Alternativen gegeben ist, hat der zugehörige Graph höchstens $n \cdot m$ Knoten und höchstens $n \cdot n(m-1)$ Kanten, die alle mittels $nm + n^2(m-1)$-maligen Durchmustern aller Alternativen bestimmt werden können.
Auch die Transformation in Beispiel 3.15 ist polynomial, wie aus der Definition von τ sofort zu sehen ist.

Lemma 3.17 *i) α ist eine transitive Relation auf der Menge der Sprachen.*
ii) Aus $L_2 \in \mathbf{P}$ und $L_1 \alpha L_2$ folgt $L_1 \in \mathbf{P}$.
iii) Aus $L_2 \in \mathbf{NP}$ und $L_1 \alpha L_2$ folgt $L_1 \in \mathbf{NP}$.

Beweis. i) folgt aus der leicht zu verifizierenden Tatsache, dass aus der Berechenbarkeit von f_1 und f_2 in polynomialer Zeit die Berechenbarkeit von $f_1 \circ f_2$ in polynomialer Zeit folgt.
ii) Wir haben zu zeigen, dass $w \in L_1$ in polynomialer Zeit durch eine deterministische TURING-Maschine entscheiden werden kann. Nach Voraussetzung können wir in polynomialer Zeit $\tau(w)$ mittels einer deterministischen TURING-Maschine M_1 bestimmen. Wegen $L_2 \in \mathbf{P}$ kann $\tau(w) \in L_2$ nun in polynomialer Zeit von einer deterministischen Turing-Maschine M_2 entschieden werden. Nach der Definition der Transformierbarkeit gilt $w \in L_1$ genau dann, wenn $\tau(w) \in L_2$ gültig ist. Somit kann $w \in L_1$ durch die deterministische TURING-Maschine, die zuerst wie M_1 und dann wie M_2 arbeitet, in polynomialer Zeit entschieden werden.
iii) wird analog zu ii) bewiesen. □

Definition 3.18 *Eine Sprache L heißt $\mathbf{NP}$-vollständig, wenn folgende Bedingungen erfüllt sind:*
i) $L \in \mathbf{NP}$,
ii) $L' \alpha L$ gilt für jede Sprache $L' \in \mathbf{NP}$.

Satz 3.19 *Die folgenden Aussagen sind gleichwertig:*
i) $\mathbf{P} = \mathbf{NP}$.
ii) $L \in \mathbf{P}$ gilt für jede $\mathbf{NP}$-vollständige Sprache L.
iii) $L \in \mathbf{P}$ gilt für eine $\mathbf{NP}$-vollständige Sprache L.

Beweis. i) $\Rightarrow$ ii). Sei L eine **NP**-vollständige Sprache. Da nach Definition $L \in \mathbf{NP}$ gilt, folgt aus $\mathbf{P} = \mathbf{NP}$ sofort $L \in \mathbf{P}$.
ii) $\Rightarrow$ iii). Diese Implikation ist trivial.
iii) $\Rightarrow$ i). Seien L eine **NP**-vollständige Sprache und L' eine Sprache aus **NP**. Aus der Definition der **NP**-Vollständigkeit folgt $L' \alpha L$. Wegen Lemma 3.17, ii) gilt nun $L' \in \mathbf{P}$ wegen der Voraussetzung $L \in \mathbf{P}$.
Damit ist die Inklusion $\mathbf{NP} \subseteq \mathbf{P}$ bewiesen. Wegen der Gültigkeit der umgekehrten Inklusion folgt die Behauptung. □

Die Bedeutung der **NP**-vollständigen Sprachen besteht nach Satz 3.19 in Folgendem: Können wir für eine **NP**-vollständige Sprache zeigen, dass sie in **P** liegt, so gilt $\mathbf{P} = \mathbf{NP}$; beweisen wir dagegen für eine **NP**-vollständige Sprache, dass sie nicht in **P** ist, so gilt $\mathbf{P} \neq \mathbf{NP}$. **NP**-vollständige Sprachen sind also Scharfrichter für die Frage „**P**=**NP** ?“.

Wir beweisen nun erst einmal die Existenz **NP**-vollständiger Probleme.

Satz 3.20 *SAT ist* **NP***-vollständig.*

Beweis. Entsprechend der Definition **NP**-vollständiger Probleme, müssen wir zum einen zeigen, dass das Erfüllbarkeitsproblem für aussagenlogische Ausdrücke in konjunktiver Normalform in **NP** liegt, und zum anderen haben wir zu zeigen, dass jede Sprache aus **NP** polynomial auf dieses Erfüllbarkeitsproblem transformierbar ist.

$SAT \in \mathbf{NP}$ haben wir bereits informell bewiesen. Ein formaler Beweis bleibt dem Leser überlassen.

Es sei L eine beliebige Sprache aus **NP**. Dann gibt es eine nichtdeterministische Turing-Maschine $M = (X, Z, z_0, Q, \tau)$, die L in polynomialer Zeit akzeptiert, die also für eine Eingabe $w \in L$ höchstens $p(|w|)$ Schritte benötigt, wobei p ein Polynom ist. Es seien $X = \{a_1, a_2, \ldots, a_r\}$, $w = a_{i_1} a_{i_2} \ldots a_{i_n}$, $* = a_0$, $Z = \{z_0, z_1, \ldots, z_m\}$ und ohne Beschränkung der Allgemeinheit $Q = \{z_1\}$. Ferner sei

$$q = \max\{\#\tau(z, a) \mid z \in Z,\ a \in X\}\,.$$

Wir nummerieren die Zellen des Bandes mit ganzen Zahlen in der Weise, dass die Zelle mit der Nummer 1 zu Beginn der Arbeit den ersten Buchstaben von w enthält und setzen nach rechts (bzw. links) durch Addition (bzw. Subtraktion) von 1 die Nummerierung fort. Setzen wir noch $t = p(|w|) + 1$, so kann der Kopf während der Arbeit von M nur über den Zellen stehen, die mit einer Zahl k, $-t \leq k \leq t$, nummeriert sind.
Wir definieren nun einen aussagenlogischen Ausdruck, der die Arbeit von M auf der Eingabe w beschreibt. Als Variablen benutzen wir

$$Z_{ij}, 1 \leq i \leq t, 0 \leq j \leq m,$$
$$H_{ik}, 1 \leq i \leq t, -t \leq k \leq t,$$
$$S_{ikl}, 1 \leq i \leq t, -t \leq k \leq t, 0 \leq l \leq r,$$

die folgende Bedeutung haben:

- Z_{ij} nimmt genau dann den Wert 1 an, wenn M zur Zeit i im Zustand z_j ist,

- H_{ik} nimmt genau dann den Wert 1 an, wenn der Kopf von M zur Zeit i über der Zelle k steht, und

- S_{ikl} nimmt genau dann den Wert 1 an, wenn zur Zeit i in der Zelle k auf dem Band von M der Buchstabe a_l steht.

Wir betrachten die folgenden Ausdrücke:

(1) $(Z_{i0} \vee Z_{i1} \vee \cdots \vee Z_{im})$ für $1 \leq i \leq t$,
(2) $(\neg Z_{ij} \vee \neg Z_{ij'})$ für $1 \leq i \leq t$, $0 \leq j < j' \leq m$,
(3) $(H_{i,-t} \vee H_{i,-t+1} \vee \cdots \vee H_{it})$ für $1 \leq i \leq t$,
(4) $(\neg H_{ik} \vee \neg H_{ik'})$ für $1 \leq i \leq t$, $-t \leq k < k' \leq t$,
(5) $(S_{ik0} \vee S_{ik1} \vee \cdots \vee S_{ikr})$ für $1 \leq i \leq t$, $-t \leq k \leq t$,
(6) $(\neg S_{ikl} \vee \neg S_{ikl'})$ für $1 \leq i \leq t$, $-t \leq k \leq t$, $0 \leq l < l' \leq r$,
(7) Z_{10},
(8) H_{11},
(9) $S_{11i_1}, S_{12i_2}, \ldots, S_{1ni_n}$ und S_{1k0} für $-t \leq k \leq t$, $k \notin \{1, 2, \ldots, n\}$,
(10) Z_{t1},
(11) $(\neg Z_{ij} \vee \neg H_{ik} \vee \neg S_{ikl} \vee (Z_{i+1,j_1} \wedge H_{i+1,k_1} \wedge S_{i+1,k,l_1}) \vee \ldots \vee (Z_{i+1,j_u} \wedge H_{i+1,k_u} \wedge S_{i+1,k,l_u}))$
für $1 \leq i \leq t-1$, $0 \leq j \neq 1 \leq m$, $-t \leq k \leq t$, $0 \leq l \leq r$,
$\delta(z_j, a_l) = \{(z_{j_1}, a_{l_1}, d_1), (z_{j_2}, a_{l_2}, d_2), \ldots, (z_{j_u}, a_{l_u}, d_u)$,
$k_p = k-1$ für $d_p = L$, $k_p = k$ für $d_p = N$, $k_p = k+1$ für $d_p = R$, $1 \leq p \leq u$,
(12) $(\neg Z_{i1} \vee \neg H_{ik} \vee \neg S_{ikl} \vee (Z_{i+1,1} \wedge H_{i+1,k} \wedge S_{i+1,k,l}))$
für $1 \leq i \leq t-1$, $-t \leq k \leq t$, $0 \leq l \leq r$,
(13) $(\neg S_{ikl} \vee \neg H_{ik'} \vee S_{i+1,k,l})$ für $1 \leq i \leq t-1$, $-t \leq k \neq k' \leq t$, $0 \leq l \leq r$.

Durch diese Wahl der Ausdrücke wird folgendes erreicht: (1) nimmt genau dann den Wert 1 an, wenn mindestens eine der Variablen Z_{ij}, $0 \leq j \leq m$, den Wert 1 annimmt, d.h. wenn sich die Maschine M zur Zeit i in mindestens einem Zustand z_j befindet. Die Alternative (2) nimmt genau dann den Wert 0 an, wenn Z_{ij} und $Z_{ij'}$ den Wert 1 annehmen, d.h. wenn sich M zur Zeit i sowohl im Zustand z_j als auch im Zustand $z_{j'}$ befindet. Die Alternativen (1) und (2) sind also genau dann beide wahr, wenn sich M zur Zeit i in genau einem Zustand befindet.
Analog sichern (3) und (4), dass sich der Kopf von M zur Zeit i über genau einer Zelle befindet, und (5) und (6) bedeuten, dass in der Zelle k zur Zeit i genau ein Buchstabe steht.
Die Alternativen (7), (8) und (9) beschreiben die Anfangskonfiguration; (10) sichert das Erreichen einer Endkonfiguration.
Der Ausdruck (11) beschreibt das Verhalten von M, wenn noch kein Endzustand erreicht ist. Bei Wahrheit von Z_{ij}, H_{ik} und S_{ikl} muss eine der Konjunktionen $(Z_{i+1,j_p} \wedge H_{i+1,k_p} \wedge S_{i+1,k,l_p})$, $1 \leq p \leq u$, wahr werden. Wenn M zur Zeit i im Zustand z_j ist und das Symbol a_l in Zelle k liest, dann schreibt M das Symbol a_{l_p} in die Zelle k, geht in den Zustand z_{j_p} und bewegt den Kopf zur Zelle k_p. Folglich wird eine der möglichen Aktionen von M ausgeführt.
Analog sichert (12), dass bei Erreichen eines Endzustandes keine Änderung mehr vorgenommen wird, d.h. wir setzen die Arbeit von M im Unterschied zur formalen Definition auch bei Erreichen des Endzustandes fort, um den Zeitpunkt t zu erreichen. Die Alterna-

tive (13) besagt, dass der Inhalt der Zelle nicht verändert wird, wenn sich der Kopf nicht über der Zelle befindet.
Es sei B die Konjunktion aller Ausdrücke aus (1) – (15). Aus obigen Bemerkungen folgt sofort, dass es genau dann eine Belegung der Variablen gibt, bei der alle Ausdrücke (1) – (15) den Wert 1 annehmen, wenn die Akzeptanz der Eingabe w höchstens $p(|w|)$ Schritte erfordert. Somit liegt eine Transformation von L auf das Erfüllbarkeitsproblem für aussagenlogische Ausdrücke vor.
Wir haben noch zu zeigen, dass diese Transformation polynomial ist. Dazu reicht es aus, festzustellen, dass der aus M und w konstruierte Ausdruck B höchstens die Länge

$$\begin{aligned}(2m+4)t + 8 \cdot \frac{1}{2}m(m+1)t + (4t+4)t + 8 \cdot \frac{1}{2}(2t+1)2t^2 + (2r+4)(2t+1)t \\ +8 \cdot \frac{1}{2}r(r+1)(2t+1)t + 2 \cdot 1 + 2 \cdot 1 + 2 \cdot (2t+1) + 2 \cdot 1 \\ +(8q+11)(m+1)(2t+1)(t-1)(r+1) + 10(r+1)(2t+1)2t^2 \\ \leq (2r+8q+40)(m^2+1)(r^2+1)2t^2(2t+1)\end{aligned}$$

hat, wobei sich die ersten 10 Summanden aus den Längen der Alternativen der Typen (1) – (10) ergeben, der elfte Summand eine obere Abschätzung der Länge der Ausdrücke aus (11) und (12) ist und der letzte Summand die Länge der Alternativen vom Typ (13) ist (dabei gibt bei jedem Summanden der erste Faktor jeweils die um Eins vergrößerte Länge eines Ausdrucks der Form an, wobei die hinzugefügte Eins das in B dem Ausdruck folgende $\wedge$ erfasst; das Produkt der anderen Faktoren gibt die Anzahl der entsprechende Ausdrücke an). □

Wir haben bereits oben auf die Bedeutung der **NP**-vollständigen Sprachen für die Lösung des Problems „**P**=**NP** ?“ hingewiesen. Daher wollen wir nun eine Reihe von **NP**-vollständigen Sprachen aus verschiedenen Bereichen der Mathematik und Informatik angeben. Auf Beweise werden wir dabei weitgehend verzichten. In den Fällen, wo wir einen Beweis geben werden, wird der folgende Satz angewandt.

Satz 3.21 *Ist die* **NP***-vollständige Sprache L polynomial auf die Sprache L' aus* **NP** *transformierbar, so ist L' auch* **NP***-vollständig.*

Beweis. Für jede Sprache Q aus **NP** gilt $Q \alpha L$. Weiterhin haben wir nach Voraussetzung $L \alpha L'$. Damit folgt $Q \alpha L'$ für alle $Q \in$ **NP**. □

Diese Methode ist also erneut die Reduktion eines Problems auf ein anderes, wobei sich die **NP**-Vollständigkeit überträgt.
Bei den folgenden Beispielen werden wir – der Anschaulichkeit halber – statt Sprachen die zugehörigen Probleme verwenden.

Satz 3.22 *Das Cliquenproblem ist* **NP***-vollständig.*

Beweis. Nach Beispiel 3.14 und der Bemerkung nach Definition 3.16 ist SAT polynomial auf das Cliquenproblem transformierbar. Außerdem ist das Cliquenproblem sicher in **NP**, da wir nichtdeterministisch in polynomialer Zeit eine k-elementige Menge V' von Knoten auswählen und dann in polynomialer Zeit testen können, ob V' eine Clique ist. Nach Satz 3.21 ist das Cliquenproblem damit als **NP**-vollständig nachgewiesen. □

Ohne Beweis geben wir nun die folgende Aussage.

Satz 3.23 *Das Problem der Existenz von* HAMILTON*-Kreisen ist* **NP***-vollständig.* □

Satz 3.24 *Das Problem des Geschäftsreisenden ist* **NP***-vollständig.*

Beweis. Nach dem Beispiel 3.15 ist das Problem der Existenz von HAMILTON-Kreisen auf das Problem des Geschäftsreisenden polynomial transformierbar. Satz 3.24 ist daher nach Satz 3.21 bewiesen, wenn wir gezeigt haben, dass das Problem des Geschäftsreisenden in **NP** liegt. Dies folgt aber leicht, wenn wir nichtdeterministisch alle möglichen Rundreisen erzeugen und dann testen, ob sich für eine Rundreise ein Wert $\leq B$ ergibt, da beide Teilschritte mit polynomialen Aufwand erledigt werden können. □

Wir betrachten noch eine Variante des Problems des Geschäftsreisenden, die ein spezielles diskretes Optimierungsproblem darstellt.

Problem:	Minimale Rundreise
Gegeben:	natürliche Zahl $n \geq 1$,
	Städte $C_1, C_2, \ldots, C_n$ mit den Abständen $d(C_i, C_j)$, $1 \leq i, j \leq n$,
Frage:	Wie groß ist der minimale Wert von $d(C_{i_n}, C_{i_1}) + \Sigma_{j=1}^{n-1} d(C_{i_j}, C_{i_{j+1}})$, wobei das Minimum über alle Permutation von $\{1, 2, \ldots n\}$ zu nehmen ist?

Satz 3.25 *Das Problem der minimalen Rundreise ist* **NP***-vollständig.*

Beweis. Sei

$$m = max\{d(C_i, C_j) : 1 \leq i, j \leq n\}.$$

Dann ist das gesuchte Minimum beim Problem der minimalen Rundreise sicher höchstens $m \cdot (n+1)$. Somit kann das Problem der minimalen Rundreise durch sequentielles Abarbeiten des Problems des Geschäftsreisenden mit den Werten $B_i = i$, $1 \leq i \leq m(n+1)$, gelöst werden.
Umgekehrt liefert die Bestimmung des Minimums auch die Antwort auf die Frage nach einer Rundreise mit einer Länge $\leq B$. □

Satz 3.26 *Das Problem der (Knoten-)Färbbarkeit von Graphen*

Gegeben:	*Graph $G = (V, E)$ und natürliche Zahl $k \geq 3$*
Frage:	*Gibt es eine Färbung der Knoten von G mit k Farben, so dass durch eine Kante verbundene Knoten jeweils verschieden gefärbt sind?*

ist **NP***-vollständig.* □

Für $k = 2$ gibt es eine Lösung des Färbbarkeitproblems mit polynomialem Aufwand.

Satz 3.27 *Das Problem der Teilmengensumme*

Gegeben:	*endliche Menge $A \subseteq \mathbf{N}$ und natürliche Zahl $b \in \mathbf{N}$*
Frage:	*Gibt es eine Teilmenge $A' \subseteq A$ derart, dass $\Sigma_{a \in A'} a = b$ gilt?*

ist **NP***-vollständig.* □

Satz 3.28 *Das Problem der Lösbarkeit diophantischer quadratischer Gleichungen*

Gegeben: *natürliche Zahlen* a, b, c
Frage: *Gibt es eine Lösung von* $ax^2 + by = c$ *in natürlichen Zahlen?*

ist **NP***-vollständig.* □

Wir wollen nun ein Problem aus der Theorie der Datenbanken betrachten, für das wir das CODDsche relationale Datenbankmodell zugrundelegen. Es besteht aus Objekten und zugeordneten Attributwerten. Die Notation erfolgt meist in Form einer Tabelle, in deren erster Spalte die Objekte stehen und in den weiteren Spalten, die den Attributen entsprechen, stehen in der Zeile von einem Objekt die ihm zugeordneten Attributwerte. Die folgende Tabelle gibt ein Beispiel.

Objekt	Name	Vorname	Immatrikulations-nummer	Universität	Fakultät/ Fachbereich
1	Meyer	Heike	12345678	RWTH Aachen	Informatik
2	Schulz	Ulrike	21436587	TU München	Elektrotechn.
3	Müller	Heike	12348765	TU Dresden	Elektrotechn.
4	Muster	Fritz	56781234	TH Darmstadt	Mathematik.
5	Meyer	Ulrich	65874321	TU Berlin	Mathematik
6	Müller	Fritz	87654321	RWTH Aaachen	Informatik

Für das Objekt i und das Attribut A sei der i zugeordnete Attributwert mit $A(i)$ bezeichnet. Wir sagen, dass das Attribut A von den Attributen $B_1, B_2, \ldots, B_k$ abhängig ist, wenn die durch $f(B_1(i), B_2(i), \ldots, B_k(i)) = A(i)$ gegebene Abbildung eine Funktion ist, d.h. wenn der Wert $A(i)$ für jedes i bereits durch die Werte $B_1(i), B_2(i), \ldots, B_k(i)$ eindeutig festgelegt ist. Wir schreiben hierfür $\{B_1, B_2, \ldots, B_k\} \succ A$.

Im obigen Beispiel gelten z.B. {Immatrikulationsnummer} $\succ$ Name und {Name, Vorname} $\succ$ Immatrikulationsnummer, aber nicht {Name} $\succ$ Vorname und nicht {Vorname} $\succ$ Name.

Es sei eine Datenbank mit der Menge H von Attributen gegeben. Eine Teilmenge K von H heißt Schlüssel, falls $K \succ B$ für jedes $B \in H$ gilt.

Satz 3.29 *Das Problem der Existenz von Schlüsseln in einer Datenbank*

Gegeben: *Datenbank mit Menge* H *von Attributen, natürliche Zahl* k
Frage: *Gibt es einen Schlüssel* K *für* F *mit* $\#(K) \leq k$ *?*

ist **NP***-vollständig.* □

Wie in Satz 3.24 kann ausgehend von Satz 3.29 anstelle von Satz 3.25 bewiesen werden, dass auch das Problem der Bestimmung eines minimalen Schlüssels (hinsichtlich der Mächtigkeit) **NP**-vollständig ist.

Das Problem, ob **P**=**NP** gilt, ist bis heute noch ungelöst. Insbesondere gibt es also für alle bekannten **NP**-vollständigen Probleme bis heute keinen deterministischen Algorithmus, der sie in polynomialer Zeit löst, aber es gibt auch kein solches Problem, für das die Nichtexistenz eines polynomialen Algorithmus gezeigt werden konnte. Hat man ein

NP-vollständiges Problem gegeben, ist daher nicht zu erwarten, dass man dafür einen polynomialen Algorithmus findet, und sollte sich mit einem exponentiellen Algorithmus zufriedengeben. Dies wird noch dadurch unterstützt, dass allgemein die Relation **P**≠**NP** vermutet wird.

Die Überlegungen, die wir in diesem Kapitel bezüglich der Zeitkomplexität durchgeführt haben, lassen sich im wesentlichen auch für die Raumkomplexität anstellen.

Übungsaufgaben

1. Gegeben sei der Graph $G = (V, E)$ mit

$$\begin{aligned} V &= \{1,2,3,4,5,6,7,8,9,10,11\}, \\ E &= \{(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10),(1,11), \\ &\quad (2,4),(2,10),(3,5),(3,7),(3,9),(3,11),(4,6),(5,7),(5,9),(5,11), \\ &\quad (6,8),(7,9),(7,11),(8,10),(9,11)\}. \end{aligned}$$

   Eine Überdeckung von G ist eine Menge $V' \subseteq V$ derart, dass $\{v, v'\} \cap V' \neq \emptyset$ für alle Kanten $(v, v') \in E$ gilt.
   Bestimmen Sie

   (a) die maximale Zahl k, für die es eine Clique aus k Elementen gibt,

   (b) die minimale Zahl k, für die G k-knotenfärbbar ist,

   (c) die minimale Zahl k, für die eine Überdeckung aus k Elementen existiert.

2. Geben Sie eine Transformation des Cliquenproblems auf das *Überdeckungsproblem*:

   Gegeben: Graph $G = (V, E)$, $k \in \mathbf{N}$,
   Frage : Gibt es eine k-elementige Überdeckung von G ?

   (Die Definition der Überdeckung ist in Übungsaufgabe 3. gegeben.
   Hinweis: Man verwende den Komplementärgraph $G' = (V, E')$ mit $E' = \{(v, v') : (v, v') \notin E\}$.)

3. Beweisen Sie die **NP**-Vollständigkeit von $3-SAT$, das sich von SAT dadurch unterscheidet, dass alle Alternativen die Form $x_i^{\sigma_i} \vee x_j^{\sigma_j} \vee x_k^{\sigma_k}$ für gewisse $1 \leq i < j < k \leq n$ haben.
   (Hinweis: Man ersetze eine beliebige Alternative A unter Einbeziehung von zusätzlichen Variablen durch eine Menge von Alternativen mit jeweils genau drei Variablen, so dass A genau dann *wahr* wird, wenn alle Alternativen aus M *wahr* werden.)

4. Konstruieren Sie entsprechend Beispiel 3.14 den Graphen für die Alternativen

$$x \vee y \vee \overline{z}, \quad \overline{x} \vee \overline{y} \vee z, \quad y \vee \overline{z}.$$

5. Beweisen Sie, dass das Cliquenproblem für festes k in **P** liegt.

6. Beweisen Sie, dass das Problem der Knotenfärbung für $k = 2$ in **P** liegt.

Kapitel 4

Ergänzungen I : Weitere Modelle der Berechenbarkeit

4.1 Rekursive Funktionen

Im ersten Teil der Vorlesung haben wir berechenbare Funktionen als von Programmen bzw. TURING-Maschinen induzierte Funktionen eingeführt. In diesem Abschnitt gehen wir einen Weg, der etwas direkter ist. Wir werden Basisfunktionen definieren und diese als berechenbar ansehen. Mittels Operationen, bei denen die Berechenbarkeit nicht verlorengeht, werden dann weitere Funktionen erzeugt.

Wir geben nun die formalen Definitionen. Als *Basisfunktionen* betrachten wir:

- die nullstellige Funktion $Z_0 : \mathbf{N}_0 \to \mathbf{N}_0$, die den (konstanten) Wert 0 liefert,
- die Funktion $S : \mathbf{N}_0 \to \mathbf{N}_0$, bei der jeder natürlichen Zahl ihr Nachfolger zugeordnet wird,
- die Funktion $P : \mathbf{N}_0 \to \mathbf{N}_0$, bei der jede natürliche Zahl $n \geq 1$ auf ihren Vorgänger und die 0 auf sich selbst abgebildet wird,
- die Funktionen $P_i^n : \mathbf{N}_0^n \to \mathbf{N}_0$, die durch

$$P_i^n(x_1, x_2, \ldots, x_n) = x_i$$

  definiert sind.

Anstelle von $S(n)$ schreiben wir zukünftig auch - wie üblich - $n + 1$. P_i^n ist die übliche Projektion eines n-Tupels auf die i-te Komponente (Koordinate).

Als *Operationen* zur Erzeugung neuer Funktionen betrachten wir die beiden folgenden Schemata:

- *Kompositionsschema*: Für eine m-stellige Funktion g und m n-stellige Funktionen $f_1, f_2, \ldots, f_m$ definieren wir die n-stellige Funktion f vermöge

$$f(x_1, x_2, \ldots, x_n) = g(f_1(x_1, x_2, \ldots, x_n), f_2(x_1, x_2, \ldots, x_n), \ldots, f_m(x_1, x_2, \ldots, x_n)).$$

- *Rekursionsschema*: Für fixierte natürliche Zahlen $x_1, x_2 \ldots, x_n$, eine n-stellige Funktion g und eine $(n+2)$-stellige Funktion h definieren wir die $(n+1)$-stellige Funktion f vermöge

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n), \\ f(x_1, x_2, \ldots, x_n, y+1) &= h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)). \end{aligned}$$

Zuerst erwähnen wir, dass das Kompositionsschema eine einfache Formalisierung des „Einsetzens“ ist.
Wir merken an, dass das gegebene Rekursionsschema eine parametrisierte Form der klassischen Rekursion

$$\begin{aligned} f(0) &= c \\ f(y+1) &= h(y, f(y)), \end{aligned}$$

wobei c eine Konstante ist, mit den Parametern $x_1, x_2, \ldots, x_n$ ist.
Für das Kompositionsschema ist sofort einzusehen, dass ausgehend von festen Funktionen $g, f_1, f_2, \ldots, f_m$ genau eine Funktion f definiert wird. Wir zeigen nun, dass dies auch für das Rekursionsschema gilt, wobei wir (um die Bezeichnungen einfach zu halten) dies nur für die klassische parameterfreie Form durchführen. Zuerst einmal ist klar, dass durch das Schema eine Funktion definiert wird (für $y = 0$ ist der Wert festgelegt, für $y \geq 1$ lässt er sich schrittweise aus der Rekursion berechnen). Wir zeigen nun die Eindeutigkeit. Seien dazu f_1 und f_2 zwei Funktionen, die den Gleichungen des Rekursionsschemas genügen. Mittels vollständiger Induktion beweisen wir nun $f_1(y) = f_2(y)$ für alle natürlichen Zahlen y. Laut Schema gilt für $y = 0$ die Beziehung

$$f_1(0) = f_2(0) = c,$$

womit der Induktionsanfang gezeigt ist. Sei die Aussage schon für alle natürlichen Zahlen $x \leq y$ bewiesen. Dann gilt

$$f_1(y+1) = h(y, f_1(y)) = h(y, f_2(y)) = f_2(y+1),$$

wobei die erste und letzte Gleichheit aus dem Rekursionsschema und die zweite Gleichheit aus der Induktionsvoraussetzung folgen.

Definition 4.1 *Eine Funktion $f : \mathbf{N}_0^n \to \mathbf{N}_0$ heißt primitiv-rekursiv, wenn sie mittels endlich oft wiederholter Anwendung von Kompositions- und Rekursionsschema aus den Basisfunktionen erzeugt werden kann.*

Wir lasssen dabei auch die 0-malige Anwendung, d. h. keine Anwendung, als endlich oftmalige Anwendung zu; sie liefert stets eine der Basisfunktionen.

