Prof. Dr. Jürgen Dassow

Skript zur Vorlesung

T H E O R E T I S C H E
I N F O R M A T I K

gehalten

am Hasso-Plattner-Institut

im Studienjahr 2013/14

Potsdam, November 2013

Einleitung

Die Informatik wird heutzutage oft in vier große Teilgebiete unterteilt:

- Theoretische Informatik,
- Technische Informatik,
- Praktische Informatik,
- Angewandte Informatik,

wobei die Grenzen zwischen diesen Disziplinen fließend sind und nicht in jedem Fall eine eindeutige Einordnung eines Problems oder Sachverhalts möglich ist.
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 wird hier eine Konzentration auf die folgenden Problemkreise vorgenommen:

- Im ersten Abschnitten werden verschiedene Aspekte des Algorithmenbegriffs, der für die gesamte Informatik ein zentrales Konzept darstellt, behandelt. Es wird eine Präzisierung des Begriffs Algorithmus vorgenommen 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).
- Der zweite Abschnitt 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 gewidmet und ihre Eigenschaften hinsichtlich Entscheidbarkeitsfragen diskutiert.
- Im dritten Abschnitt 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.

Jürgen Dassow Potsdam, November 2013

Vorbemerkungen

Im Folgenden setzen wir voraus, dass der Leser über mathematische Kenntnisse verfügt, wie sie üblicherweise in einer Vorlesung über (Diskrete) Mathematik vermittelt werden. Das erforderliche Wissen besteht im Wesentlichen aus Formeln für kombinatorische Anzahlprobleme und Summen, Basiswissen in Zahlentheorie, linearer und abstrakter Algebra und Graphentheorie.

Wir verwenden folgende Bezeichnungen für Zahlbereiche:

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

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).

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 hat eindeutig bestimmte Ausgabedaten geliefert,
  - 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:*

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

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

$\Pi_1; \Pi_2$,
LOOP x_i **BEGIN** Π **END** ,
WHILE $x_i \neq 0$ **BEGIN** Π **END** .

Wir geben nun einige Beispiele.

Beispiel 1.1

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 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

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

$\Pi_1; \Pi_2$ 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 1.2 *Π sei 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 offenbar durch Π auch n Funktionen $\Phi_{\Pi,i}(x_1, x_2, \ldots, x_n)$, $1 \leq i \leq n$, 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

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

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

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

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 1.3 *Eine Funktion $f(x_1, x_2, \ldots, x_n)$ heißt* **LOOP/ WHILE**-*berechenbar, wenn es ein Programm Π mit m Variablen, $m \geq n$, derart gibt, dass*

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

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 1.3 gegebene Einschränkung auf die erste Variable durch die Auswahl von $\Phi_{\Pi,1}$ ist nur scheinbar, 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 1.2 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

$$a_{i+2} = a_i + a_{i+1}.$$

Wir wollen nun zeigen, dass die Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ 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 1.3). 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 definieren nun die *Tiefe* eines **LOOP**/**WHILE**-Programms, die sich im folgenden als nützliches Hilfsmittel erweisen wird.

Definition 1.4 *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) $t(\textbf{LOOP } x_i \textbf{ BEGIN } \Pi \textbf{ END}) = t(\Pi) + 1$,
iv) $t(\textbf{WHILE } x_i \neq 0 \textbf{ BEGIN } \Pi \textbf{ END}) = t(\Pi) + 1$.

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

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

Die Programme der Tiefe 3 haben 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

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

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

Satz 1.1 *Es gibt (mindestens) eine totale Funktion* $f : \mathbb{N}_0 \to \mathbb{N}_0$*, die nicht* **LOOP**/ **WHILE**-*berechenbar ist.*

Beweis. Wir geben zwei Beweise für diese Aussage.
a) Wir erinnern zuerst daran, dass 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 $k \in \mathbb{N}$ 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 $k \in \mathbb{N}$ 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 $i \in \mathbb{N}$ 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 $i \in \mathbb{N}$, $j \in \mathbb{N}$ und $i + j = k + 1$ abgebildet werden. Daher ergibt sich, dass die Menge aller Programme dieser Form gleichmächtig zu

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

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 $\mathbb{N} \times Q_k$. 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 $\mathbb{N}_0$ in $\mathbb{N}_0$ gibt. Es sei nämlich die Menge E dieser Funktionen abzählbar, so gibt es eine eineindeutige Funktion von $\mathbb{N}_0$ auf E. Für $i \in \mathbb{N}_0$ 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 Funktion $f \in E$ mittels der Setzung $f(r) = f_r(r) + 1$ für $r \in \mathbb{N}_0$. Offenbar ist f nicht eine der Funktionen der Matrix, da für jedes $t \in \mathbb{N}_0$ 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.

