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:

- **N** für die Menge der natürlichen Zahlen $\{1, 2, \ldots\}$,
- $\mathbf{N}_0 = \mathbf{N} \cup \{0\}$,
- **Z** für die Menge der ganzen Zahlen,
- **Q** für die Menge der rationalen Zahlen,
- **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_1w_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

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

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

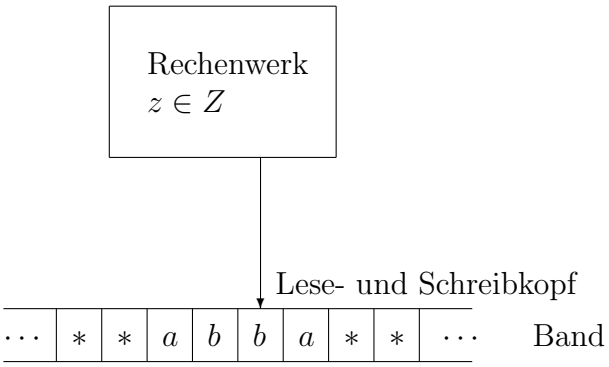

Abbildung 1.2: TURING-Maschine

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 Zusatand 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

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

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 $0, 1, 2, \dots, 9$ 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

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

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*

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

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 **??** 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 $i_1 i_2 \ldots i_m$ 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

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

   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 $i_1 i_2 \dots i_k$ 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) $x^4 - 2x^2y^2 + 2y^4 - 3 = 0$.

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.