Beispiel 4.2 a) Ausgehend von der Basisfunktion S gewinnen wir mittels Kompositionsschema die Funktion f mit $f(n) = S(S(n))$, aus der durch erneute Anwendung des Kompositionsschemas f' mit $f'(n) = S(f(n)) = S(S(S(n)))$ erzeugt werden kann. Offenbar ordnen f bzw. f' jeder natürlichen Zahl ihren zweiten bzw. dritten Nachfolger zu. Beide Funktionen sind nach Definition primitiv-rekursiv.

b) Wegen $x = P(S(x))$ ist die identische Funktion $id : \mathbf{N}_0 \to \mathbf{N}_0$ mit $id(x) = x$ ebenfalls primitiv-rekursiv.

c) Die nullstellige konstante Funktion Z_0 gehört zu den Basisfunktionen und ist daher primitiv-rekursiv. Wir zeigen nun, dass auch die n-stellige Funktion Z_n mit $Z_n(x_1, x_2, \ldots, x_n) = 0$ für alle $x_1, x_2, \ldots, x_n$ ebenfalls primitiv-rekursiv ist.
Es sei $n = 1$. Dann betrachten wir das Rekursionsschema

$$Z_1(0) = Z_0 \quad \text{und} \quad Z_1(y+1) = P_2^2(y, Z_1(y))\,.$$

Wir zeigen mittels vollständiger Induktion, dass Z_1 die einstellige konstante Funktion mit dem Wertevorrat 0 ist. Offensichtlich gilt $Z_1(0) = 0$, da Z_0 den Wert 0 liefert. Sei nun schon $Z_1(y) = 0$ gezeigt. Dann ergibt sich aus der zweiten Rekursionsgleichung sofort $Z_1(y+1) = P_2^2(y, 0) = 0$, womit der Induktionsschritt vollzogen ist.
Nehmen wir nun an, dass wir bereits die n-stellige konstante Funktion Z_n mit dem Wert 0 als primitiv-rekursiv nachgewiesen haben, so können wir analog zu Obigem zeigen, dass das Rekursionsschema

$$\begin{aligned} Z_{n+1}(x_1, x_2, \ldots x_n, 0) &= Z_n(x_1, x_2, \ldots x_n)\,, \\ Z_{n+1}(x_1, x_2, \ldots x_n, y+1) &= P_{n+2}^{n+2}(x_1, x_2, \ldots, x_n, y, Z_{n+1}(x_1, x_2, \ldots x_n, y)) \end{aligned}$$

die $(n+1)$-stellige konstante Funktion Z_{n+1} mit dem Wert 0 liefert.

d) Die Addition und Multiplikation natürlicher Zahlen lassen sich mittels der Rekursionsschemata

$$\begin{aligned} add(x, 0) &= id(x), \\ add(x, y+1) &= S(P_3^3(x, y, add(x, y))) \end{aligned}$$

und

$$\begin{aligned} mult(x, 0) &= Z_1(x), \\ mult(x, y+1) &= add(P_1^3(x, y, mult(x, y)), P_3^3(x, y, mult(x, y))) \end{aligned}$$

definieren. Da die Identität, S, Z und die Projektionen bereits als primitiv-rekursiv nachgewiesen sind, ergibt sich damit die primitive Rekursivität von *add* und aus der dann die von *mult*.
Entsprechend unserer obigen Bemerkung ist klar, dass durch diese Schemata eindeutige Funktionen definiert sind. Durch einfaches „Nachrechnen“ überzeugt man sich davon, dass es sich wirklich um Addition und Multiplikation handelt, z.B. bedeutet die letzte Relation mit der üblichen Notation $add(x, y) = x + y$ und $mult(x, y) = x \cdot y$ nichts anderes als das bekannte Distributivgesetz

$$mult(x, y+1) = x \cdot (y+1) = x + x \cdot y = add(x, mult(x, y)).$$

d) Durch das Rekursionsschema

$$sum(0) = 0 \quad \text{und} \quad sum(y+1) = S(add(y, sum(y)))$$

wird die Funktion

$$sum(y) = \sum_{i=0}^{y} i = \frac{y(y+1)}{2}$$

definiert, wovon man sich leicht mittels vollständiger Induktion überzeugen kann.

Wir betrachten nun die folgende rekursive Definition der Fibonacci-Folge:

$$\begin{aligned} f(0) &= 1, \quad f(1) = 1, \\ f(y+2) &= f(y+1) + f(y). \end{aligned}$$

Für diese Rekursion ist nicht offensichtlich, dass sie durch das obige Rekursionsschema realisiert werden kann, da nicht nur auf den Wert $f(y)$ rekursiv zurückgegriffen wird. Die Rekursion für die Fibonacci-Folge lässt sich aber unter Verwendung von zwei Funktionen so umschreiben, dass jeweils nur die Kenntnis der Werte an der Stelle y erforderlich ist. Dies wird durch das Schema

$$\begin{aligned} f_1(0) &= 1, & f_2(1) &= 1, \\ f_1(y+1) &= f_2(y), & f_2(y+1) &= f_1(y) + f_2(y) \end{aligned}$$

geleistet. Hiervon ausgehend führen wir die folgende Verallgemeinerung des Rekursionsschemas, simultane Rekursion genannt, ein: Für n-stellige Funktionen g_i und die $(n+m+1)$-stelligen Funktionen h_i, $1 \leq i \leq m$, definieren wir simultan die $(n+1)$-stelligen Funktionen f_i, $1 \leq i \leq m$, durch

$$\begin{aligned} f_i(x_1, \ldots, x_n, 0) &= g_i(x_1, \ldots, x_n), \ 1 \leq i \leq m, \\ f_i(x_1, \ldots, x_n, y+1) &= h_i(x_1, \ldots, x_n, y, f_1(x_1, \ldots, x_n, y), \ldots, f_m(x_1, \ldots, x_n, y)). \end{aligned}$$

Wir wollen nun zeigen, dass die simultane Rekursion auch nur die Erzeugung primitiv-rekursiver Funktionen gestattet. Um die Notation nicht unnötig zu verkomplizieren werden wir die Betrachtungen nur für den Fall $n = 1$ und $m = 2$ durchführen.
Seien die Funktionen C, E, D_1 und D_2 mittels der Funktionen $\ominus$ und div aus Übungsaufgabe 10 von Abschnitt 1 durch

$$\begin{aligned} C(x_1, x_2) &= sum(x_1 + x_2) + x_2, \\ E(0) &= 0, \quad E(n+1) = E(n) + (n \ div \ sum(E(n)+1)), \\ D_1(n) &= E(n) + sum(E(n)) \ominus n, \\ D_2(n) &= E(n) \ominus D_1(n) \end{aligned}$$

definiert. Entsprechend der Konstruktion und Übungsaufgabe 10 von Abschnitt 1 sind alle diese Funktionen primitiv-rekursiv. Weiterhin rechnet man nach, dass die folgenden Bedingungen erfüllt sind: Für alle natürlichen Zahlen n, n_1 und n_2 gilt

$$C(D_1(n), D_2(n)) = n, \ D_1(C(n_1, n_2)) = n_1, \ D_2(C(n_1, n_2)) = n_2.$$

Zur Veranschaulichung betrachte man Abbildung 4.1 und prüfe nach, dass durch $E(n)$ die Nummer der Diagonalen in der n steht, durch $x_1 + x_2$ die Nummer der Diagonalen, in der sich die Spalte von x_1 und die Zeile von x_2 kreuzen, durch $C(x_1, x_2)$ das im Kreuzungspunkt der Spalte zu x_1 und der Zeile zu x_2 stehende Element, durch D_1 und D_2 die Projektionen von einem Element gegeben werden.
Dann definieren wir für die gegebenen Funktionen g_i und h_i, $1 \leq i \leq m$, die Funktionen g und h durch

$$\begin{aligned} g(x) &= C(g_1(x), g_2(x)), \\ h(x, y, z) &= C(h_1(x, y, D_1(z), D_2(z)), h_2(x, y, D_1(z), D_2(z))) \end{aligned}$$

x_2 \ x_1	0	1	2	3	4	...
0	0	1	3	6	10	...
1	2	4	7	11	...	
2	5	8	12	...		
3	9	13	...			
4	14	...				
...	...					

Abbildung 4.1:

die Funktion f durch das Rekursionsschema

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

und die Funktionen f_1 und f_2, die durch das simultane Rekursionsschema erzeugt werden sollen, durch

$$f_1(x,y) = D_1(f(x,y)) \text{ und } f_2(x,y) = D_2(f(x,y)).$$

Wegen

$$f_i(x,0) = D_i(f(x,0)) = D_i(g(x)) = D_i(C(g_1(x), g_2(x)) = g_i(x)$$

für $i \in \{1,2\}$, sind die Ausgangsbedingungen des verallgemeinerten Rekursionsschemas erfüllt, und analog zeigt man, dass die Rekursionsbedingungen befriedigt werden. Diese Konstruktion von f_1 und f_2 erfordert nur das ursprüngliche Rekursionsschema (für die Funktion f) und das Kompositionsschema, womit gezeigt ist, dass diese beiden Funktionen primitiv-rekursiv sind.

Aufgrund der eben gezeigten Äquivalenz von Rekursionsschema und simultanem Rekursionsschema werden wir zukünftig auch von der simultanen Rekursion Gebrauch machen, um zu zeigen, dass gewisse Funktionen primitiv-rekursiv sind.

Satz 4.3 *Eine Funktion f ist genau dann primitiv-rekursiv, wenn sie* **LOOP***-berechenbar ist.*

Beweis: Wir zeigen zuerst mittels Induktion über die Anzahl k der Operationen zur Erzeugung der primitiv-rekursiven Funktion f, dass f auch **LOOP**-berechenbar ist.
Sei $k = 0$. Dann muss die zu betrachtende Funktion f eine Basisfunktion sein. Die Tabelle in Abbildung 4.2 gibt zu jeder Basisfunktion f ein **LOOP**-Programm Π mit $\Phi_{\Pi,1} = f$. Damit ist der Induktionsanfang gesichert.
Wir führen nun den Induktionsschritt durch. Sei dazu f eine Funktion, die durch k-malige, $k \geq 1$, Anwendung der Operationen erzeugt wurde. Dann gibt es eine Operation, die als letzte angewendet wurde. Hiernach unterscheiden wir zwei Fälle, welche der beiden Operationen dies ist.
Fall 1. Kompositionsschema. Dann gilt

$$f(x_1, x_2, \ldots, x_n) = g(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)),$$

f	Π
Z	$x_1 := 0$
S	$x_1 := S(x_1)$
P	$x_1 := P(x_1)$
P_i^n	$x_1 := x_i$

Abbildung 4.2:

wobei die Funktionen $g, f_1, f_2, \ldots, f_m$ alle durch höchstens $(k-1)$-malige Anwendung der Operationen entstanden sind. Nach Induktionsannahme gibt es also Programme $\Pi, \Pi_1, \Pi_2, \ldots, \Pi_m$ derart, dass

$$\Phi_{\Pi,1} = g \text{ und } \Phi_{\Pi_i,1} = f_i \text{ für } 1 \leq i \leq m$$

gelten. Nun prüft man leicht nach, dass das Programm

$x_{n+1} := x_1; x_{n+2} := x_2; \ldots; x_{2n} := x_n;$
$\Pi_1; x_{2n+1} := x_1; x_1 := x_{n+1}; x_2 := x_{n+2}; \ldots; x_n := x_{2n};$
$\Pi_2; x_{2n+2} := x_1; x_1 := x_{n+1}; x_2 := x_{n+2}; \ldots; x_n := x_{2n};$
...
$\Pi_m; x_{2n+m} := x_1;$
$x_1 := x_{2n+1}; x_2 := x_{2n+2}; \ldots; x_m := x_{2n+m}; \Pi$

die Funktion f berechnet (die Setzungen $x_{n+i} := x_i$ stellen ein Abspeichern der Eingangswerte für die Variablen x_i dar; durch die Anweisungen $x_i := x_{n+i}$ wird jeweils gesichert, dass die Programme Π_j mit der Eingangsbelegung der x_i arbeiten, denn bei der Abarbeitung von Π_{j-1} kann die Belegung der x_i geändert worden sein; die Setzungen $x_{2n+j} := x_1$ speichern die Werte $f_j(x_1, x_2, \ldots, x_n)$, die durch die Programme Π_j bei der Variablen x_1 entsprechend der berechneten Funktion erhalten werden; mit diesen Werten wird dann aufgrund der Anweisungen $x_j := x_{2n+j}$ das Programm Π gestartet und damit der nach Kompositionsschema gewünschte Wert berechnet).
Fall 2. Rekursionsschema. Die Funktion f werde mittels Rekursionsschema aus den n- bzw. $(n+2)$-stelligen Funktionen g (an der Stelle $y = 0$) und h (für die eigentliche Rekursion) erzeugt. Da sich diese beiden Funktionen durch höchstens $(k-1)$-malige Anwendung der Schemata erzeugen lassen können, gibt es für sie Programme Π über den Variablen $x_1, x_2, \ldots x_n$ und Π' über den Variablen $x_1, x_2, \ldots, y, z$ mit $\Phi_{\Pi,1} = g$ und $\Phi_{\Pi',1} = h$ (wobei wir zur Vereinfachung nicht nur Variable der Form x_i, wie in Abschnitt 1.1.1 gefordert, verwenden). Wir betrachten das folgende Programm:

$y := 0; x_{n+1} := x_1; x_{n+2} := x_2; \ldots x_{2n} := x_n; \Pi; z := x_1;$
LOOP y' **BEGIN** $x_1 := x_{n+1}; \ldots x_n := x_{2n}; \Pi'; z := x_1; y := S(y)$ **END**;
$x_1 := z$

und zeigen, dass dadurch der Wert $f(x_1, x_2, \ldots, x_n, y')$ berechnet wird.
Erneut wird durch die Variablen x_{n+i} die Speicherung der Anfangsbelegung der Variablen x_i gewährleistet. Ist $y' = 0$, so werden nur die erste und dritte Zeile des Programms realisiert. Daher ergibt sich der Wert von Π bei der ersten Variablen, und weil Π die Funktion g berechnet, erhalten wir $g(x_1, x_2, \ldots, x_n)$, wie es das Rekursionsschema für

$f(x_1, x_2, \ldots, x_n, 0)$ erfordert. Ist dagegen $y' > 0$, so wird innerhalb der **LOOP**-Anweisung mit $z = f(x_1, x_2, \ldots, x_n, y)$ der Wert $f(x_1, x_2, \ldots, y+1)$ berechnet und die Variable y um Eins erhöht. Da dies insgesamt von $y = 0$ und $f(x_1, x_2, \ldots, x_n, 0) = g(x_1, x_2, \ldots, x_n)$ (aus der ersten Zeile) ausgehend, y'-mal zu erfolgen hat, wird tatsächlich $f(x_1, x_2, \ldots, x_n, y')$ als Ergebnis geliefert.
Damit ist der Induktionsbeweis vollständig.

Wir zeigen nun die umgekehrte Richtung. Wir gehen analog vor, werden vollständige Induktion über die Programmtiefe t benutzen und sogar zeigen, dass jede von einem **LOOP**-Programm Π berechnete Funktion $\Phi_{\Pi,j}$, $j \geq 1$ eine primitiv-rekursive Funktion ist.
Es sei $t = 1$. Dann bestehen die Programme aus den Wertzuweisungen. Wenn wir die im ersten Teil dieses Beweises gegebenen Tabelle von rechts nach links lesen, finden wir zu jeder derartigen Wertzuweisung die zugehörige primitiv-rekursive Funktion, die identisch mit der vom Programm berechneten Funktion ist. Damit ist der Induktionsanfang gesichert.
Sei nun Π ein Programm der Tiefe $t > 1$. Dann gilt $\Pi = \Pi_1; \Pi_2$ oder Π=**LOOP** y **BEGIN** Π' **END** für gewisse Programme Π_1, Π_2, Π' mit einer Tiefe $\leq t - 1$. Nach Induktionsannahme sind dann alle Funktionen $\Phi_{\Pi_1,j}, \Phi_{\Pi_2,j}, \Phi_{\Pi',j}$ primitiv-rekursiv.
Ist Π als Nacheinanderausführung von Π_1 und Π_2 gegeben, so ergeben sich für die von Π berechneten Funktionen die Beziehungen

$$\Phi_{\Pi,j}(x_1, \ldots x_n) = \Phi_{\Pi_2,j}(\Phi_{\Pi_1,1}(x_1, \ldots, x_n), \Phi_{\Pi_1,2}(x_1, \ldots, x_n), \ldots, \Phi_{\Pi_1,m}(x_1, \ldots, x_n)).$$

Damit entstehen die von Π berechneten Funktionen mittels des Kompositionsschems aus primitiv-rekursiven Funktionen und sind daher selbst primitiv-rekursiv.
Sei nun Π=**LOOP** y **BEGIN** Π' **END**, wobei wir ohne Beschränkung der Allgemeinheit annehmen, dass y nicht unter den Variablen $x_1, x_2, \ldots, x_n$ des Programms Π' vorkommt (siehe Übungsaufgabe 5). Dann werden die von Π berechneten Funktionen durch das folgende simultane Rekursionsschema bestimmt:

$$\begin{aligned} \Phi_{\Pi,j}(x_1, \ldots, x_n, 0) &= P^n_j(x_1, \ldots, x_n), \\ \Phi_{\Pi,j}(x_1, \ldots, x_n, y+1) &= \Phi_{\Pi',j}(\Phi_{\Pi,1}(x_1, \ldots, x_n, y), \ldots, \Phi_{\Pi,n}(x_1, \ldots, x_n, y)) \end{aligned}$$

(die erste Gleichung legt den Wert der Variablen vor Abarbeitung des Programms fest; um zum Wert für $y > 0$ zu kommen, wird das Programm Π' entsprechend der zweiten Gleichung stets wieder ausgeführt, wobei die im vorhergehenden Schritt erhaltenen Funktionswerte als Eingaben dienen (siehe Kompositionsschema); wie beim **LOOP**-Programm ist y-malige Nacheinanderausführung zur Gewinnung von $\Phi_{\Pi,j}(x_1, \ldots, x_n, y)$ notwendig.
Damit ist der Induktionsbeweis auch für diese Richtung geführt. □

Wir wollen nun eine weitere Operation zur Erzeugung von Funktionen einführen, die es uns gestattet, auch partielle Funktionen zu erhalten (mittels Kompositions- und Rekursionsschema erzeugte Funktionen sind offenbar total).

- *μ-Operator*: Für eine $(n+1)$-stellige Funktion h definieren wir die n-stellige Funktion f wie folgt. $f(x_1, x_2, \ldots, x_n) = z$ gilt genau dann, wenn die folgenden Bedingungen erfüllt sind:

- $h(x_1, x_2 \ldots, x_n, y)$ ist für alle $y \leq z$ definiert,
- $h(x_1, x_2, \ldots, x_n, y) \neq 0$ für $y < z$,
- $h(x_1, x_2, \ldots, x_n, z) = 0$.

Wir benutzen die Bezeichnungen

$$f(x_1, \ldots, x_n) = (\mu y)[h(x_1, \ldots, x_n, y) = 0] \quad \text{bzw.} \quad f = (\mu y)[h].$$

Intuitiv bedeutet dies, dass für die festen Parameter $x_1, x_2, \ldots, x_n$ der kleinste Wert von z bestimmt wird, für den $h(x_1, x_2, \ldots, x_n, z) = 0$ gilt (wobei bei nicht überall definierten Funktionen zusätzlich verlangt wird, dass für alle kleineren Werte y als z das Tupel $(x_1, x_2, \ldots x_n, y)$ im Definitionsbereich liegt, sonst ist f an dieser Stelle nicht definiert).

Beispiel 4.4 a) Es gilt

$$(\mu y)[add(x, y)] = \begin{cases} 0 & \text{für } x = 0 \\ nicht\ definiert & \text{sonst} \end{cases}$$

(für $x = 0$ ist wegen $0 + 0 = 0$ offenbar $z = 0$ der gesuchte minimale Wert; für $x > 0$ gilt auch $x + y > 0$ für alle y, und daher existiert kein z mit der dritten Eigenschaft aus der Definition des μ-Operators).

b) Es sei

$$h(x, y) = |9x^2 - 10xy + y^2|.$$

Durch Anwendung des μ-Operators auf h entsteht die Identität, d.h. f mit $f(x) = x$ für alle x.
Dies ist wie folgt leicht zu sehen. Für einen fixierten Wert von x ist $(\mu y)[h(x, y)]$ die kleinste natürliche Nullstelle des Polynoms $9x^2 - 10xy + y^2$ in der Unbestimmten y. Eine einfache Rechnung ergibt die Nullstellen x und $9x$. Somit gilt

$$f(x) = (\mu y)[h(x, y)] = x.$$

Wir wollen nun eine Erweiterung der primitiv-rekursiven Funktionen mittels μ-Operator entsprechend Definition 4.1 vornehmen. Da jedoch durch die Anwendung des μ-Operators Funktionen entstehen können, deren Definitionsbereiche echte Teilmengen von $\mathbf{N}_0^n$ sind, müssen wir zuerst Kompositions- und Rekursionsschema auf diesen Fall ausdehnen.
Beim Kompositionsschema ist $f(x_1, x_2, \ldots, x_n)$ genau dann definiert, wenn für $1 \leq i \leq n$ die Funktionen f_i auf dem Tupel $\underline{x} = (x_1, x_2, \ldots, x_n)$ und g auf $(f_1(\underline{x}), f_2(\underline{x}), \ldots, f_m(\underline{x}))$ definiert sind. In ähnlicher Weise kann das Rekursionsschema erweitert werden; die Details dazu überlassen wir dem Leser.

Definition 4.5 *Eine Funktion $f : \mathbf{N}_0^n \to \mathbf{N}_0$ heißt partiell-rekursiv, wenn sie mittels endlich oft wiederholter Anwendung von Kompositionsschema, Rekursionsschema und μ-Operator aus den Basisfunktionen erzeugt werden kann.*

Satz 4.6 *Eine Funktion ist genau dann partiell-rekursiv, wenn sie* **LOOP/WHILE**-*berechenbar ist.*

Beweis: Wir gehen wie beim Beweis von Satz 4.3 vor.
Daher reicht es in der ersten Richtung zusätzlich zu den dortigen Fakten zu zeigen, dass jede partiell-rekursive Funktion f, die durch Anwendung des μ-Operators auf h entsteht, **LOOP**/**WHILE**-berechenbar ist. Nach Induktionsannahme ist h **LOOP**/**WHILE**-berechenbar, also $h = \Phi_{\Pi,1}$ für ein Programm Π. Um den minimalen Wert z zu berechnen, berechnen wir der Reihe nach die Werte y' an den Stellen mit $0, 1, 2, \ldots$ für die Variable y und testen jeweils, ob der aktuelle Wert von y' von Null verschieden ist. Formal ergibt sich folgendes Programm für f:

$y := 0; x_{n+1} := x_1; \ldots x_{2n} := x_n; \Pi; y' := x_1;$
WHILE $y' \neq 0$ **BEGIN** $y := S(y); x_1 := x_{n+1}, \ldots, x_n := x_{2n}; \Pi; y' := x_1$ **END**;
$x_1 := y$

In der umgekehrten Richtung ist noch die **WHILE**-Anweisung zusätzlich zu betrachten. Sei also $\Pi'' =$ **WHILE** $x_k \neq 0$ **BEGIN** Π' **END**, wobei nach Induktionsannahme alle Funktionen $\Phi_{\Pi',j}$ partiell-rekursiv sind. Wir konstruieren zuerst die gleichen Funktionen $\Phi_{\Pi,j}$ wie bei der Umsetzung der **LOOP**-Anweisung im Beweis von Satz 4.3. Nach den dortigen Überlegungen gibt $\Phi_{\Pi,j}(x_1, x_2, \ldots x_n, y)$ den Wert der Variablen x_j nach y-maliger Hintereinanderausführung von Π' an. Die $\Phi_{\Pi,j}$ sind partiell-rekursive Funktionen, da sie durch Anwendung des simultanen Rekursionsschemas auf partiell-rekursive Funktionen entstanden sind. Wir betrachten nun die Funktion

$$w(x_1, \ldots, x_n) = (\mu y)[\Phi_{\Pi,k}(x_1, \ldots, x_n, y)],$$

die nach Definition die kleinste Zahl von Durchläufen von Π' liefert, um den Wert 0 bei der Variablen x_k zu erreichen. Entsprechend der Semantik der **WHILE**-Anweisung bricht diese genau nach $w(x_1, \ldots, x_n)$ Schritten ab. Folglich gilt

$$\Phi_{\Pi'',i}(x_1, \ldots, x_n) = \Phi_{\Pi,i}(x_1, \ldots, x_n, w(x_1, \ldots, x_n))$$

(da die rechten Seiten den Wert der Variablen x_i nach $w(x_1, \ldots, x_n)$ Hintereinanderausführungen von Π' und damit bei Abbruch der **WHILE**-Anweisung angeben). Entsprechend dieser Konstruktion ist damit jede von Π'' berechnete Funktion partiell-rekursiv. □

Wir bemerken, dass die Beweise der Sätze 4.3 und 4.6 eine enge Nachbarschaft zwischen dem Berechenbarkeitsbegriff auf der Basis von **LOOP**/**WHILE**-Programmen einerseits und partiell-rekursiven Funktionen andererseits ergibt, da die Wertzuweisungen den Basisfunktionen, die Hintereinanderausführung von Programmen dem Kompositionsschema, die **LOOP**-Anweisung dem Rekursionsschema und die **WHILE**-Anweisung dem μ-Operator im wesentlichen entsprechen.

Durch Kombination der Sätze 1.10 und 4.6 erhalten wir die folgende Aussage.

Folgerung 4.7 *Es gibt eine totale Funktion, die nicht partiell-rekursiv ist.* □

4.2 Registermaschinen

Wir wollen nun einen Berechenbarkeitsbegriff behandeln, der auf einer Modellierung der realen Rechner basiert.

Definition 4.8 *i) Eine Registermaschine besteht aus den Registern*

$$B,\ C_0, C_1, C_2, \ldots, C_n, \ldots$$

und einem Programm.
B heißt Befehlszähler, C_0 heißt Arbeitsregister oder Akkumulator, und jedes der Register C_i, $i \geq 1$, heißt Speicherregister.
Jedes Register enthält als Wert eine natürliche Zahl.
ii) Unter einer Konfiguration der Registermaschine verstehen wir das unendliche Tupel

$$(b, c_0, c_1, \ldots, c_n, \ldots),$$

wobei

- *das Register B die Zahl $b \in \mathbf{N}_0$ enthält,*
- *für $n \geq 0$ das Register C_n die Zahl $c_n \in \mathbf{N}_0$ enthält.*

iii) Das Programm ist eine endliche Folge von Befehlen. Durch die Anwendung eines Befehls wird die Konfiguration der Registermaschine geändert. Die folgende Liste gibt die zugelassenen Befehle und die von ihnen jeweils bewirkte Änderung der Konfiguration $(b, c_o, c_1, \ldots, c_n, \ldots)$ in die Konfiguration $(b', c'_0, c'_1, \ldots, c'_n, \ldots)$ an, wobei für die nicht angegebenen Komponenten $u' = u$ gilt:

Ein- und Ausgabebefehle:

`LOAD` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = c_i$
`ILOAD` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = c_{c_i}$
`CLOAD` i ,	$i \in \mathbf{N}_0$	$b' = b+1$	$c'_0 = i$
`STORE` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_i = c_0$
`ISTORE` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_{c_i} = c_0$

Arithmetische Befehle:

`ADD` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = c_0 + c_i$
`CADD` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = c_0 + i$
`SUB` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = \begin{cases} c_0 - c_i & \text{für } c_0 \geq c_i \\ 0 & \text{sonst} \end{cases}$
`CSUB` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = \begin{cases} c_0 - i & \text{für } c_0 \geq i \\ 0 & \text{sonst} \end{cases}$
`MULT` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = c_0 * c_i$
`CMULT` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = c_0 + i$
`DIV` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = \begin{cases} \lfloor c_0/c_i \rfloor & \text{für } c_i > 0 \\ \textit{nicht definiert} & \text{sonst} \end{cases}$
`CDIV` i ,	$i \in \mathbf{N}$	$b' = b+1$	$c'_0 = \lfloor c_0/i \rfloor$

Sprungbefehle:

`GOTO` i ,	$i \in \mathbf{N}$	$b' = i$
`IF` $c_0 = 0$ `GOTO` i ,	$i \in \mathbf{N}$	$b' = \begin{cases} i & \textit{falls } c_0 = 0 \\ b+1 & \textit{sonst} \end{cases}$

Stopbefehl:

`END`

Eine Registermaschine lässt sich entsprechend Abb. 4.3 veranschaulichen.

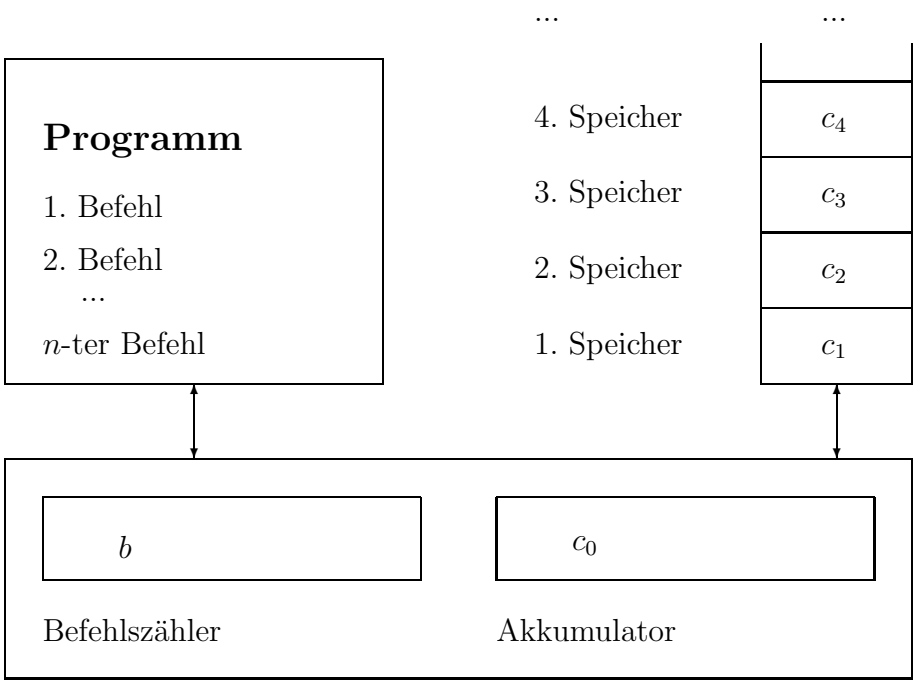


Abbildung 4.3: Registermaschine

Bei den Eingabebefehlen `LOAD` i bzw. `CLOAD` i wird der Wert des i-ten Registers bzw. die Zahl i in den Akkumulator geladen; bei `STORE` i wird der Wert des Akkumulators in das i-te Speicherregister eingetragen. Es sei j der Inhalt des i-ten Registers (d.h. $c_i = j$); dann werden durch die Befehle `ILOAD` i bzw. `ISTORE` i mit indirekter Adressierung der Inhalt des Registers j in den Akkumulator geladen bzw. der Inhalt des Akkumulators in das j-te Register gespeichert.
Bei den Befehlen `ADD` i, `SUB` i, `MULT` i und `DIV` i erfolgt eine Addition, Subtraktion, Multiplikation und Division des Wertes des Akkumulators mit dem Wert des i-ten Speicherregisters. Da die Operationen nicht aus dem Bereich der natürlichen Zahlen herausführen sollen, wird die Subtraktion nur dann wirklich ausgeführt, wenn der Subtrahend nicht kleiner als der Minuend ist und sonst 0 ausgegeben; analog erfolgt die Division nur ganzzahlig.
Die Befehle `CADD` i, `CSUB` i, `CMULT` i und `CDIV` i arbeiten analog, nur dass anstelle des Wertes des i-ten Registers die natürliche Zahl i benutzt wird. Dadurch werden auch arithmetische Operationen mit Konstanten möglich.

In all diesen Fällen wird der Wert des Befehlsregisters um 1 erhöht, d.h. der nächste Befehl des Programms wird abgearbeitet. Dies ist bei den Sprungbefehlen grundsätzlich anders. Bei `GOTO` i wird als nächster Befehl der i-te Befehl des Programms festgelegt, während bei der `IF`-Anweisung in Abhängigkeit von dem Erfülltsein der Bedingung $c_0 = 0$ der nächste Befehl der i-te bzw. der im Programm auf die `IF`-Anweisung folgende Befehl des Programms ist.
Der Befehl `END` ist ein Stopbefehl.

Definition 4.9 *Es sei M eine Registermaschine wie in Definition 4.8. Die von M induzierte Funktion $f_M : \mathbf{N}_0^n \longrightarrow \mathbf{N}_0$ ist wie folgt definiert: $f_M(x_1, x_2, \ldots x_n) = y$ gilt genau dann, wenn M ausgehend von der Konfiguration $(1, 0, x_1, x_2, \ldots, x_n, 0, 0, \ldots)$ die Konfiguration $(b, c_0, y, c_2, c_3, \ldots)$ für gewisse $b, c_0, c_2, c_3, \ldots$ erreicht und der b-te Befehl des Programm* `END` *ist.*

Entsprechend dieser Definition gehen wir davon aus, dass zu Beginn der Arbeit der Registermaschine die ersten n Speicherregister die Werte $x_1, x_2, \ldots, x_n$ und der Akkumulator und alle anderen Speicherregister den Wert 0 enthalten, die Abarbeitung des Programms mit dem ersten Befehl begonnen wird und bei Erreichen eines `END`-Befehls beendet wird und dann das Ergebnis y im ersten Speicherregister abgelegt ist (die Inhalte der anderen Register interessieren nicht).