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

Abbildung 1.1: Matrixdarstellung von der Menge E der einstelligen Funktionen

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 $f : \mathbb{N}_0 \to \mathbb{N}_0$, die durch $f(0) = 0$ und

$$f(n) = \max\{\Phi_{\Pi,1}(0, 0, \ldots, 0) \mid \Pi \text{ ist ein } \textbf{LOOP/WHILE}\text{-Programm, } t(\Pi) \leq n\}$$

für $n \geq 1$ definiert ist.
Somit gibt $f(n)$ die größte Zahl, 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 $x_1 := S(x_1)$ 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 folgende Programm

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

Das Programm Π' aus ersten sechs Zeilen erfüllt $t(\Pi') = 8$ und $\Phi_{\Pi,1}(0, 0, 0) = 12$, während für das gesamte Programm Π die Beziehungen $t(\Pi) = 12$ und $\Phi_{\Pi,1}(0, 0, 0) = 60$ gelten. Damit erhalten wir $f(8) \geq 12$ und $f(12 \geq 60$, womit f nicht die Identität ist.
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. Entsprechend der Definition von f ist $f(0) < f(1)$ gesichert.
Es sei nun 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 gilt für das Programm Π', das durch Hintereinanderausführung von Π und $x_1 := S(x_1)$ entsteht und damit die Tiefe $n+1$ hat,

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

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

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

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

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

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

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

die Funktion g. Es sei

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

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

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

Dann gelten

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

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

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

wodurch offensichtlich ein Widerspruch gegeben ist. □

Für spätere Anwendungen benötigen wir die folgende Modifikation von Satz 1.1.

Folgerung 1.2 *Es gibt eine Funktion $f : \mathbb{N}_0^2 \to \mathbb{N}_0$ mit folgenden Eigenschaften:*
- f ist total,
- der Wertebereich von f ist $\{0,1\}$,
- f ist nicht **LOOP/WHILE***-berechenbar.*

Beweis. Nach Satz 1.1 gibt es eine totale Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$, die nicht **LOOP**/**WHILE**-berechenbar ist. Wir konstruieren nun die Funktion $g : \mathbb{N}_0^2 \to \mathbb{N}_0$ mit

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

Offenbar genügt g den ersten beiden Forderungen aus Folgerung 1.2. 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':

$x_2 := 0; x_3 := x_1; x_1 := 0; x_1 := S(x_1)$
WHILE $x_1 \neq 0$ **BEGIN**
$x_1 := x_4; x_2 := x_5; \Pi; x_3 = S(x_3)$
END;
$x_1 := P(x_3)$.

Dieses Programm berechnet f, was aus folgenden Überlegungen folgt: Die Variable x_3 dienen der Speicherung des Wertes, mit dem die Variable x_1 zu Beginn belegt ist. Durch die anschließende Wertzuweisungen $x_1 := 0$ und $x_1 := S(x_1)$ wird x_1 auf $1 \neq 0$ gesetzt. Daher wird nun die **WHILE**-Anweisung mindestens einmal durchlaufen. Aufgrund der Wertzuweisung $x_2 := S(x_2)$ und durch die stets erfolgende Setzung der Variablen x_1 auf den Wert der Anfangsbelegung werden mittels der **WHILE**-Anweisung der Reihe nach die Werte $g(x_1, 0), g(x_1, 1), g(x_1, 2), \ldots$ berechnet, bis i mit $g(x_1, i) = 0$ erreicht wird. Dann wird durch die letzte Wertzuweisung des Programms x_1 mit i belegt. Andererseits gilt nach Definition von g auch $f(x_1) = i$. 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. Dies trifft jedoch auf eine eingeschränkte Menge von Funktionen zu, die zu ihrer Berechnung nur die Wertzuweisungen, Hintereinanderausführung von Programmen und die **LOOP**-Anweisung benötigen. Formal wird dies durch die folgende Definition und Satz 1.3 gegeben.

Definition 1.5 *Eine Funktion f 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 1.3 *Der Definitionsbereich jeder n-stelligen* **LOOP***-berechenbaren Funktion ist die Menge $\mathbb{N}_0^n$, 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.
Es 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 1.4 *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 Rekursive Funktionen

Im letzten Abschnitt haben wir berechenbare Funktionen als von Programmen induzierte Funktionen eingeführt. In diesem Abschnitt gehen wir einen Weg, der etwas direkter ist. Wir werden Basisfunktionen definieren und diese als berechenbar ansehen. Mittels Operationen, bei denen die Berechenbarkeit nicht verlorengeht, werden dann weitere Funktionen erzeugt.

Wir geben nun die formalen Definitionen. Als *Basisfunktionen* betrachten wir:

- die nullstellige Funktion Z_0, die den konstanten Wert 0 liefert,
- die Funktion $S : \mathbb{N}_0 \to \mathbb{N}_0$, bei der jeder natürlichen Zahl ihr Nachfolger zugeordnet wird,
- die Funktion $P : \mathbb{N}_0 \to \mathbb{N}_0$, bei der jede natürliche Zahl $n \geq 1$ auf ihren Vorgänger und die 0 auf sich selbst abgebildet wird,
- die Funktionen $P_i^n : \mathbb{N}_0^n \to \mathbb{N}_0$, die durch

$$P_i^n(x_1, x_2, \ldots, x_n) = x_i$$

definiert sind.

