Studienbücherei

B. A. Trachtenbrot

Algorithmen und Rechenautomaten

Studienbücherei

Algorithmen und Rechenautomaten

B. A. Trachtenbrot

Mit 55 Abbildungen
und 3 Tabellen

VEB Deutscher Verlag
der Wissenschaften
Berlin 1977

Б. А. Трахтенброт
Алгоритмы и вычислительные автоматы
Издательство „Советское радио“
Москва 1974

Übersetzung aus dem Russischen:
Prof. Dr. Günter Asser, Dr. Hans-Dietrich Hecker, Dr. Lutz Voelkel

Das Original trägt die Widmung:
Dieses Buch ist dem Andenken an Alexej Andreevič Ljapunov gewidmet.

Verlagslektoren: Dipl.-Math. Gesine Reiher, Dipl.-Math. Brigitte Mai
Umschlaggestaltung: Rudolf Wendt

Printed in the German Democratic Republic
Lizenz-Nr. 206 · 435/105/77
Gesamtherstellung: VEB Druckhaus „Maxim Gorki“, 74 Altenburg
LSV 1094
Bestellnummer 570 496 8
DDR 17,80 M

Vorwort zur russischen Ausgabe

Das vorliegende Buch ist eine Einführung in die Algorithmentheorie. Es ist der Erklärung eines der wichtigsten Begriffe der Mathematik, des Algorithmenbegriffs, gewidmet. In ihm wird ein Kreis von Fragen behandelt, die in das Grenzgebiet zwischen mathematischer Logik und Theorie der Rechenautomaten gehören.

Das Buch besteht aus drei fast umfangsgleichen Kapiteln. Das erste Kapitel „Algorithmen" gehört noch nicht zur eigentlichen Algorithmentheorie, obwohl in ihm ständig von Algorithmen die Rede ist. Grund hierfür ist, daß dort das Wort „Algorithmus" im intuitiven Sinne benutzt wird, die eigentliche Theorie aber erst dann beginnt, wenn dieser Terminus einen exakt definierten mathematischen Begriff bezeichnet. Trotzdem ist der Inhalt dieses Kapitels, das vorbereitenden Charakter trägt, sehr wichtig, da es nützliche Einsichten in wesentliche Züge der Theorie vermittelt. Das zweite Kapitel „Turing-Maschinen" führt unmittelbar in den Kreis der Grundbegriffe der Algorithmentheorie. Das dritte Kapitel „Algorithmische Probleme" bildet den Höhepunkt des Buches. Es enthält solche Resultate der Theorie, für die eine einigermaßen elementare Darstellung möglich ist. Ein ausführlicher Überblick über den Inhalt der Kapitel wird in der Einleitung gegeben.

Die Anregung zum Versuch einer Popularisierung der Algorithmentheorie in schriftlicher Form erhielt der Autor durch Schwierigkeiten und Ärgernisse, die sich bei seinen ersten Versuchen einer mündlichen Popularisierung ergaben. Das war vor mehr als 20 Jahren der Fall, als der Autor vor seinen damaligen Kollegen, den Mathematikern des Pädagogischen Instituts in Penza, über Fragen der Algorithmentheorie vortrug. Solche Ideen und Resultate der Algorithmentheorie, wie die Formalisierung des Berechenbarkeitsbegriffs und die Existenz von algorithmisch unlösbaren Problemen, erschienen damals vielen nicht nur ungewöhnlich, sie wirkten auch auf die meisten Mathematiker abschreckend. Damit ergab sich für den Autor die dringende Aufgabe, in populären Artikeln und kleinen Büchern den Leser mit den Grundbegriffen der Algorithmentheorie vertraut zu machen und ihn auf möglichst kurzem Wege zu den Sätzen über die algorithmische Unlösbarkeit konkreter Probleme zu führen. Im Ergebnis dessen erschien 1956 in der Zeitschrift „Математика

в школе" ein Artikel mit dem Titel „Алгоритмы и машинное решение задач" (Algorithmen und die maschinelle Lösung von Aufgaben), der in erweiterter Form zweimal (Гостехиздат 1957, Физматгиз 1960) als Buch herausgegeben wurde, das dann auch in eine Reihe anderer Sprachen übersetzt wurde.[1])

Die Abschnitte 1.1 bis 2.3 und 2.7 bis 3.3 des vorliegenden Buches enthalten den wesentlichen Inhalt der genannten Publikationen, der einer sorgfältigen Durchsicht und Überarbeitung unterzogen wurde. Darüber hinaus enthält das Buch aber auch umfangreiches neues Material, insbesondere in den Abschnitten 2.4 bis 2.6, 3.4 bis 3.10 und den Schlußbemerkungen, denen Vorlesungen zugrunde liegen, die vom Autor an der Universität Novosibirsk für Studenten der Abteilung Mathematische Linguistik und für Hörer der Fakultät für Weiterbildung gehalten wurden. Die hier behandelten Fragen sind unserer Meinung nach von Bedeutung für das Verständnis von Gebieten der Algorithmentheorie, die nicht mehr unmittelbar mit dem Problemkreis der algorithmischen Unlösbarkeit verknüpft sind. Dabei werden hauptsächlich die folgenden drei Problemkreise betrachtet: Programmierung (Abschnitt 2.4 bis 2.6), Güte von Algorithmen (Abschnitt 3.4 bis 3.7), Automaten mit Parallelarbeit (Abschnitt 3.8 bis 3.10). Diese Dinge konnten schon deshalb nicht in den früheren Publikationen behandelt werden, weil die meisten der hier verarbeiteten Resultate erst später erhalten wurden.

Die Abschnitte, deren Lektüre etwas mehr Anstrengung erfordert, obwohl auch sie keine speziellen Vorkenntnisse voraussetzen, sind im Inhaltsverzeichnis mit einem Stern versehen. Sie können bei einer ersten Lektüre übergangen werden. Bibliographische Angaben finden sich hauptsächlich in den Schlußbemerkungen am Ende des Buches.

Am 23. Juni 1973, als sich das Buch bereits im Umbruch befand, verstarb ALEXEJ ANDREEVIČ LJAPUNOV, der hervorragende Mathematiker und Kybernetiker, bedeutende Pädagoge und Propagandist der Wissenschaft. Die langjährige Verbindung mit ihm hat dem Autor viel gegeben und nicht zuletzt auch auf den Inhalt dieses Buches gewirkt. Seine moralische Unterstützung und ständige Ermunterung hatte große Bedeutung sowohl für die Abfassung der älteren Teile des Buches als auch für das spätere Schaffen des Autors in der Zeit der gemeinsamen Arbeit an dem von A. A. LJAPUNOV gegründeten und geleiteten Lehrstuhl für Theoretische Kybernetik der Universität Novosibirsk. Der Autor sieht in diesem Buch einen bescheidenen Beitrag zum Gedenken an ALEXEJ ANDREEVIČ LJAPUNOV.