Wir geben nun drei Beispiele.

Beispiel 4.10 Wir betrachten die Registermaschine M_1 mit dem Programm aus Abb. 4.4.

```
 1  CLOAD 1
 2  STORE 3
 3  LOAD 2
 4  IF c0 = 0 GOTO 12
 5  LOAD 3
 6  MULT 1
 7  STORE 3
 8  LOAD 2
 9  CSUB 1
10  STORE 2
11  GOTO 4
12  LOAD 3
13  STORE 1
14  END
```

Abbildung 4.4: Programm der Registermaschine aus Beispiel 4.10

Das Programm geht davon aus, dass im ersten und zweiten Register Werte stehen (Befehle 3 bzw. 6), so dass wir davon ausgehen, dass M_1 eine zweistellige Funktion $f_{M_1}(x, y)$ berechnet, wobei x und y zu Beginn im ersten bzw. zweiten Speicherregister stehen.

M_1 verhält sich wie folgt: Mittels der ersten zwei Befehle wird der Wert 1 in das dritte Register geschrieben. Die Befehle 4 – 11 bilden eine Schleife, die sooft durchlaufen wird, wie y angibt, denn bei jedem Durchlauf wird y um 1 verringert (Befehle 8 – 10). Ferner erfolgt bei jedem Durchlauf der Schleife eine Multiplikation des Inhalts des dritten Registers mit x (Befehle 5 – 7). Abschließend wird der Inhalt des dritten Registers in das erste umgespeichert, weil dort nach Definition das Ergebnis zu finden ist. Folglich induziert diese Registermaschine die Funktion

$$f_{M_1}(x, y) = 1 \cdot \underbrace{x \cdot x \cdot \ldots \cdot x}_{y \text{ mal}} = x^y.$$

M_1 berechnet also die Potenzfunktion.

Beispiel 4.11 Wir betrachten die Registermaschine M_2 mit dem Programm aus Abb. 4.5 und zu Beginn der Arbeit stehe nur im ersten Register ein möglicherweise von Null verschiedener Wert (in den anderen Registern steht also eine Null).

```
 1  LOAD 1
 2  IF c_0 = 0 GOTO 12
 3  LOAD 2
 4  CADD 1
 5  STORE 2
 6  ADD 3
 7  STORE 3
 8  LOAD 1
 9  CSUB 1
10  STORE 1
11  GOTO 1
12  LOAD 3
13  STORE 1
14  END
```

Abbildung 4.5: Programm der Registermaschine aus Beispiel 4.11

Steht im ersten Register eine Null, so werden wegen des zweiten Befehls die Befehle 12–14 abgearbeitet, durch die die Ausgabe 0 erzeugt wird. Anderenfalls erfolgt ein Durchlaufen der Befehle 3–5, durch die der Inhalt des Registers 2 um 1 erhöht wird, und anschließend der Befehle 6–7, durch die eine Addition der Werte der Register 2 (nach der Erhöhung) und 3 erfolgt, deren Resultat wieder in Register 3 abgelegt wird. Danach wird der Wert des Registers 1 um 1 erniedrigt. Diese Befehle werden solange durchgeführt, wie in Register 1 keine 0 steht, d.h. diese Schleife wird n mal durchlaufen, wenn n zu Beginn in Register 1 steht, da dieses Register bei jedem Durchlauf um 1 verringert wird. In Register 2 stehen während der Durchläufe nacheinander die Zahlen 1, 2, $\ldots n$, die in Register 3 aufaddiert werden. Da der Inhalt des dritten Registers das Resultat liefert, erhalten wir

$$f_{M_2}(n) = \sum_{i=1}^{n} i\,.$$

Beispiel 4.12 Wir gehen jetzt umgekehrt vor. Wir geben uns eine Funktion vor und wollen eine Registermaschine konstruieren, die diese Funktion induziert. Dazu betrachten wir die auf einem Feld (oder Vektor) $(x_1, x_2, \ldots, x_n)$ und seiner Länge n durch definierte Funktion

$$f(n, x_1, x_2, \ldots, x_n) = \begin{cases} \sum_{i=1}^{n} x_i & n \geq 1 \\ 0 & n = 0 \end{cases} \tag{4.1}$$

definierte Funktion.
Wir konstruieren eine Registermaschine, bei der zu Beginn n im ersten Register, x_i im $i+1$-ten Register und 0 in allen anderen Registern steht. Die Addition der Elemente des Feldes realisieren wir, indem wir zum Inhalt des zweiten Registers der Reihe nach die Werte $x_n, x_{n-1}, \ldots, x_2$ aus den Registern $n+1, n, \ldots, 3$ addieren. Hierbei greifen wir durch indirekte Adressierung immer auf das entsprechende i-te Register zu, indem wir das erste Register zuerst auf $n+1$ setzen und dann bei jeder Addition um 1 verringern. Die Addition erfolgt solange, wie die Registernummer (im ersten Register), auf die wir zugreifen wollen, mindestens 3 ist. Für diesen ganzen Prozess konstruieren wir eine Schleife.
Die beiden Sonderfälle, $n = 0$ und $n = 1$ (bei denen eigentlich keine Addition erfolgt) lassen sich einfach dadurch realisieren, dass der Inhalt des zweiten Registers (0 bei $n = 0$ und x_1 bei $n = 1$) direkt in das Ergebnisregister 1 umgespeichert wird. Diese beiden Fällen werden wir außerhalb der Schleife vorab erledigen.
Abschließend speichern wir das Ergebnis, das in jedem Fall im zweiten Register steht, in das erste Register um.
Formal ergibt dies das Programm aus Abb. 4.6.

1	`LOAD` 1	
2	`CSUB` 1	Befehle 1–3 testen, ob Sonderfall vorliegt
3	`IF` $c_0 = 0$ `GOTO` 15	
4	`LOAD` 1	
5	`CADD` 1	Befehle 4–5 setzen Registernummer für Addition auf $n+1$
6	`STORE` 1	
7	`ILOAD` 1	
8	`ADD` 2	Befehle 7–9 addieren $c_{i+1} = x_i$, $n \geq i \geq 2$, zu c_2
9	`STORE` 2	
10	`LOAD` 1	
11	`CSUB` 3	Befehle 10–12 testen, ob $c_3 = x_2$ schon addiert wurde
12	`IF` $c_0 = 0$ `GOTO` 15	
13	`CADD` 2	Verringern der Registernummer $i+1$ um 1
14	`GOTO` 6	
15	`LOAD` 2	
16	`STORE` 1	Befehle 15 und 16 speichern das Ergebnis in das erste Register
17	`END`	

Abbildung 4.6: Programm einer Registermaschine zur Berechnung von (4.1)

Wir wollen nun zeigen, dass Registermaschinen die Funktionen, die von **LOOP/WHILE**-Programmen induziert werden, berechnen können.

Satz 4.13 *Zu jedem* **LOOP/WHILE**-*Programm* Π *gibt es eine Registermaschine* M *derart, dass* $f_M = \Phi_{\Pi,1}$ *gilt.*

Beweis. Wir beweisen den Satz mittels Induktion über die Tiefe der **LOOP/WHILE**-Programme.
Induktionsanfang $k = 1$. Dann ist die gegebene Funktion eine der Wertzuweisungen.
Ist $x_i := 0$ die Anweisung, so liefert die Registermaschine mit dem Programm

```
1  CLOAD 0
2  STORE i
3  END
```

bereits das gewünschte Verhalten.
Ist $x_i := S(x_j)$, so leistet die Registermaschine mit dem Programm

```
1  LOAD j
2  CADD 1
3  STORE i
4  END
```

die gewünschte Simulation.
Für $x_i := P(x_j)$ und $x_i := x_j$ geben wir analoge Konstruktionen.
Induktionsschritt von $< k$ *auf* k. Wir haben zwei Fälle zu unterscheiden, nämlich ob als letzte Operation beim Aufbau der Programme eine Hintereinanderausführung oder eine **WHILE**-Schleife angewendet wurde (auf die Betrachtung der **LOOP**-Schleife können wir wegen der Bemerkung am Ende des Abschnitts 1.1.1 verzichten).
Hintereinanderausführung. Es sei $\Pi = \Pi_1; \Pi_2$, und für $i \in \{1, 2\}$ sei M_i die nach Induktionsvoraussetzung existierende Registermaschine mit $f_{M_i} = \Phi_{\Pi_i,1}$. Ferner habe M_i das Programm P_i, das aus r_i Befehlen bestehen möge. Weiterhin bezeichnen wir mit $p_{j,i}$ den j-ten Befehl von P_i. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass jedes der Programme P_i nur einen `END`-Befehl enthält, der am Ende des Programms steht. Wir modifizieren nun die Befehle des Programms P_2 dahingehend, dass wir alle Befehlsnummer j in einem Sprungbefehl oder einem bedingten Befehl durch $j + r_1 - 1$ ersetzen. Dadurch entstehe q_j aus $p_{j,2}$ für $1 \leq j \leq r_2$. Dann berechnet die Registermaschine mit dem Programm

$$\begin{array}{rl} 1 & p_{1,1} \\ 2 & p_{2,1} \\ \ldots & \ldots \\ r_1 - 1 & p_{r_1-1,1} \\ r_1 & q_1 \\ r_1 + 2 & q_2 \\ \ldots & \ldots \\ r_1 + r_2 - 1 & q_{r_2} \end{array}$$

die Funktion $\Phi_{\Pi,1}$.

WHILE-*Schleife* Sei $\Pi' = \textbf{WHILE}x_i \neq 0\textbf{BEGIN}\Pi\textbf{END}$, und seien $p_1, p_2, \ldots p_r$ die Befehle einer Registermaschine M mit $f_M = \Phi_{\Pi,1}$, wobei wiederum $p_r = \texttt{END}$ gelte. Für $1 \leq i \leq m$ sei q_i der Befehl, der aus p_i entsteht, indem jede in ihm vorkommende Befehlsnummern um 2 erhöht werden. Dann berechnet das Programm

1	$\texttt{LOAD}i$
2	$\texttt{IF}c_0 = 0\texttt{GOTO}r + 3$
3	q_1
4	q_2
...	...
$r+1$	q_{r-1}
$r+2$	$\texttt{GOTO}1$
$r+3$	$\texttt{END}$

die von Π' indizierte Funktion. □

Um zu zeigen, dass auch umgekehrt jede von einer Registermaschine induzierte Funktion **LOOP**/**WHILE**-berechenbar ist, geben wir nun eine Simulation von Registermaschinen durch Mehrband-TURING-Maschinen an. Dies ist hinreichend, weil wir aus dem ersten Teil der Vorlesung wissen, dass bis auf eine Kodierung von TURING-Maschinen induzierte Funktionen **LOOP**/**WHILE**-berechenbar sind.
Da die von TURING-Maschinen induzierten Funktionen Wörter in Wörter abbilden, während die von Registermaschinen berechneten Funktionen Tupel natürlicher Zahlen auf natürliche Zahlen abbilden, müssen wir natürliche Zahlen durch Wörter kodieren. Dies kann z.B. durch die Binär- oder Dezimaldarstellung der Zahlen erfolgen.
Für eine natürliche Zahl m bezeichne $dec(m)$ die Dezimaldarstellung von n.
Der folgende Satz besagt nun, dass es zu jeder Registermaschine M eine mehrbändige TURING-Maschine gibt, die im wesentlichen dasselbe wie M leistet, d.h. auf eine Eingabe der Dezimaldarstellungen von $m_1, m_2, \ldots m_n$ liefert die TURING-Maschine die Dezimaldarstellung von $f_M(m_1, m_2, \ldots, m_n)$, also die Dezimaldarstellung des Ergebnisses der Berechnung von M.

Satz 4.14 *Es seien M eine Registermaschine M mit $f_M : \mathbf{N}^n \to \mathbf{N}$. Dann gibt es eine 3-Band-TURING-Maschine M', deren Eingabealphabet außer den Ziffern 0,1,2,... 9 noch das Trennsymbol # und das Fehlersymbol F enthält und deren induzierte Funktion*

$$f_{M'}(w) = \begin{cases} dec(f_M(m_1, m_2, m_3, \ldots, m_n)) & w = dec(m_1)\#dec(m_2)\#dec(m_3)\ldots\#dec(m_n) \\ F & sonst \end{cases}$$

gilt (auf einer Eingabe, die einem Zahlentupel entspricht, verhält sich M' wie M und gibt bei allen anderen Eingaben eine Fehlermeldung).

Beweis. Wir geben hier keinen vollständigen formalen Beweis; wir geben nur die Idee des Beweises wider; der formale Beweis lässt sich unter Verwendung der Konstruktionen und Ideen aus den Beweisen der Lemmata 1.21 und 1.22 erbringen.
Wir konstruieren eine 3-Band-TURING-Maschine M', die schrittweise die Arbeit von M simuliert:

Auf dem ersten Arbeitsband speichern wir im Wesentlichen die Konfiguration der Registermaschine. Da diese ein unendliches Tupel ist, kann dies nicht direkt geschehen. Wir geben dort im wesentlichen die Folge der Nummern und Inhalte der Register an, die im Laufe der schon simulierten Schritte belegt worden sind. Formal steht auf dem ersten Band das Wort

$$\#\#0\#dec(c_0)\#\#dec(k_1)\#dec(c_{k_1})\#\#dec(k_2)\#dec(c_{k_2})\ldots\#\#dec(k_s)\#dec(c_{k_s})\#\#$$

(d.h. durch ## werden die verschiedenen Register voneinander getrennt; es wird stets die Nummer 0 bzw. k_i des Registers, $1 \leq i \leq s$, und der Inhalt c_o bzw. c_{k_i} angegeben die durch ein # getrennt sind; in allen anderen Registern steht eine 0; da stets nur endlich viele Register einer Registermaschine mit von Null verschiedenen Werten belegt werden, enthält das erste Band stets nur endlich viele Symbole).
Die gegebene Registermaschine M habe ein Programm mit r Befehlen. Für $1 \leq i \leq r$ konstruieren wir eine TURING-Maschine M_i, die eine Änderung des ersten Bandes entsprechend dem i-ten Befehl vornimmt. M_i arbeitet nur auf den drei Arbeitsbändern. Nach der eigentlichen Simulation des Befehls der Registermaschine werden das zweite und dritte Arbeitsband stets geleert und auf dem ersten Arbeitsband der Kopf zum Anfang bewegt (d.h. er steht über dem ersten #). $z_{i,0}$ sei der Anfangszustand und q_i der Stoppzustand von M_i. Um festzuhalten, welcher Befehl gerade simuliert wird, haben die Zustände von M' die Form (i, z), wobei $1 \leq i \leq r$ gilt und z ein Zustand von M_i ist. Für eine Anfangsphase werden noch weitere Zustände benötigt.
M' arbeitet nun wie folgt: Zuerst testet M', ob die Eingabe die Form $w_1\#w_2\#\ldots\#w_n$, wobei für $1 \leq i \leq n$ entweder $w_i = 0$ oder $w_i = x_iy_i$ mit $x_i \in \{1, 2, \ldots, 9\}$ und $y_i \in \{0, 1, \ldots, 9\}^*$ gilt, d.h. ob die Eingabe die Kodierung eines n-Tupels natürlicher Zahlen ist.
Ist dies nicht der Fall, so schreibt M' das Fehlersymbol F auf das Ausgabeband und stoppt.
Im anderen Fall schreibt M' auf das erste Arbeitsband die entsprechend modifizierte Eingabe

$$\#\#0\#0\#\#1\#dec(m_1)\#\#2\#dec(m_2)\ldots\#\#dec(n)\#dec(m_n)\#\#,$$

geht in den Zustand $(1, z_{1,0})$ über, und M_1 beginnt mit der Simulation des ersten Befehls. M_i, $1 \leq i \leq r$, beendet seine Simulation im Zustand (i, q_i), da während der Simulation nur die zu M_i gehörende zweite Komponente geändert wird. M' geht nun in den Zustand $(j, z_{j,0})$, wobei j der nach dem i-ten Befehl abzuarbeitende Befehl ist.

Wir geben nun die Arbeitsweise einiger TURING-Maschinen zu Befehlen der Registermaschine an, wobei wir nur die Änderung des Inhalts des ersten Arbeitsbandes angeben (es folgen dann noch die Löschungen auf den anderen Arbeitsbändern und die Kopfbewegung zum Anfang des ersten Arbeitsbandes).
a) Sei `LOAD t` der i-te Befehl. Dann schreibt M_i zuerst $dec(t)$ auf das zweite Arbeitsband und testet dann, ob im t-ten Register schon etwas gespeichert ist. Dazu läuft sie über das Wort auf dem ersten Arbeitsband und vergleicht stets ob nach zwei aufeinanderfolgenden # das Wort $dec(t)$ folgt. Hinter $dec(t)$ steht dann $\#dec(c_t)\#$ auf dem ersten Band. M_i leert das zweite Band und kopiert $dec(c_t)$ auf das zweite Band. Dann ersetzt M_i den Inhalt

des Akkumulators (zwischen dem dritten und vierten # auf dem Band) durch den Inhalt $dec(c_t)$ des t-ten Registers. Findet M_i keinen Eintrag im t-ten Register (ist bei Erreichen eines $*$ gegeben), so wird eine 0 in den Akkumulator geschrieben.
b) Im Fall der indirekten Adressierung `ILOAD` t schreiben wir zuerst den Inhalt des t-ten Registers in Dezimaldarstellung auf das zweite Band (dieser wird analog zu a) gesucht) und mit diesem Wert anstelle von $dec(t)$ verfahren wir wie bei a).
c) Ist der i-te Befehl `CLOAD` t, so schreiben wir gleich $dec(t)$ auf das zweite Band und kopieren dies in den Akkumulator (zwischen dem dritten und dem vierten Vorkommen von #).
d) Ist `STORE` t der i-te Befehl, so kopiert M_i den Inhalt $dec(c_0)$ des Akkumulators auf das dritte Band, schreibt $dec(t)$ auf das zweite Band, sucht die Stelle, wo der Inhalt des t-Registers steht, (dies steht hinter $\#\#dec(t)\#$) und ersetzt diesen durch den Inhalt des dritten Bandes. Wird $\#\#dec(t)\#$ nicht gefunden (d.h. es erfolgte noch kein Eintrag in dieses Register, so wird $dec(t)\#dec(c_0)\#\#$ an das Wort auf dem ersten Band angefügt.
e) Ist der i-te Befehl `ADD` t, so wird zuerst der Inhalt des t-ten Registers gesucht und auf das zweite Band geschrieben. Durch ein # getrennt schreibt M_i den Inhalt des Akkumulators dahinter und addiert beide Zahlen, wobei das Ergebnis auf das dritte Band geschrieben wird. Dieses Ergebnis schreiben wir in den Akkumulator.
f) Beim Befehl `GOTO` t ändern wir direkt den Zustand $(i, z_{i,0})$ zu $(t, z_{t,0})$.
g) Beim Befehl `IF` $c_o \neq 0$ `GOTO` t testet M_i, ob zwischen dem dritten und vierten # genau eine 0 steht. Ist dies der Fall geht M' in den Zustand $(t, z_{t,0})$, andererfalls in $(i+1, z_{i+1,0})$.
h) Beim Befehl `END` wird der Inhalt des ersten Registers (zwischen dem sechsten und siebenten #) auf das Ausgabeband geschrieben und gestoppt.
Die Konstruktionen für die anderen Fälle sind analog.

Aus diesen Erklärungen folgt sofort, dass M' bei Eingabe von der Kodierung eines Zahlentupels schrittweise die Befehle der Registermaschine simuliert und am Ende die Kodierung des Inhalts des ersten Registers, in dem das Ergebnis der Berechnung der Registermaschine steht, ausgibt. □

Fassen wir unsere Ergebnisse über Beziehungen zwischen Berechenbarkeitsbegriffen zusammen, so ergibt sich folgender Satz.

Satz 4.15 *Für eine Funktion f sind die folgenden Aussagen gleichwertig:*

- *f ist durch ein* **LOOP/WHILE**-*Programm berechenbar.*
- *f ist partiell-rekursiv.*
- *f ist durch eine Registermaschine berechnenbar.*
- *f ist bis auf Konvertierung der Zahlendarstellung durch eine* Turing-*Maschine berechenbar.*
- *f ist bis auf Konvertierung der Zahlendarstellung durch eine k-Band-*Turing-*Maschine, $k \geq 1$, berechenbar.* □

4.3 Komplexitätstheoretische Beziehungen

Im vorhergehenden Abschnitt haben wir nur gezeigt, dass eine gegenseitige Simulation von TURING-Maschinen und Registermaschinen möglich ist. In diesem Abschnitt wollen eine genauere Analyse durchführen, indem wir noch zusätzlich die Komplexität der Berechnungen betrachten. Daher definieren wir zuerst die Komplexität bei Registermaschinen. Hierbei nehmen wir eine Beschränkung der (indirekten) arithmetischen Befehle auf Addition und Subtraktion vor, um etwas realistischer zu sein, da die Komplexität der Berechnung einer Multiplikation bzw. Division erheblich höher ist als die von Addition und Subtraktion. Werden Multiplikation und Division dagegen auf Addition und Subtraktion zurückgeführt, entstehen realistischere Maße.

Definition 4.16 *Es sei M eine Registermaschine. Die (uniforme) Komplexität von M auf $(x_1, x_2, \ldots, x_n)$, bezeichnet durch $t_M(x_1, x_2, \ldots, x_n)$ ist als Anzahl der Schritte definiert, die M bis zum Erreichen des* `END`*-Befehls auf der Eingabe $(x_1, x_2, \ldots, x_n)$ ausführt.*

Definition 4.17 *Es sei M eine Registermaschine. Wir sagen, dass M $t(n)$-zeitbeschränkt ist, wenn*

$$t_M(x_1, x_2, \ldots, x_n) \leq t(n)$$

für jede Eingabe $(x_1, x_2, \ldots, x_n)$ gilt.

Satz 4.18 *i) Jede $t(n)$-zeitbeschränkte Registermaschine kann durch eine $O((t(n))^3)$-zeitbeschränkten 3-Band-*TURING*-Maschine simuliert werden.*
ii) Jede $t(n)$-zeitbeschränkte Registermaschine kann durch eine $O((t(n))^6)$-zeitbeschränkten TURING*-Maschine simuliert werden.*

Beweis. i) Wir analysieren die im Beweis von Satz 4.14 gegebene TURING-Maschine. Da in jedem Schritt der Registermaschine ein Registerinhalt hinsichtlich der Länge höchstens um 1 erhöht wird (z.B. bei einer Addition einer Zahl mit sich selbst)[1]. Daher ist die Länge der Registerinhalte durch $O(t(n))$ beschränkt. Außerdem können in $t(n)$ höchstens $O(t(n))$ Register gefüllt werden. Folglich ist jeder Teil $dec(k)\#dec(c_k)$ in der Länge durch $O(t(n))$ beschränkt. Damit steht auf dem ersten Band höchstens ein Wort der Länge $O(t(n)^2)$. Folglich erfordert die Arbeit einer jeder Maschine, die einen Befehl simuliert, höchstens $O(t(N)^2)$ Schritte (Lesen der Konfiguration auf Band 1, umspeichern, vergleichen etc. von Wörtern der Länge $\leq O(t(n))$). Da insgesamt $t(n)$ mal Befehle simuliert werden, benötigt die TURING-Maschine insgesamt höchstens $O(t(n)^3)$ Schritte.
ii) Die Aussage ergibt sich aus dem ersten Teil unter Verwendung von Satz 3.7 □

Wir wollen nun zeigen, dass auch die Simulation in der umgekehrten Richtung polynomiale Berechenbarkeit erhalten bleibt. Wir machen dabei aber darauf aufmerksam, dass wir damit sogar eine direkte Simulation von TURING-Maschinen durch Registermaschinen erhalten (im vorhergehenden Abschnitt haben wir einen Umweg über **LOOP**/**WHILE**-Berechenbarkeit gewählt.
Wir benötigen das folgende Lemma.

[1] Man beachte, dass diese Aussage bei Verwendung der Multiplikation nicht mehr gilt. Die Multiplikation kann zur Verdopplung der Länge führen.

Lemma 4.19 *Es sei $k \geq 2$. Jede $t(n)$-zeitbeschränkte k-Band-*Turing*-Maschine kann durch eine $O(t(n))$-zeitbeschränkte k-Band-*Turing*-Maschine simuliert werden, die keine Zelle links der Eingabe betritt/verändert.*

Beweis. Wir geben keinen vollständigen formalen Beweis sondern nur die Idee der Konstruktion. Es sei $M = (k, X, Z, z_0, Q, \delta)$ eine k-Band-Turing-Maschine, bei der wir die Zellen eines jeden Bandes entsprechend den natürlichen Zahlen durchnummerieren. Wir konstruieren eine k-Band-Turing-Maschine, bei der wir ebenfalls eine Nummerierung der Zellen eines jeden Bandes vornehmen. In jede Zelle des Bandes k mit der Nummer i, wobei $i > 0$ ist, speichern wir jetzt aber ein Paar von Buchstaben aus $X \cup \{*\}$ ab. Das Paar (a, b) in der Zelle mit Nummer i des Bandes k bei M' beschreibt die Situation, dass bei M in der Zelle des Bandes k mit der Nummer i das Symbol a und in der Zelle mit der Nummer $-i$ das Element b steht. Daher muss sich M' nie auf Zellen mit nicht positiven Nummern begeben. Die Maschine M' muss sich aber im Zustand merken, ob es sich über dem negativen oder positiven Bereich von M befindet. Dies kann dadurch erreicht werden, dass die Zustände die Form $(u_1, u_2, \ldots u_k, q)$ mit $u_i \in \{+, -\}$, $1 \leq i \leq k$, und $Q \in Z$ haben. Dabei gibt u_i, $1 \leq i \leq k$, an, ob sich der Kopf des Bandes i über dem positiven oder negativen Teil des Bandes befindet. Da die Zelle mit der Nummer 0 nur ein Element aus $X \cup \{*\}$ (und kein Paar) enthält, ist es einfach den Wechsel von $+$ zu $-$ und umgekehrt zu vollziehen.
Offensichtlich braucht M' genau die gleiche Anzahl von Schritten wie M. □

Satz 4.20 *Jede $t(n)$-zeitbeschränkte k-Band-*Turing*-Maschine mit $k \geq$ kann durch eine $O(t(n))$-zeitbeschränkte Registermaschine simuliert werden.*

Beweis. Es sei $M = (k, X, Z, z_0, Q, \delta)$ eine k-Band-Turing-Maschine. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass $X = \{1, 2, \ldots, n\}$ und $Z = \{0, 1, 2, \ldots, m\}$ gelten, wobei $z_0 = 0$ gilt. Außerdem wollen wir anstelle von $*$ die Zahl 0 verwenden. Wir nehmen nach Lemma 4.19 an, dass sich die Köpfe nur über Zellen mit den Nummern aus $\mathbf{N}_0$ befinden.
Wir konstruieren nun eine Registermaschine, die wie folgt aufgebaut ist und arbeitet: Im Register i, $1 \leq i \leq k$, steht die Position, in der sich der Kopf des Bandes i befindet, im Register $k + 1$ steht die Position, in der sich der Kopf des Eingabebandes befindet, im Register $k+2$ steht der Zustand der Turing-Maschine, im Register $C+(k+1)p$ steht der Inhalt der Zelle mit Nummer p vom Eingabeband, im Register $C + kp + j$ steht der Inhalt der Zelle mit Nummer p vom Band j, wobei C eine hinreichend große Konstante ist, so dass die notwendigen Zwischenrechnungen in den Registern $k+3$ – $C-1$ durchgeführt werden können. Wir illustrieren dies durch ein Beispiel mit $k = 2$. Wenn sich M im Zustand 5 befindet, auf dem Eingabeband 1232, auf dem ersten Arbeitsband 221 und auf dem zweiten Arbeitsband 1133 stehen (der erste Buchstabe stehe stets in der Zelle mit der Nummer 0), und der Kopf des Eingabebandes, ersten und zweiten Arbeitsbandes auf den Positionen 2, 3, und 1 stehen, so ergibt sich die folgende Konfiguration der Registermaschine

$$(b, c_0, 3, 1, 2, 5, c_5, c_6, \ldots, c_{C-1}, 1, 2, 1, 2, 2, 1, 3, 2, 3, 2, 0, 3, 0, 0, 0, \ldots).$$

Das Programm der Registermaschine besteht aus „kleinen“ Programmen, die nacheinander abgearbeitet werden. Ein dieser Programme überträgt die Eingabe der Turing-Maschine in die entsprechenden Register. Die restlichen Programme simulieren einen Arbeitsschritt der **Turing-Maschine**, d.h. sie berechnen jeweils die Funktionen δ, wobei sie die notwendigen Eingaben aus den entsprechenden Registern holen, genauer den Zustand aus Register $k+2$, das Symbol des Eingabebandes aus Register $C + c_{k+1}(k+1)$, das Symbol des j-ten Arbeitsbandes aus Register $C + c_j(k+1) + j$, $1 \leq j \leq k$, und die ermittelten Werte (einschließlich des neuen Zustandes und der neuen Kopfpositionen) wieder entsprechend ablegen. Die Durchführung einer derartigen Berechnung von δ geschieht in konstanter Zeit (man beachte, dass k fest gewählt ist). Damit erfordert die Simulation nur $O(t(n))$ Schritte der Registermaschine. □

Entsprechend den vorstehenden Sätzen ist es hinsichtlich der Zugehörigkeit zur Klasse **P** nicht wichtig, ob wir mitTuring-Maschinen oder Registermaschinen arbeiten.

Kapitel 5

Ergänzung II: Abschluss- und Entscheidbarkeitseigenschaften formaler Sprachen

5.1 Abschlusseigenschaften formaler Sprachen

Im ersten Teil der Vorlesung haben wir das Verhalten von regulären Sprachen hinsichtlich der Operationen Vereinigung, Durchschnitt, Komplement, Produkt und Kleene-Abschluss untersucht. Daraus haben wir eine Charakterisierung der regulären sprachen (durch reguläre Ausdrücke gewonnen.
In diesem Abschnitt wollen wir untersuchen, wie sich die Sprachen der anderen Familien der Chomsky-Hierarchie gegenüber diesen Operationen verhalten. Weiterhin werden wir weitere Operationen einführen, mittels derer uns eine weitere Charakterisierung der regulären Sprachen gelingen wird.
Wir geben zuerst die hierfür grundlegende Definition.

Definition 5.1 *Es seien $\mathcal{L}$ eine Menge von Sprachen und τ eine n-stellige Operation, die über Sprachen definiert ist. Dann heißt $\mathcal{L}$ abgeschlossen unter τ, wenn für beliebige Sprachen $L_1, L_2, \ldots, L_n \in \mathcal{L}$ auch*

$$\tau(L_1, L_2, \ldots, L_n) \in \mathcal{L}$$

gilt.