Anstelle von $S(n)$ schreiben wir zukünftig auch - wie üblich - $n+1$. P_i^n ist die übliche Projektion eines n-Tupels auf die i-te Komponente (Koordinate).
Als *Operationen* zur Erzeugung neuer Funktionen betrachten wir die beiden folgenden Schemata:

- *Kompositionsschema*: Für eine m-stellige Funktion g und m n-stellige Funktionen $f_1, f_2, \ldots, f_m$ definieren wir die n-stellige Funktion f vermöge

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

- *Rekursionsschema*: Für fixierte natürliche Zahlen $x_1, x_2 \ldots, x_n$, eine n-stellige Funktion g und eine $(n+2)$-stellige Funktion h definieren wir die $(n+1)$-stellige Funktion f vermöge

$$\begin{aligned} f(x_1, x_2, \ldots, x_n, 0) &= g(x_1, x_2, \ldots, x_n), \\ f(x_1, x_2, \ldots, x_n, y+1) &= h(x_1, x_2, \ldots, x_n, y, f(x_1, x_2, \ldots, x_n, y)). \end{aligned}$$

Zuerst erwähnen wir, dass das Kompositionsschema eine einfache Formalisierung des „Einsetzens“ ist.
Wir merken an, dass das gegebene Rekursionsschema eine parametrisierte Form der klassischen Rekursion

$$\begin{aligned} f(0) &= c \\ f(y+1) &= h(y, f(y)), \end{aligned}$$

wobei c eine Konstante ist, mit den Parametern $x_1, x_2, \ldots, x_n$ ist.
Für das Kompositionsschema ist sofort einzusehen, dass ausgehend von festen Funktionen $g, f_1, f_2, \ldots, f_m$ genau eine Funktion f definiert wird. Wir zeigen nun, dass dies auch für das Rekursionsschema gilt, wobei wir (um die Bezeichnungen einfach zu halten) dies nur für die klassische parameterfreie Form durchführen. Zuerst einmal ist klar, dass durch das Schema eine Funktion definiert wird (für $n = 0$ ist der Wert festgelegt, für $n \geq 1$ lässt er sich schrittweise aus der Rekursion berechnen). Wir zeigen nun die Eindeutigkeit. Es seien dazu f_1 und f_2 zwei Funktionen, die den Gleichungen des Rekursionsschemas genügen. Mittels vollständiger Induktion beweisen wir nun $f_1(y) = f_2(y)$ für alle natürlichen Zahlen y. Laut Schema gilt für $y = 0$ die Beziehung

$$f_1(0) = f_2(0) = c,$$

womit der Induktionsanfang gezeigt ist. Es sei die Aussage schon für alle natürlichen Zahlen $x \leq y$ bewiesen. Dann gilt

$$f_1(y+1) = h(y, f_1(y)) = h(y, f_2(y)) = f_2(y+1),$$

wobei die erste und letzte Gleichheit aus dem Rekursionsschema und die zweite Gleichheit aus der Induktionsvoraussetzung folgen.

Definition 1.6 *Eine Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ heißt primitiv-rekursiv, wenn sie mittels endlich oft wiederholter Anwendung von Kompositions- und Rekursionsschema aus den Basisfunktionen erzeugt werden kann.*

Wir lasssen dabei auch die 0-malige Anwendung, d. h. keine Anwendung, als endlich oft malige Anwendung zu; sie liefert stets eine der Basisfunktionen.

Beispiel 1.3 a) Ausgehend von der Basisfunktion S gewinnen wir mittels Kompositionsschema die Funktion f mit $f(n) = S(S(n))$, aus der durch erneute Anwendung des Kompositionsschemas f' mit $f'(n) = S(f(n)) = S(S(S(n)))$ erzeugt werden kann. Offenbar ordnen f bzw. f' jeder natürlichen Zahl ihren zweiten bzw. dritten Nachfolger zu. Beide Funktionen sind nach Definition primitiv-rekursiv.

b) Wegen $x = P(S(x))$ ist die identische Funktion $id : \mathbb{N}_0 \to \mathbb{N}_0$ mit $id(x) = x$ ebenfalls primitiv-rekursiv.