Novosibirsk, Akademgorodok, November 1973

B. TRACHTENBROT

[1]) Eine deutschsprachige Übersetzung erschien in mehreren Auflagen unter dem Titel „Wieso können Automaten rechnen?" im VEB Deutscher Verlag der Wissenschaften. [*Anm. d. Übers.*]

Inhalt

Einleitung

Der Begriff des Algorithmus gehört zu den Grundbegriffen der Mathematik. Unter einem Algorithmus versteht man eine genaue Vorschrift, nach der ein gewisses System von Operationen in einer bestimmten Reihenfolge auszuführen ist und nach der man alle Aufgaben eines gegebenen Typs lösen kann.

Das ist natürlich keine genaue Definition des Begriffs Algorithmus. Es wird nur ungefähr die inhaltliche Bedeutung des Wortes „Algorithmus" erklärt. Dennoch ist diese Formulierung jedem Mathematiker geläufig und verständlich. Sie erfaßt den Begriff des Algorithmus, wie er sich im Laufe der Zeit herausgebildet hat und schon im Altertum in der Mathematik verwendet wurde.

Einfache Beispiele für Algorithmen sind die Regeln, nach denen im dezimalen Zahlensystem die vier Grundrechenarten ausgeführt werden. Das Wort „Algorithmus" geht auf den Namen des usbekischen Gelehrten ALHWARIZMI zurück, der schon im 9. Jahrhundert solche Regeln aufstellte. In der Mathematik gilt eine Schar von Aufgaben eines bestimmten Typs als gelöst, wenn ein Algorithmus zur Lösung jeder Aufgabe dieses Typs aufgestellt worden ist. So gibt es z. B. in der Algebra Algorithmen, mit deren Hilfe man aus den Koeffizienten einer beliebigen algebraischen Gleichung die Anzahl und auch die Vielfachheit der verschiedenen Wurzeln der betreffenden Gleichung ermitteln und zugleich diese Wurzeln mit beliebig vorgegebener Genauigkeit berechnen kann.[1]) Das Aufstellen von Algorithmen gehört zu den natürlichen Aufgaben der Mathematik.

---

[1]) Man muß deutlich zwischen der Lösung einer einzelnen, konkreten Aufgabe und einer Schar von Aufgaben unterscheiden. Eine konkrete Aufgabe liegt z. B. vor, wenn die Wurzeln einer ganz bestimmten algebraischen Gleichung zu ermitteln sind. Die Lösung dieser Aufgabe besteht in der Berechnung der Wurzeln eben dieser Gleichung. Eine *Aufgabenschar* wird dagegen meistens als „Problem" formuliert, z. B. eine „Methode" zu finden, nach der man für jede algebraische Gleichung ihre Wurzeln bestimmen kann. Die Lösung einer Aufgabenschar besteht also in der Angabe einer einheitlichen Vorschrift (einer Methode, eines Algorithmus), mittels derer man jede konkrete Aufgabe aus der gegebenen Schar lösen kann, wenn man die konkreten Werte der veränderlichen Parameter, die die einzelne Aufgabe der Schar charakterisieren, kennt.

Prozesse, die nach streng definierten formalen Vorschriften, d. h. nach einem Algorithmus, ablaufen, sind aber nicht nur in der Mathematik, sondern auch in vielen anderen Bereichen der menschlichen Tätigkeit anzutreffen. So setzen sich z. B. in der Buchhaltung und in der Planung die Analyse der anfallenden Daten, deren Verarbeitung, die Aufstellung von Materialbilanzen zur Erzielung optimaler Resultate usw. aus einer langen Reihe von Elementaroperationen bestimmter Typen zusammen, die nach strengen Instruktionen und Schemata auszuführen sind. Die Zahl derartiger Beispiele ließe sich beliebig vergrößern. In den Abschnitten 1.1 bis 1.4 werden wir an einer Reihe von Beispielen die Arbeitsweise von Algorithmen erklären und Algorithmen zur Lösung gewisser Klassen von mathematischen und logischen Aufgaben angeben.

Das Aufstellen eines Algorithmus (insbesondere eines „guten", bequemen Algorithmus) zur Lösung von Aufgaben eines bestimmten Typs bedarf, wenn es überhaupt gelingt, im allgemeinen schwieriger und komplizierter Überlegungen, die eine hohe Qualifikation und große Erfindungsgabe erfordern. Wenn jedoch einmal ein solcher Algorithmus vorhanden ist, wird der Lösungsprozeß für Aufgaben des betreffenden Typs so einfach, daß er ohne die geringsten Kenntnisse über das Wesen der vorgelegten Aufgabe vollzogen werden kann. Derjenige, welcher die Aufgabe löst, muß nur fähig sein, die Elementaroperationen, aus denen sich der Prozeß zusammensetzt, auszuführen, und er muß gewissenhaft und akkurat nach der vorliegenden Beschreibung (dem Algorithmus) vorgehen. Ein Mensch, der so rein mechanisch oder maschinell vorgeht, kann jede Aufgabe des betrachteten Typs erfolgreich lösen. Die Redeweise „maschinelles Vorgehen", die meistens im übertragenen Sinne gebraucht wird, bekommt beim heutigen Stand von Wissenschaft und Technik auch einen direkten Sinn. Ein Mensch nämlich, der, von einem Algorithmus geführt, eine Aufgabe löst, kann prinzipiell auch durch eine Maschine ersetzt werden, die denselben Prozeß ausführt. Maschinen, die diese Fähigkeit haben, sind z. B. die modernen „Rechenautomaten".

In den letzten 20 Jahren sind die Rechenautomaten wesentlich weiterentwickelt worden, und sie werden heute zur Lösung der verschiedensten Probleme herangezogen. Die Besonderheit dieser Maschinen ist, daß sie vom Moment der Eingabe der Anfangsdaten (der Parameter der konkreten Aufgabe) und des Programms (des Lösungsalgorithmus) an ohne jede Einwirkung des Menschen bis zur Ausgabe des Endresultats arbeiten. Die Leistungsfähigkeit der modernen Rechenautomaten ist gewaltig: Sie können in einer Sekunde mehrere Millionen arithmetischer Operationen ausführen. Auch ihr Anwendungsbereich erweitert sich ständig: Rechenautomaten lösen komplizierte Gleichungen und Gleichungssysteme, sie übersetzen von einer Sprache in eine andere, sie spielen Schach, u. v. a. m. Eine große Anwendungsperspektive in der Produktion haben Automaten, die technologische Prozesse im Maßstab von Großbetrieben steuern. Darüber hinaus ermöglichen Rechenautomaten eine schnelle und sichere mathematische Bearbeitung und Analyse experimenteller Daten und schaffen damit die Voraussetzungen für die Entwicklung neuer, früher

unerreichbarer Forschungsmethoden in vielen Wissenschaftsgebieten. Es ist heute wohl allgemein bekannt, daß die Rechenautomaten ein mächtiges Werkzeug bilden, das nicht nur die Arbeit des Menschen erleichtert, sondern ihn vielfach sogar vollständig von gewissen Formen umfangreicher und anstrengender geistiger Arbeit befreit.