Wir untersuchen nun, ob die Sprachmengen der Chomsky-Hierarchie unter den üblichen mengentheoretischen Operationen Vereinigung, Durchschnitt und Komplement und den algebraischen Operationen Produkt und Kleene-Abschluss abgeschlossen sind. Dabei werden wir zwar der Vollständigkeit halber auch die Ergebnisse für die Menge der regulären Mengen angeben (und damit jeweils Satz 2.63 partiell wiederholen), aber den Beweis dafür nicht erneut geben.

Lemma 5.2 *$\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen unter der (üblichen) Vereinigung von Sprachen.*

Beweis. Wir zeigen die Aussage zuerst für $\mathcal{L}(CF)$. Seien L_1 und L_2 zwei kontextfreie Sprachen über dem Alphabet T. Wir haben zu zeigen, dass auch $L_1 \cup L_2$ eine kontextfreie Sprache (über T) ist.
Dazu seien

$$G_1 = (N_1, T_1, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T_2, P_2, S_2)$$

zwei kontextfreie Grammatiken mit

$$L(G_1) = L_1 \quad \text{und} \quad L(G_2) = L_2.$$

Offenbar können wir ohne Beschränkung der Allgemeinheit annehmen, dass

$$T_1 = T_2 = T \quad \text{und} \quad N_1 \cap N_2 = \emptyset$$

gelten (notfalls sind die Nichtterminale umzubenennen). Ferner sei S ein Symbol, das nicht in $N_1 \cup N_2 \cup T$ liegt. Wir betrachten nun die kontextfreie Grammatik

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S).$$

Offenbar hat jede Ableitung in G die Form

$$(*) \qquad S \Longrightarrow S_i \Longrightarrow^* w,$$

wobei $i = 1$ oder $i = 2$ gilt und $S_i \Longrightarrow^* w$ eine Ableitung in G_i ist (da wegen $N_1 \cap N_2 = \emptyset$ keine Symbole aus N_j, $j \neq i$ entstehen können und damit keine Regeln aus P_j anwendbar sind). Folglich gilt $w \in L(G_i)$. Hieraus folgt sofort

$$L(G) \subseteq L(G_1) \cup L(G_2) = L_1 \cup L_2.$$

Man sieht aber auch aus $(*)$ sofort, dass jedes Element aus $L(G_i)$, $i \in \{1, 2\}$, erzeugt werden kann, womit auch die umgekehrte Inklusion

$$L(G) \supseteq L(G_1) \cup L(G_2) = L_1 \cup L_2$$

gezeigt ist.
Da die bei der Konstruktion von G hinzugenommenen Regeln bei kontextabhängigen oder allgemeinen Regelgrammatiken zugelassen sind, kann mit dem gleichen Beweis auch gezeigt werden, dass $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ gegenüber Vereinigung abgeschlossen sind. □

Lemma 5.3 *$\mathcal{L}(REG)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen unter Durchschnitt, und $\mathcal{L}(CF)$ ist gegenüber Durchschnitt nicht abgeschlossen.*

Beweis. i) $\mathcal{L}(RE)$. Es seien $L_1 \in \mathcal{L}(RE)$ und $L_2 \in \mathcal{L}(RE)$ gegeben. Nach Satz 2.43 und 2.32 gibt es TURING-Maschinen

$$M_1 = (X, Z_1, z_{01}, Q_1, \delta_1, Q_1) \quad \text{und} \quad M_2 = (X, Z_2, z_{02}, Q_2, \delta_2, Q_2)$$

mit

$$T(M_1) = L_1 \quad \text{und} \quad T(M_2) = L_2,$$

wobei wir wieder annehmen können, dass $Z_1 \cap Z_2 = \emptyset$ gilt. Wir betrachten nun die TURING-Maschine M, die wie folgt arbeitet (die formale Beschreibung von M bleibt dem Leser überlassen). Zuerst ersetzt sie jeden Buchstaben x auf dem Band durch das Paar (x, x). Dann arbeitet sie wie M_1, wobei sie stets nur den Inhalt der ersten Komponente entsprechend δ_1 verändert und sich somit in der zweiten Komponente das ursprünglich auf dem Band befindliche Wort speichert (werden Leerzeichen gelesen, so sind wie ein Paar $(*, *)$ zu behandeln). Wird ein Zustand aus Q_1 erreicht, so ersetzt die Maschine alle Paare auf dem Band durch ihre zweite Komponente, womit das Ausgangswort wieder auf dem Band steht. Dann bewegt sie den Kopf zum ersten Buchstaben, geht in den Zustand z_{02} und beginnt nun wie M_2 zu arbeiten. Die Maschine stoppt, wenn sie einen Zustand aus Q_2 erreicht.
Entsprechend dieser Arbeitsweise wird mittels der ersten Komponente getestet, ob das Wort von M_1 akzeptiert wird; ist dies der Fall wird auch noch getestet, ob es in $T(M_2)$ liegt. Folglich erreicht M genau dann einen Stoppzustand, wenn das Wort sowohl in $T(M_1)$ als auch $T(M_2)$ liegt. Wenn wir alle Stoppzustände zur Akzeptanz verwenden, erhalten wir

$$T(M) = T(M_1) \cap T(M_2) = L_1 \cap L_2,$$

womit die Behauptung aus Satz 2.43 folgt.

ii) $\mathcal{L}(CS)$. Der Beweis kann genauso wie unter i) geführt werden, wobei wir von linear beschränkten Automaten ausgehen und Satz 2.45 benutzen.

iii) $\mathcal{L}(CF)$. Um diese Aussage zu beweisen, reicht es, zwei kontextfreie Sprachen L_1 und L_2 anzugeben, deren Durchschnitt keine kontextfreie Sprache ist. Dazu betrachten wir

$$L_1 = \{a^n b^n c^m : n \geq 1, m \geq 1\} \quad \text{und} \quad L_2 = \{a^m b^n c^n : n \geq 1, m \geq 1\}.$$

Es ist leicht zu zeigen, dass diese beiden Sprachen kontextfrei sind (wir überlassen die Konstruktion zugehöriger Grammatiken dem Leser). Dann gilt

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geq 1\},$$

und dies ist nach Lemma 2.28 keine kontextfreie Sprache. □

Lemma 5.4 *i) $\mathcal{L}(REG)$ und $\mathcal{L}(CS)$ sind abgeschlossen bezüglich Komplement.*
ii) $\mathcal{L}(CF)$ und $\mathcal{L}(RE)$ sind nicht abgeschlossen unter Komplement.

Beweis. Wir beweisen die Aussage nur für $\mathcal{L}(RE)$, $\mathcal{L}(REG)$ und $\mathcal{L}(CF)$, da der Beweis für $\mathcal{L}(CS)$ mit den bisher zur Verfügung stehenden Mitteln zu aufwändig ist (ein Beweis ist z.B. in [24] zu finden).

$\mathcal{L}(RE)$. Wäre $\mathcal{L}(RE)$ unter Komplementbildung abgeschlossen, so wäre wegen Satz 2.35 jede rekursiv-aufzählbare Sprache rekursiv. Dies steht aber im Widerspruch zu Satz 2.37.

$\mathcal{L}(CF)$. Wir nehmen an, dass $\mathcal{L}(CF)$ gegenüber Komplement abgeschlossen ist. Es seien nun zwei beliebige kontextfreie Sprachen L_1 und L_2 gegeben. Wir setzen

$$X = alph(L_1) \cup alph(L_2), \quad X_1 = X \setminus alph(L_1), \quad X_2 = X \setminus alph(L_2).$$

Ferner seien R_1 und R_2 die Mengen aller Wörter über X, die mindestens einen Buchstaben aus X_1 bzw. X_2 enthalten. Wir zeigen nun, dass diese beiden Sprachen regulär und damit auch kontextfrei sind. Hierfür reicht es festzustellen, dass für $i \in \{1,2\}$ die reguläre Grammatik

$$G_i = (\{S,A\}, X, \bigcup_{a \in alph(L_i)} \{S \to aS\} \cup \bigcup_{b \in X_i} \{S \to bA, S \to b\} \cup \bigcup_{x \in X} \{A \to xA, A \to x\}, S)$$

die Sprache R_i erzeugt. Dies folgt daher, dass ein Übergang zum Nichtterminal A oder zum direkten Terminieren aus S nur möglich sind, wenn mindestens ein Element aus X_i erzeugt wurde.
Aufgrund einfacher mengentheoretischer Fakten gilt

$$X^* \setminus L_i = ((alph(L_i))^* \setminus L_i) \cup R_i = \overline{L_i} \cup R_i$$

für $i \in \{1,2\}$. Nach unserer Annahme und Lemma 5.2 sind damit $X^* \setminus L_1$ und $X^* \setminus L_2$ kontextfreie Sprachen. Damit ist auch

$$R = (X^* \setminus L_1) \cup (X^* \setminus L_2)$$

eine kontextfreie Sprache. Wegen unserer Annahme und

$$L_1 \cap L_2 = X^* \setminus ((X^* \setminus L_1) \cup (X^* \setminus L_2)) = (alph(R))^* \setminus R$$

ist damit $L_1 \cap L_2$ als kontextfrei nachgewiesen. Dies steht im Widerspruch zu Lemma 5.3. Folglich muss die gemachte Annahme falsch sein, womit die Aussage des Lemmas gezeigt ist. □

Lemma 5.5 *$\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen unter Produkt.*

Beweis. $\mathcal{L}(CF)$. Erneut gehen wir von zwei kontextfreien Grammatiken

$$G_1 = (N_1, T, P_1, S_1) \quad \text{und} \quad G_2 = (N_2, T, P_2, S_2)$$

mit $N_1 \cap N_2 = \emptyset$ aus und zeigen, dass die Grammatik

$$G = (N_1 \cup N_2 \cup \{S\}, T, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$$

die Sprache

$$L(G) = L(G_1) \cdot L(G_2)$$

erzeugt. Hierzu reicht zu bemerken, dass bis auf die Reihenfolge der Anwendung der Regeln jede Ableitung in G die Form

$$S \Longrightarrow S_1 S_2 \Longrightarrow^* w_1 S_2 \Longrightarrow^* w_1 w_2$$

hat, wobei für $i \in \{1,2\}$ die Ableitung $S_i \Longrightarrow^* w_i$ auch eine Ableitung in G_i ist, d.h. nur mittels Regeln aus P_i entsteht.
$\mathcal{L}(CS)$ und $\mathcal{L}(RE)$. Wir können den Beweis wie bei $\mathcal{L}(CF)$ führen, wenn wir voraussetzen, dass die Grammatiken in der Normalform aus Satz 2.16 sind (diese Voraussetzung sichert, dass sich die Ableitungen in G_1 und G_2 nicht über den Kontext beeinflussen können. Außerdem haben wir das Leerwort, falls es in $L(G_1)$ und/oder $L(G_2)$ liegt, gesondert zu behandeln (dies erfordert nur die Vereinigung einiger Sprachen). □

Lemma 5.6 *$\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind abgeschlossen gegenüber der Bildung des (positiven) Kleene-Abschlusses.*

Beweis. Wir beweisen die Aussage zuerst für den positiven KLEENE-Abschluss.
$\mathcal{L}(CF)$. Es sei die kontextfreie Sprache L von der kontextfreien Grammatik $G = (N, T, P, S)$ erzeugt. Wir setzen dann

$$G' = (N \cup \{S'\}, T, P \cup \{S' \to SS', S' \to S\}, S')$$

(wobei S' wieder ein zusätzliches Symbol ist). Bis auf die Reihenfolge der Anwendung der Regeln hat jede Ableitung in G' die Form

$$\begin{aligned} S' &\Longrightarrow SS' \Longrightarrow^* w_1S' \Longrightarrow w_1SS' \Longrightarrow^* w_1w_2S' \Longrightarrow w_1w_2SS' \Longrightarrow ... \\ &\Longrightarrow w_1w_2 \dots w_{n-1}S' \Longrightarrow w_1w_2 \dots w_{n-1}S \Longrightarrow^* w_1w_2 \dots w_{n-1}w_n, \end{aligned}$$

wobei die Ableitungen $S \Longrightarrow^* w_i$ für $1 \leq i \leq n$ stets nur Regeln aus P benutzen. Daher gilt $w_i \in L(G) = L$ für $1 \leq i \leq n$ und $w_1w_2 \dots w_n \in L^n$ ist bewiesen.
Umgekehrt ist klar, dass auf diese Weise jedes aus L^n, $n \geq 1$, erzeugt werden kann. Folglich gilt

$$L(G') = \bigcup_{n \geq 1} L^n = L^+,$$

womit L^+ als kontextfrei nachgewiesen ist.
$\mathcal{L}(CS)$ und $\mathcal{L}(RE)$. Wir gehen erneut wie bei $\mathcal{L}(CF)$ vor, benutzen die Normalform nach Satz 2.16 und ändern die zu P hinzugefügten Regeln, um zu sichern, dass sich die Kontexte nicht gegenseitig beeinflussen, d.h. dass die Ableitung des Wortes w_i erst beginnt, wenn die von w_{i-1} hierdurch nicht mehr beeinflusst werden kann. Dies geschieht durch Hinzufügen der Regeln

$$\begin{aligned} &S' \to S,\ S' \to SS'', \\ &xS'' \to xSS'',\ xS'' \to xS \quad \text{für } x \in T. \end{aligned}$$

Die Details des Beweises überlassen wir dem Leser.

Wir geben nun die Modifikationen für den KLEENE-$*$. Gilt $\lambda \in L$, so können wir wegen der dann gegebenen Gültigkeit von $L^* = L^+$ wie oben vorgehen. Ist $\lambda \notin L$, so haben wir nur noch das Leerwort zusätzlich zu L^+ zu erzeugen. Ist $G = (N, T, P, S)$ eine Grammatik mit $L(G) = L^+$, so ist dies einfach dadurch zu realisieren, dass wir zu N ein weiteres Nichtterminal S' und zu P die Regeln $S' \to \lambda$ und $S' \to S$ hinzufügen und S' als Axiom nehmen. □

Wir kommen nun zu Operationen, die aus der Algebra bekannt sind und dort eine wichtige Rolle spielen.

Definition 5.7 *Es seien X und Y zwei Alphabete. Ein Homomorphismus h von X^* in Y^* ist eine eindeutige Abbildung von X^* in Y^*, bei der $h(w_1w_2) = h(w_1)h(w_2)$ für beliebige Wörter w_1 und w_2 aus X^* gilt.*
Ein Homomorphismus heißt nichtlöschend, wenn für alle Wörter $w \neq \lambda$ auch $h(w) \neq \lambda$ gilt.

Wegen $h(w) = h(w \cdot \lambda) = h(w)h(\lambda)$ gilt für jeden Homomorphismus $h(\lambda) = \lambda$.
Es sei $w = a_1 a_2 \dots a_n \in X^+$ mit $a_i \in X$. Dann gilt $h(w) = h(a_1)h(a_2)\dots h(a_n)$. Daher ist $h(w)$ berechenbar, wenn die Werte $h(a_i)$, $1 \leq i \leq n$, bekannt sind. Folglich reicht es aus die Wörter $h(a)$ für $a \in X$ zu kennen, um für jedes Wort $w \in X^*$ das Bild $h(w)$ zu bestimmen.
Für einen nichtlöschenden Homomorphismus ist dann $h(a) \neq \lambda$ für alle $a \in X$.

Beispiel 5.8 Die Homomorphismen h_1 und h_2 von $\{a,b\}^*$ in $\{a,b,c\}^*$ seien durch

$$h_1(a) = ab,\ h_1(b) = bb \quad \text{und} \quad h_2(a) = ac,\ h_2(b) = \lambda$$

gegeben. Dann ergeben sich

$$h_1(abba) = abbbbbab,\ h_1(bab) = bbabbb,\ h_2(abba) = acac),\ h_2(bab) = ac\,.$$

Wir erweitern nun den Homomorphismusbegriff auf Sprachen durch die folgenden Festlegungen.

Definition 5.9 *Es seien X und Y zwei Alphabete, $L \subseteq X^*$ und $L' \subseteq Y^*$ zwei Sprachen und h ein Homomorphismus von X^* in Y^*. Dann setzen wir*

$$h(L) = \{h(w) \mid w \in L\} \quad \textit{und} \quad h^{-1}(L') = \{w \mid w \in X^*,\ h(w) \in L'\}.$$

Beispiel 5.10 *(Fortsetzung von Beispiel 5.8)* Wir betrachten die Homomorphismen h_1 und h_2 aus Beispiel 5.8. Dann ergeben sich

$$\begin{aligned}
h_1(\{a\}^* \cup \{b\}^*) &= \{(ab)^n \mid n \geq 0\} \cup \{b^{2n} \mid n \geq 0\},\\
h_2(\{a\}^* \cup \{b\}^*) &= \{(ac)^n \mid n \geq 0\}\\
&\qquad \text{(die Potenzen von } b \text{ liefern nur das schon vorhandene Leerwort)},\\
h_1(\{a^n b^n \mid n \geq 1\}) &= \{(ab)^n b^{2n} \mid n \geq 1\},\\
h_2(\{a^n b^n \mid n \geq 1\}) &= \{(ac)^n \mid n \geq 1\},\\
h_1^{-1}(\{ab^n \mid n \geq 1\} &= \{ab^n \mid n \geq 0\},\\
h_2^{-1}(\{ac, acac\} &= \{b^i ab^j \mid i \geq 0, j \geq 0\} \cup \{b^r ab^s ab^t \mid r \geq 0, s \geq 0, t \geq 0\},\\
h_1^{-1}(\{a^n b^n \mid n \geq 1\}) &= \{a\},\\
h_2^{-1}(\{a^n b^n \mid n \geq 1\}) &= \emptyset.
\end{aligned}$$

Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter (nichtlöschenden) Homomorphismen abgeschlossen ist, wenn für beliebige Alphabete X und Y, beliebige Sprachen $L \subseteq X^*$ und jeden (nichtlöschenden) Homomorphismusn $h : X^* \to Y*$ aus $L \in \mathcal{L}$ auch $h(L) \in \mathcal{L}$ folgt.
Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter inversen Homomorphismen abgeschlossen ist, wenn für beliebige Alphabete X und Y, beliebige Sprachen $L \subseteq Y^*$ und beliebige Homomorphismen $h : X^* \to Y*$ aus $L \in \mathcal{L}$ auch $h^{-1}(L) \in \mathcal{L}$ folgt.

Lemma 5.11 *i) Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$ und $\mathcal{L}(RE)$ sind unter Homomorphismen abgeschlossen, $\mathcal{L}(CS)$ ist nicht unter Homomorphismen abgeschlossen.*
ii) iii) Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind unter nichtlöschenden Homomorphismen abgeschlossen.
iii) Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind unter inversen Homomorphismen abgeschlossen.

Beweis. Wir beweisen nur die Aussagen i) und ii).
a) $\mathcal{L}(RE)$. Es seien $L \in \mathcal{L}(RE)$ eine Sprache über T und $h : T^* \to Y^*$ ein beliebiger Homomorphismus. Dann gibt es eine Regelgrammatik $G = (N, T, P, S)$ mit $L = L(G)$. Ohne Beschränkung der Allgemeinheit können wir annehmen, dass alle Regeln in P von der Form $A_1A_2 \dots A_n \to B_1B_2 \dots B_s$ mit $A_i \in N$ für $1 \leq i \leq n$ und $B_j \in N$ für $1 \leq j \leq s$ mit gewissen $n \geq 1$ und $s \geq 1$ oder von der Form $A \to a$ oder $A \to \lambda$ mit $A \in N$ und $a \in T$ sind (siehe Lemma 2.15. Wir konstruieren die Regelgrammatik $G' = (N, Y, P', S)$, wobei P' aus P entsteht, indem jede Regel der Form $A \to a$ durch $A \to h(a)$ ersetzt wird. Es ist leicht zu sehen, dass $L(G') = h(L(G))$ gilt. Wegen $h(L) = h(L(G))$ wird daher $h(L)$ durch eine Regelgrammatik erzeugt, womit bewiesen ist, dass $h(L) \in \mathcal{L}(RE)$ liegt.
b) $\mathcal{L}(CF)$. Wir geben den gleichen Beweis unter Verwendung der Chomsky-Normalform (siehe Satz 2.23).
c) $\mathcal{L}(REG)$. Es seien $L \in \mathcal{L}(RE)$ eine Sprache über T und $h : T^* \to Y^*$ ein beliebiger Homomorphismus. Dann gibt es eine reguläre Grammatik $G = (N, T, P, S)$ mit $L = L(G)$, bei der alle Regeln von der Form $A \to aB$, $A \to a$ oder $A \to \lambda$ mit $A, B \in N$ und $a \in T$ sind (siehe Satz 2.24) sind. Wir konstruieren die reguläre Grammatik $G' = (N, Y, P', S)$, wobei P' aus P entsteht, indem jede Regel der Form $A \to aB$ durch $A \to h(a)B$ und jede Regel der Form $A \to a$ durch $A \to h(a)$ ersetzt wird. Es ist leicht zu sehen, dass $L(G') = h(L(G))$ gilt. Wegen $h(L) = h(L(G))$ wird daher $h(L)$ durch eine reguläre Grammatik erzeugt, womit bewiesen ist, dass $h(L) \in \mathcal{L}(RE)$ liegt.
d) $\mathcal{L}(CS)$. Die obige Beweisidee kann für nichtlöschende Homomorphismen unter Verwendung der Normalform aus Satz 2.16 benutzt werden, da die neuen Regel $A \to h(a)$ in monotonen Grammatiken erlaubt sind. Ist der Homomorphismus dagegen löschend, so entsteht eine Regel $A \to \lambda$ aus mindestens einer Regel $A \to a$, die bei monotonen Grammatiken nicht erlaubt sind.
Es sei L eine Sprache aus $\mathcal{L}(RE) \setminus \mathcal{L}(CS)$ und $G = (N, T, P, S)$ eine Grammatik der in a) gegebenen Form mit $L = L(G)$. Wir konstruieren nun die monotone Grammatik $G' = (N, T \cup \{c\}, P', S)$, indem wir jede Regel $A \to \lambda$ aus P durch $A \to c$ ersetzen, wobei $c \notin N \cup T$ gilt. Die Sprache $L(G')$ unterscheidet sich von der Sprache $L(G)$ nur dadurch, dass in den Wörtern zusätzlich an einigen Stellen der Buchstabe c steht. Betrachten wir nun den Homomorphismus h mit $h(a) = a$ für $a \in T$ und $h(c) = \lambda$, so erhalten wir $h(L(G')) = L(G) = L \notin \mathcal{L}(CS)$. Somit ist $\mathcal{L}(CS)$ nicht abgeschlossen unter Homomorphismen. □

Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter Durchschnitten mit regulären Sprachen abgeschlossen ist, wenn für eine beliebige Sprache $L \in \mathcal{L}$ und eine beliebige reguläre Sprache R auch $L \cap R \in \mathcal{L}$ gilt.

Lemma 5.12 *Die Sprachfamilien $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ sind unter Durchschnitten mit regulären Sprachen abgeschlossen.*

Beweis. Da jede reguläre Sprache auch in $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ liegt, folgt die Behauptung für $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ aus Lemma 5.3.
Um die Aussage für $\mathcal{L}(CF)$ zu beweisen, konstruieren wir einen Kellerautomaten $\mathcal{M}$, der $L \cap R$ akzeptiert. Es seien $\mathcal{M}_1 = (X, Z_1, \Gamma, z_{0,1}, F_1, \delta_1)$ ein Kellerautomat, der L akzeptiert, und $\mathcal{A}_2 = (X, Z_2.z_{0,2}, F_2, \delta_2)$ ein deterministischer endlicher Automat, der R akzeptiert. Wir konstruieren nun den Kellerautomaten

$$\mathcal{M} = (X, Z_1 \times Z_2, \Gamma, (z_{0,1}, z_{0,2}), F_1 \times F_2, \delta)$$

mit

$$(z_1', z_2'), R, \beta) \in \delta((z_1, z_2), a, \gamma) \quad \text{falls} \quad (z_1', \beta) \in \delta_1(z_1, a, \gamma) \text{ und } \delta_2(z_2, a) = z_2',$$
$$(z_1', z_2), N, \beta) \in \delta((z_1, z_2), a, \gamma) \quad \text{falls} \quad (z_1', \beta) \in \delta_1(z_1, a, \gamma).$$

Entsprechend dieser Definition verhält sich $\mathcal{M}$ in der ersten Komponente der Zustände und hinsichtlich des Bandes wie $\mathcal{M}_1$ und in der zweiten Komponente der Zustände wie $\mathcal{A}_2$ (wobei von $\mathcal{A}_2$ nur dann ein Buchstabe gelesen wird, wenn auch $\mathcal{M}_1$ diesen Buchstaben liest und nach rechts geht). Damit erfolgte eine Akzeptanz nur, wenn sowohl $\mathcal{M}_1$ und cA_2 das Eingabewort akzeptieren. Damit wird $L \cap R$ akzeptiert. □

Wir fassen die Resultate zu Abschlusseigenschaften in der Tabelle aus Abb. 5.1 zusammen, wobei ein + bzw. – im Schnittpunkt der Spalte mit der Familie $\mathcal{L}$ von Sprachen und der Zeile mit der Operation τ bedeutet, dass $\mathcal{L}$ unter τ abgeschlossen bzw. nicht abgeschlossen ist.

	$\mathcal{L}(RE)$	$\mathcal{L}(CS)$	$\mathcal{L}(CF)$	$\mathcal{L}(REG)$
Vereinigung	+	+	+	+
Durchschnitt	+	+	–	+
Komplement	–	+	–	+
Produkt	+	+	+	+
(positiver) KLEENE-Abschluss	+	+	+	+
Homomorphismen	+	–	+	+
nichtlöschende Homomorphismen	+	+	+	+
inverse Homomorphismen	+	+	+	+
Durchschnitt mit regulären Mengen	+	+	+	+

Abbildung 5.1: Tabelle der Abschlusseigenschaften

Wir geben nun eine Bezeichnung für Sprachfamilien, die gegenüber allen bisher behandelten Operationen abgeschlossen sind, unter denen jede der Familien der Chomsky-Hierarchie abgeschlossen ist.

Definition 5.13 *Eine Menge $\mathcal{L}$ von Sprachen heißt abstrakte Familie von Sprachen (kurz AFL),*

- *wenn sie mindestens eine nichtleere Sprache enthält und*
- *wenn sie unter Vereingung, Produkt, Kleene-Abschluss, nichtlöschenden Homomorphismen, inversen Homomorphismen und Durchschnitten mit regulären Sprachen abgeschlossen ist.*

Die Menge $\mathcal{L}$ heißt volle AFL, wenn sie zusätzlich unter (beliebigen) Homomorphismen abgeschlossen ist.

Entsprechend der Tabelle aus Abb. 5.1 sind $\mathcal{L}(REG)$, $\mathcal{L}(CF)$, $\mathcal{L}(CS)$ und $\mathcal{L}(RE)$ AFLs, und $\mathcal{L}(REG)$, $\mathcal{L}(CF)$ und $\mathcal{L}(RE)$ sind sogar volle AFLs.
Wir wollen jetzt eine weitere Charakterisierung der Familie $\mathcal{L}(REG)$ geben. Sie wird sich als die kleinste volle AFL erweisen.

Satz 5.14 *Für jede volle AFL $\mathcal{L}$ gilt $\mathcal{L}(REG) \subseteq \mathcal{L}$.*

Beweis. Es sei $\mathcal{L}$ eine volle AFL und R eine beliebige reguläre Menge über dem Alphabet X. Dann enthält $\mathcal{L}$ eine nichtleere Sprache L. Es sei w ein Wort aus L. Da die endliche Menge $\{w\}$ eine reguläre Menge ist, liegt $L \cap \{w\} = \{w\}$ auch in $\mathcal{L}$. Wir definieren nun für jedes Element $a \in X$ den Homomorphismus h_a durch $h_a(a) = w$ und $h_a(b) = wa$ für jedes $b \in X$ mit $b \neq a$. Offenbar gilt dann $h_a^{-1}(\{w\}) = \{a\}$. Somit liegt jede Menge $\{a\}$ mit $a \in X$ in $\mathcal{L}$. Unter Verwendung des Homomorphismus h mit $h(a) = \lambda$ für alle $a \in X$ und der regulären Menge $\{b\}$, wobei $b \in X$ und $b \neq a$ gelte, erhalten wir, dass

$$\{\lambda\} = h(\{a\}) \text{ und } \emptyset = \{a\} \cap \{b\}$$

in $\mathcal{L}$ liegen. Aus diesen Mengen kann R nach dem Satz von Kleene durch (mehrfache, iterierte) Anwendung von Vereinigung, Produkt und Kleene-Abschluss erzeugt werden. Da $\mathcal{L}$ unter diesen Operationen abgeschlossen ist, ist $R \in \mathcal{L}$ gezeigt. □

Damit ergibt sich sofort die folgende Aussage.

Folgerung 5.15 *Die Menge $\mathcal{L}(REG)$ ist die (bez. der Inklusion) kleinste volle AFL.* □

Die AFLs wurden eingeführt, weil man bemerkte, dass die zu ihrer Definition verwendeten Abschlüsse unter gewissen Operationen den Abschluss unter weiteren Operationen nach sich ziehen. Daher brauchten diese neuen Abschlusseigenschaften nicht für jede Familie einzeln nachgewiesen werden, sondern es kann ein einheitlicher Beweis für alle AFLs gegeben werden. Wir demonstrieren dies an zwei Beispielen.
Wir sagen, dass eine Familie $\mathcal{L}$ von Sprachen unter mengentheoretischer Subtraktion regulärer Sprachen abgeschlossen ist, wenn für eine beliebige Sprache $L \in \mathcal{L}$ und eine beliebige reguläre Sprache R auch $L \setminus R \in \mathcal{L}$ gilt.

Satz 5.16 *Jede AFL ist unter mengentheoretischer Subtraktion regulärer Mengen abgeschlossen.*

Beweis. Es sei $\mathcal{L}$ eine AFL. Für eine Sprache $L \subseteq X^*$ aus $\mathcal{L}$ und eine reguläre Sprache $R \subseteq X^*$ gilt $L \setminus R = L \cap (X^* \setminus R)$. Da das Komplement $X^* \setminus R$ von R nach Lemma 5.4 regulär ist, ist $L \setminus R$ der Durchschnitt von einer Sprache in $\mathcal{L}$ und einer regulären Sprachen, d.h. $L \setminus R$ liegt in $\mathcal{L}$. □

Definition 5.17 *Es seien X und Y Alphabete. Für jeden Buchstaben $a \in X$ sei $\sigma(a)$ eine Sprache über Y. Für eine Sprache $L \subseteq X^*$ definieren die Sprache $\sigma(L) \subseteq Y^*$ durch*

$$\sigma(L) = \{w_1 w_2 \dots w_n \mid a_1 a_2 \dots a_n \in L,\ a_i \in X \text{ und } w_i \in \sigma(a_i) \text{ für } 1 \leq i \leq n,\ n \geq 0\}.$$

Beispiel 5.18 Für die Sprache $L = \{aba, aa\}$ und die Substitutionen σ_1 und σ_2, die durch

$$\sigma_1(a) = \{a^2\},\ \sigma_1(b) = \{ab\} \quad \text{und} \quad \sigma_2(a) = \{a, a^2\},\ \sigma_2(b) = \{b, b^2\}$$

gegeben sind, erhalten wir

$$\begin{aligned}
\sigma_1(L) &= \{a^2aba^2, a^2a^2\} = \{a^3ba^2, a^4\},\\
\sigma_2(L) &= \{aba, a^2ba, aba^2, a^2ba^2, ab^2a, a^2b^2a, ab^2a^2, a^2b^2a^2, aa, a^2a, aa^2, a^2a^2\}\\
&= \{aba, a^2ba, aba^2, a^2ba^2, ab^2a, a^2b^2a, ab^2a^2, a^2b^2a^2, a^2, a^3, a^4\}.
\end{aligned}$$

Wir bemerken, dass ein Homomorphismus $h : X^* \to Y^*$ als Substitution $\sigma : X^* \to Y^*$ aufgefasst werden kann, bei der die Mengen $\sigma(a)$ für $a \in X$ nur ein Wort enthalten.