c) Die nullstellige konstante Funktion Z_0 gehört zu den Basisfunktionen und ist daher primitiv-rekursiv. Wir zeigen nun, dass die n-stellige Funktion Z_n mit $Z_n(x_1, \ldots, x_n) = 0$ für alle $x_1, x_2, \ldots, x_n$ ebenfalls primitiv-rekursiv ist.
Es sei $n = 1$. Dann betrachten wir das Rekursionsschema

$$Z_1(0) = Z_0 \quad \text{und} \quad Z_1(y+1) = P_2^2(y, Z_1(y))\,.$$

Wir zeigen mittels vollständiger Induktion, dass Z_1 die einstellige konstante Funktion mit dem Wertevorrat 0 ist. Offensichtlich gilt $Z_1(0) = 0$, da Z_0 den Wert 0 liefert. Es sei nun schon $Z_1(y) = 0$ gezeigt. Dann ergibt sich Aus der zweiten Rekursionsgleichung sofort $Z_1(y+1) = P_2^2(y, 0) = 0$, womit der Induktionsschritt vollzogen ist.
Nehmen wir nun an, dass wir bereits die n-stellige konstante Funktion Z_n mit dem Wert 0 als primitiv-rekursiv nachgewiesen haben, so können wir analog zu Obigem zeigen, dass das Rekursionsschema

$$\begin{aligned} Z_{n+1}(x_1, x_2, \ldots x_n, 0) &= Z_n(x_1, x_2, \ldots x_n)\,, \\ Z_{n+1}(x_1, x_2, \ldots x_n, y+1) &= P_{n+2}^{n+2}(x_1, x_2, \ldots, x_n, y, Z_{n+1}(x_1, x_2, \ldots x_n, y)) \end{aligned}$$

die $(n+1)$-stellige konstante Funktion Z_{n+1} mit dem Wert 0 liefert.

d) Die Addition und Multiplikation natürlicher Zahlen lassen sich mittels der Rekursionsschema

$$\begin{aligned} add(x, 0) &= id(x), \\ add(x, y+1) &= S(P_3^3(x, y, add(x, y))) \end{aligned}$$

und

$$\begin{aligned} mult(x, 0) &= Z_1(x), \\ mult(x, y+1) &= add(P_1^3(x, y, mult(x, y)), P_3^3(x, y, mult(x, y))) \end{aligned}$$

definieren. Da die Identität, S, Z und die Projektionen bereits als primitiv-rekursiv nachgewiesen sind, ergibt sich damit die primitive Rekursivität von *add* und aus der dann die von *mult*.
Entsprechend unserer obigen Bemerkung ist klar, dass durch diese Schemata eindeutige Funktionen definiert sind. Durch einfaches „Nachrechnen“ überzeugt man sich davon, dass es sich wirklich um Addition und Multiplikation handelt, z.B. bedeutet die letzte Relation

mit der üblichen Notation $add(x, y) = x + y$ und $mult(x, y) = x \cdot y$ nichts anderes als das bekannte Distributivgesetz

$$mult(x, y + 1) = x \cdot (y + 1) = x + x \cdot y = add(x, mult(x, y)).$$

d) Durch das Rekursionsschema

$$sum(0) = Z \quad \text{und} \quad sum(y + 1) = S(add(y, sum(y)))$$

wird die Funktion

$$sum(y) = \sum_{i=0}^{y} i = \frac{y(y + 1)}{2}$$

definiert, wovon man sich leicht mittels vollständiger Induktion überzeugen kann.

Wir betrachten nun die folgende rekursive Definition der Fibonacci-Folge:

$$\begin{aligned} f(0) &= 1, \quad f(1) = 1, \\ f(y + 2) &= f(y + 1) + f(y). \end{aligned}$$

Für diese Rekursion ist nicht offensichtlich, dass sie durch das obige Rekursionsschema realisiert werden kann, da nicht nur auf den Wert $f(y)$ rekursiv zurückgegriffen wird. Die Rekursion für die Fibonacci-Folge lässt sich aber unter Verwendung von zwei Funktionen so umschreiben, dass jeweils nur die Kenntnis der Werte an der Stelle y erforderlich ist. Dies wird durch das Schema

$$\begin{aligned} &f_1(0) = 1, && f_2(0) = 1, \\ &f_1(y + 1) = f_2(y), && f_2(y + 1) = f_1(y) + f_2(y) \end{aligned}$$

geleistet. Hiervon ausgehend führen wir die folgende Verallgemeinerung des Rekursionsschemas, simultane Rekursion genannt, ein: Für n-stellige Funktionen g_i und die $(n + m + 1)$-stelligen Funktionen h_i, $1 \leq i \leq m$, definieren wir simultan die $(n + 1)$-stelligen Funktionen f_i, $1 \leq i \leq m$, durch

$$\begin{aligned} f_i(x_1, \ldots, x_n, 0) &= g_i(x_1, \ldots, x_n), \ 1 \leq i \leq m, \\ f_i(x_1, \ldots, x_n, y + 1) &= h_i(x_1, \ldots, x_n, y, f_1(x_1, \ldots, x_n, y), \ldots, f_m(x_1, \ldots, x_n, y)). \end{aligned}$$