In den Abschnitten 1.4 und 1.5 werden der Aufbau und die Arbeitsweise von elektronischen Rechenautomaten sowie die Grundprinzipien für deren Programmierung, d. h. der Realisierung von Algorithmen in solchen Maschinen, kurz behandelt. Moderne Maschinen mit automatischer Steuerung heißen elektronisch, weil ihre wichtigsten Bauelemente nach Prinzipien der Elektronik funktionieren. Das ermöglicht gerade die ökonomisch wichtigen hohen Operationsgeschwindigkeiten dieser Maschinen. Das Grundprinzip dieser Maschinen, die automatische Steuerung der Prozesse, hängt jedoch nicht von der Verwendung der Elektronik ab. Die ersten funktionstüchtigen Rechenautomaten, die um 1940 konstruiert wurden, arbeiteten elektromechanisch. Im Prinzip könnte man die elektronischen Bauelemente sogar durch mechanische Elemente (Zahnräder, Hebel u. ä.) ersetzen, d. h. mechanische Rechenmaschinen mit automatischer Steuerung bauen, die genau dasselbe leisten wie die elektronischen Rechenautomaten, nur natürlich bedeutend langsamer.[1]) Das Entstehen der modernen Rechenautomaten darf also nicht ausschließlich als Resultat der Erfindungsgabe der Ingenieure oder der Entwicklung der Elektronik angesehen werden. Schon einige Jahre vor der Konstruktion der ersten arbeitsfähigen Exemplare von Rechenmaschinen mit automatischer Steuerung wurden deren Grundprinzipien von Mathematikern, genauer gesagt, von mathematischen Logikern, vorweggenommen. Im Jahre 1936 entwickelten der englische Mathematiker A. M. Turing und der amerikanische Mathematiker E. L. Post unabhängig voneinander den Begriff einer abstrakten Rechenmaschine. Derartige Turing-Post-Maschinen wurden allerdings niemals gebaut, obwohl dies technisch durchaus möglich gewesen wäre; der technische Fortschritt schlug einen anderen Weg zur Verwirklichung großer (sogenannter universeller) Rechenautomaten ein. Ungeachtet dessen sind die Konstruktionen von Turing und Post ein glänzendes Beispiel für eine wissenschaftliche Voraussage, denn gerade mit diesen abstrakten mathematischen Modellen wurde die Existenz von universellen Rechenautomaten streng bewiesen (vgl. Abschnitt 3.1). Die Turing-Maschine wird heute in der Algorithmentheorie verbreitet als Grundmodell zur Präzisierung des Inhalts der Begriffe Berechenbarkeit und Algorithmus sowie zur Klärung der Beziehungen zwischen Algorithmen und Rechenautomaten verwendet.[2])

[1]) Ein erstes Projekt für eine automatische Rechenmaschine, die man als einen gewissen Vorläufer heutiger Rechenautomaten ansehen kann, wurde 1833 von dem englischen Erfinder John Babbage in Form einer mechanischen Vorrichtung ausgearbeitet. Aufgrund materieller Schwierigkeiten gelang es ihm jedoch nicht, sein Projekt voll zu verwirklichen.

[2]) Die modernen elektronischen Rechenautomaten unterscheiden sich in ihrem Aufbau beträchtlich von den gedanklichen Konstruktionen von Turing und Post und entsprechen viel stärker den Konstruktionen von Babbage und von anderen Erfindern. Aus diesem Grunde

Das zweite Kapitel des vorliegenden Buches ist einem detaillierten Studium der Turing-Maschinen und der Realisierung von Algorithmen auf Turing-Maschinen gewidmet. Wesentlichen Raum nehmen hierbei Fragen der Turing-Programmierung ein, die auch für das Verständnis von Problemen der Programmierung realer Rechenautomaten nützlich sein dürften. Hierzu gehört insbesondere das Problem der automatischen Programmierung (vgl. Abschnitt 2.4). Neben den Turing-Maschinen gibt es zahlreiche andere abstrakte Modelle für Rechenautomaten. Sie sind jedoch alle untereinander und damit speziell dem Turing-Modell in folgendem Sinne gleichwertig: Die Klasse der Probleme, die man auf abstrakten Maschinen des einen Typs lösen kann, stimmt mit der Klasse der Probleme überein, die auf abstrakten Maschinen des anderen Typs, also speziell auch auf Turing-Maschinen, lösbar sind. Damit bekommen die folgenden beiden Fragen einen mathematisch exakten Sinn und können mit einer mathematisch exakten Antwort versehen werden:

Was können Automaten grundsätzlich leisten?
Wie tun sie das?

Das dritte Kapitel enthält Material, aus dem man bestimmte Antworten auf diese Fragen entnehmen kann.

Die erste Frage erfordert eine Erklärung, welche Formen geistiger Arbeit durch Rechenautomaten übernommen werden können. Ihre Aktualität und Tragweite kommt vor allem darin zum Ausdruck, daß die durch die modernen realen Rechenautomaten erzielten Erfolge zum Teil phantastische Prognosen und unerfüllbare Illusionen hinsichtlich allmächtiger Maschinen („gigantischer Elektronengehirne") hervorgerufen haben. Die genaue Analyse der Prozesse, die nach einem gegebenen Algorithmus ablaufen, wie auch der in realen Rechenautomaten ablaufenden Prozesse führte demgegenüber zu einer ganz anderen Erkenntnis. Es wurde streng bewiesen, daß es Typen von Aufgaben gibt, zu denen kein einheitliches effektives Verfahren, kein Algorithmus existiert, mit dessen Hilfe man alle Aufgaben dieses Typs lösen kann; in diesem Sinne ist also eine maschinelle Lösung der Aufgaben eines solchen Typs unmöglich. Seitdem konzentrieren sich zahlreiche Untersuchungen in

kann man TURING und POST nicht unmittelbar zu den Schöpfern oder Pionieren auf dem Gebiet der modernen Rechentechnik zählen. [TURING arbeitete allerdings von 1945 bis 1948 am National Laboratory of Physics in einer Gruppe, die sich mit dem Bau und dem Einsatz einer automatischen Großrechenanlage befaßte und das Projekt der automatischen Rechenmaschine ACE erarbeitete; er stellte als erster eine Reihe von Programmen auf. Als 1948 an der Universität Manchester mit dem Bau einer Rechenmaschine begonnen wurde, wechselte TURING nach dort über und übernahm die Leitung der mit diesem Projekt zusammenhängenden mathematischen Arbeiten. Insbesondere befaßte er sich mit dem Problem der Zusammensetzung großer Programme aus Unterprogrammen. Auch die Frage, ob und wie Maschinen lernen können, hat ihn als einen der ersten bewegt. — *Anm. d. Übers.*] Vielmehr begründeten sie das philosophische Verständnis dafür, daß automatisch arbeitende Rechenmaschinen möglich sind und bewiesen die Existenz universeller Maschinen, und das zu einer Zeit, als es solche Maschinen real noch nicht gab. An der Entwicklung und Konstruktion der realen modernen programmgesteuerten Rechenautomaten hatte der zu den bedeutendsten Mathematikern des 20. Jahrhunderts zu rechnende J. v. NEUMANN großen Anteil.

der Algorithmentheorie (und auch in anderen Gebieten der Mathematik) auf die Frage nach der Existenz von Lösungsalgorithmen für bestimmte Typen von Aufgaben. Die in dieser Richtung erzielten großen Erfolge führten zu einem besseren Verständnis dafür, was Rechenautomaten leisten bzw. nicht leisten können. Hierzu gehören insbesondere die relativ einfach zu formulierenden Resultate von A. A. Markov und P. S. Novikov, die wir in den Abschnitten 1.4, 3.2 und 3.3 darstellen werden.