Satz 5.19 *Jede volle AFL ist unter Substitutionen mit regulären Sprachen abgeschlossen.*

Beweis. Es seien $\mathcal{L}$ eine volle AFL, $L \subseteq X^*$ eine Sprache aus $\mathcal{L}$ und $\tau : X^* \to Y^*$ eine Substitution, bei der $\tau(a)$ für jedes $a \in X$ eine reguläre Sprache über Y ist. Es sei $X = \{a_1, a_2, \ldots, a_n\}$. Ferner gelte $\tau(a_i) = R_i$. Wir definieren nun

$$
\begin{aligned}
&X' = \{a' \mid a \in X\},\\
&h_1 : (X' \cup Y)^* \to X^* \text{ durch } h_1(x') = x \text{ für } x \in X \text{ und } h_1(y) = \lambda \text{ für } y \in Y,\\
&h_2 : (X' \cup Y)^* \to Y^* \text{ durch } h_2(x') = \lambda \text{ für } x \in X \text{ und } h_2(y) = y \text{ für } y \in Y,\\
&R = \bigcup_{i=1}^{n} a_i' R_i.
\end{aligned}
$$

Dann ergibt sich der Reihe nach

$$
\begin{aligned}
h_1^{-1}(L) &= \{u_0 x_1' u_1 x_2' u_2 \ldots x_r' u_r \mid x_1 x_2 \ldots x_r \in L,\ u_i \in Y^* \text{ für } 1 \le i \le r\},\\
h_1^{-1}(L) \cap R &= \{x_1' u_1 x_2' u_2 \ldots x_r' u_r \mid x_1 x_2 \ldots x_r \in L,\ u_i \in \tau(x_i) \text{ für } 1 \le i \le r\},\\
h_2(h_1^{-1}(L) \cap R) &= \{u_1 u_2 \ldots u_r \mid x_1 x_2 \ldots x_r \in L,\ u_i \in \tau(x_i) \text{ für } 1 \le i \le r\}.
\end{aligned}
$$

Damit gilt

$$\tau(L) = h_2(h_1^{-1}(L) \cap R),$$

und nach den für eine AFL geltenden Abschlußeigenschaften ist somit $\tau(L)$ in $\mathcal{L}$. □

5.2 Entscheidbarkeitsprobleme bei formalen Sprachen

Im ersten Teil der Vorlesung haben wir bereits das Mitgliedsproblem hinsichtlich Entscheidbarkeit und Komplexität für die Sprachfamilien der Chomsky-Hierarchie untersucht. In diesem Abschnitt behandeln wir nun weitere Entscheidbarkeitsprobleme.
Das *Leerheitsproblem* ist die Frage, ob eine gegebene Grammatik mindestens ein Wort erzeugt. Hierbei ist aber wichtig, wie die Sprache gegeben ist. Entsprechend den vorhergehenden Abschnitten kann dies sowohl durch eine Grammatik als auch durch einen akzeptierenden Automaten (und im Fall einer regulären Sprache auch durch einen regulären Ausdruck) geschehen. Daraus resultieren mindestens die zwei folgenden Varianten des Leerheitsproblems für kontextfreie Sprachen:

Gegeben:	kontextfreie Grammatik G
Frage:	Ist $L(G)$ leer ?

oder

Gegeben:	Kellerautomat $\mathcal{M}$
Frage:	Ist $T(\mathcal{M})$ leer ?

Im Folgenden interessieren wir uns zuerst wieder dafür, ob das Problem entscheidbar ist oder nicht, d.h. wir untersuchen, ob es einen Algorithmus gibt, der die Frage beantwortet. Die Antwort ist dann unabhängig von der Formulierung des Problems, da sowohl der Übergang von einer kontextfreien Grammatik G zu einem Kellerautomaten $\mathcal{M}$ mit $L(G) = T(\mathcal{M})$ als auch der umgekehrte Übergang von einem Kellerautomaten zu einer kontextfreien Grammatik konstruktiv - also mittels eines Algorithmus - erfolgen. Folglich haben beide Formulierungen stets die gleiche Antwort.
Eine analoge Situation ist auch hinsichtlich der anderen Typen von Grammatiken und zugehörigen Automaten gegeben.

Im Fall der Existenz eines Algorithmus zur Beantwortung des Problems ist natürlich auch die Komplexität des Algorithmus von großem Interesse. Hier ist eine Abhängigkeit vom Problem gegeben, da schon die Größe der Eingabe Grammatik bzw. Automat (Maschine) unterschiedlich sind. Wir geben hier stets nur die Komplexität des Algorithmus bezogen auf die Größe der Grammatik an. Dafür definieren wir die Größe einer Grammatik $G = (N, T, P, S)$ durch

$$k(G) = \sum_{\alpha \to \beta \in P} |\alpha| + |\beta| + 1,$$

d.h. wir summieren die Längen der Regeln auf, wobei wir eine Regel $\alpha \to \beta$ als Wort über $V \cup T \cup \{\to\}$ auffassen. Geht man davon aus, dass Nichtterminale und Terminale und das Axiom einfach erkannt werden können (beispielsweise durch die Forderung, dass Nichtterminale stets (indizierte) Großbuchstaben, Terminale stets (indizierte) Kleinbuchstaben des lateinischen Alphabets sind und stets S das Axiom ist), so können aus den Regeln auch die anderen Komponenten von G abgelesen werden, so dass zur Angabe der Grammatik eigentlich die Menge der Regeln ausreichend ist.
Ist man an der Komplexität bezogen auf die (hier noch nicht definierte) Größe des Automaten interessiert, so lässt sich diese meist leicht dadurch ermitteln, dass man den Aufwand für den Übergang vom Automaten zur Grammatik noch hinzufügt. Letzterer Aufwand kann aus den Konstruktionen in Abschnitt 2.2 relativ einfach ermittelt werden.

Wir geben jetzt noch zwei weitere wichtige Probleme an, wobei wir stets nur die grammatikalische Variante angeben.

Endlichkeitsproblem
Gegeben: Grammatik G
Frage: Ist $L(G)$ endlich ?

Äquivalenzproblem
Gegeben: Grammatiken G_1 und G_2
Frage: Gilt $L(G_1) = L(G_2)$?

Die obigen Formulierungen der Probleme sind sehr allgemein, da sie keine Spezifikation des Typs der Grammatiken vornehmen. In den folgenden Aussagen werden wir dann vom entsprechenden Problem für reguläre, kontextfreie usw. Grammatiken sprechen, wenn die gegebene Grammatik regulär bzw. kontextfrei usw. ist.

Satz 5.20 *Das Leerheits- und das Endlichkeitsproblem sind für beliebige Regelgrammatiken und monotone (kontextsensitive) Grammatiken unentscheidbar.*

Beweis. Wir geben zuerst den Beweis für die Unentscheidbarkeit des Leerheitsproblems für beliebige Regelgrammatiken.
Es sei G eine Regelgrammatik. Wir betrachten ein beliebiges Wort w über dem Terminalalphabet von G. Dann ist $\{w\}$ eine reguläre Sprache. Wegen der CHOMSKY-Hierachie und Lemma 5.3 gibt es eine Regelgrammatik G' mit $L(G') = L(G) \cap \{w\}$. Offenbar gilt damit

$$L(G') = \begin{cases} \{w\} & \text{falls } w \in L(G) \\ \emptyset & \text{sonst.} \end{cases}$$

Folglich ist $L(G')$ genau dann leer, wenn w nicht in $L(G)$ liegt. Aus der Entscheidbarkeit des Leerheitsproblems für G' würde somit die Entscheidbarkeit des Mitgliedsproblems für G folgen. Wegen Satz 2.69 erhalten wir daher die Unentscheidbarkeit des Leerheitsproblems.

Wir kommen nun zum Leerheitsproblem für monotone Grammatiken. Es sei $G = (N, T, P, S)$ erneut eine beliebige Regelgrammatik. Wir konstruieren zuerst zu G die Regelgrammatik $G' = (N', T, P', S)$ aus Lemma 2.15, deren Regeln alle von der Form $\alpha \to \beta$ mit $\alpha, \beta \in (N')^*$ oder $A \to a$ mit $A \in N', a \in T$ sind und für die $L(G') = L(G)$ gilt. Es sei \$ ein zusätzliches Symbol. Für jede Regel $p = \alpha \to \beta \in P'$ seien

$$\begin{aligned} k(p) &= \max\{0, |\alpha| - |\beta|\} \\ p' &= \alpha \to \beta\$^{k(p)}\,. \end{aligned}$$

Wegen der Wahl von $k(p)$ gilt stets $|\alpha| \leq |\beta| + k(p)$. Die Regelgrammatik

$$G'' = (N', T \cup \{\$\}, \{p' \mid p \in P\} \cup \bigcup_{A \in N'} \{\$A \to A\$\}, S)$$

ist somit monoton.
Zu jedem Wort $w \in L(G) = L(G')$ gibt es ein Wort $w'' \in L(G'')$ mit $w'' = w\$^r$ für ein gewisses $r \geq 0$. Um dies zu sehen, erinnern wir uns, dass es eine Ableitung von w in G' gibt, bei der wir zuerst nur Regeln der Form $\alpha \to \beta$ mit $\alpha, \beta \in (N')^*$ anwenden und anschließend nur die terminierenden Regeln der Form $A \to a$. Wir simulieren diese Ableitung und transportieren nach jeder Anwendung einer Regel aus G'' die unter Umständen entstandenen \$s nach rechts. Dadurch entstehen nur Satzformen der Form $z\$^s$, bei denen z eine Satzform von G' ist. Hieraus ergibt sich die Existenz eines $r \geq 0$ mit $w\$^r \in L(G'')$ für jedes $w \in L(G') = L(G)$.
Umgekehrt ist leicht zu sehen, dass jedes Wort w'' aus $L(G'')$ die Form

$$w'' = w_1\$^{r_1} w_2\$^{r_2} \ldots w_k\$^{r_k} w_{k+1}$$

mit

$$w = w_1 w_2 \ldots w_k w_{k+1} \in L(G') = L(G)$$

hat.
Hieraus ergibt sich, dass $L(G'')$ genau dann leer ist, wenn auch $L(G)$ leer ist. Die Entscheidbarkeit des Leerheitsproblems für monotone Grammatiken G'' würde daher die Entscheidbarkeit des Leerheitsproblems für beliebige Regelgrammatiken nach sich ziehen. Dies widerspricht der gerade gezeigten Unentscheidbarkeit des Leerheitsproblems für beliebige Regelgrammatiken.

Wir kommen nun zum Endlichkeitsproblem, für dessen Unentscheidbarkeit wir einen gemeinsamen Beweis für monotone und beliebige Regelgrammatiken geben.
Es sei $H = (N, T, P, S)$ eine beliebige bzw. monotone Regelgrammatik. Ferner sei $c \notin T$. Aus H konstruieren wir zuerst eine beliebige bzw. monotone Regelgrammatik H' mit $L(H') = L(H) \cdot \{c^n \mid n \geq 1\}$ (siehe Lemma 5.5 bei Beachtung von $\{c^n \mid n \geq 1\} \in \mathcal{L}(MON) \subset \mathcal{L}(RE)$). Offensichtlich ist $L(H')$ genau dann endlich, wenn $L(H)$ leer ist. Somit würde die Entscheidbarkeit der Endlichkeit die Entscheidbarkeit der Leerheit implizieren. Die vorstehend bewiesene Aussagen liefern daher die Unentscheidbarkeit der Endlichkeit. □

Satz 5.21 *Für kontextfreie Grammatiken sind das Leerheits- und Endlichkeitsproblem in der Zeit $O(\#(V) \cdot k(G))$ entscheidbar.*

Beweis. i) Leerheitsproblem.
Es sei $G = (N, T, P, S)$ eine kontextfreie Grammatik. Wir geben nun einen Algorithmus zur Bestimmung der Menge N' aller Nichtterminale A, für die $A \Longrightarrow^* w_A$ für ein gewisses $w_A \in T^*$ gilt.
Dazu setzen wir $N'_0 = \emptyset$ und $P_0 = P$. Sind N'_i und P_i schon definiert, so setzen wir weiterhin

$$N'_{i+1} = \{A \mid A \in N, A \to w \in P_i \text{ für ein } w \in T^*\},$$

und P_{i+1} sei die Menge aller Regeln, die aus denen aus P dadurch entstehen, dass wir alle Symbole aus N'_{i+1} streichen. Offensichtlich gilt dann

$$N' = \bigcup_{i=0}^{\#(N)} N'_i.$$

Offenbar ist $L(G)$ genau dann leer, wenn S kein Element aus N' ist, womit die Leerheit von $L(G)$ entschieden werden kann.

Die Konstruktion der Menge N' erfolgt in Analogie zur Konstruktion von M im Beweis von Lemma 2.18 und hat daher auch die gleiche Komplexität. Damit ist gezeigt, dass die Leerheit von $L(G)$ in der Zeit $O(\#(N) \cdot k(G))$ entschieden werden kann.

ii) Endlichkeitsproblem.
Wir entscheiden zuerst, ob $L(G)$ leer ist. Bei positiver Antwort ist $L(G)$ auch endlich. Andernfalls gilt $S \in N'$ und wir setzen wie folgt fort.
Ausgehend von $N''_1 = \{S\}$ konstruieren wir induktiv die Mengen

$$N''_{i+1} = N''_i \cup \{A \mid B \to xAy \text{ für gewisse } x, y \in V^*, B \in N''_i\}$$

und setzen

$$N'' = \bigcup_{i=1}^{\#(N)} N''_i.$$

Die Bestimmung von N'' ist in der Zeit $\#(N) \cdot k(G)$ möglich, da erneut die Konstruktion von N''_{i+1} aus N''_i mittels Durchmustern aller Regeln gewonnen werden kann.
Offenbar gilt

$$N'' = \{A \mid S \Longrightarrow^* xAy \text{ für gewisse } x, y \in V^*\}.$$

Daher kann ein Nichtterminal $C \notin N''$ in keiner Satzform von G vorkommen.
Wir definieren nun $G' = (N' \cap N'', T, P', S)$, wobei N' in Teil i) dieses Beweises definiert wurde P' die Menge aller Regeln aus P ist, die nur Nichtterminale aus $N' \cap N''$ enthalten. Hierdurch ist gesichert, dass $L(G') = L(G)$ gilt und es keine Satzform gibt, aus der nicht ein terminales Wort hergeleitet werden kann.
Wir erzeugen nun aus G' eine Grammatik $G'' = (N' \cap N'', T, P'', S)$, die keine Regeln der Form $A \to B$ mit $A, B \in N' \cap N''$ enthält (siehe Lemma 2.21).
Dann konstruieren wir den gerichteten Graphen $H = (N' \cap N'', E)$, wobei (A, B) genau dann in E liegt, wenn es eine Regel $A \to xBy$ mit gewissen x, y in P'' gibt. Wir testen nun, ob H einen Kreis enthält. Ist dies der Fall, gibt es eine Ableitung $A \Longrightarrow^* z_1 A z_2$ mit $|z_1 z_2| > 0$ und folglich für $1 \leq i$ die Ableitungen

$$S \Longrightarrow^* u_1 A u_2 \Longrightarrow^* u_1 z_1 A z_2 u_2 \Longrightarrow^* u_1 z_1^2 A z_2^2 u_2 \Longrightarrow^* \ldots \Longrightarrow^* u_1 z_1^i A z_2^i u_2.$$

Damit existieren auch Ableitungen

$$S \Longrightarrow^* u_1 z_1^i A z_2^i u_2 \Longrightarrow^* u_1'(z_1')^i v (z_2')^i u_2' \in T^*,$$

womit die Unendlichkeit von $L(G) = L(G') = L(G'')$ nachgewiesen ist. Hat H dagegen keinen Kreis, so kann es nur endlich viele Ableitungen geben, woraus die Endlichkeit von $L(G)$ folgt. Somit ist $L(G)$ genau dann endlich, wenn H keinen Kreis enthält.
Entsprechend den Aussagen zur Komplexität der Bestimmung von N' (siehe Teil i) dieses Beweises) und N'' (siehe oben) und Lemma 2.21 ist klar, dass der Graph H in der Zeit $O(\#(N) \cdot k(G))$ gewonnen werden kann. Die Existenz eines Kreises kann mittels Tiefensuche daher ebenfalls in der Zeit $O(\#(N) \cdot k(G))$ erfolgen. □

Wir kommen nun zum Äquivalenzproblem.

Satz 5.22 *Das Äquivalenzproblem ist für kontextfreie Grammatiken unentscheidbar.*

Beweis. Wir beweisen die Aussage durch Reduktion auf das Postsche Korrespondenzproblem. Dazu sei die Menge $U = \{(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)\}$ von Paaren mit $u_i, v_i \in T^+$ für $1 \leq i \leq n$ gegeben.
Wir betrachten die kontextfreien Grammatiken

$$G_1 = (N, T \cup \{c\}, P, S) \quad \text{und} \quad G_2 = (N \cup \{S', S''\}, T \cup \{c\}, P \cup P', S')$$

mit

$$\begin{aligned}
N &= \{S, S_u, S_r, S_l\}, \\
P &= \{S_u \to c, S_l \to c, S_r \to c\} \cup \{S \to xS_u y \mid x, y \in T, x \neq y\} \\
&\quad \cup \{S_u \to xS_u y \mid x, y \in T\} \\
&\quad \cup \bigcup_{x \in T} \{S \to xSx, S \to xS_l, S \to S_r x, S_l \to xS_l, S_r \to S_r x\}, \\
P' &= \{S' \to S, S' \to S''\} \cup \bigcup_{i=1}^{n} \{S'' \to u_i S'' v_i^R, S'' \to u_i c v_i^R\}.
\end{aligned}$$

Die erzeugten Sprachen sind

$$L(G_1) = \{\alpha c \beta^R \mid \alpha, \beta \in T^+, \alpha \neq \beta\}$$

und

$$L(G_2) = L(G_1) \cup \{u_{i_1} u_{i_2} \dots u_{i_k} c v_{i_k} v_{i_{k-1}} \dots v_{i_1} \mid k \geq 1, 1 \leq i_j \leq n, 1 \leq j \leq k\}.$$

Dies ist wie folgt zu sehen. Alle nichtterminalen Satzformen von G_1 sind von einer der folgenden Formen:

$$\begin{array}{lll} \alpha S \beta^R & \text{mit} & |\alpha| = |\beta|, \alpha = \beta, \\ \alpha S_u \beta^R & \text{mit} & |\alpha| = |\beta|, \alpha \neq \beta, \\ \alpha S_r \beta^R & \text{mit} & |\alpha| < |\beta|, \\ \alpha S_l \beta^R & \text{mit} & |\alpha| > |\beta|. \end{array}$$

Da das Terminieren nur durch die Regeln $S_u \to c$, $S_r \to c$ und $S_l \to c$ erfolgen kann, ist damit gezeigt, dass die Wörter aus $L(G_1)$ von der Gestalt $\alpha c \beta^R$ mit $\alpha \neq \beta$ sind. Es ist leicht zu sehen, dass alle Wörter dieser Gestalt von G_1 erzeugt werden können.
Das Axiom von G_2 generiert entweder S, woraus genau die Wörter aus $L(G_1)$ erzeugt werden können, oder S'', wodurch nach links stets ein u_i und nach rechts das Spiegelbild des zugehörigen v_i erzeugt wird. Damit ist obige Aussage für $L(G_2)$ nachgewiesen.
Weiterhin ergibt sich $L(G_1) = L(G_2)$ genau dann, wenn $u_{i_1} u_{i_2} \dots u_{i_k} \neq v_{i_1} v_{i_2} \dots v_{i_k}$ für alle Folgen $i_1 i_2 \dots i_k$ gilt, d.h. wenn das Postsche Korrespondenzproblem für die Paarmenge U keine Lösung hat. Daher folgt die Behauptung aus der Unentscheidbarkeit des Postschen Korrespondenzproblems. □

Satz 5.23 *Das Äquivalenzproblem für reguläre Grammatiken $G_1 = (N_1, T, P_1, S_1)$ und $G_2 = (N_2, T, P_2, S_2)$ ist in der Zeit $O(n \cdot k^2)$ mit*

$$n = max\{\#(N_1), \#(N_2)\} \quad \text{und} \quad k = max\{k(G_1), k(G_2)\}$$

entscheidbar.

Beweis. Zu gegebenen regulären Grammatiken G_1 und G_2 können wir eine reguläre Grammatik $G = (N, T, P, S)$ entsprechend den Ergebnissen der vorhergehenden Abschnitte derart konstruieren, dass

$$L(G) = (L(G_1) \cap \overline{L(G_2)}) \cup (L(G_2) \cap \overline{L(G_1)}$$

gilt. Entsprechend den mengentheoretischen Beziehungen ist daher $L(G)$ genau dann leer, wenn die Sprachen $L(G_1)$ und $L(G_2)$ gleich sind.
Die Komplexität dieses Verfahrens läßt sich wie folgt ermitteln: Für die Konstruktion von G ist zuerst die Umwandlung von G_1 und G_2 in Grammatiken

$$G_1' = (N_1', T, P_1', S_1') \quad \text{und} \quad G_2' = (N_2', T, P_2', S_2')$$

erforderlich, die in der Normalform aus Satz 4.11 sind und für die

$$\#(N_i') = O(k(G_i)) = O(k) \quad \text{und} \quad k(G_i') = O(\#(N_i) \cdot k(G_i)) = O(n \cdot k)$$

für $i \in \{1,2\}$ gelten. Hierzu ist für $i \in \{1,2\}$ die Zeit $O(\#(N_i) \cdot k(G_i)) = O(n \cdot k)$ erforderlich. Die Bestimmung von G aus G'_1 und G'_2 ist in $O(max\{k(G'_1), k(G'_2)\}) = O(n \cdot k)$ möglich, und es gelten

$$\#(N) = O(max\{\#(N'_1), \#(N'_2)\}) = O(k)$$

und

$$k(G) = O(max\{k(G'_1), k(G'_2)\}) = O(n \cdot k).$$

Aus Satz 5.21 ergibt sich für die Entscheidbarkeit der Leerheit von $L(G)$ die Komplexität

$$O(\#(N) \cdot k(G)) = O(k \cdot n \cdot k) = O(n \cdot k^2).$$

Die Komplexitätsaussage des Satzes ergibt sich nun durch Addition der Komplexitäten der einzelnen Schritte. □

Wir fassen die Aussagen zur Entscheidbarkeit in einer Tabelle zusammen, wobei ein + bzw. − im Schnittpunkt der Zeile zum Problem und der Spalte zur Sprachfamilie, bedeutet, dass dieses Problem bei dieser Familie entscheidbar bzw. unentscheidbar ist. Dabei geben wir der Vollständigkeit halber auch die Resultate für das Mitgliedsproblem an.

	$\mathcal{L}(REG)$	$\mathcal{L}(CF)$	$\mathcal{L}(CS)$	$\mathcal{L}(RE)$
Mitgliedsproblem	+	+	+	−
Leerheitsproblem	+	+	−	−
Endlichkeitsproblem	+	+	−	−
Äquivalenzproblem	+	−	−	−

Kapitel 6

Ergänzung III : Beschreibungskomplexität endlicher Automaten

6.1 Eine algebraische Charakterisierung der Klasse der regulären Sprachen

Ein (deterministischer) *endlicher Automat* ist ein Quintupel

$$\mathcal{A} = (X, Z, z_0, \delta, F),$$

wobei X und Z Alphabete sind, z_0 ein Element von Z ist, δ eine Funktion von $Z \times X$ in Z ist, und F eine nichtleere Teilmenge von Z ist. X ist die Menge der Eingabesymbole, und Z ist die Menge der Zustände. z_0 und F sind der Anfangszustand und die Menge der akzeptierenden Zustände.
δ ist die Überführungsfunktion von $\mathcal{A}$.
Durch

$$\begin{aligned} \delta^*(z, \lambda) &= z, \text{ für } z \in Z, \\ \delta^*(z, wa) &= \delta(\delta^*(z, w), a) \text{ für } w \in X^*,\ a \in X \end{aligned}$$

erweitern wir δ zu einer Funktion δ^* von $Z \times X^*$ in Z.
Die von $\mathcal{A}$ *akzeptierte* Sprache ist durch

$$T(\mathcal{A}) = \{w \in X^* \mid \delta^*(z_0, w) \in F\}$$

definiert.
Es ist bekannt, dass *eine Sprache L genau dann regulär ist, wenn es einen endlichen Automaten $\mathcal{A}$ mit $L = T(\mathcal{A})$ gibt.*
Wir wollen zuerst zwei algebraische Charakterisierungen regulärer Sprachen herleiten. Dazu benötigen wir die folgenden Definitionen.
Eine Äquivalenzrelation $\sim$ auf der Menge X^* heißt *Rechtskongruenz*, falls für beliebige Wörter $x, y \in X^*$ und $a \in X$ aus $x \sim y$ auch $xa \sim ya$ folgt. Bei Multiplikation von rechts werden äquivalente Wörter wieder in äquivalente Wörter überführt.

Eine Äquivalenzrelation $\sim$ auf X^* heißt *Verfeinerung* der Menge $R \subseteq X^*$, wenn für beliebige Wörter $x, y \in X^*$ aus $x \sim y$ folgt, dass $x \in R$ genau dann gilt, wenn auch $y \in R$ gilt. Dies bedeutet, dass mit einem Wort $x \in R$ auch alle zu x äquivalenten Wörter in R liegen. Folglich liegt die zu $x \in R$ gehörige Äquivalenzklasse vollständig in R. Hieraus resultiert die Bezeichnung Verfeinerung für diese Eigenschaft.
Für eine Äquivalenzrelation $\sim$ bezeichnen wir die Anzahl der Äquivalenzklassen von $\sim$ als Index von $\sim$ und bezeichnen sie mit $Ind(\sim)$. Die Äquivalenzrelation $\sim$ hat *endlichen Index*, falls $Ind(\sim)$ endlich ist.
Wir nennen eine Äquivalenzrelation $\sim$ eine *R-Relation*, falls sie eine Rechtskongruenz mit endlichem Index ist und eine Verfeinerung von R ist.

Beispiel 6.1 Seien $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein endlicher Automat und $R = T(\mathcal{A})$. Wir definieren auf X^* die Relation $\sim_{\mathcal{A}}$ durch

$$x \sim_{\mathcal{A}} y \quad \text{genau dann, wenn} \quad \delta^*(z_0, x) = \delta^*(z_0, y)$$

gilt. Offenbar ist $\sim_{\mathcal{A}}$ eine Äquivalenzrelation. Wir zeigen nun, dass $\sim_{\mathcal{A}}$ eine R-Relation ist.
Sei $x \sim_{\mathcal{A}} y$. Dann gilt nach Definition $\delta^*(z_0, x) = \delta^*(z_0, y)$. Damit erhalten wir

$$\delta^*(z_0, xa) = \delta(\delta^*(z_0, x), a) = \delta(\delta^*(z_0, y), a) = \delta^*(z_0, ya)$$

und somit $xa \sim_{\mathcal{A}} ya$, womit nachgewiesen ist, dass $\sim_{\mathcal{A}}$ eine Rechtskongruenz ist.
Sei erneut $x \sim_{\mathcal{A}} y$. Ferner sei $x \in R$. Dies bedeutet $\delta^*(z_0, x) = \delta^*(z_0, y)$ und $\delta^*(z_0, x) \in F$. Folglich gilt $\delta^*(z_0, y) \in F$, woraus $y \in R$ folgt. Analog folgt aus $y \in R$ auch $x \in R$. Damit ist $\sim_{\mathcal{A}}$ Verfeinerung von R.
Sei $x \in X^*$. Ferner sei $\delta^*(z_0, x) = z$. Dann ergibt sich für die zu x bez. $\sim_{\mathcal{A}}$ gehörende Äquivalenzklasse $K_{\mathcal{A}}(x)$

$$\begin{aligned} K_{\mathcal{A}}(x) &= \{y \mid x \sim_{\mathcal{A}} y\} \\ &= \{y \mid \delta^*(z_0, x) = \delta^*(z_0, y)\} \\ &= \{y \mid \delta^*(z_0, y) = z\}. \end{aligned}$$

Sei nun $z \in Z$ ein von z_0 erreichbarer Zustand. Dann gibt es ein Wort x mit $\delta^*(z_0, x) = z$ und die Beziehungen

$$\begin{aligned} \{y \mid \delta^*(z_0, y) = z\} &= \{y \mid \delta^*(z_0, y) = \delta^*(z_0, x)\} \\ &= \{y \mid y \sim_{\mathcal{A}} x\} \\ &= K_{\mathcal{A}}(x) \end{aligned}$$

sind gültig. Daher gibt es eine eineindeutige Beziehung zwischen den Äquivalenzklassen von $\sim_{\mathcal{A}}$ und den Zuständen von $\mathcal{A}$, die vom Anfangszustand erreichbar sind. Dies bedeutet, dass die Anzahl der Zustände von $\mathcal{A}$ mit dem Index von $\sim_{\mathcal{A}}$ übereinstimmt. Wegen der Endlichkeit der Zustandsmenge Z ist also auch der Index von $\sim_{\mathcal{A}}$ endlich.

Beispiel 6.2 Für eine Sprache $R \subseteq X^*$ definieren wir die Relation $\sim_R$ wie folgt: $x \sim_R y$ gilt genau dann, wenn für alle $w \in X^*$ das Wort xw genau dann in R ist, falls dies auch für yw der Fall ist. Wir zeigen, dass $\sim_R$ eine Rechtskongruenz ist, die R verfeinert.

Seien dazu $x \sim_R y$, $a \in X$ und $w \in X^*$. Dann ist $aw \in X^*$ und nach Definition von $\sim_R$ gilt $xaw \in R$ genau dann, wenn $yaw \in R$ gültig ist. Dies liefert, da w beliebig ist, die Aussage $xa \sim_R ya$. Somit ist $\sim_R$ eine Rechtskongruenz.
Wählen wir $w =$
lambda, so ergibt sich aus $x \sim_R y$ nach Definition, dass $x \in R$ genau dann gilt, wenn auch $y \in R$ erfüllt ist. Deshalb ist $\sim_R$ eine Verfeinerung von R.
Wir bemerken, dass $\sim_R$ nicht notwendigerweise einen endlichen Index haben muss. Dazu betrachten wir

$$R = \{a^n b^n \mid n \geq 1\}$$

und zwei Wörter a^k und a^ℓ mit $k \neq \ell$. Wegen $a^k b^k \in R$ und $a^\ell b^k \notin R$ sind offensichtlich a^ℓ und a^k nicht äquivalent. Somit gibt es mindestens soviel Äquivalenzklassen wie Potenzen von a und damit soviel wie natürliche Zahlen. Dies zeigt die Unendlichkeit des Index.