Wir wollen nun zeigen, dass die simultane Rekursion auch nur die Erzeugung primitivrekursiver Funktionen gestattet. Um die Notation nicht unnötig zu verkomplizieren werden wir die Betrachtungen nur für den Fall $n = 1$ und $m = 2$ durchführen.
Es seien die Funktionen C, E, D_1 und D_2 mittels der Funktionen $\ominus$ und div aus Übungsaufgabe 14 durch

$$\begin{aligned} C(x_1, x_2) &= sum(x_1 + x_2) + x_2, \\ E(0) &= 0, \quad E(n + 1) = E(n) + (n \ div \ sum(E(n) + 1)) + 1, \\ D_1(n) &= E(n) + sum(E(n)) \ominus n, \\ D_2(n) &= E(n) \ominus D_1(n) \end{aligned}$$

definiert. Entsprechend der Konstruktion und Übungsaufgabe 14 sind alle diese Funktionen primitiv-rekursiv. Weiterhin rechnet man nach, dass die folgenden Bedingungen erfüllt sind: Für alle natürlichen Zahlen n, n_1 und n_2 gilt

$$C(D_1(n), D_2(n)) = n,\ D_1(C(n_1, n_2)) = n_1,\ D_2(C(n_1, n_2)) = n_2.$$

Zur Veranschaulichung betrachte man Abbildung 1.2 und prüfe nach, dass durch $E(n)$ die Nummer der Diagonalen in der n steht, durch $x_1 + x_2$ die Nummer der Diagonalen, in der sich die Spalte von x_1 und die Zeile von x_2 kreuzen, durch $C(x_1, x_2)$ das im Kreuzungspunkt der Spalte zu x_1 und der Zeile zu x_2 stehende Element, durch D_1 und D_2 die Projektionen von einem Element gegeben werden.

x_1 / x_2	0	1	2	3	4	...
0	0	1	3	6	10	...
1	2	4	7	11	...	
2	5	8	12	...		
3	9	13	...			
4	14	...				
...	...					

Abbildung 1.2:

Dann definieren wir für die gegebenen Funktionen g_i und h_i, $1 \leq i \leq m$, die Funktionen g und h durch

$$\begin{aligned} g(x) &= C(g_1(x), g_2(x)), \\ h(x, y, z) &= C(h_1(x, y, D_1(z), D_2(z)), h_2(x, y, D_1(z), D_2(z))) \end{aligned}$$

die Funktion f durch das Rekursionsschema

$$\begin{aligned} f(x, 0) &= g(x), \\ f(x, y + 1) &= h(x, y, f(x, y)) \end{aligned}$$

und die Funktionen f_1 und f_2, die durch das simultane Rekursionsschema erzeugt werden sollen, durch

$$f_1(x, y) = D_1(f(x, y)) \ \text{ und } \ f_2(x, y) = D_2(f(x, y)).$$

Wegen

$$f_i(x, 0) = D_i(f(x, 0)) = D_i(g(x)) = D_i(C(g_1(x), g_2(x)) = g_i(x)$$

für $i \in \{1, 2\}$, sind die Ausgangsbedingungen des verallgemeinerten Rekursionsschemas erfüllt, und analog zeigt man, dass die Rekursionsbedingungen befriedigt werden. Diese Konstruktion von f_1 und f_2 erfordert nur das ursprüngliche Rekursionsschema (für die Funktion f) und das Kompositionsschema, womit gezeigt ist, dass diese beiden Funktionen primitiv-rekursiv sind.

Aufgrund der eben gezeigten Äquivalenz von Rekursionsschema und simultanem Rekursionsschema werden wir zukünftig auch von der simultanen Rekursion Gebrauch machen, um zu zeigen, dass gewisse Funktionen primitiv-rekursiv sind.

Satz 1.5 *Eine Funktion f ist genau dann primitiv-rekursiv, wenn sie* **LOOP***-berechenbar ist.*

Beweis: Wir zeigen zuerst mittels Induktion über die Anzahl k der Operationen zur Erzeugung der primitiv-rekursiven Funktion f, dass f auch **LOOP**-berechenbar ist.
Es sei $k = 0$. Dann muss die zu betrachtende Funktion f eine Basisfunktion sein. Die Tabelle in Abbildung 1.3 gibt zu jeder Basisfunktion f ein **LOOP**-Programm Π mit $\Phi_{\Pi,1} = f$ (man beachte, dass bei der Erzeugung von Z die Variable eine zusätzliche Variable ist, $n = 0$ und $m = 1$ in Definition 1.3). Damit ist der Induktionsanfang gesichert.

f	Π
Z	$x_1 := 0$
S	$x_1 := S(x_1)$
P	$x_1 := P(x_1)$
P_i^n	$x_1 := x_i$

Abbildung 1.3:

Wir führen nun den Induktionsschritt durch. Es sei dazu f eine Funktion, die durch k-malige, $k \geq 1$, Anwendung der Operationen erzeugt wurde. Dann gibt es eine Operation, die als letzte angewendet wurde. Hiernach unterscheiden wir zwei Fälle, welche der beiden Operationen dies ist.
Fall 1. Kompositionsschema. Dann gilt

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

wobei die Funktionen $g, f_1, f_2, \ldots, f_m$ alle durch höchstens $(k-1)$-malige Anwendung der Operationen entstanden sind. Nach Induktionsannahme gibt es also Programme $\Pi, \Pi_1, \Pi_2, \ldots, \Pi_m$ derart, dass

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

gelten. Nun prüft man leicht nach, dass das Programm

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

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

aufgrund der Anweisungen $x_j := x_{2n+j}$ das Programm Π gestartet und damit der nach Kompositionsschema gewünschte Wert berechnet).
Hierbei haben so getan, als ob Π, Π_1, $\Pi_2, \ldots,$ Π_n keine zusätzlichen Variablen benutzen. Sollte dies aber der fall sein, so sind für diese größere Indices zu benutzen und sie sind jeweils vor der Anwendung von Π auf 0 zu setzen. Auch in den folgenden Beweisen werden wir keine zusätzlichen variablen betrachten und überlassen es dem Leser, die notwendigen Modifikationen vorzunehmen, sollten welche benutzt werden.
Fall 2. Rekursionsschema. Die Funktion f werde mittels Rekursionsschema aus den n- bzw. $(n+2)$-stelligen Funktionen g (an der Stelle $y = 0$) und h (für die eigentliche Rekursion) erzeugt. Da sich diese beiden Funktionen durch höchstens $(k-1)$-malige Anwendung der Schemata erzeugen lassen können, gibt es für sie Programme Π über den Variablen $x_1, x_2, \ldots x_n$ und Π' über den Variablen $x_1, x_2, \ldots, y, z$ mit $\Phi_{\Pi,1} = g$ und $\Phi_{\Pi',1} = h$ (wobei wir zur Vereinfachung nicht nur Variable der Form x_i, wie in Abschnitt 1.1.1 gefordert, verwenden). Wir betrachten das folgende Programm:

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

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

Wir zeigen nun die umgekehrte Richtung. Wir gehen analog vor, werden vollständige Induktion über die Programmtiefe t benutzen und sogar zeigen, dass jede von einem **LOOP**-Programm Π berechnete Funktion $\Phi_{\Pi,j}$, $j \geq 1$ eine primitiv-rekursive Funktion ist.
Es sei $t = 0$. Dann bestehen die Programme aus den Wertzuweisungen. Wenn wir die im ersten Teil dieses Beweises gegebenen Tabelle von rechts nach links lesen, finden wir zu jeder derartigen Wertzuweisung die zugehörige primitiv-rekursive Funktion, die identisch mit der vom Programm berechneten Funktion ist. Damit ist der Induktionsanfang gesichert.
Es sei nun Π ein Programm der Tiefe $t > 1$. Dann gilt

$$\Pi = \Pi_1; \Pi_2 \text{ oder } \Pi = \textbf{LOOP } y \textbf{ BEGIN } \Pi' \textbf{ END}$$

für gewisse Programme Π_1, Π_2, Π' mit einer Tiefe $\leq t - 1$. Nach Induktionsannahme sind dann alle Funktionen $\Phi_{\Pi_1,j}, \Phi_{\Pi_2,j}, \Phi_{\Pi',j}$ primitiv-rekursiv.
Ist Π als Nacheinanderausführung von Π_1 und Π_2 gegeben, so ergeben sich für die von Π berechneten Funktionen die Beziehungen

$$\Phi_{\Pi,j}(x_1, \ldots x_n) = \Phi_{\Pi_2,j}(\Phi_{\Pi_1,1}(x_1, \ldots, x_n), \Phi_{\Pi_1,2}(x_1, \ldots, x_n), \ldots, \Phi_{\Pi_1,m}(x_1, \ldots, x_n)).$$

Damit entstehen die von Π berechneten Funktionen mittels des Kompositionsschems aus primitiv-rekursiven Funktionen und sind daher selbst primitiv-rekursiv.
Es sei nun Π=**LOOP** y **BEGIN** Π' **END**, wobei wir ohne Beschränkung der Allgemeinheit annehmen, dass y nicht unter den Variablen $x_1, x_2, \ldots, x_n$ des Programms Π' vorkommt (siehe Übungsaufgabe 5). Dann werden die von Π berechneten Funktionen durch das folgende simultane Rekursionsschema bestimmt:

$$\begin{aligned} \Phi_{\Pi,j}(x_1, \ldots, x_n, 0) &= P_j^n(x_1, \ldots, x_n), \\ \Phi_{\Pi,j}(x_1, \ldots, x_n, y+1) &= \Phi_{\Pi',j}(\Phi_{\Pi,1}(x_1, \ldots, x_n, y), \ldots, \Phi_{\Pi,n}(x_1, \ldots, x_n, y)) \end{aligned}$$