Nun zur zweiten Frage: Wenn zwei Maschinen dasselbe tun oder zu tun fähig sind, so bedeutet das nicht, daß sie es in derselben Weise tun. Diese Bemerkung betrifft sowohl verschiedene Maschinen desselben Typs (z. B. zwei Turing-Maschinen) als auch Vertreter verschiedener Maschinentypen. So setzt sich z. B. der in einer Turing-Maschine ablaufende Prozeß aus einer Folge von Einzelschritten zusammen, wobei in jedem Einzelschritt nur ein kleiner Teil der Maschine in Aktion ist. Demgegenüber ist ein anderer Maschinentyp, der sogenannte v.-Neumann-Automat durch einen hohen Grad von Parallelarbeit charakterisiert, d. h., in einem solchen Automaten wird jeder Einzelschritt gleichzeitig „überall" wirksam. Umstände dieser oder ähnlicher Art können dazu führen, daß bei einem speziellen Typ von Aufgaben ein bestimmter Typ von Automaten oder Algorithmen den Vorzug verdient, selbst wenn sie alle gleichwertig sind. Gleichwertige Algorithmen oder Automaten können sich in bestimmten Situationen durchaus als besser oder schlechter erweisen. Damit führt die sehr allgemein formulierte zweite Frage zu einer Vielzahl ganz spezieller Fragen, die mit verschiedenen Klassifikationen der abstrakten Rechenautomaten und einer Abschätzung der Güte der von ihnen realisierten Algorithmen verknüpft sind. Problemen dieser Art sind die Abschnitte 3.4 bis 3.10 gewidmet.

Abschließend noch zwei Bemerkungen zu der von uns gewählten Art der Darstellung: Wegen ihrer Länge können nicht alle Beweise vollständig gebracht werden. Daher ist die Darstellung an einzelnen Stellen weder streng noch vollständig, was jedoch, wie uns scheint, kein Nachteil ist, sondern eher das Verständnis des Wesens der Sache erleichtert. Zur Abrundung des Bildes werden in den Abschnitten 2.1 und 3.2 einige Fakten in Form eines Überblicks gegeben. Bei der Beschreibung des Aufbaus von Maschinen gehen wir nicht auf technische Einzelheiten ein. Vielmehr richten wir unser Augenmerk auf das Zusammenwirken der einzelnen Teile der Maschine. Das entspricht der Zielsetzung des Buches, die in der Erörterung der mathematischen und logischen Möglichkeiten und nicht in der Beschreibung technischer Einzelheiten besteht.

1. Algorithmen

1.1. Numerische Algorithmen

1.1.1. Die Grundrechenarten. Der Euklidische Algorithmus

Wie schon in der Einleitung gesagt, sind einfache Beispiele für Algorithmen die Regeln, nach denen im Dezimalsystem die Grundrechenarten ausgeführt werden. So läßt sich beispielsweise die Addition zweier mehrstelliger natürlicher Zahlen auf eine Kette elementarer Operationen zurückführen, bei denen der Rechner jeweils nur zwei entsprechende Ziffern der Summanden zu beachten braucht, von denen eine mit einem Zeichen versehen sein kann, das auf einen Zehner-Übertrag hinweist. Hierbei treten zwei Arten von Operationen auf:

1. das Niederschreiben der entsprechenden Ziffer der Summe;
2. der Hinweis auf einen Übertrag an die links benachbarte Ziffer.

Ferner ist vorgeschrieben, in welcher Reihenfolge die Operationen auszuführen sind (von rechts nach links). Der formale Charakter dieser elementaren Operationen besteht darin, daß sie sich mit Hilfe einer ein für allemal vorgegebenen Additions- und Übertragstabelle für die Ziffern automatisch ausführen lassen, von deren inhaltlichen Bedeutung völlig abgesehen werden kann.

Ähnlich steht es mit den anderen drei Grundrechenarten sowie dem Quadratwurzelziehen und ähnlichen Operationen. Der formale Charakter der entsprechenden Vorschriften (Algorithmen) läßt offenbar keinerlei Zweifel aufkommen (bei Schülern kann das besonders bei der Vorschrift für das Quadratwurzelziehen beobachtet werden).

Als weiteres Beispiel wollen wir den Euklidischen Algorithmus betrachten, mit dem man alle Aufgaben des folgenden Typs lösen kann:

Zu zwei natürlichen Zahlen a, b ist der größte gemeinsame Teiler zu bestimmen.

Die verschiedenen Aufgaben dieses Typs entsprechen den verschiedenen Zahlenpaaren a, b.

Bekanntlich läßt sich die Lösung jeder solchen Aufgabe mittels einer fallenden Folge von Zahlen konstruieren, von denen die erste die größere der beiden vorgegebenen Zahlen, die zweite die kleinere, die dritte der Rest bei der Division der ersten

durch die zweite, die vierte der Rest bei der Division der zweiten durch die dritte ist, und so fort. Der Prozeß wird so lange fortgesetzt, bis die Division ohne Rest aufgeht. Der Divisor der letzten Division ist der gesuchte größte gemeinsame Teiler der beiden Zahlen.

Da man die Division auf wiederholte Subtraktionen zurückführen kann, läßt sich die Vorschrift zur Lösung einer solchen Aufgabe auch in Form folgender, nacheinander auszuführender Anweisungen angeben:

Anweisung 1. Notiere die Zahlen a und b. Gehe zur folgenden Anweisung über.

Anweisung 2. Vergleiche die notierten Zahlen ($a = b$ oder $a < b$ oder $a > b$). Gehe zur folgenden Anweisung über.

Anweisung 3. Wenn die beiden zuletzt notierten Zahlen gleich sind, so leistet jede von ihnen das Gewünschte; dann ist die Rechnung abzubrechen. Ist das nicht der Fall, dann gehe zur folgenden Anweisung über.

Anweisung 4. Wenn die erste der zuletzt notierten Zahlen kleiner als die zweite ist, so vertausche ihre Plätze und gehe zur folgenden Anweisung über.

Anweisung 5. Subtrahiere die zweite Zahl von der ersten und notiere die beiden folgenden Zahlen: den Subtrahenden und die Differenz. Gehe zur Anweisung 2 über.