Ziel dieses Abschnitts ist eine Charakterisierung der regulären Sprachen durch Äquivalenzrelationen mit den oben definierten Eigenschaften.

Satz 6.3 *Für eine Sprache $R \subseteq X^*$ sind die folgenden Aussagen gleichwertig:*
i) R ist regulär.
ii) Es gibt eine R-Relation.
iii) Die Relation $\sim_R$ (aus Beispiel 6.2) hat endlichen Index.

Beweis. i) $\Longrightarrow$ ii). Zu einer regulären Sprache R gibt es einen endlichen Automaten $\mathcal{A}$ mit $T(\mathcal{A}) = R$. Dann können wir entsprechend Beispiel 6.1 die Relation $\sim_{\mathcal{A}}$ konstruieren. Diese ist nach Beispiel 6.1 eine R-Relation.
ii) $\Longrightarrow$ iii) Nach Voraussetzung gibt es eine R-Relation $\sim$ mit endlichem Index. Wir beweisen nun $Ind(\sim_R) \leq Ind(\sim)$.
Dazu zeigen wir zuerst, dass aus $x \sim y$ auch $xw \sim yw$ für alle $w \in X^*$ folgt. Für $w =$ *lambda* ist dies klar. Für $w \in X$ ist es klar, da $\sim$ eine Rechtskongruenz ist. Sei die Aussage nun schon für $|w| < n$ bewiesen. Wir betrachten das Wort v der Länge n. Dann gibt es ein w der Länge $n-1$ und ein $a \in X$ mit $v = wa$. Nach Induktionvoraussetzung haben wir $xw \sim yw$. Nach der Definition der Rechtskongruenz folgt daraus $xwa \sim ywa$ und damit $xv \sim yv$.
Sei nun $x \sim y$ und $w \in X^*$. Dann gilt auch $xw \sim yw$. Da $\sim$ eine R-Relation ist, folgt $xw \in R$ gilt genau dann, wenn $yw \in R$ gilt. Nach Definition $\sim_R$ bedeutet dies aber $x \sim_R y$. Damit ist gezeigt, dass $x \sim y$ auch $x \sim_R y$ impliziert. Hieraus ergibt sich aber

$$\{y \mid y \sim x\} \subseteq \{y \mid y \sim_R x\}.$$

Jede Äquivalenzklasse von $\sim$ ist also in einer Äquivalenzklasse von $\sim_R$ enthalten. Damit gilt $Ind(\sim_R) \leq Ind(\sim)$. Da $Ind(\sim)$ endlich ist, hat auch $\sim_R$ endlichen Index.
iii) $\Longrightarrow$ i) Mit $K_R(x)$ bezeichnen wir die Äquivalenzklasse von $x \in X^*$ bez. $\sim_R$. Wir betrachten den endlichen Automaten

$$\mathcal{A} = (X, \{K_R(x) \mid x \in X^*\}, K_R(lambda), \delta, \{K_R(y) \mid y \in R\})$$

mit

$$\delta(K_R(x), a) = K_R(xa).$$

Wir bemerken zuerst, dass aufgrund der Voraussetzung, dass $\sim_R$ eine R-Relation ist, die Definition von $\mathcal{A}$ korrekt ist. (Die Endlichkeit der Zustandsmenge von $\mathcal{A}$ folgt aus der Endlichkeit des Index von $\sim_R$. Falls $K_R(x) = K_R(y)$, so ist auch $K_R(xa) = K_R(ya)$, denn $\sim_R$ ist eine Rechtskongruenz und daher gilt mit $x \sim_R y$, d. h. $K_R(x) = K_R(y)$, auch $xa \sim_R ya$.) Ferner beweist man mittels vollständiger Induktion über die Länge von x leicht, dass $\delta^*(K_R($
$lambda), x) = K_R(x)$ für alle $x \in X^*$ gilt. Damit folgt

$$T(\mathcal{A}) = \{x \mid \delta^*(K_R(lambda), x) \in \{K_R(y) \mid y \in R\}\} = \{x \mid K_R(x) \in \{K_R(y) \mid y \in R\}\} = R.$$

Damit ist R als regulär nachgewiesen. □

Im Beweisteil ii) $\Longrightarrow$ iii) wurde eigentlich die folgende Aussage bewiesen.

Folgerung 6.4 *Sei R eine beliebige Sprache. Dann gilt für jede R-Relation $\sim$ die Beziehung $Ind(\sim) \geq Ind(\sim_R)$.* □

6.2 Minimierung deterministischer endlicher Automaten

In diesem Abschnitt diskutieren wir als Komplexitätsmaß die Anzahl der Zustände eines endlichen Automaten. Dazu setzen wir für einen endlichen Automaten $\mathcal{A} = (X, Z, z_0, \delta, F)$ und eine reguläre Sprache R

$$\begin{aligned} z(\mathcal{A}) &= \#(Z), \\ z(R) &= \min\{z(\mathcal{A}) \mid T(\mathcal{A}) = R\}. \end{aligned}$$

Wir sagen, dass $\mathcal{A}$ *minimaler* Automat für R ist, falls $T(\mathcal{A}) = R$ und $z(\mathcal{A}) = z(R)$ gelten. Als erstes geben wir ein Resultat an, dass die Größe $z(R)$ für eine Sprache bestimmt.

Satz 6.5 *Für eine reguläre Sprache R gilt $z(R) = Ind(\sim_R)$.*

Beweis. Wir bemerken zuerst, dass aus dem Teil iii) $\Longrightarrow$ i) des Beweises von Satz 6.3 folgt, dass es einen Automaten $\mathcal{A}$ mit $z(\mathcal{A}) = Ind(\sim_R)$ gibt.
Sei nun $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein beliebiger endlicher Automat mit $T(\mathcal{A}) = R$. Dann ist nach Beispiel 6.1 die Relation $\sim_{\mathcal{A}}$ eine R-Relation, für die $z(\mathcal{A}) = Ind(\sim_{\mathcal{A}})$ gilt. Wegen Folgerung 6.4 erhalten wir daraus sofort $z(\mathcal{A}) \geq Ind(\sim_R)$.
Die Kombination dieser beiden Erkenntnisse besagt gerade die Behauptung. □

Wir beschreiben nun für einen Automaten $\mathcal{A}$ einen Automaten, der minimal für $T(\mathcal{A})$ ist. Sei $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein endlicher Automat. Für $z \in Z$ und $z' \in Z$ setzen wir $z \approx_{\mathcal{A}} z'$ genau dann, wenn für alle $x \in X^*$ genau dann $\delta^*(z, x)$ in F liegt, wenn auch $\delta^*(z', x)$ in F liegt. Falls der Automat $\mathcal{A}$ aus dem Kontext klar ist, schreiben wir einfach $\approx$ anstelle von $\approx_{\mathcal{A}}$

Lemma 6.6 *$\approx_{\mathcal{A}}$ ist eine Äquivalenzrelation auf der Menge der Zustände des Automaten $\mathcal{A}$.*

Beweis. Wir verzichten auf den einfachen Standardbeweis. □

Mit $K_\approx(z)$ bezeichnen wir die Äquivalenzklasse von $z \in Z$ bezüglich $\approx$ $(= \approx_\mathcal{A})$. Wir setzen

$$Z_\approx = \{K_\approx(z) \mid z \in Z\} \quad \text{und} \quad F_\approx = \{K_\approx(z) \mid z \in F\}$$

und konstruieren den Automaten

$$\mathcal{A}_\approx = (X, Z_\approx, K_\approx(z_0), \delta_\approx, F_\approx)$$

mit

$$\delta_\approx(K_\approx(z), a) = K_\approx(\delta(z, a)).$$

Wir zeigen, dass die Definition von $\mathcal{A}_\approx$ korrekt ist. Dazu haben wir nachzuweisen, dass die Definition von $\delta_\approx$ unabhängig von der Auswahl des Repräsentanten der Äquivalenzklasse ist. Seien daher z und z' zwei Zustände mit $K_\approx(z) = K_\approx(z')$ und a ein beliebiges Element aus X. Dann muss $z \approx z'$ gelten. Wir betrachten ein beliebiges Wort $w \in X^*$. Dann liegt $\delta^*(z, aw)$ in F genau dann wenn $\delta^*(z', aw)$ in F liegt. Wegen $\delta^*(z, aw) = \delta^*(\delta(z, a), w)$ und $\delta^*(z', aw) = \delta^*(\delta(z', a), w)$ folgt $\delta(z, a) \approx \delta(z', a)$ und damit auch $K_\approx(\delta(z, a)) = K_\approx(\delta(z', a))$, was zu zeigen war.
Wir zeigen nun, dass $\mathcal{A}_\approx$ minimal für die von $\mathcal{A}$ akzeptierte reguläre Sprache ist.

Satz 6.7 *Für jeden Automaten $\mathcal{A}$ ist $\mathcal{A}_\approx$ minimaler Automat für $T(\mathcal{A})$.*

Beweis. Wir zeigen zuerst mittels Induktion über die Wortlänge, dass sich die Definition von $\delta_\approx : Z_\approx \times X \to Z_\approx$ auch auf die Erweiterung $\delta^*_\approx : Z_\approx \times X^* \to Z_\approx$ übertragen lässt, d. h. dass $\delta^*_\approx(K_\approx(z), y) = K_\approx(\delta^*(z, y))$ für alle $y \in X^*$ gilt. Nach (genereller) Definition der Erweiterung haben wir

$$\begin{aligned}
\delta^*_\approx(K_\approx(z), & \\
\quad lambda) &= K_\approx(z) = K_\approx(\delta^*(z, \\
lambda)), & \\
\delta^*_\approx(K_\approx(z), a) &= \delta_\approx(K_\approx(z), a) = K_\approx(\delta(z, a)) = K_\approx(\delta^*(z, a)),
\end{aligned}$$

womit der Induktionsanfang gesichert ist. Der Induktionsschluss ist durch

$$\begin{aligned}
\delta^*_\approx(K_\approx(z), ya) &= \delta_\approx(\delta^*_\approx(K_\approx(z), y), a) \\
&= \delta_\approx(K_\approx(\delta^*(z, y)), a) \\
&= K_\approx(\delta(\delta^*(z, y), a)) \\
&= K_\approx(\delta^*(z, ya))
\end{aligned}$$

gegeben.
Hieraus erhalten wir sofort

$$\begin{aligned}
T(\mathcal{A}_\approx) &= \{x \mid \delta^*_\approx(K_\approx(z_0), x) \in F_\approx\} \\
&= \{x \mid K_\approx(\delta^*(z_0, x)) \in F_\approx\} \\
&= \{x \mid \delta^*(z_0, x) \in F\} \\
&= T(\mathcal{A}).
\end{aligned}$$

Damit bleibt nur noch $z(\mathcal{A}_\approx) = z(T(\mathcal{A}))$ zu zeigen. Wegen Satz 6.5 reicht es nachzuweisen, dass $z(\mathcal{A}_\approx) = Ind(\sim_{T(\mathcal{A})})$ erfüllt ist. Dazu betrachten wir die Relation $\sim_{\mathcal{A}_\approx}$ entsprechend Beispiel 6.1, für die $z(\mathcal{A}_\approx) = Ind(\sim_{\mathcal{A}_\approx})$ gilt, und beweisen $\sim_{\mathcal{A}_\approx} = \sim_{T(\mathcal{A})}$, womit der Beweis vollständig ist.
Seien zuerst $x \in X^*$ und $y \in X^*$ zwei Wörter mit $x \not\sim_{\mathcal{A}_\approx} y$. Dann gilt $\delta^*_\approx(K_\approx(z_0), x) \neq \delta^*_\approx(K_\approx(z_0), y)$. Hieraus erhalten wir $K_\approx(\delta^*(z_0, x)) \neq K_\approx(\delta^*(z_0, y))$ und deshalb $\delta^*(z_0, x) \not\approx \delta^*(z_0, y)$. Nach der Definition von $\approx$ heißt dies, dass es ein Wort $w \in X^*$ gibt, für das entweder

$$\delta^*(\delta^*(z_0, x), w) \in F \quad \text{und} \quad \delta^*(\delta^*(z_0, y), w) \notin F$$

oder

$$\delta^*(\delta^*(z_0, x), w) \notin F \quad \text{und} \quad \delta^*(\delta^*(z_0, y), w) \in F$$

erfüllt ist. Daher gilt entweder $\delta^*(z_0, xw) \in F$ und $\delta^*(z_0, yw) \notin F$ oder $\delta^*(z_0, xw) \notin F$ und $\delta^*(z_0, yw) \in F$ und folglich entweder $xw \in T(\mathcal{A})$ und $yw \notin T(\mathcal{A})$ oder $xw \notin T(\mathcal{A})$ und $yw \in T(\mathcal{A})$. Letzteres bedeutet aber gerade $x \not\sim_{T(\mathcal{A})} y$.
Analog beweist man, dass $x \sim_{\mathcal{A}_\approx} y$ für zwei Wörter $x \in X^*$ und $y \in X^*$ auch $x \sim_{T(\mathcal{A})} y$ zur Folge hat.
Somit stimmen die Äquivalenzrelationen $\sim_{\mathcal{A}_\approx}$ und $\sim_{T(\mathcal{A})}$ überein, was zu zeigen war. □

Leider geben die bisherigen Betrachtungen keine Hinweis, wie der Index von $\sim_R$ zu gegebenem R zu ermitteln ist, denn nach Definition von $\sim_R$ sind unendlich viele Wörter zu untersuchen, um $x \sim_R y$ festzustellen. Analog ist die Konstruktion von $\mathcal{A}_\approx$ nicht algorithmisch, denn auch die Feststellung, ob $x \approx y$ gilt, erfordert die Betrachtung von unendlich vielen Wörtern.
Deshalb beschreiben wir jetzt einen Algorithmus zur Bestimmung von $\approx = \approx_{\mathcal{A}}$ für einen endlichen Automaten $\mathcal{A}$. Dies versetzt uns dann in die Lage, zu $\mathcal{A}$ den minimalen Automaten $\mathcal{A}_\approx$ zu konstruieren. Dazu beginnen wir mit einer Liste aller Paare von Zuständen und markieren jeweils ein Paar, wenn sich aus dem Verhalten der beide Zustände des Paares entsprechend der Überführungsfunktion bzw. der Menge der akzeptierenden Zustände und der aktuellen Liste ergibt, dass die beiden Zustände des Paares nicht äquivalent sind. Der *Reduktionsalgorithmus* besteht aus den folgenden Schritten:

1. Erstelle eine Liste aller Paare (z, z') von Zuständen $z \in Z$ und $z' \in Z$.

2. Streiche ein Paar (z, z') falls entweder $z \in F$ und $z' \notin F$ oder $z \notin F$ und $z' \in F$ gilt.

3. Führe die folgende Aktion solange aus, bis keine Markierungen mehr in der Liste möglich sind: Falls für ein Paar (z, z') und ein $a \in X$ das Paar $(\delta(z, a), \delta(z', a))$ nicht in der Liste ist, so streiche (z, z').

Beispiel 6.8 Wir betrachten den endlichen Automaten

$$\mathcal{A} = (\{a, b\}, \{0, 1, 2, 3, 4, 5\}, 0, \delta, \{1, 2, 5\}),$$

dessen Überführungsfunktion δ dem Zustandsgraphen aus Abb. 6.1 zu entnehmen ist. Wir

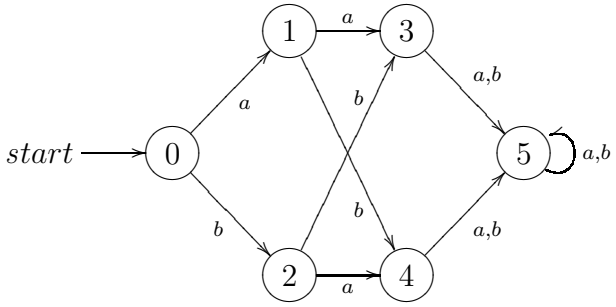


Abbildung 6.1: Zustandsgraph des Automaten $\mathcal{A}$ aus Beispiel 6.8

erhalten entsprechend Schritt 1 des Algorithmus zuerst die Liste

$$\begin{array}{llllll}(0,0), & (0,1), & (0,2), & (0,3), & (0,4), & (0,5),\\ (1,0), & (1,1), & (1,2), & (1,3), & (1,4), & (1,5),\\ (2,0), & (2,1), & (2,2), & (2,3), & (2,4), & (2,5),\\ (3,0), & (3,1), & (3,2), & (3,3), & (3,4), & (3,5),\\ (4,0), & (4,1), & (4,2), & (4,3), & (4,4), & (4,5),\\ (5,0), & (5,1), & (5,2), & (5,3), & (5,4), & (5,5).\end{array}$$

Hieraus entsteht nach Schritt 2 die Liste

$$\begin{array}{llllll}(0,0), & (0,3), & (0,4), & (1,1), & (1,2), & (1,5),\\ (2,1), & (2,2), & (2,5), & (3,0), & (3,3), & (3,4),\\ (4,0), & (4,3), & (4,4), & (5,1), & (5,2), & (5,5).\end{array}$$

Wir haben nun solange wie möglich Schritt 3 anzuwenden. Hierzu gehen wir stets die Liste durch und streichen die entsprechenden Paare und wiederholen diesen Prozess. Nach dem ersten Durchlauf streichen wir nur die Paare $(1,5)$ (da $(\delta(1,a),\delta(5,a)) = (3,5)$ nicht mehr in der Liste ist), $(2,5)$, $(5,1)$ und $(5,2)$. Beim zweiten Durchlauf werden die Paare $(0,3)$ (da das Paar $(\delta(0,a),\delta(3,a)) = (1,5)$ beim ersten Durchlauf gerade gestrichen wurde), $(0,4)$, $(3,0)$ und $(4,0)$ gestrichen. Beim dritten Durchlauf werden keine Streichungen mehr vorgenommen. Damit ergibt sich abschließend die Liste

$$(0,0),\ (1,1),\ (1,2),\ (2,1),\ (2,2),\ (3,3),\ (3,4),\ (4,3),\ (4,4),\ (5,5).$$

Wir zeigen nun, dass mittels des oben angegebenen Algorithmus wirklich die Relation $\approx_{\mathcal{A}}$ berechnet wird. Dies ist im Wesentlichen die Aussage des folgenden Lemmas.

Lemma 6.9 *Sei $\mathcal{A} = (X, Z, z_0, \delta, F)$ ein endlicher Automat, und seien $z \in Z$ und $z' \in Z$ zwei Zustände des Automaten. Dann ist das Paar (z, z') genau dann in der durch den Reduktionsalgorithmus erzeugten Liste enthalten, wenn $z \approx_{\mathcal{A}} z'$ gilt.*

Beweis. Wir beweisen mittels Induktion über die Anzahl der Schritte des Reduktionsalgorithmus die folgende äquivalente Aussage: Das Paar (z, z') wird genau dann durch den Reduktionsalgorithmus aus der Liste gestrichen, wenn es ein Wort $x \in X^*$ gibt, für das entweder $\delta^*(z, x) \in F$ und $\delta^*(z', x) \notin F$ oder $\delta^*(z, x) \notin F$ und $\delta^*(z', x) \in F$ gelten.

Im Folgenden nehmen wir stets ohne Beschränkung der Allgemeinheit an, dass der erste dieser beiden Fälle eintritt.
Erfolgt ein Streichen des Paares (z, z') im zweiten Schritt, so besitzt wegen $\delta^*(z,$ $lambda) = z$ und $\delta^*(z',$
$lambda) = z'$ das leere Wort die gewünschte Eigenschaft. Hat umgekehrt das Leerwort die Eigenschaft, so wird das Paar im zweiten Schritt gestrichen.
Das Paar (z, z') werde nun im dritten Schritt gestrichen. Dann gibt es ein Element $a \in X$ so, dass das Paar $(\delta(z,a), \delta(z',a))$ bereits früher gestrichen wurde. Nach Induktionsannahme gibt es ein Wort $x \in X^*$ mit $\delta^*(\delta(z,a),x) \in F$ und $\delta^*(\delta(z',a),x) \notin F$. Damit haben wir auch $\delta^*(z,ax) \in F$ und $\delta^*(z',ax) \notin F$, womit die Behauptung bewiesen ist. Durch Umkehrung der Schlüsse zeigt man, dass aus der Existenz eines Wortes ax mit $\delta^*(z,ax) \in F$ und $\delta^*(z',ax) \notin F$ folgt, dass (z,z') gestrichen wird. □

Beispiel 6.8 (Fortsetzung) Wir erhalten aus der bereits oben erzeugten Liste die folgenden Äquivalenzklassen bez. $\approx$:

$$K_\approx(0) = \{0\},\ K_\approx(1) = K_\approx(2) = \{1,2\},\ K_\approx(3) = K_\approx(4) = \{3,4\},\ K_\approx(5) = \{5\}.$$

Hieraus resultiert nun entsprechend obiger Konstruktion der minimale Automat

$$\mathcal{A}_\approx = (X, \{K_\approx(0), K_\approx(1), K_\approx(3), K_\approx(5)\}, K_\approx(0), \delta_\approx, \{K_\approx(1), K_\approx(5)\}),$$

dessen Überführungsfunktion aus Abb. 6.2, in der wir $[i]$ anstelle von $K_\approx(i)$, $1 \le i \le 5$, schreiben, entnommen werden kann.

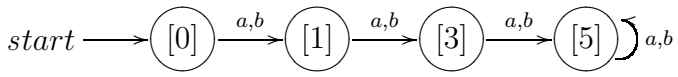


Abbildung 6.2: Zustandsgraph des minimalen Automaten zu $\mathcal{A}$ aus Beispiel 6.8

Wir bemerken, dass aus der Darstellung des minimalen Automaten sofort

$$T(\mathcal{A}) = T(\mathcal{A}_\approx) = X \cup X^3X^*$$

zu sehen ist.

Wir untersuchen nun die Komplexität des Reduktionsalgorithmus. Es gibt n^2 Paare von Zuständen. Bei Schritt 2 oder jedem Durchlauf entsprechend Schritt 3 streichen wir ein Paar oder stoppen. Das Durchmustern einer Liste der Länge r erfordert $O(r)$ Schritte. Damit ist durch

$$\sum_{i=1}^{n^2} O(i) = O\Big(\sum_{i=1}^{n^2} i\Big) = O(n^3)$$

eine obere Schranke für die Komplexität gegeben.
Wir merken hier ohne Beweis an, dass es erheblich effektivere Algorithmen zur Minimierung von endlichen Automaten gibt.

Satz 6.10 *Die Konstruktion des minimalen Automaten zu einem gegebenem endlichen Automaten mit n Zuständen ist in $O(n \cdot \log(n))$ Schritten möglich.* □

Wir wollen nun beweisen, dass der minimale Automat im Wesentlichen eindeutig bestimmt ist. Um das „im Wesentlichen“ exakt zu fassen, geben wir die folgende Definition.

Definition 6.11 *Zwei Automaten $\mathcal{A} = (X, Z, z_0, \delta, F)$ und $\mathcal{A}' = (X, Z', z_0', \delta', F')$ heißen isomorph, wenn es eine eineindeutige Funktion φ von Z auf Z' mit folgenden Eigenschaften gibt:*
– Es ist $\varphi(z_0) = z_0'$.
– Für $z \in Z$ gilt $z \in F$ genau dann, wenn auch $\varphi(z) \in F'$ gilt.
– Für alle $z \in Z$ und $a \in X$ gilt $\delta'(\varphi(z), a) = \varphi(\delta(z, a))$.

Intuitiv liegt Isomorphie zwischen zwei Automaten vor, wenn diese sich nur in der Bezeichnung der Zustände unterscheiden (und ansonsten das gleiche Verhalten zeigen).

Satz 6.12 *Sind $\mathcal{A}$ und $\mathcal{A}'$ zwei minimale Automaten für die reguläre Menge R, so sind $\mathcal{A}$ und $\mathcal{A}'$ isomorph.*

Beweis. Seien $\mathcal{A} = (X, Z, z_0, \delta, F)$ und $\mathcal{A}' = (X, Z', z_0', \delta', F')$ zwei minimale Automaten für R.
Für zwei Zustände $z \in Z$ und $z' \in Z$ von $\mathcal{A}$ ist $z \not\approx_{\mathcal{A}} z'$. Dies folgt daraus, dass aufgrund der Minimalität von $\mathcal{A}$ der Automat $\mathcal{A}_{\approx}$ genau so viel Zustände wie $\mathcal{A}$ hat. Also können keine zwei Zustände von $\mathcal{A}$ in einer Äquivalenzklasse bez. $\approx_{\mathcal{A}}$ liegen. Nun beweist man analog zum letzten Teil des Beweises von Satz 6.7, dass die Relationen $\sim_{\mathcal{A}}$ und $\sim_R$ übereinstimmen.
Entsprechend ergibt sich auch $\sim_{\mathcal{A}'} = \sim_R$ und damit $\sim_{\mathcal{A}} = \sim_{\mathcal{A}'}$.
Sei nun $z \in Z$. Dann gibt es ein Wort $x \in X^*$ mit $\delta^*(z_0, x) = z$. Wir setzen nun

$$\varphi(z) = (\delta')^*(z_0', x).$$

Diese Setzung liefert eine korrekte Definition von φ, denn der Wert von $\varphi(z)$ hängt nicht von der Wahl von x ab. Seien nämlich $x \in X^*$ und $y \in X^*$ mit $\delta^*(z_0, x) = \delta^*(z_0, y)$, so ergibt sich zuerst $x \sim_{\mathcal{A}} y$ und damit auch $x \sim_{\mathcal{A}'} y$, woraus $(\delta')^*(z_0', x) = (\delta')^*(z_0', y)$ folgt.
Wir zeigen, dass φ ein Isomorphismus ist.
Wegen $z_0 = \delta^*(z_0,$
lambda) ergibt sich $\varphi(z_0) = (\delta')^*(z_0', \lambda) = z_0'$.
Sei nun $z \in F$. Dann gibt es ein $x \in R$ mit $z = \delta^*(z_0, x)$. Dann gilt $\varphi(z) = (\delta')^*(z_0', x)$. Wegen $x \in R$ und $R = T(\mathcal{A}')$ erhalten wir $\varphi(z) \in F'$. Sei umgekehrt $\varphi(z) \in F'$. Dann liegt x mit $\varphi(z) = (\delta')^*(z_0', x)$ in R und somit ist $z = \delta(z_0, x) \in F$.
Außerdem gilt

$$\begin{aligned} \delta'(\varphi(z), a) &= \delta'((\delta')^*(z_0', x), a) = (\delta')^*(z_0', xa) \\ &= \varphi(\delta^*(z_0, xa)) = \varphi(\delta(\delta^*(z_0, x), a)) \\ &= \varphi(\delta(z, a)). \end{aligned}$$

□

Kapitel 7

Ergänzungen IV: Erweiterungen von kontextfreien Sprachen

Bei der Untersuchung von Programmiersprachen und natürlichen Sprachen wird häufig ausgenutzt, dass sie sich zu einem großen Teil durch kontextfreie Sprachen beschreiben lassen. Jedoch ist festzustellen, dass nicht alle Aspekte natürlicher Sprachen bzw. von Programmiersprachen durch kontextfreie Grammatiken beschreiben lassen. Wir geben dafür jetzt drei Beispiele.

Wir beginnen mit der englischen Sprache. Dazu betrachten wir die Sätze.

Mary	and John	are	
a woman	and a man,	respectively.	

Mary,	John	and William	are	
a woman,	a man	and a man,	respectively.	

Mary,	John,	William	and Jenny	are	
a woman,	a man,	a man	and a woman,	respectively.	

Sie sind offensichtlich von der gleichen Bauart: Zuerst wird eine Folge von Frauen- und Männernamen gegeben und dann wird hinzugesetzt, ob es sich bei der Person mit dem Namen, um eine Frau oder einen Mann handelt. Derartige Sätze beliebiger Länge sind im Englischen zugelassen (obwohl man sie praktisch natürlich höchstens mit fünf oder sechs Namen gebrauchen wird). Wir betrachten nun die Menge

$$R = \{x, \mid x \in X\}\{x, \mid x \in X\}^+\{and\}X\{are\}YY^+\{and\}Y\{respectively.\} \ ,$$

wobei X die Menge der Vornamen der englischen Sprache und $Y = \{a\ woman,\ ,\ a\ man,\}$ sind. Die Menge R beschreibt die Sätze obiger Form, ohne allerdings darauf zu achten, ob Name und Geschlecht zueinander passen und ob die Folgen der Namen und die Folge aus *a woman* und *a man* die gleiche Länge haben. Diese beiden zusätzlichen Eigenschaften sind aber bei der englischen Sprache gefordert. Folglich beschreibt die Menge $Englisch \cap R$ genau die obigen Sätze. Wir betrachten nun einen Homomorphismus h, der durch

$h(and) = h(are) = h(a) = h(respectively) = h(,) = h(.) = \lambda$
$h(woman) = a$, $h(x) = a$ für jeden Frauennamen x,
$h(man) = b$, $h(y) = b$ für jeden Männernamen y

gegeben ist. Dann erhalten wir

$$R' = h(Englisch \cap R) = \{ww \mid w \in \{a,b\}^+, |w| \geq 3\}.$$

Wir wissen, dass R' keine kontextfreie Sprache ist. Nehmen wir nun an, dass *Englisch* eine kontextfreie Sprache ist. Nach den Abschlusseigenschaften von $\mathcal{L}(CF)$ ist dann auch R' kontextfrei, was falsch ist. Somit ist unsere Annahme falsch, also ist *Englisch* keine kontextfreie Sprache.

Dieses Beispiel ist nicht ganz befriedigend, weil die Beziehung zwischen den Namen und dem Geschlecht eine semantische Relation ist, die Theorie formaler Sprachen aber nur die Syntax beschreiben kann. Wir betrachten daher die Sätze

> *Jan säit das mer em Hans hälfed.*
> (Jan sagt, dass wir Hans helfen.)
>
> *Jan säit das mer em Hans es Huus hälfed aastriche.*
> (Jan sagt, dass wir Hans helfen, das Haus zu streichen.)
>
> *Jan säit das mer d'chind em Hans es Huus lönd hälfed aastriche.*
> (Jan sagt, dass wir den Kindern erlauben, Hans zu helfen, das Haus zu streichen.)

aus einem Dialekt des Schweizerdeutschen. Auch hier liegt die gleiche Struktur vor, weil die Tätigkeiten und die zugehörigen Objekte in der gleichen Reihenfolge stehen (man beachte, dass dies bei der Übersetzung ins Deutsche nicht mehr der Fall ist). Daher ergibt sich hier wie oben

$$h'(Schweizerdeutsch \cap Q) = \{ww \mid w \in \{a,b\}^+\}$$

für einen passenden Homomorphismus h' und eine passende reguläre Sprache Q. Folglich ist Schweizerdeutsch keine kontextfreie Sprache (was sich ebenfalls analog ergibt). Hierbei haben wir die syntaktische Beziehungen Verb – Objekt ausgenutzt.