(die erste Gleichung legt den Wert der Variablen vor Abarbeitung des Programms fest; um zum Wert für $y > 0$ zu kommen, wird das Programm Π' entsprechend der zweiten Gleichung stets wieder ausgeführt, wobei die im vorhergehenden Schritt erhaltenen Funktionswerte als Eingaben dienen (siehe Kompositionsschema); wie beim **LOOP**-Programm ist y-malige Nacheinanderausführung zur Gewinnung von $\Phi_{\Pi,j}(x_1, \ldots, x_n, y)$ notwendig. Damit ist der Induktionsbeweis auch für diese Richtung geführt. □

Wir wollen nun eine weitere Operation zur Erzeugung von Funktionen einführen, die es uns gestattet, auch partielle Funktionen zu erhalten (mittels Kompositions- und Rekursionsschema erzeugte Funktionen sind offenbar total).

- *μ-Operator*: Für eine $(n+1)$-stellige Funktion h definieren wir die n-stellige Funktion f wie folgt. $f(x_1, x_2, \ldots, x_n) = z$ gilt genau dann, wenn die folgenden Bedingungen erfüllt sind:
  - $h(x_1, x_2 \ldots, x_n, y)$ ist für alle $y \leq z$ definiert,
  - $h(x_1, x_2, \ldots, x_n, y) \neq 0$ für $y < z$,
  - $h(x_1, x_2, \ldots, x_n, z) = 0$.

Wir benutzen die Bezeichnungen

$$f(x_1, \ldots, x_n) = (\mu y)[h(x_1, \ldots, x_n, y) = 0] \quad \text{bzw.} \quad f = (\mu y)[h].$$

Intuitiv bedeutet dies, dass für die festen Parameter $x_1, x_2, \ldots, x_n$ der kleinste Wert von z bestimmt wird, für den $h(x_1, x_2, \ldots, x_n, z) = 0$ gilt (wobei bei nicht überall definierten Funktionen zusätzlich verlangt wird, dass für alle kleineren Werte y als z das Tupel $(x_1, x_2, \ldots x_n, y)$ im Definitionsbereich liegt, sonst ist f an dieser Stelle nicht definiert).

Beispiel 1.4 a) Es gilt

$$(\mu y)[add(x, y)] = \begin{cases} 0 & \text{für } x = 0 \\ nicht\ definiert & \text{sonst} \end{cases}$$

(für $x = 0$ ist wegen $0 + 0 = 0$ offenbar $z = 0$ der gesuchte minimale Wert; für $x > 0$ gilt auch $x + y > 0$ für alle y, und daher existiert kein z mit der dritten Eigenschaft aus der Definition des μ-Operators).

b) Es sei

$$h(x, y) = |9x^2 - 10xy + y^2|.$$

Durch Anwendung des μ-Operators auf h entsteht die Identität, d.h. f mit $f(x) = x$ für alle x.
Dies ist wie folgt leicht zu sehen. Für einen fixierten Wert von x ist $(\mu y)[h(x, y)]$ die kleinste natürliche Nullstelle des Polynoms $x^2 - 10xy + y^2$ in der Unbestimmten y. Eine einfache Rechnung ergibt die Nullstellen x und $9x$. Somit gilt

$$f(x) = (\mu y)[h(x, y)] = x.$$

Wir wollen nun eine Erweiterung der primitiv-rekursiven Funktionen mittels μ-Operator entsprechend Definition 1.6 vornehmen. Da jedoch durch die Anwendung des μ-Operators Funktionen entstehen können, deren Definitionsbereiche echte Teilmengen von $\mathbb{N}_0^n$ sind, müssen wir zuerst Kompositions- und Rekursionsschema auf diesen Fall ausdehnen.
Beim Kompositionsschema ist $f(x_1, x_2, \ldots, x_n)$ genau dann definiert, wenn für $1 \leq i \leq n$ die Funktionen f_i auf dem Tupel $\underline{x} = (x_1, x_2, \ldots, x_n)$ und g auf $(f_1(\underline{x}), f_2(\underline{x}), \ldots, f_m(\underline{x}))$ definiert sind. In ähnlicher Weise kann das Rekursionsschema erweitert werden; die Details dazu überlassen wir dem Leser.