Wenn also alle fünf Anweisungen ausgeführt worden sind, soll man zur zweiten zurückkehren, danach zur dritten, vierten, fünften und wiederum zur zweiten, dritten usw. übergehen, bis zwei gleiche Zahlen auftreten; dann ist die in der dritten Anweisung enthaltene Bedingung erfüllt, und der Prozeß bricht ab.

In der Mathematik werden Algorithmen natürlich nicht immer so pedantisch formal beschrieben. Es dürfte aber einleuchten, daß jeder der bekannten Algorithmen in dieser Form angegeben werden kann.

In der angeführten Beschreibung des Euklidischen Algorithmus ist der Lösungsprozeß in mehrere elementare Operationen aufgegliedert worden: Subtraktion zweier Zahlen, Vergleich zweier Zahlen, Vertauschung der Plätze zweier Zahlen. Man sieht leicht ein, daß diese Aufgliederung noch viel weiter getrieben werden kann. So läßt sich z. B. die Anweisung 5 bezüglich der Subtraktion der notierten Zahlen in ein System von Anweisungen zerlegen, das den Algorithmus der Subtraktion zweier Zahlen beschreibt; da aber die Algorithmen für die Grundrechenarten hinreichend bekannt und einfach sind, wird man von einer weiteren Aufgliederung Abstand nehmen.

1.1.2. Numerische Algorithmen

Algorithmen, in denen die vier Grundrechenarten eine wesentliche Rolle spielen, werden häufig als *numerische Algorithmen*[1]) bezeichnet. Wir finden sie in allen Gebieten der elementaren und auch der höheren Mathematik. Sie werden in Worten,

[1]) Man beachte, daß dieser Terminus keine fest umrissene Bedeutung hat.

durch Formeln oder durch verschiedenartige Schemata beschrieben. So kann beispielsweise der Algorithmus zur Auflösung eines Systems von zwei linearen Gleichungen mit zwei Unbekannten

$$a_1x + b_1y = c_1,$$
$$a_2x + b_2y = c_2$$

durch die Formeln

$$x = \frac{c_1b_2 - c_2b_1}{a_1b_2 - a_2b_1}, \quad y = \frac{a_1c_2 - a_2c_1}{a_1b_2 - a_2b_1}$$

beschrieben werden. Durch diese Formeln wird die Art der auszuführenden Operationen sowie ihre Reihenfolge festgelegt.[1]) Diese Formeln beschreiben für alle Aufgaben des angegebenen Typs (d. h. für beliebige Koeffizienten $a_1, b_1, c_1, a_2, b_2, c_2$) dieselbe Kette von Rechenoperationen.

Es sei aber darauf hingewiesen, daß im allgemeinen die Anzahl der Operationen, die zur Ausführung eines Algorithmus notwendig sind, im voraus nicht bekannt zu sein braucht. Diese Anzahl kann von den konkreten Bedingungen jeder einzelnen Aufgabe abhängen und braucht sich erst im Lösungsprozeß selbst zu ergeben. Zum Beispiel gilt das für den Euklidischen Algorithmus. Hier hängt die Anzahl der notwendigen Subtraktionen von der speziellen Auswahl des Zahlenpaares a, b ab.

Die weite Verbreitung der numerischen Algorithmen erklärt sich daraus, daß man viele andere Operationen auf die vier Grundrechenarten zurückführen kann. In vielen Fällen ist diese Rückführung nicht völlig exakt, kann jedoch mit jeder beliebig vorgegebenen Genauigkeit ausgeführt werden. Das läßt sich sehr gut am Beispiel des üblichen Algorithmus zum Quadratwurzelziehen erkennen, der die Quadratwurzel im allgemeinen zwar nur angenähert, aber mit beliebig vorgegebener Genauigkeit mit Hilfe von aufeinanderfolgenden Divisionen, Multiplikationen und Subtraktionen zu ermitteln gestattet. In einem speziellen Zweig der modernen Mathematik, der praktischen Analysis, werden Verfahren erarbeitet, welche komplizierte Operationen der Analysis, wie Integration, Differentiation usw., auf die vier Grundrechenarten zurückführen.

In Fällen, in denen man keinen Algorithmus zur Lösung aller Aufgaben eines gegebenen Typs besitzt, kann es durchaus möglich sein, einzelne Aufgaben dieses Typs zu lösen. Ein solches für den Einzelfall geschaffenes Verfahren wird jedoch für die Mehrheit der übrigen Fälle unbrauchbar sein und ist daher kein Algorithmus für die betrachtete Aufgabenklasse.

[1]) Bei diesen Überlegungen wird stillschweigend angenommen, daß es sich um eindeutig lösbare Gleichungssysteme handelt, d. h. solche, bei denen $a_1b_2 - a_2b_1 \neq 0$ ist. [*Anm. d. Übers.*]

1.1.3. Diophantische Gleichungen

Wir kommen nun zu einem Beispiel für einen Typ von Aufgaben, zu deren Lösung die Mathematik nachweisbar über keinen Algorithmus verfügt.

Wir betrachten alle möglichen diophantischen Gleichungen, d. h. Gleichungen der Form

$$P = 0,$$

wobei P ein Polynom mit ganzzahligen Koeffizienten ist.[1]) Beispiele für solche Gleichungen sind

$$x^2 + y^2 - z^2 = 0,$$
$$6x^{18} - x + 3 = 0.$$

Die erste dieser Gleichungen enthält drei, die zweite eine Unbestimmte (im allgemeinen werden Gleichungen mit beliebig vielen Unbekannten betrachtet). Eine solche Gleichung kann ganzzahlige Lösungen besitzen oder auch nicht. Die erste Gleichung besitzt z. B. die ganzzahlige Lösung

$$x = 3, \quad y = 4, \quad z = 5,$$

während die zweite Gleichung keine ganzzahlige Lösung besitzt; man beweist nämlich leicht, daß für jedes ganzzahlige x die Ungleichung

$$6x^{18} > x - 3$$

gilt.

Auf dem Internationalen Mathematikerkongreß in Paris im Jahre 1900 legte der berühmte Göttinger Mathematiker David Hilbert eine Liste von 23 schwierigen Problemen vor und lenkte die Aufmerksamkeit der mathematischen Öffentlichkeit auf die Wichtigkeit ihrer Lösung.[2]) Unter ihnen befand sich auch das folgende Problem (das 10. Hilbertsche Problem):

Es ist ein Algorithmus anzugeben, mit dessen Hilfe man von einer beliebigen diophantischen Gleichung feststellen kann, ob sie eine ganzzahlige Lösung besitzt oder nicht.