Wir kommen nun Programmiersprache ALGOL60, einer der klassische deklarativen Programmiersprachen (ähnliche Konstruktionen lassen sich aber auch für andere Programmiersprachen durchführen). Bei ALGOL60 ist es erforderlich, jede auftretende Variable zu deklarieren. Wir betrachten das Programm

```
begin integer x;
      y := 1
end
```

wobei die beiden auftretenden Variablen x und y jeweils durch ein Wort über $\{a,b\}$ gegeben seien. Damit das Programm richtig ist, muss an beiden Stellen des Auftretens einer Variablen jeweils das gleiche Wort stehen. Daher ergibt sich erneut

$$h''(ALGOL \cap S) = \{xx \mid x \in \{a,b\}^*\}$$

für eine entsprechend gewählte reguläre Menge S und einen passenden Homomorphismus h''. Damit ist auch ALGOL60 keine kontextfreie Sprache.

Zur Beschreibung von natürlichen Sprachen und Programmiersprachen bedarf es also Mechanismen, die auch gewisse nicht-kontextfreie Sprachen erzeugen können. Dies ist sicher für die kontextsensitiven Grammatiken der Fall, die aber zwei negative Aspekte haben: Einige Entscheidbarkeitsprobleme sind für kontextsensitive Grammatiken unentscheidbar bzw. sehr schwer; so sind z.B. nach Satz 5.20 das Endlichkeits- und das Leerheitsproblem unentscheidbar. Zum anderen gibt es für kontextsensitive Grammatiken keine vernünftigen Ableitungsbäume, da mehrere Nichtterminale aus verschiedenen Schichten des Baumes von einer regel betroffen sein können. Ableitungsbäume haben sich aber als sehr gutes Instrument für die Analyse von Sätzen bzw. Programmen erwiesen.
Daher ist man an Erweiterungen der kontextfreien Grammatiken interessiert, die zum einen Ableitungsbäume ermöglichen (d.h. jede angewendete Regel muss kontextfrei sein) und zum anderen möglichst einfach zu entscheidende Probleme haben. Wir präsentieren in diesem Abschnitt zwei derartigen Grammatiken, wobei dr Fokus stärker auf dem erstgenannten Aspekt der Existenz von Ableitungsbäumen liegt.

Definition 7.1 *i) Eine Grammatik mit Auswahlkontext ist ein Quadrupel $G = (N, T, P, S)$, wobei*

- *N, T, S wie bei einer Regelgrammatik spezifiziert sind,*
- *P eine endliche Menge von Tripeln $p = (r_p, E_p, F_p)$ ist, wobei jeweils $r_p = A_p \to w_p$ eine kontextfreie Regel mit $w_p \neq \lambda$ ist, und E_p und F_p Teilmengen von N mit $E_p \cap F_p = \emptyset$ sind.*

ii) Für zwei nichtleere Wörter x und y über $N \cup T$ sagen wir, dass y aus x durch Anwendung von $(A \to w, E, F)$ erzeugt wird (geschrieben als $X \Longrightarrow_p y$), wenn folgende Bedingungen erfüllt sind:
— $x = uAv$, $y = uwv$ (kontextfreie Ersetzung)
— jedes Symbol aus E kommt in uv vor,
— kein Symbol aus F kommt in uv vor.
Wir sagen, dass y aus x in G durch einen Ableitungsschritt entsteht (geschrieben als $x \Longrightarrow_G y$), wenn es eine Regel p mit $x \Longrightarrow_p y$ gibt.
iii) Die von einer Grammatik $G = (N, T, P, S)$ mit Auswahlkontext erzeugte Sprache $L(G)$ ist

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

wobei $\Longrightarrow_G^*$ der reflexive und transitive Abschluss von $\Longrightarrow_G$ ist.

Bei einer Regel $p = (A \to w, E, F)$ heißen E und F der geforderte bzw. verbotene Kontext der Regel $A \to w$
Die Definition der Grammatiken mit Auswahlkontext ist speziell darauf abgestellt, einen Zusammenhang zwischen Deklaration und Benutzung von Variablen in Programmiersprachen zu ermöglichen, d.h. den Grund dafür, dass oben im dritten Beispiel eine nicht kontextfreie Sprache auftritt, zu eliminieren. Dies geschieht dadurch, das eine Benutzung

einer Variablen entsprechend einer Regel nur möglich ist, wenn diese schon in der Satzform vorkommt, was dadurch erreicht wird, dass die Variable in der zur Regel gehörenden Menge E liegt.
Wir bemerken weiterhin, dass Grammatiken mit Auswahlkontext eine Erweiterung der kontextfreien Grammatiken sind, denn bei der Wahl von $F = \emptyset$ für jede Regel, ergibt sich eine kontextfreie Grammatik, da nun keine Bedingungen mehr für die Anwendung von Regeln vorliegen (also jede Regel jederzeit anwendbar ist); und umgekehrt kann jede kontextfreie Grammatik als eine Grammatik mit Auswahlkontext aufgefasst werden, bei der alle geforderten und verbotenen Kontexte leer sind.
Wir zeigen nun, dass Grammatiken mit Auswahlkontext Sprachen erzeugen können, die nicht kontextfrei sind.

Beispiel 7.2 Wir betrachten die Grammatik mit Auswahlkontext

$$G_1 = (\{S, A, A', A_a, A_b, B, B'\}, \{a, b, c\}, \{p_0, p_1, \ldots, p_{10}\}, S)$$

mit den Regeln

$$\begin{array}{ll}
p_0 = (S \to AB, \emptyset, \emptyset), & p_1 = (A \to aA_a, \{B\}, \emptyset), \\
p_2 = (A \to bA_b, \{B\}, \emptyset), & p_3 = (B \to aB', \{A_a\}, \emptyset), \\
p_4 = (B \to bB', \{A_b\}, \emptyset), & p_5 = (A_a \to A, \{B'\}, \emptyset), \\
p_6 = (A_b \to A, \{B'\}, \emptyset), & p_7 = (B' \to B, \{A\}, \emptyset) \\
p_8 = (A \to A', \{B\}, \emptyset), & p_9 = (B \to c, \{A'\}, \emptyset), \\
p_{10} = (A' \to c, \emptyset, \emptyset). &
\end{array}$$

Offensichtlich beginnt jede Ableitung mit der einzigen Regel für S, d.h. wir erhalten $S \Longrightarrow AB$. Wir wollen etwas allgemeiner von einer Satzform $wAwB$ ausgehen (das in einem Schritt erhaltene Wort AB wird gerade bei $w = \lambda$ erhalten). Wir können keine der Regeln mit linker Seite B anwenden, da der jeweilige geforderte Kontext A_a bzw. A_b bzw.A' in der Satzform nicht vorhanden ist. Folglich muss eine Regel mit linker Seite A verwendet werden. Wir unterscheiden drei Fälle.

Fall 1. Es wird die Regel p_1 angewendet. Dadurch erhalten wir waA_awB. Da die einzige Regel für A_a den geforderten Kontext B', ist eine Regel für B anzuwenden. Es kommt nur Regel p_3 in Frage, da bei den anderen Regeln wieder der geforderte Kontext fehlt. Daher ergibt sich waA_awaB'. Jetzt ist nur Regel p_5 anwendbar, wodurch $waAwaB'$ entsteht. Nun ist Regel p_7 anzuwenden, und wir erhalten $waAwaB$. Damit haben wir wieder die Ausgangssituation erhalten, nur dass das Wort an beiden Stellen um den Buchstaben a verlängert wurde.

Fall 2. Es wird die Regel p_2 angewendet. Dann ergibt sich analog zu Fall 1 als einzig mögliche Ableitung

$$wAwB \Longrightarrow_{p_2} wbA_bwB \Longrightarrow_{p_4} wbA_bwbB' \Longrightarrow_{p_6} wbAwbB' \Longrightarrow_{p_7} wbAwbB,$$

d.h. wir haben das Wort an beiden Stellen um den Buchstaben b verlängert.

Fall 3. Es wird die Regel p_8 angewendet. Dann ergibt sich die eindeutige Ableitung

$$wAwB \Longrightarrow_{p_8} wA'wB \Longrightarrow_{p_9} wA'wc \Longrightarrow_{p_{10}} wcwc$$

(man beachte, dass die Anwendung von p_{10} auf $wA'wB$ zwar möglich ist, aber $wcwB$ liefert, worauf keine Regel mehr anwendbar ist und somit kein terminales Wort erreicht werden kann).
Aufgrund der drei Fälle ist sofort zu sehen, dass

$$L(G_1) = \{wcwc \mid w \in \{a,b\}^*\}$$

gilt.
Wir machen darauf aufmerksam, dass in allen Regeln von G_1 kein verbotener Kontext vorkommt.

Beispiel 7.3 Es sei die Grammatik mit Auswahlkontext

$$G_2 = (\{S, A, A', B, C, D\}, \{a\}, \{p_0, p_1, \ldots, p_8\}, S)$$

mit den Regeln

$$\begin{array}{lll} p_0 = (S \to CA, \emptyset, \emptyset), & p_1 = (S \to BA, \emptyset, \emptyset), & p_2 = (A \to a, \{C\}, \emptyset), \\ p_3 = (C \to a, \emptyset, \{A, A'\}), & p_4 = (A \to A'A', \{B\}, \emptyset), & p_5 = (B \to D, \emptyset\{A\}), \\ p_6 = (A' \to A, \{D\}, \emptyset), & p_7 = (D \to B, \emptyset, \{A'\}), & p_8 = (D \to C, \emptyset, \{A'\}) \end{array}$$

gegeben. Zu Beginn ist eine der Regeln p_0 oder p_1 anzuwenden. Hierdurch entstehen CA und BA. Wir betrachten erneut allgemeiner die Situation, dass die Satzform CA^{2^n} oder BA^{2^n} für eine nicht-negative ganze Zahl n vorliegt (CA und BA entsprechen $n = 0$). Sei zuerst CA^{2^n} gegeben. Die einzige Regel für C ist nicht anwendbar, da A zum verbotenen Kontext gehört. Die Regel p_4 ist ebenfalls nicht anwendbar, da bei ihr der Kontext B gefordert wird. Folglich ist nur p_2 anwendbar, wodurch CA^raA^s mit $r + s = 2^n - 1$ entsteht. Solange noch ein A und kein B vorhanden sind, sind p_3 und p_4 nicht anwendbar. Damit erhalten wir die Ableitung

$$CA^{2^n} \Longrightarrow_{p_2} CA^raA^s \Longrightarrow_{p_2} CA^naA^maA^k \Longrightarrow_{p_2} \ldots \Longrightarrow_{p_2} Ca^{2^n}.$$

Nun ist nur p_3 anwendbar und wir terminieren mit $aa^{2^n} = a^{2^n+1}$.
Starten wir mit BA^{2^n}, so erhalten wir durch analoge Überlegungen die eindeutige Ableitung

$$\begin{array}{lll} BA^{2^n} & \Longrightarrow_{p_4} & BA^raA'A'A^s \Longrightarrow_{p_4} BA^nA'A'A^mA'A'A^k \Longrightarrow_{p_4} \ldots \Longrightarrow_{p_4} B(A'A')^{2^n} = B(A')^{2^{n+1}} \\ & \Longrightarrow_{p_5} & D(A')^{2^{n+1}} \Longrightarrow_{p_6} D^(A')^pA(A')^q \Longrightarrow_{p_6} \ldots \Longrightarrow_{p_6} DA^{2^{n+1}} \end{array}$$

Jetzt gibt es zwei Fortsetzung mittels der regeln p_7 oder p_8, wodurch $CA^{2^{n+1}}$ und $BA^{2^{n+1}}$ entstehen. Damit erhalten wir wieder unsere Ausgangssituation nur mit einer um Eins erhöhten Zweierpotenz. Hieraus folgt nun sofort

$$L(G_2) = \{a^{2^n+1} \mid n \geq 0\}.$$

Wir merken ohne Beweis an, dass die Erzeugung von $L(G_2)$ verbotene Kontexte erfordert.

Wir definieren nun die zweite Art von Grammatiken, die zu einer Erweiterung der kontextfreien Sprachen führt.

Definition 7.4 *i) Eine programmierte Grammatik ist eine Quintupel* $G = (N, T, Lab, P, S)$, *wobei*

- N, T, S *wie bei einer Regelgrammatik spezifiziert sind,*
- Lab *eine endliche Menge von Labels ist,*
- P *eine endliche Menge von Regeln* $p = (l_p, A_p \to w_p, \sigma_p, \varphi_p)$ *ist, wobei jeweils* $l_p \in Lab$, $A_p \in N$, $w_p \in (N \cup T)^+$, $\sigma_p \subseteq Lab$ *und* $\varphi_p \subseteq Lab$ *gelten, und*
- *für zwei verschiedene Regeln* p *und* q *die zugehörigen Labels* l_p *und* l_q *auch verschieden sind.*

ii) Die von einer programmierten Grammatik $G = (N, T, Lab, P, S)$ *erzeugte Sprache besteht aus allen Wörtern* $w \in T^*$, *für die es eine Ableitung*

$$S = w_0 \Longrightarrow_{p_1} w_1 \Longrightarrow_{p_2} w_2 \Longrightarrow_{p_3} \ldots \Longrightarrow_{p_k} w_k = w,$$

mit $k \geq 1$ *so gibt, dass für* $1 \leq i \leq k$ *mit* $p_i = (l_i, A_i \to v_i, \sigma_i, \varphi_i)$, *entweder*

$$w_{i-1} = w'_{i-1} A_i w''_{i-1},\ w_i = w'_{i-1} v_i w''_{i-1} \text{ für gewisse } w'_{i-1}, w''_{i-1} \in V_G^*,\ l_{i+1} \in \sigma_i$$

oder

$$A_i \text{ kommt in } w_{i-1} \text{ nicht vor, } w_{i-1} = w_i,\ l_{i+1} \in \varphi_i.$$

gilt.

Die Mengen σ_p und φ_p einer Regel p heißen Erfolgs- bzw. Fehlerfeld. Ist die Regel p anwendbar (hatte ihre Anwendung Erfolg), so wird mit einer Regel fortgesetzt, deren Label im Erfolgsfeld σ_p liegt; ist die Regel dagegen nicht anwendbar, so wird ohne Änderung der Satzform mit einer Regel fortgesetzt, deren Label im Fehlerfeld φ_p liegt. Mit einer Regel ist also immer eine Angabe der möglichen nächsten Regeln verbunden.
Erneut ist einfach zu sehen, dass programmierte Grammatiken eine Erweiterung der kontextfreien Grammatiken darstellen. Die kontextfreien Grammatiken ergeben sich gerade durch die Festsetzung $\sigma_p = \varphi_p = Lab$, da dann keine Steuerung der Reihenfolge der Regeln mehr gegeben ist, sondern jede Regel nach jeder Regel angewendet werden kann, wie es bei den kontextfreien Grammatiken der Fall ist.

Beispiel 7.5 Wir betrachten die programmierte Grammatik

$$G'_1 = (\{S, A, B\}, \{a, b\}, \{q_0, q_1, \ldots, q_8\}, \{r_0, r_1, r_2, \ldots, r_8\}, S)$$

mit

$$\begin{array}{ll}
r_0 = (q_0, S \to AB, \{q_1, q_3, q_5, q_7\}, \emptyset), & \\
r_1 = (q_1, A \to aA, \{q_2\}, \emptyset), & r_2 = (q_2, B \to aB, \{q_1, q_3, q_5, q_7\}, \emptyset), \\
r_3 = (q_3, A \to bA, \{q_4\}, \emptyset), & r_4 = (q_4, B \to bB, \{q_1, q_3, q_5, q_7\}, \emptyset), \\
r_5 = (q_5, A \to a, \{q_6\}, \emptyset), & r_6 = (q_6, B \to a, \emptyset, \emptyset), \\
r_7 = (q_7, A \to b, \{q_8\}, \emptyset), & r_8 = (q_8, B \to b, \emptyset, \emptyset).
\end{array}$$

Jede Ableitung in G beginnt mit einer Anwendung von r_0, wodurch die Satzform AB entsteht. Da r_0 anwendbar war, sind im nächsten Schritt die Regeln r_1, r_3, r_5 und r_7 anwendbar.
Wir diskutieren nun die möglichen Ableitungen einer Satzform $wAwB$, wobei wir davon ausgehen, dass darauf die Regeln r_1, r_3, r_5 und r_7 im nächsten Schritt anwendbar sind. Dann ergeben sich die folgenden Ableitungen

$$\begin{aligned}
&wAwB \Longrightarrow_{r_1} waAwB \Longrightarrow_{r_2} waAwaB,\\
&wAwB \Longrightarrow_{r_3} wbAwB \Longrightarrow_{r_4} wbAwbB,\\
&wAwB \Longrightarrow_{r_5} wawB \Longrightarrow_{r_6} wawa,\\
&wAwB \Longrightarrow_{r_7} wbwB \Longrightarrow_{r_8} wbwb.
\end{aligned}$$

In allen Fällen wird das vorhandene Wort w an beiden Stellen seines Auftretens um jeweils den gleichen Buchstaben verlängert. In den ersten beiden Fällen kann die Ableitung erneut mit den Regeln r_1, r_3, r_5 und r_7 fortgesetzt werden; in den beiden letzten Fällen ist ein terminales Wort erreicht und die Ableitung beendet. Damit ergibt sich

$$L(G_1') = \{ww \mid w \in \{a,b\}^+\}.$$

Beispiel 7.6 Es sei die programmierte Grammatik

$$G_2' = (\{S,A\}, \{a\}, \{q_1,q_2,q_3\}, \{r_1,r_2,r_3\}, S)$$

mit den Regeln

$$r_1 = (q_1, S \to AA, \{q_1\}, \{q_2\}),\ r_2 = (q_2, A \to S, \{q_2\}, \{q_1,q_3\}), r_3 = (q_3, S \to a, \{q_3\}, \emptyset)$$

gegeben. Auf das Startsymbol S sind die Regeln r_1 und r_3 anwendbar. Wir betrachten die Situation, dass auf eine Satzform S^n die Regeln r_1 und r_3 anwendbar sind. Wenn wir r_3 anwenden, so haben wir das immer wieder zu tun, da das Erfolgsfeld von r_3 nur den Label q_3 enthält, d.h. wir haben der Reihe nach alle vorkommenden S jeweils durch a ersetzt. Dies liefert

$$S^n \Longrightarrow_{r_3} S^p a S^q \Longrightarrow_{r_3} \ldots \Longrightarrow_{r_3} a^n.$$

Damit hat die Ableitung terminiert. Wenden wir dagegen r_1 an, so ist dies weiterhin zu tun, bis alle S durch AA ersetzt sind. Die erneute Anwendung von r_3 schlägt jetzt fehl, so dass mit r_2 fortzusetzen ist. Diese Regel ist wiederum solange anzuwenden bis alle A wieder durch S ersetzt sind. Formal ergibt sich also

$$S^n \Longrightarrow_{r_1} S^p AAS^q \Longrightarrow_{r_1} \ldots \Longrightarrow_{r_1} (AA)^n = A^{2n} \Longrightarrow_{r_2} A^s S A^t \Longrightarrow_{r_2} \ldots \Longrightarrow_{r_2} S^{2n}.$$

Eine Fortsetzung muss nun durch r_1 oder r_3 erfolgen, d.h. wir haben die gleiche Situation erreicht, aber die Anzahl der Ss verdoppelt. Daher ergibt sich

$$L(G_2') = \{a^{2^n} \mid n \geq 0\}.$$

Mit $\mathcal{L}(P)$ und $\mathcal{L}(RC)$ bezeichnen wir die Mengen der von programmierten Grammatiken bzw. von Grammatiken mit Auswahlkontext erzeugten Sprachen. Wir vergleichen zuerst die beiden Sprachmengen.

Lemma 7.7 $\mathcal{L}(RC) \subseteq \mathcal{L}(P)$.

Beweis. Es sei $L \in \mathcal{L}(RC)$. Dann gibt es eine Grammatik $G = (N, T, P, S)$ mit Auswahlkontext so, dass $L = L(G)$ gilt. Wir setzen zuerst

$$N' = N \cup \{A' \mid A \in N\}.$$

Es sei $p = (A \to w, \{A_1, A_2, \ldots, A_r\}, \{B_1, B_2, \ldots, B_s\})$ eine Regel aus P. Diese ist auf eine Satzform xAy nur anwendbar, wenn xy alle Symbole $A_1, A_2 \ldots, A_r$ und keinen der Buchstaben $B_1, B_2, \ldots, B_s$ enthält; die Anwendung liefert dann xwy. Für p führen wir die Labels (p, i) mit $1 \leq i \leq r + s + 2$ und die zugehörigen Regeln

$$\begin{aligned}
r_{p,1} &= ((p,1), A \to A', \{(p,2)\}, \emptyset),\\
r_{p,2} &= ((p,2), A_1 \to A_1, \{(p,3)\}, \emptyset),\\
r_{p,3} &= ((p,2), A_2 \to A_2, \{(p,4)\}, \emptyset),\\
\ldots & \qquad\qquad \ldots\\
r_{p,r} &= ((p,r-1), A_{r-1} \to A_{r-1}, \{(p,r+1)\}, \emptyset),\\
r_{p,r+1} &= ((p,r+1), A_r \to A_r, \{(p,r+2)\}, \emptyset),\\
r_{p,r+2} &= ((p,r+2), B_1 \to B_1, \emptyset, \{(p,r+3)\}),\\
r_{p,r+3} &= ((p,r+3), B_2 \to B_2, \emptyset, \{(p,r+4)\}),\\
\ldots & \qquad\qquad \ldots\\
r_{p,r+s} &= ((p,r+s), B_{s-1} \to B_{s-1}, \emptyset, \{(p,r+s+1)\}),\\
r_{p,r+s+1} &= ((p,r+s+1), B_s \to B_s, \emptyset, \{(p,r+s+2)\}),\\
r_{p,r+s+2} &= ((p,r+s+2), A' \to w, \{(p,1) \mid p \in P\}, \emptyset)
\end{aligned}$$

ein. Durch die Regel $r_{p,i+1}$ testen wir, ob A_i in der Satzform vorhanden ist, denn $A_i \to A_i$ ist nur dann anwendbar, ändert aber die Satzform nicht. Ist A_i nicht vorhanden, kann die Ableitung nicht fortgesetzt werden, da das Fehlerfeld von $r_{p,i+1}$ leer ist. Analog testen wir durch die Regeln $r_{p,r+s+j+1}$ ob B_j in der Satzform vorkommt, jedoch ist jetzt nur eine Fortsetzung möglich, wenn B_j nicht vorhanden ist, da das Erfolgsfeld leer ist. Weiterhin sind die Regeln genau in der gegebenen Reihenfolge anzuwenden, da die Erfolgs- bzw. Fehlerfelder genau eine Nachfolgeregel spezifizieren. Da die Regeln $r_{p,2} - r_{p,r+s+1}$ keine Veränderung der Satzform bewirken, wird nur entsprechend $r_{p,1}$ und $r_{p,r+s+2}$ eine Veränderung erreicht, die in $xAy \Longrightarrow xA'y \Longrightarrow xwy$ resultieren. Damit wird die Satzform erreicht, die auch bei Anwendung von p entsteht und die Anwendbarkeitsbedingungen für p wurden auch getestet. Das Nichtterminal A' wurde eingeführt, damit es beim Testen der Vorkommen von $A_1, A_2, \ldots, A_r, B_1, B_2, \ldots, B_s$ nicht berücksichtigt wird.
Wir konstruieren die programmierte Grammatik $G' = (N', T, Lab, P', S)$, wobei Lab und P' aus allen Labels und Regeln bestehen, die sich entsprechend obigem Verfahren ergeben. Es ist nun leicht einzusehen, dass die Ableitung $xAy \Longrightarrow_p xwy$ in G genau dann existiert, wenn wir in G' die Ableitung $xAy \Longrightarrow_{r_{p,1}} xA'y \Longrightarrow_{r_{p,2}} \ldots \Longrightarrow_{r_{p,r+s+2}} xwy$ gibt. Daher erzeugen beide Grammatiken die gleiche Sprache. Somit haben wir $L = L(G) = L(G') \in \mathcal{L}(P)$. □

Für den Beweis der Umkehrung benötigen wir das folgende Lemma.

Lemma 7.8 *Es seien $L \in \mathcal{L}(P)$ eine Sprache mit $L \subseteq T^*$ und $a \in T$ gegeben. Wenn die Menge*

$$L_a = \{w \mid aw \in L\} \in \mathcal{L}(P)$$

nicht nur aus dem Leerwort besteht, so liegt sie in $\mathcal{L}(P)$.

Beweis. Es sei L eine Sprache aus $\mathcal{L}(P)$. Dann gibt es eine programmierte Grammatik $G = (N, T, Lab, P, S)$, die L erzeugt. Ferner sei

$$q = \max\{|w| \mid (l, A \to w, \sigma, \varphi) \in P\}.$$

Wenn $q = 1$ gilt, so werden nur Wörter der Länge 1 erzeugt. Daher ist L_a entweder leer (wenn a nicht erzeugt wird), oder es ist $L_a = \{\lambda\}$. Im ersten Fall wird L_a von der programmierten Grammatik $(\{S\}, T, \{l\}, \{(l, S \to S, \{l\}, \emptyset)\}, S)$ erzeugt. Der zweite Fall braucht wegen der Voraussetzungen des Lemmas nicht betrachtet zu werden. Wir können daher im Folgenden annehmen, dass $q \geq 2$ gilt.
Sei nun w eine Satzform der Länge k mit $q + 1 \leq k \leq 2q$. Dann gibt es eine Ableitung

$$D : S \Longrightarrow_{p_1} w_1 \Longrightarrow_{p_2} w_2 \Longrightarrow_{p_3} w_3 \Longrightarrow_{p_4} \ldots \Longrightarrow_{p_n} w_n = w.$$

Wir definieren dann $s(w, D)$ und $v(w, D)$ als das Erfolgs- bzw. Fehlerfeld von p_n. Offensichtlich gibt es nur endliche viele w und für jedes w nur endlich viele verschiedene Ableitungen.
Wir setzen zuerst

$$N' = N \cup \{(x, y) \mid x, y \in N \cup T\} \cup \{S'\}$$

(die neuen Buchstaben (x, y) stehen für ein Wort der Länge 2 und werden im Folgenden anstelle des Anfangsstückes der Länge 2 verwendet). Für eine Regel $(l, A \to x_1 x_2 \ldots x_n, \sigma, \varphi)$ aus P konstruieren wir die Regeln

$$(l, A \to x_1 x_2 \ldots x_n, \bigcup_{k \in \sigma} \{k, k', k''\} \cup \{u_x \mid x \in T\}, \bigcup_{t \in \varphi} \{t, t', t''\} \cup \{u_x \mid x \in T\}),$$

$$(l', (y, A) \to (y, x_1) x_2 x_3 \ldots x_n, \bigcup_{k \in \sigma} \{k, k', k''\} \cup \{u_x \mid x \in T\}, \bigcup_{t \in \varphi} \{t, t', t''\} \cup \{u_x \mid x \in T\}),$$

$$(l'', (A, y) \to (x_1, y), \bigcup_{k \in \sigma} \{k, k', k''\} \cup \{u_x \mid x \in T\}, \bigcup_{t \in \varphi} \{t, t', t''\} \cup \{u_x \mid x \in T\})$$

für $n = 1$,

$$(l'', (A, y) \to (x_1, x_2) x_3 \ldots x_n y, \bigcup_{k \in \sigma} \{k, k', k''\} \cup \{u_x \mid x \in T\}, \bigcup_{t \in \varphi} \{t, t', t''\} \cup \{u_x \mid x \in T\})$$

für $n \geq 2$.

Durch Regeln mit den Labels l' und l'' wird abgesichert, dass auch eine Anwendung auf den beiden ersten gekoppelten Buchstaben möglich wird. Außerdem benutzen wir Regeln der Form

$((w), S' \to w, \emptyset, \emptyset)$ für $w \in L, |w| \leq q$,

$$((w, D), S' \to w, \bigcup_{k \in s(w,D)} \{k, k', k''\} \cup \{u_x \mid x \in T\}, \bigcup_{t \in v(w,D)} \{t, t', t''\} \cup \{u_x \mid x \in T\})$$

für eine Satzform w von G mit $q + 1 \leq |w| \leq 2q$ und eine Ableitung D von w,

$(u_x, (a, x) \to x, \emptyset, \emptyset)$ für $x \in T$.

Durch die Regeln der beiden erstgenannten Formen werden die erforderlichen kurzen Wörter der Sprache bzw. Satzformen der Grammatik G direkt erzeugt. Durch die Anwendung der letztgenannten Regeln wird der erste Buchstabe a gestrichen, wodurch ein Wort aus L_a erzeugt wird; ist der erste Buchstabe nicht a, so kann der gekoppelte Buchstabe nicht eliminiert werden, d.h., die Ableitung kann nicht terminieren; falls eine solche Regel angewendet wird, so stoppt die Ableitung, da Erfolgs- und Fehlerfeld leer sind, d.h. eine solche Regel kann nur als letzte der Ableitung angewendet werden.
Nach diesen Ausführungen ist leicht zu sehen, dass für die programmierte Grammatik

$$G' = (N', T, \bigcup_{l \in Lab} \{l, l', l''\}, P', S'),$$

bei der P' aus allen oben konstruierten Regeln besteht, $L(G') = L_a$ gilt. □

Lemma 7.9 $\mathcal{L}(P) \subseteq \mathcal{L}(RC)$.

Beweis. Es sei L eine Sprache aus $\mathcal{L}(P)$ über dem Alphabet V. Für $a \in V$ setzen wir $L_a = \{w \mid aw \in L\}$. Dann gilt

$$L = \bigcup_{a \in V} aL_a.$$

Wir werden nun zeigen, dass für jedes $a \in V$ die Sprache aL_a in $\mathcal{L}(RC)$ liegt. DA es leicht zu zeigen ist, dass $\mathcal{L}(RC)$ unter Vereinigung abgeschlossen ist (der Standardbeweis aus Abschnitt 5 ist nur leicht zu modifizieren), ist dann auch L in $\mathcal{L}(RC)$.
Nach Lemma 7.8 wird L_a von einer programmierten Grammatik $G = (N, T, Lab, P, S)$ erzeugt. Dabei sei p_l stets die Regel mit Label l. Wir setzen

$$G' = (N \cup \{A_l \mid A \in N, l \in Lab\} \cup Lab \cup \{S'\}, T, P', S'),$$

wobei P' aus allen Regeln besteht, die wie folgt aussehen: Für das neue Startsymbol S' gibt es die Regeln

$(S' \to lS, \emptyset, \emptyset)$, wobei l Label einer Regel mit linker Seite S ist

(mit einer solchen Regel beginnt die Ableitung und es soll anschließend die Anwendung der Regel $(l, S \to v_l, \sigma(l), \varphi(l))$ in G simuliert werden). Für jede Regel $(l, A \to w, \sigma(l), \varphi(l)) \in P$ führen wir die Regeln

$$\begin{aligned}
&(A \to A_l, \{l\}\{B_k \mid B \in N, k \in Lab\}),\\
&(l \in l', \{A_l\}, \emptyset,) \text{ für alle } l' \in \sigma(l),\\
&(A_l \to w, \emptyset, \{l\}),\\
&(l \to l', \emptyset, \{A\}) \text{ für alle } l' \in \varphi(l),\\
&(l \to a, \emptyset, N \cup \{B_k \mid B \in N, k \in Lab\})
\end{aligned}$$

ein (liegt die Satzform $luAv$ vor, so ist nur die Ableitung $luAv \Longrightarrow luA_lv \Longrightarrow l'uA_lv \Longrightarrow l'uwv$ möglich, d.h. ein Ableitungsschritt aus G wurde simuliert und danach ist sowohl in G als auch in G' mit einer Regel $l' \in \sigma(l)$ fortzusetzen, kommt A in der Satzform lw nicht vor so ist nur $(l \to l', \emptyset, \{A\})$ anwendbar, wodurch $l'w$ entsteht, d.h. wir haben erneut

einen Ableitungschritt aus G simuliert; die Regel $l \to a$ ist nur als letzte anwendbar, da sonst kein anderes Nichtterminal mehr vorhanden sein darf).
Aus diesen Erklärungen folgt, dass eine Ableitung

$$S \Longrightarrow_{p_{l_1}} w_1 \Longrightarrow_{p_{l_2}} w_2 \Longrightarrow_{p_{l_3}} w_3 \Longrightarrow_{p_{l_4}} \ldots \Longrightarrow_{p_{l_n}} w_n = z \in L(G) = L_a$$

in G genau dann existiert, wenn in G' eine Ableitung der Form

$$S' \Longrightarrow l_1 S \Longrightarrow l_2 w_1 \Longrightarrow l_3 w_2 \Longrightarrow l_4 w_3 \Longrightarrow \ldots \Longrightarrow l_n w_{n-1} \Longrightarrow l_{n+1} w_n \Longrightarrow a w_n = az$$

existiert. Damit folgt $L(G') = aL_a$. □

Satz 7.10 $\mathcal{L}(CF) \subset \mathcal{L}(RC) = \mathcal{L}(P) \subset \mathcal{L}(CS)$.