Definition 1.7 *Eine Funktion $f : \mathbb{N}_0^n \to \mathbb{N}_0$ heißt partiell-rekursiv, wenn sie mittels endlich oft wiederholter Anwendung von Kompositionsschema, Rekursionsschema und μ-Operator aus den Basisfunktionen erzeugt werden kann.*

Satz 1.6 *Eine Funktion ist genau dann partiell-rekursiv, wenn sie* **LOOP/WHILE***-berechenbar ist.*

Beweis: Wir gehen wie beim Beweis von Satz 1.5 vor.
Daher reicht es in der ersten Richtung zusätzlich zu den dortigen Fakten zu zeigen, dass jede partiell-rekursive Funktion f, die durch Anwendung des μ-Operators auf h entsteht, **LOOP/WHILE**-berechenbar ist. Nach Induktionsannahme ist h **LOOP/WHILE**-berechenbar, also $h = \Phi_{\Pi,1}$ für ein Programm Π. Um den minimalen Wert z zu berechnen, berechnen wir der Reihe nach die Werte y' an den Stellen mit $0, 1, 2, \ldots$ für die Variable y und testen jeweils, ob der aktuelle Wert von y' von Null verschieden ist. Formal ergibt sich folgendes Programm für f:

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

In der umgekehrten Richtung ist noch die **WHILE**-Anweisung zusätzlich zu betrachten. Es sei also $\Pi'' =$ **WHILE** $x_k \neq 0$ **BEGIN** Π' **END**, wobei nach Induktionsannahme alle Funktionen $\Phi_{\Pi',j}$ partiell-rekursiv sind. Wir konstruieren zuerst die gleichen Funktionen $\Phi_{\Pi,j}$ wie bei der Umsetzung der **LOOP**-Anweisung im Beweis von Satz 1.5. Nach den dortigen Überlegungen gibt $\Phi_{\Pi,j}(x_1, x_2, \ldots x_n, y)$ den Wert der Variablen x_j nach y-maliger Hintereinanderausführung von Π' an. Die $\Phi_{\Pi,j}$ sind partiell-rekursive Funktionen, da sie durch Anwendung des simultanen Rekursionsschemas auf partiell-rekursive Funktionen entstanden sind. Wir betrachten nun die Funktion

$$w(x_1, \ldots, x_n) = (\mu y)[\Phi_{\Pi,k}(x_1, \ldots, x_n, y)],$$

die nach Definition die kleinste Zahl von Durchäufen von Π' liefert, um den Wert 0 bei der Variablen x_k zu erreichen. Entsprechend der Semantik der **WHILE**-Anweisung bricht diese genau nach $w(x_1, \ldots, x_n)$ Schritten ab. Folglich gilt

$$\Phi_{\Pi'',i}(x_1, \ldots, x_n) = \Phi_{\Pi,i}(x_1, \ldots, x_n, w(x_1, \ldots, x_n))$$

(da die rechten Seiten den Wert der Variablen x_i nach $w(x_1, \ldots, x_n)$ Hintereinanderausführungen von Π' und damit bei Abbruch der **WHILE**-Anweisung angeben). Entsprechend dieser Konstruktion ist damit jede von Π'' berechnete Funktion partiell-rekursiv. □

Wir bemerken, dass die Beweise der Sätze 1.5 und 1.6 eine enge Nachbarschaft zwischen dem Berechenbarkeitsbegriff auf der Basis von **LOOP**/**WHILE**-Programmen einerseits und partiell-rekursiven Funktionen andererseits ergibt, da die Wertzuweisungen den Basisfunktionen, die Hintereinanderausführung von Programmen dem Kompositionsschema, die **LOOP**-Anweisung dem Rekursionsschema und die **WHILE**-Anweisung dem μ-Operator im wesentlichen entsprechen.

Durch Kombination der Sätze 1.1 und 1.6 erhalten wir die folgende Aussage.

Folgerung 1.7 *Es gibt eine totale Funktion, die nicht partiell-rekursiv ist.* □

1.1.3 Registermaschinen

Wir wollen nun einen Berechenbarkeitsbegriff behandeln, der auf einer Modellierung der realen Rechner basiert.

Definition 1.8 *i) Eine Registermaschine besteht aus den Registern*

$$B,\ C_0,\ ,C_1, C_2, \ldots, C_n, \ldots$$

und einem Programm.
B heißt Befehlszähler, C_0 heißt Arbeitsregister oder Akkumulator, und jedes der Register C_n, $n \geq 1$, heißt Speicherregister.
Jedes Register enthält als Wert eine natürliche Zahl.
ii) Unter einer Konfiguration der Registermaschine verstehen wir das unendliche Tupel

$$(b, c_0, c_1, \ldots, c_n, \ldots),$$

wobei

- *das Register B die Zahl b enthält,*
- *für $n \geq 0$ das Register C_n die Zahl c_n enthält.*

iii) Das Programm ist eine endliche Folge von Befehlen. Durch die Anwendung eines Befehls wird die Konfiguration der Registermaschine geändert. Die folgende Liste gibt die zugelassenen Befehle und die von ihnen jeweils bewirkte Änderung der Konfiguration $(b, c_o, c_1, \ldots c_n, \ldots)$ in die Konfiguration $(b', c'_0, c'_1, \ldots, c'_n, \ldots)$ an, wobei für die nicht angegebenen Komponenten $u' = u$ gilt:

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.