Für den Spezialfall der diophantischen Gleichungen mit einer einzigen Unbekannten ist ein solcher Algorithmus längst bekannt. Wenn nämlich eine Gleichung

[1]) Näheres über solche Gleichungen findet man z. B. in dem Bändchen von A. O. Gelfond, Die Auflösung von Gleichungen in ganzen Zahlen (Diophantische Gleichungen), 5. Auflage, VEB Deutscher Verlag der Wissenschaften, Berlin 1973. [*Anm. d. Übers.*]

[2]) Ein Neudruck der Hilbertschen Probleme mit ausführlichen Kommentaren hervorragender sowjetischer Mathematiker über die Entwicklung und den gegenwärtigen Stand der Lösung dieser Probleme, die sich zum überwiegenden Teil als Schlüsselprobleme für den Fortschritt in den ausgewählten Gebieten der Mathematik erwiesen haben, ist 1971 unter dem Titel „Die Hilbertschen Probleme" als Band 252 der Reihe „Ostwald's Klassiker der exakten Wissenschaften" erschienen. [*Anm. d. Übers.*]

mit ganzzahligen Koeffizienten

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0$$

die ganzzahlige Wurzel x_0 haben soll, so muß a_0 durch x_0 teilbar sein. Damit läßt sich folgender Algorithmus angeben:

1. Bestimme alle Teiler der Zahl a_0 (es sind endlich viele).

2. Setze jeden Teiler in die linke Seite der Gleichung ein und berechne deren numerischen Wert.

3. Nimmt die linke Seite für einen gewissen Teiler den Wert Null an, so ist dieser Teiler eine Wurzel der Gleichung; wenn aber die linke Seite der Gleichung für jeden Teiler von Null verschieden ist, so besitzt die Gleichung keine ganzzahlige Lösung.

Mit dem 10. Hilbertschen Problem befaßten sich viele hervorragende Mathematiker, ohne daß es ihnen gelang, auch nur für Polynome zweiten Grades einen solchen Algorithmus anzugeben. Im Zusammenhang mit der Entwicklung der Algorithmentheorie und der Entdeckung algorithmisch unlösbarer Probleme verdichtete sich der Verdacht, daß es vielleicht nachweisbar keinen derartigen Algorithmus gibt, eine Möglichkeit, die wahrscheinlich von HILBERT gar nicht in Betracht gezogen wurde. Aber auch die Bemühungen in dieser Richtung blieben lange Zeit ohne Erfolg, bis es Ende 1969 dem jungen Leningrader Mathematiker JU. V. MATIJASEVIČ gelang, die Nichtexistenz eines solchen Algorithmus zu beweisen. Der genaue Sinn dieses negativen Resultats wird dem Leser erst aus den späteren Darlegungen klar werden.

Die folgenden Eigenschaften numerischer Algorithmen, die in den bisher betrachteten Beispielen genügend klar erkennbar sein dürften, sind auch für jeden anderen Algorithmus charakteristisch:

Die Determiniertheit eines Algorithmus. Es wird gefordert, daß man das Rechenverfahren jeder anderen Person in Form endlich vieler Anweisungen mitteilen kann, die besagen, wie man in den einzelnen Stadien der Rechnung vorzugehen hat. Die Rechnung gemäß diesen Anweisungen unterliegt keinerlei Willkür des jeweiligen Rechners und ist ein wohldeterminierter Prozeß, der zu beliebiger Zeit wiederholt und mit demselben Ergebnis auch von anderen Personen durchgeführt werden kann.

Die Allgemeinverwendbarkeit eines Algorithmus. Ein Algorithmus ist eine einheitliche Vorschrift, die einen Rechenprozeß bestimmt, der bei unterschiedlichen Anfangswerten beginnen kann und in allen Fällen zum entsprechenden Resultat führt. Ein Algorithmus dient also nicht nur zur Lösung einer individuellen Aufgabe, sondern zur Lösung einer ganzen Schar von Aufgaben gleichen Typs.

1.2. Algorithmische Spiele

Die Aufgaben aus Abschnitt 1.1 waren der Arithmetik und der Algebra entnommen. Sie sind für die Problematik dieser mathematischen Gebiete geradezu typisch und besitzen sozusagen traditionellen mathematischen Charakter. Wir werden jetzt

zwei Scharen von Aufgaben etwas näher betrachten, die völlig anders geartet sind. Man könnte kurz sagen, daß sie logischer statt mathematischer Natur sind, obwohl es ziemlich schwierig sein dürfte, ein exaktes Kriterium anzugeben, nach welchem logische Aufgaben von üblichen mathematischen Aufgaben zu unterscheiden sind. Ohne uns auf weitere terminologische Betrachtungen einzulassen, wollen wir nur bemerken, daß es sich in beiden Fällen darum handelt, Algorithmen zu finden, d. h. Vorschriften, nach denen jeweils wieder ganze Scharen von Aufgaben gleichen Typs gelöst werden können. Bei den in diesem Paragraphen zu betrachtenden Fällen handelt es sich jedoch nicht um numerische Algorithmen.

1.2.1. Das Spiel „Elf Gegenstände"

In seinem Buch „Математическая смекалка" („Mathematische Geschicklichkeit")[1]) stellt B. A. Kordemskij eine Reihe von Spielen vor, bei denen der Erfolg nicht vom zufälligen Vorliegen günstiger Umstände, sondern allein vom Geschick und den Fähigkeiten der Spieler abhängt. Spiele dieser Art werden heute meistens *strategische Spiele* genannt. Wir beginnen mit der Betrachtung des Spiels „Elf Gegenstände", einer speziellen Variante der sogenannten Nimm-Spiele.

Auf dem Tisch liegen elf Gegenstände, z. B. Streichhölzer. Der erste Spieler (A) nimmt sich von diesen, nach seinem Ermessen, ein, zwei oder drei Stück. Danach nimmt der zweite Spieler (B) von den verbleibenden Streichhölzern, ebenfalls nach seinem Ermessen, ein, zwei oder drei Stück. Sodann ist wieder Spieler A an der Reihe. In jedem Zug des Spiels kann der Spieler, der an der Reihe ist, nach seinem Ermessen, ein, zwei oder drei Streichhölzer aufnehmen. Derjenige Spieler hat das Spiel verloren, der das letzte Streichholz aufnimmt.

Kann der Spieler A, der das Spiel beginnt, seinen Partner B zwingen, das letzte Streichholz aufzunehmen? Eine Analyse des Spieles zeigt, daß das tatsächlich der Fall ist. Der Spieler A muß dabei dem folgenden System von Anweisungen folgen:

1. Erster Zug: A nimmt 2 Streichhölzer.

2. Folgezug: Hat B bei seinem letzten Zug l ($1 \leqq l \leqq 3$) Streichhölzer genommen, so nimmt A im Folgezug $4 - l$ Streichhölzer.