Beweis. $\mathcal{L}(CF) \subseteq \mathcal{L}(RC)$ ergibt sich einfach daraus, dass jede kontextfreie Grammatik als eine Grammatik mit Auswahlkontext aufgefasst werden kann, wir wie oben festgestellt haben. Die Echtheit der Inklusion folgt aus den Beispielen 7.2 und 7.3. Die Gleichheit $\mathcal{L}(RC) = \mathcal{L}(P)$ folgt aus den Lemmata 7.7 und 7.8.
Der Beweis der Inklusion $\mathcal{L}(RC) \subseteq \mathcal{L}(CS)$ ist leicht dadurch zu erbringen, dass eine kontextsensitive Grammatik erzeugt wird, bei der stets die Anwendbarkeit in Analogie zum Beweis von Lemma 7.7 getestet wird (durch Regeln $XA \to AX$ bewegt sich ein Symbol von links nach rechts über die Satzform und merkt sich dabei die vorkommenden Nichtterminale). Den Beweis der Echtheit der Inklusion geben wir hier nicht. □

Damit haben wir eines der eingangs formulierten Ziele erreicht; wir haben eine Sprachfamilie, die echt zwischen denen der kontextfreien und kontextsensitiven Sprachen liegt. Wir merken noch an, dass auch hinsichtlich der Entscheidbarkeits- bzw. Komplexitätsfragen eine verbesserte Situation vorliegt, denn es gelten die folgenden Aussagen, auf deren Beweis wir hier verzichten.

Satz 7.11 *i)* $\mathcal{L}(P)$ *ist eine AFL.*
ii) Das Mitgliedsproblem für programmierte Grammatiken is **NP***-vollständig.*
iii) Das Leerheitsproblem für programmierte Grammatiken, bei denen jedes Fehlerfeld leer ist, ist entscheidbar. □

Kapitel 8

Ergänzungen V : Eindeutigkeit kontextfreier Grammatiken

Ein Problem, das bei natürlicher Sprachen, Programmiersprachen und formalen Sprachen gleichermaßen störend ist, ist die Mehrdeutigkeit von Sätzen. Als ein Beispiel aus der deutschen Sprachen nehmen wir den Satz

Er öffnete die Kiste mit dem Hammer.

Hier ist nicht klar, ob der Hammer zum Öffnen benutzt wird oder ob der Hammer zur Kiste gehört. Ein bekanntes Beispiel der englischen Sprache ist der Satz

They are flying machines.

Welche Bedeutung hier gemeint ist kann beispielsweise von der Ableitung dieses Satzes gewonnen werden. Wir haben die folgenden beiden Ableitungen:

$$\begin{aligned}
\text{Satz} &\Longrightarrow \text{Subjekt Prädikat Objekt .}\\
&\Longrightarrow \text{They Prädikat Objekt .}\\
&\Longrightarrow \text{They (Verlaufsform eines Verbs) Objekt .}\\
&\Longrightarrow \text{They are flying Objekt .}\\
&\Longrightarrow \text{They are flying machines .}
\end{aligned}$$

und

$$\begin{aligned}
\text{Satz} &\Longrightarrow \text{Subjekt Prädikat Objekt .}\\
&\Longrightarrow \text{They Prädikat Objekt .}\\
&\Longrightarrow \text{They are Objekt .}\\
&\Longrightarrow \text{They are Adjektiv Substantiv .}\\
&\Longrightarrow \text{They are flying Substantiv .}\\
&\Longrightarrow \text{They are flying machines .}
\end{aligned}$$

Um Eindeutigkeit zu erreichen scheint es daher sinnvoll zu sein, dass die Ableitung eindeutig ist. In dieser Allgemeinheit ist die Forderung aber unbrauchbar, da in einer kontextfreien Grammatik keine Beschränkung hinsichtlich des Buchstaben, auf den die nächste Regel angewendet wird, gegeben ist. Dies ändert sich aber, wenn wir uns auf Linksableitungen beschränken, bei denen immer das am weitesten links stehende Nichtterminal in der Satzform zu ersetzen ist. Die beiden oben angegebenen Ableitungen des englischen Satzes sind beide Linksableitungen, aber sie sind verschieden.

Definition 8.1 *Eine kontextfreie Grammatik G heißt eindeutig, wenn für jedes Wort aus $w \in L(G)$ genau eine Linksableitung bez. G existiert. Anderenfalls heißt G mehrdeutig.*

Es ist offensichtlich, dass dies äquivalent zu der Forderung ist, dass es für jedes Wort w aus $L(G)$ genau einen Ableitungsbaum gibt, da ein jeder Ableitungsbaum auch Ableitungsbaum einer Linksableitung ist.
Wir betrachten zwei Beispiele.

Beispiel 8.2 Die Grammatik $GH_1 = (\{S\}, \{a, b\}, \{S \to aSb, S \to ab\}, S)$ (die $\{a^n b^n \mid n \geq 1\}$ erzeugt), ist offensichtlich eindeutig, denn zuerst ist die erste Regel eine beliebige Anzahl von Malen anzuwenden und zum Schluss die zweite Regel genau einmal.

Beispiel 8.3 Wir betrachten die lineare Grammatik

$$H_2 = (\{S\}, \{a, b\}, \{S \to aSa, S \to aS, S \to b\}, S),$$

die die Sprache

$$L(H_2) = \{a^n b a^m \mid n \geq m \geq 0\}$$

erzeugt. Die Grammatik H_2 ist mehrdeutig, denn für das Wort a^2ba haben wir die Linksableitungen

$$S \Longrightarrow aSa \Longrightarrow aaSa \Longrightarrow aaca \text{ und } S \Longrightarrow aS \Longrightarrow aaSa \Longrightarrow aaca,$$

die voneinander verschieden sind.

Der folgende Satz, dessen Beweis (durch Reduktion auf das Postsche Korrespondenzproblem) wir dem Leser als Übungsaufgabe überlassen, zeigt, dass Eindeutigkeit kein einfaches Konzept für Grammatiken ist.

Satz 8.4 *Es ist algorithmisch unentscheidbar, ob eine gegebene kontextfreie Grammatik eindeutig oder mehrdeutig ist.* □

Wir übertragen nun den Begriff der Eindeutigkeit auf Sprachen.

Definition 8.5 *Eine (kontextfreie Sprache Sprache L heißt eindeutig, wenn es eine eindeutige kontextfreie Grammatik G mit $L = L(G)$ gibt. Ansonsten heißt eine kontextfreie Sprache mehrdeutig.*

Eindeutige Sprachen sind also diejenigen, die von eindeutigen Grammatiken erzeugt werden. Es ist aber nicht ausgeschlossen, dass es auch eine mehrdeutige Grammatik gibt, die die eindeutige Sprache erzeugt (siehe das folgende Beispiel). Für mehrdeutige Sprachen gibt es keine sie erzeugende eindeutige Grammatiken, d.h. alle die Sprache erzeugenden Grammatiken sind mehrdeutig.

Beispiel 8.6 Die Sprache $L(H_2)$ aus Beispiel 8.3 ist eindeutig. Wir müssen also zeigen, dass es außer der mehrdeutigen Grammatik H_2 noch eine eindeutige Grammatik H_3 gibt, die auch L erzeugt. Dies leistet z.B. die Grammatik

$$H_3 = (\{S, A\}, \{a, b\}, \{S \to aSa, S \to A, A \to aA, A \to b\}, S).$$

Die Grammatik ist eindeutig, da bei jeder Ableitung in H_3 zuerst die Regel $S \to aSA$ eine gewisse Anzahl von Malen angewendet wird (sagen wir m mal), dann eine Anwendung von $S \to A$ folgt, worauf $A \to aA$ wieder eine beliebige Anzahl von Malen verwendet wird (sagen wir n mal), um dann mit $A \to b$ zu terminieren. Offensichtlich wird so $a^n a^m b a^m = a^{n+m} b a^m$ erzeugt. Hiermit ist dann auch $L(H_3) = L(H_2)$ gezeigt.

Eigentlich ist es aufgrund der Definition 8.5 nicht klar, dass es eine mehrdeutige Sprache gibt (denn es könnte sein, dass jede kontextfreie Sprache durch eine eindeutige Grammatik erzeugbar ist. Wir wollen nun zeigen, dass es mehrdeutige kontextfreie Sprachen gibt. Für den Beweis brauchen wir die folgende Normalform für eindeutige Grammatiken.

Definition 8.7 *i) Für eine kontextfreie Grammatik $G = (N, T, P, S)$ definieren wir die Menge $U(G)$ durch*

$$U(G) = \{A \mid A \in N,\ A \Longrightarrow^* xAy \text{ für gewisse } x, y \in T^* \text{ mit } xy \neq \lambda\}.$$

ii) Eine kontextfreie Grammatik $G = (N, T, P, S)$ heißt selbstzyklisch, wenn sie folgende Bedingungen erfüllt:

1. *Für jedes Nichtterminal $A \in N$gibt es eine terminierende Ableitung, d.h. eine Ableitung $A \Longrightarrow^* w$ mit $w \in T^*$.*
2. *Für jedes Nichtterminal $A \in N$ gibt es eine Ableitung $S \Longrightarrow^* w_1 A w_2$ mit gewissen $w_1, w_2 \in T^*$.*
3. *Entweder ist S in $U(G)$ enthalten oder S kommt in jeder Ableitung für jedes Wort $w \in L(G)$ genau einmal vor.*
4. *Jedes Nichtterminal $A \in N \setminus \{S\}$ ist in $U(G)$ enthalten.*

Lemma 8.8 *Für jede eindeutige Grammatik G mit $L(G) \neq \emptyset$ gibt es eine selbstzyklische eindeutige Grammatik G' mit $L(G) = L(G')$.*

Beweis. Die Bedingungen 1. und 2. lassen sich einfach dadurch erfüllen, dass wir alle Nichtterminale, die 1. oder 2. (oder beide Bedingungen) nicht erfüllen und alle Regeln, in denen diese auf der linken oder rechten Seite vorkommen, streichen. Es ist einfach zu sehen, dass hierdurch die erzeugte Sprache Sprache nicht verändert wird, wenn $L(G)$ nicht leer ist (ein formaler Beweis ist im Wesentlichen schon im Beweis von Satz 5.21 enthalten). Durch diesen Schritt werden existierende terminierende Ableitungen nicht verändert, womit Eindeutigkeit erhalten bleibt.
Wir zeigen nun, dass Bedingung 3. auch erfüllt ist. Dazu nehmen wir an, dass es eine Ableitung gibt, in der S mindestens zweimal vorkommt. Dann gilt $S \Longrightarrow^* w_1 S w_2 \Longrightarrow^* w_1 w w_2 \in T^*$. Falls $w_1 w_2 \neq lambda$ gilt, so ist $S \in U(G)$. Daher nehmen wir nun an,

dass $w_1 = w_2 = \lambda$ ist. Dann gibt es Linksableitungen $S \Longrightarrow^* SU$ und $U \Longrightarrow^* \lambda$, wobei $U \in (N \cup T)^*$ ist. Dann haben wir für ein Wort z aus $L(G)$ aber die zwei verschiedenen Linksableitungen

$$S \Longrightarrow^* z \quad \text{und} \quad S \Longrightarrow^* SU \Longrightarrow^* zU \Longrightarrow z.$$

Folglich ist G nicht eindeutig im Widerspruch zur Voraussetzung.
Es sei $N = \{S, A_1, A_2, \ldots A_k\}$. Wir konstruieren nun eine Folge von eindeutigen Grammatiken $G_0, G_1, \ldots, G_k$ derart, dass für $1 \leq i \leq k$ für die Grammatik $G_j = (N_j, T, P_j, S)$ gilt, dass

$$A_j \notin N_i \text{ oder } A_j \in U(G_i) \text{ für } 1 \leq j \leq i \tag{8.1}$$

gilt. Die Konstruktion der Grammatiken erfolgt induktiv.
Offenbar erfüllt $G_0 = G$ die Forderung (8.1).
Sei nun $G_{i-1} = (N_{i-1}, T, P_{i-1}, S)$ bereits konstruiert. Jedes A_j mit $1 \leq j \leq i-1$ kommt also entweder in N_{j-1} nicht vor oder liegt in $U(G_{-1})$. Falls $A_i \in U(G_{i-1})$ liegt, so setzen wir $G_i = G_{i-1}$ und folglich erfolgt G_i die Forderung (8.1). Falls $A_i \notin U(G_{i-1})$ gilt, so seien $A_i \to p_1,\ A_i \to p_2, \ldots,\ A \to p_r$ die Regeln mit rechter Seite A_i in G_{i-1}. Dann kann A_i in keinem der Wörter p_s, $1 \leq s \leq r$, vorkommen. Denn wenn $A_i \to p_s = q_1 A_i q_2$ für ein s, $1 \leq s \leq r$, gilt so gibt es auch eine Ableitung $A_i \Longrightarrow^* q_1' A_i q_2'$ mit $q_1', q_2' \in T^*$, $q_1 \Longrightarrow^* q_1'$ und $q_2 \Longrightarrow^* q_2'$. Bei $q_1' q_2' \neq \lambda$ ist A_i in $U(G_{i-1})$ im Widerspruch zu unserer Voraussetzung, und bei $q_1' q_2' = \lambda$ existiert eine Ableitung $A_i \Longrightarrow^* A_i$, die analog zu Obigem zu einem Widerspruch zur Eindeutigkeit führt. Nun konstruieren wir G_i, indem wir die Regel mit rechter Seite A_i alle streichen und jedes Vorkommen von A_i in einer linken Seite durch alle p_s, $1 \leq s \leq r$, ersetzen. Die so konstruierte erzeugt die gleiche Sprache wie $L(G_{i-1})$, da nur die einmal vorzunehmenden Ersetzungen $A_i \to p_s$ schon in der Regel realisiert sind. Außerdem ist klar, dass die Konstruktion die Eindeutigkeit erhält. Weiterhin folgt aus der Konstruktion sofort, dass G_i Forderung (8.1) erfüllt. [1] □

Satz 8.9 *Die Sprache*

$$L = \{a^n b^n c^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m \mid n \geq 1, m \geq 1\}$$

ist mehrdeutig.

Beweis. Angenommen, L ist nicht mehrdeutig. Dann gibt es eine eindeutige Grammatik G mit $L = L(G)$. Wegen Lemma 8.8 können wir ohne Beschränkung der Allgemeinheit voraussetzen, dass G selbstzyklisch ist.
Wir zeigen nun zuerst, dass jedes Nichtterminal $A \neq S$ von genau einem der folgenden Typen ist:

Typ (a) Es gibt eine natürliche Zahl $m \geq 1$ und Wörter u und v so, dass $A \Longrightarrow uAv$ und entweder $uv \in \{a\}^+$ oder $uv \in \{c\}^+$ gelten.

Typ (b) Es gibt eine natürliche Zahl $m \geq 1$ derart, dass $A \Longrightarrow^* a^m A b^m$ gilt.

[1] Der Leser mache sich klar, dass die Aussage von Lemma 8.8 auch ohne die Eindeutigkeit gilt. Denn die Eindeutigkeit wird nur benutzt, um Ableitungen der Form $A \Longrightarrow^* A$ auszuschließen. Dies kann aber auch dadurch geschehen, dass man zuerst Regel der Form $A \to \lambda$ und $A \to B$ eliminiert, was nach den Normalformen aus Abschnitt 2.2 möglich ist.

Typ (c) Es gibt eine natürliche Zahl $m \geq 1$ derart, dass $A \Longrightarrow^* b^m A c^m$ gilt.

Wegen der Selbstzyklizität von G gibt es für A eine Ableitung $A \Longrightarrow^* uAv$ mit $uv \in T^*$ und $uv \neq \lambda$. Falls in u zwei Buchstaben aus $\{a, b, c\}$ vorkommen, so hat u die Form $u = u_1 x u_2 y u_3$ mit $x, y \in \{a, b, c\}$ und $x \neq y$. Wegen der Existenz einer Ableitung $S \Longrightarrow^* z_1 A z_2 \Longrightarrow^* z_1 z z_2$ mit $z_1, z_2, z \in T^*$ (wegen der Forderungen 1. und 2. bei der Selbstzyklizität) haben wir auch eine Ableitung der Form

$$\begin{aligned} S &\Longrightarrow^* z_1 A z_2 \Longrightarrow^* z_1 u_1 x u_2 y u_3 A v z_2 \Longrightarrow z_1 u_1 x u_2 y u_3 u_1 x u_2 y u_3 A v v z_2 \\ &\Longrightarrow^* z_1 u_1 x u_2 y u_3 u_1 x u_2 y u_3 z v v z_2 \in T^*. \end{aligned}$$

Damit liegt $z_1 u_1 x u_2 y u_3 u_1 x u_2 y u_3 z v v z_2$ in $L(G)$. Dies widerspricht aber $L(G) = L$, da in L der Buchstabe x nicht gleichzeitig vor und nach dem Buchstaben y mit $x \neq y$ auftreten kann. Analog kann man zeigen, dass auch q keine Vorkommen von zwei verschiedenen Buchstaben enthalten kann. Wir diskutieren nun die möglichen Fälle für u und v.
Es sei $u = a^m$ für eine Zahl $m \geq 1$. Ist auch $v \in \{a\}^*$, so ist A vom Typ (a). Es sei nun $q = b^n$ für ein $n \geq 1$. Wegen den Forderungen 1. und 2. der Selbstzyklizität gibt es eine Ableitung

$$S \Longrightarrow^* z_1 A z_2 \Longrightarrow^* a^r b^s c^t \tag{8.2}$$

mit $r = s$ oder $s = t$. Wenden wir auf A α-mal hintereineinander $A \Longrightarrow^* a^m A b^n$ an, so ergibt sich das Wort $a^{r+\alpha m} b^{s+\alpha n} c^t$. Bei der Wahl $\alpha = t$ gilt sicher $s + tn \neq t$. Folglich gilt $r + \alpha m = s + \alpha n$. Dann ergibt sich bei $(\alpha + 1)$-maliger Anwendung noch $r + (\alpha + 1)m = s + (\alpha + 1)n$, woraus dann $n = m$ folgt. Damit ist A vom Typ (b). Es sei nun $v = c^n$ für ein $n \geq 1$. Dann ergibt wieder aus (8.2) die Ableitbarkeit von $a^{r+\alpha m} b^s c^{t+\alpha n} \in L(G)$. Durch die Wahl von α kann erreicht werden, dass $r + \alpha m$ und $t + \alpha n$ von s verschieden sind. Dies widerspricht $L = L(G)$.
Es sei $u = b^m$ für ein $m \geq 1$. Dann gilt wegen der Reihenfolge der Buchstaben in den Wörtern von L entweder $v = b^n$ oder $v = c^n$ für ein $n \geq 0$. Im ersten Fall erhalten wir aus (8.2), dass $a^r b^{s+\alpha m+\alpha n} c^t$ in $L(G)$ liegt, und erhalten einen Widerspruch, da bei passender Wahl von α das Wort $a^r b^{s+\alpha m+\alpha n} c^t$ nicht in L liegt. Im zweiten Fall können wir wie oben $m = n$ zeigen, womit A vom Typ (c) ist.
Es sei $u = c^m$ für ein $m \geq 1$. Dann gilt auch $v \in \{c\}^+$ und damit liegt Typ (a) vor.
Falls $u = \lambda$ ist, so ist $v \neq \lambda$. Bei $v \in \{a\}^+$ und $v \in \{c\}^+$ ist A vom Typ (a). Bei $v = b^n$ für ein $n \geq 0$ können wir alle Wörter $a^r b^{t+\alpha n} c^t$ in G erzeugen, die aber bei passender Wahl von α nicht in L liegen.
Damit ist gezeigt, dass jedes Nichtterminal $A \neq S$ mindestens einem Typ angehört. Wir zeigen jetzt, dass es nicht zwei verschiedenen Typen angehören kann. Angenommen, A ist vom Typ (a) und vom Typ (b). Dann gibt es Ableitungen $A \Longrightarrow^* x^f A x^g$ mit $x \in \{a, c\}$, $f + g \geq 1$ und $A \Longrightarrow^* a^m A b^m$ für ein $m \geq 1$. Dann gibt es aber auch eine Ableitung $A \Longrightarrow^* a^m A b^m \Longrightarrow^* a^m x^f A x^g b^m$. Wegen der Reihenfolge der Buchstaben ist $x = c$ nicht möglich. Bei $x = a$ und $g \neq 0$ haben wir eine Ableitung $A \Longrightarrow^* uAv$, bei der v zwei verschiedene Buchstaben enthält. Oben haben wir gezeigt, dass dies unmöglich ist. Falls $g = 0$ ist, so muss nach Obigem $m + f = m$ gelten. Dies impliziert aber $f = 0$ im Widerspruch zu $f + g \geq 1$.
Analog erhalten wir, dass es nicht möglich ist, dass A sowohl vom Typ (a) als auch vom Typ (b) ist.

Sei daher A vom Typ (b) und vom Typ (c). Dann haben wir Ableitungen $A \Longrightarrow^* a^m A b^m$ und $A \Longrightarrow^* b^{m'} A c^{m'}$, aus denen sich die Ableitung $A \Longrightarrow^* a^m A b^m \Longrightarrow^* a^m b^{m'} A c^{m'} b^m$ ergibt, die nach Obigem unmöglich ist.
Folglich ist jedes Nichtterminal A von genau einem der Typen (a), (b) oder (c).
Wegen der Selbstzyklizität der Grammatik G kommt S in jeder genau einmal vor, oder $S \in U(G)$. Wir zeigen nun, dass die zweite Möglichkeit nicht eintreten. Wir nehmen das Gegenteil an und werden einen Widerspruch herleiten.
Falls $S \in U(G)$ gilt, so gibt es terminale Wörter x und y mit $S \Longrightarrow^* xSy$ und $xy \neq \lambda$. Da abc^2, a^2bc und abc in L liegen, gibt es Ableitungen $S \Longrightarrow^* abc^2$ und $S \Longrightarrow^* a^2bc$. Damit gibt es auch die Ableitungen

$$S \Longrightarrow xSy \Longrightarrow xabc^2y \text{ und } S \Longrightarrow xSy \Longrightarrow xa^2bcy.$$

Somit liegen $xabc^2y$ und xa^2bcy auch in L. Da außerdem x und y Potenzen von einem Buchstaben a oder b oder c sein müssen, gibt es für x und y nur die Möglichkeiten

$$x = a^m \text{ und } y = \lambda \quad \text{oder} \quad x = \lambda \text{ und } y = c^n \quad \text{oder} \quad x = a^m \text{ und } y = c^n$$

mit gewissen positiven ganzen Zahlen m und n. Dann gilt im ersten Fall $xabc^2y = a^{m+1}bc^2$, im zweiten Fall $xa^2bcy = a^2bc^{n+1}$ und im dritten Fall $xa^2bcy = a^{m+2}bc^{n+1}$ mit gewissen $m, n \geq 1$. Dies widerspricht aber in allen Fällender Form der Wörter in L.
Es sei

$$\begin{aligned} S &\Longrightarrow^* u_1A_1v_1 \Longrightarrow^* u_1u_2A_2v_2 \Longrightarrow^* \ldots \Longrightarrow^* u_1u_2\ldots u_rA_rv_rv_{r-1}\ldots v_1 \\ &\Longrightarrow^* u_1u_2\ldots u_rwv_rv_{r-1}\ldots v_1 \end{aligned}$$

eine Ableitung, in der keine Nichtterminale vom Typ (b) oder (c) vorkommen. Ferner beginne jeder Teilableitung

$$u_1u_2\ldots u_{i-1}A_{i-1}v_{i-1}\ldots v_1 \Longrightarrow^* u_1u_2\ldots u_{i-1}u_iA_iv_iv_{i-1}\ldots v_1$$

mit der Anwendung einer Regel, deren rechte Seite mindestens einmal den Buchstaben b enthält. Dann sind die Buchstaben $A_1, A_2, \ldots, A_r$ alle paarweise verschieden. Wäre dies nicht der Fall, so gibt es eine Ableitung $A_i \Longrightarrow^* x_1bx_2A_ix_3$ oder $A_i \Longrightarrow^* x_1Ax_2bx_3$, womit A_i vom Typ (b) oder (c) wäre. Damit werden Regeln mit Vorkommen von b auf der rechten Seite in einer Ableitung ohne Nichtterminale vom Typ (b) oder (c) höchstens μ mal für ein passendes μ angewendet. Daher enthält jedes Wort, das durch eine Ableitung, in der keine Nichtterminale vom Typ (b) oder (c) vorkommen, höchsten $s \cdot \mu$ Vorkommen von b (jede Regel mit einem Vorkommen von b produziert höchstens s Vorkommen von b.
Für jedes Nichtterminal A vom Typ (b) oder (c) wählen wir nun eine Zahl $m(A)$ derart, dass $A \Longrightarrow^* a^{m(A)}Ab^{m(A)}$ bzw. $A \Longrightarrow^* b^{m(A)}Ac^{m(A)}$ gilt. Wir wählen nun p so, dass
— $p > s \cdot \mu$ ist,
— für jedes X vom Typ (b) oder (c) $m(X)$ ein Teiler von p ist. Wir betrachten nun eine Ableitung von $w = a^pb^pc^{2p}$. Da mehr als $s \cdot \mu$ Vorkommen von b in w sind, muss die Ableitung mindestens ein Nichtterminal A vom Typ (b) oder (c) enthalten. Wir nehmen nun an, dass ein Nichtterminal vom Typ (c) vorkommt. Dann hat die Ableitung die Form $S \Longrightarrow^* u_1Au_2 \Longrightarrow^* w$ mit A vom Typ (c). Es sei $m(A) \cdot r = p$. Durch r-malige

Anwendung der Ableitung $A \Longrightarrow^* b^{m(A)} A c^{m(A)}$ erzeugen wir p zusätzliche Buchstaben b und p zusätzliche Buchstaben c, d.h. wir erhalten $S \Longrightarrow^* a^p b^{2p} c^{3p}$ und damit ein Wort, dass nicht in L liegt. Daher kommen in der Ableitung von w nur Nichtterminale der Typen (a) und (b) und mindestens ein Nichtterminal A' vom Typ (b) vor. Dann gibt es ein s mit $m(A') \cdot s = p$. Wir wenden $A' \Longrightarrow^* a^{m(A')} A' b^{m(A')}$ zusätzlich s-mal an, wodurch $a^{2p} b^{2p} c^{2p}$ entsteht. Daher gibt es eine Ableitung von $a^{2p} b^{2p} c^{2p}$, in der nur Nichtterminale der Typen (a) und (b) und mindestens ein Nichtterminal vom Typ (b) vorkommen. Mit gleicher Argumentation kann man (ausgehend von $a^{2p} b^p c^p$) zeigen, dass es eine Ableitung von $a^{2p} b^{2p} c^{2p}$ gibt, in der nur Nichtterminale der Typen (a) und (c) und mindestens ein Nichtterminal vom Typ (c) vorkommen. Damit gibt es für $a^{2p} b^{2p} c^{2p}$ zwei verschiedene Linksableitungen, was der Eindeutigkeit widerspricht. □

Literaturverzeichnis

[1] J.Albert, Th.Ottmann: Automaten, Sprachen und Maschinen für Anwender. B.-I.-Wissenschaftsverlag, 1983.

[2] A.Aho, J.E.Hopcroft, J.D.Ullman: The Design and Analysis of Algorithms. Reading, Mass., 1974.

[3] A.Aho, R.Sethi, J.D.Ullman: Compilerbau. Band 1 und 2, Addison-Wesley, 1990.

[4] A.Asteroth, Ch.Baier: Theoretische Informatik. Pearson Studium, 2002.

[5] L.Balke, K.H.Böhling: Einführung in die Automatentheorie und Theorie formaler Sprachen. B.-I.-Wissenschaftsverlag, 1993.

[6] W.Bucher, H.Maurer: Theoretische Grundlagen der Programmiersprachen. B.-I.-Wissenschaftsverlag, 1983.

[7] J.Carrol, D.Long: Theory of Finite Automata (with an Introduction to Formal Languages). Prentice Hall, London, 1983.

[8] E.Engeler, P.Läuchli: Berechnungstheorie für Informatiker. Teubner-Verlag, 1988.

[9] M.R.Garey, D.S.Johnson: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, 1979.

[10] J.Hopcroft, J.Ullman: Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie. 2. Aufl., Addison-Wesley, 1990.

[11] E.Horowitz, S.Sahni: Fundamentals of Computer Algorithms. Computer Science Press, 1978.

[12] D.E.Knuth: The Art of Computer Programming. Volumes 1-3, Addison-Wesley, 1968-1975.

[13] U.Manber, Introduction to Algorithms. Addison-Wesley, 1990.

[14] K.Mehlhorn: Effiziente Algorithmen. Teubner-Verlag, 1977.

[15] Ch.Meinel: Effiziente Algorithmen. Fachbuchverlag Leipzig, 1991.

[16] W.Paul: Komplexitätstheorie. Teubner-Verlag, 1978.

[17] Ch.Posthoff, K.Schulz: Grundkurs Theoretische Informatik. Teubner-Verlag, 1992.

[18] U.Schöning: Theoretische Informatik kurz gefaßt. B.I.Wissenschaftsverlag, 1992.

[19] R.Sedgewick: Algorithmen. Addison-Wesley, 1990.

[20] B.A.Trachtenbrot: Algorithmen und Rechenautomaten. Berlin, 1977.

[21] G. Vossen, K.-U. Witt: Grundlagen der Theoretischen Informatik mit Anwendungen. Vieweg-Verlag, Braunschweig, 2000.

[22] K.Wagner: Einführung in die Theoretische Informatik. Springer-Verlag, 1994.

[23] D.Wätjen: Theoretische Informatik. Oldenbourg-Verlag, 1994.

[24] I.Wegener: Theoretische Informatik. Teubner-Verlag, 1993.

[25] D.Wood: Theory of Computation. Harper & Row Publ., 1987.