Dieses System von Anweisungen ist in dem Sinne *vollständig*, daß es für den Spieler A in jedem Zug ein ganz bestimmtes Verhalten vorschreibt. Ein derartiges vollständiges System von Anweisungen heißt in der Spieltheorie eine *Strategie*. Eine Strategie des Spielers A heißt eine *Gewinnstrategie* (für A), wenn jede nach ihr gespielte Partie für A siegreich endet. Man kann zeigen, daß die oben angegebene Strategie tatsächlich

[1]) Das Buch ist unter dem Titel „Köpfchen, Köpfchen!" auch in deutscher Übersetzung erschienen (10. Auflage, Urania-Verlag, Leipzig–Jena–Berlin 1974). Die Spiele „Elf Gegenstände" und „Sieg der geraden Zahl" finden sich dort auf S. 126. [*Anm. d. Übers.*]

eine Gewinnstrategie für A ist, d. h. A unter allen Umständen zum Sieg führt, wie sich B auch verhalten mag. Die folgenden beiden Partien mögen diesen Sachverhalt illustrieren:

A	B	A	B	A	B		A	B	A	B	A	B
2	2	2	1	3	1,		2	3	1	1	3	1.

Falls A im ersten Zug ein oder drei Streichhölzer nimmt, kann Spieler B die Partie so fortsetzen, daß er sie mit Sicherheit gewinnt. Wir empfehlen dem Leser, die entsprechenden Gewinnstrategien für B zu notieren.

1.2.2. Das Spiel „Sieg der geraden Zahl"

Wir betrachten als nächstes das Spiel „Sieg der geraden Zahl" aus dem Buch von B. A. KORDEMSKIJ.

Von am Anfang 27 Streichhölzern nehmen die Spieler A und B abwechselnd mindestens ein und höchstens vier Stück. Gewonnen hat derjenige Spieler, der zum Schluß über eine gerade Anzahl von Streichhölzern verfügt. Es zeigt sich, daß in diesem Spiel der Spieler A, der das Spiel eröffnet, folgende Gewinnstrategie besitzt:

1. Erster Zug: A nimmt 2 Streichhölzer.

2. Folgezug von A, wenn B eine gerade Anzahl von Streichhölzern besitzt: Liegen mehr als 7 Streichhölzer auf dem Tisch, so nimmt A so viele Streichhölzer, daß die verbleibende Anzahl um 1 größer als ein Vielfaches von 6 (d. h. 19, 13 oder 7) ist. Liegen 5 oder 3 Streichhölzer auf dem Tisch, so nimmt A im ersten Fall 4, im zweiten Fall 2 Streichhölzer.

3. Folgezug von A, wenn B eine ungerade Anzahl von Streichhölzern hat: Liegen mehr als 3 Streichhölzer auf dem Tisch, so nimmt A so viele Streichhölzer, daß die verbleibende Anzahl um 1 kleiner ist als ein Vielfaches von 6 (d. h. 23, 17, 11 oder 5). Wenn das nicht möglich ist, so nimmt A so viele Streichhölzer, daß die verbleibende Anzahl ein Vielfaches von 6 ist. Liegen 3 oder weniger Streichhölzer auf dem Tisch, so werden diese alle genommen.

Man überlege sich, daß dieses System von Anweisungen vollständig, d. h. in der Tat eine Strategie für A ist. Die folgende Partie läuft entsprechend der geschilderten Strategie ab:

A	B	A	B	A	B	A	B	A	B	A	B
2	1	1	3	1	3	4	1	4	2	4	1.

Wie auch im vorangehenden Abschnitt, wollen wir auf eine Begründung der beschriebenen Gewinnstrategie verzichten. Sie ist für die betrachteten und eine Reihe weiterer Spiele im Buch von KORDEMSKIJ zu finden. Wir möchten nur bemerken, daß die anzustellenden Überlegungen wesentlich von der Spezifik des betrachteten Spiels abhängen und die Ausarbeitung einer Gewinnstrategie nicht selten großes Geschick und Erfindungsgabe erfordert.

1.2.3. Der Baum eines Spiels

Unser nächstes Ziel ist die Beschreibung eines Algorithmus, der auf jedes Spiel einer umfassenden Klasse von Spielen anwendbar ist und für jeden Spieler aus der Menge aller für ihn möglichen Strategien eine „optimale" ausgewählt. Um eine übermäßige Formalisierung zu vermeiden, werden wir die Bedeutung der meisten im folgenden benötigten Begriffe nur an Beispielen erläutern.[1])

Wir weisen zunächst auf die folgenden Besonderheiten der in den Abschnitten 1.2.1 und 1.2.2 betrachteten Spiele hin:

1. Am Spiel sind zwei Spieler beteiligt, die abwechselnd einen Zug ausführen.

2. Für jede Partie ist genau eines der folgenden beiden Resultate möglich:

a) Es gewinnt der Spieler A, der die Partie eröffnet (im folgenden durch „+" symbolisiert),

b) es gewinnt der Spieler B (im folgenden durch „−" symbolisiert).

3. In jedem Zug wählt der Spieler, der an der Reihe ist, eine Variante aus einer gegebenen endlichen Anzahl von Möglichkeiten aus, wobei diese Auswahl nicht durch einen Zufalls-Mechanismus (z. B. einen Würfel) gesteuert wird.

4. Bei jedem Zug ist jeder der beiden Spieler über den bisherigen Verlauf des Spiels, d. h. die Auswahlen und Resultate der vorangehenden Züge, voll informiert. Wir werden im folgenden unter einem Spiel stets ein solches verstehen, das diese Eigenschaften 1 bis 4 besitzt.

Zunächst ist ganz offensichtlich, daß nicht beide Spieler eine Gewinnstrategie besitzen können. Inhaltsreicher ist dagegen die Behauptung, daß in jedem Spiel einer der beiden Spieler eine Gewinnstrategie besitzt. Bevor wir das beweisen, wollen wir die bequeme Methode der Beschreibung von Spielen durch *Bäume* erläutern.

In Abb. 1 ist als Beispiel der Baum des Spiels „Sechs Gegenstände" dargestellt. In Abwandlung des Spiels „Elf Gegenstände" liegen zu Beginn des Spiels sechs Streichhölzer auf dem Tisch, von denen die beiden Spieler abwechselnd nach Belieben ein oder zwei Stück fortnehmen. Verloren hat wieder derjenige Spieler, der das letzte Streichholz nimmt.

Die *Knoten* des Baumes entsprechen den verschiedenen Situationen, die in den nach den Regeln des Spiels ablaufenden Partien eintreten können. Die aus einem Knoten nach oben herausführenden *Äste* oder *Kanten* entsprechen den im anschließenden Zug möglichen Varianten.

Im gewählten Beispiel sind in jeder Situation, ausgenommen die Situationen, in denen kein Streichholz mehr auf dem Tisch liegt — die entsprechenden Knoten des Baumes wollen wird *Endknoten* nennen — im nachfolgenden Zug genau zwei Varianten möglich. Wir vereinbaren, daß der nach links oben gerichtete Ast der Auswahl

[1]) Man vgl. etwa Wentzel, J. S., Elemente der Spieltheorie, 4. Auflage, BSB B. G. Teubner, Leipzig 1970, oder Owen, G., Spieltheorie, Springer-Verlag, Berlin—Heidelberg—New York 1971.

eines Streichholzes und der nach rechts oben gerichtete Ast der Auswahl von zwei Streichhölzern entspricht. Jede mögliche Partie des Spiels wird durch einen bestimmten Weg charakterisiert, der vom untersten Punkt α, der *Wurzel* des Baumes, zu einem Endknoten führt. Jedem Zug der Partie entspricht der Übergang von einem Knotenpunkt des Weges zu einem oberen Nachbarn. In den Endpunkten haben wir in einem kleinen Kreis das Resultat der in diesem Endpunkt endenden Partie angegeben. Unter den Knoten ist jeweils die Gesamtzahl der in der entsprechenden Situation vom Tisch genommenen Streichhölzer vermerkt. Der hervorgehobene Weg entspricht der Partie

A B A B
2 1 2 1

die von A gewonnen wird.

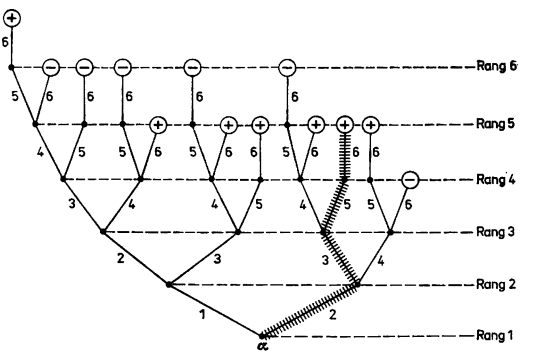


Abb. 1

Jedem Knoten des Spielbaumes, der kein Endknoten ist, ordnen wir seinen *Rang* zu (vgl. Abb. 1). Den höchstmöglichen Rang der Knotenpunkte nennen wir den *Rang des Baumes*. Er gibt die maximale Länge einer Partie des betrachteten Spiels an. Die Knoten ungeraden Ranges entsprechen den Situationen, in denen der Spieler A, der die Partie eröffnet, am Zuge ist, die Knoten geraden Ranges den Situationen, in denen der andere Spieler B am Zuge ist.

Bislang haben wir den Spielbaum nur als graphische Illustration des Spiels angesehen, dessen Regeln zunächst auf eine andere Weise, nämlich verbal, gegeben wurden. Es hindert uns jedoch nichts daran, einen Baum selbst als Beschreibung

eines Spiels anzusehen. So beschreibt z. B. der Baum in Abb. 2a ein Spiel, bei dem jede Partie aus zwei Zügen besteht, wobei Spieler A im ersten Zug eine von drei Varianten auswählen kann und Spieler B in jeder Situation im anschließenden zweiten Zug sich für eine von genau zwei Varianten entscheiden muß. In dem Spiel ist nur eine Partie möglich, bei der A gewinnt; sie ist durch den hervorgehobenen Weg charakterisiert. Der Baum in Abb. 2b beschreibt ein (ausgeartetes) Spiel, das sich auf einen einzigen Zug beschränkt.

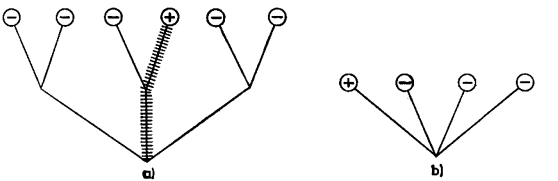


Abb. 2

Jeder Knoten eines Baumes, der kein Endknoten ist, kann als Wurzel eines *Teilbaumes* angesehen werden, der seinerseits ein bestimmtes Spiel beschreibt.

Die Darstellung eines Spiels durch seinen Baum ermöglicht es, jede Strategie des Spielers A als ein System von Pfeilen aufzufassen, die von Knoten ungeraden Ranges zu oberen Nachbarn geraden Ranges führen. Dabei müssen folgende Bedingungen erfüllt sein:

a) aus keinem Knoten (ungeraden Ranges) führt mehr als ein Pfeil heraus (die Strategie des Spielers A legt in jeder Situation, in der er am Zuge ist, eindeutig die von ihm gewählte Variante fest),

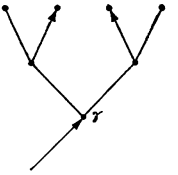


Abb. 3

b) wenn ein Pfeil zu einem Knoten γ geraden Ranges führt, so ist jeder im Spielbaum mit γ verbundene Knoten nächsthöheren Ranges, der kein Endknoten ist, Anfang genau eines Pfeiles (Abb. 3) (diese Bedingung garantiert, daß die Strategie jede Situation berücksichtigt, in die A durch den Spieler B gebracht werden kann),

c) es gibt genau einen Pfeil, der von α ausgeht.

Analog kann man jede Strategie des Spielers B als ein System von Pfeilen auffassen, die von Knoten geraden Ranges zu oberen Nachbarknoten ungeraden Ranges führen, und das entsprechende Eigenschaften hat.

In Abb. 4 ist in den Baum des Spiels „Sechs Gegenstände" die Strategie für A eingetragen, die darin besteht, daß A in jeder Situation, in der er am Zuge ist, ein Streichholz aufnimmt. Mit dieser Strategie sind drei Partien möglich, von denen A zwei verliert und eine gewinnt. Sie ist also keine Gewinnstrategie für A.

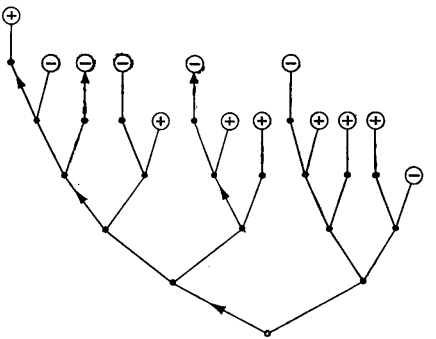


Abb. 4

1.2.4. Die Existenz einer Gewinnstrategie

Im vorliegenden Abschnitt wollen wir den folgenden Satz beweisen:

Satz. *In jedem Spiel hat einer der beiden Spieler eine Gewinnstrategie.*

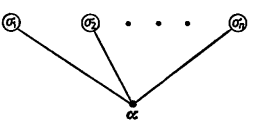


Abb. 5

Der Beweis besteht in der Angabe eines Algorithmus, durch den bei einem beliebig vorgegebenen Spiel für einen der beiden Spieler eine Gewinnstrategie aufgestellt wird. Die Konstruktion des Algorithmus erfolgt durch Induktion über die maximale Partienlänge ν des gegebenen Spiels, d. h. den Rang des Spielbaumes.

Anfangsschritt $\nu = 1$. In diesem Fall besteht jede Partie aus einem einzigen Zug, wobei wir zunächst annehmen, daß dieser Zug vom Spieler A ausgeführt wird (im vorliegenden „ausgearbeiteten" Spiel ist also B niemals am Zuge). Der Spielbaum werde durch Abb. 5 gegeben, wobei $\sigma_1, \ldots, \sigma_n$ entweder gleich „+" (A gewinnt) oder gleich „−" (B gewinnt) ist. Kommt unter $\sigma_1, \ldots, \sigma_n$ wenigstens einmal das Symbol „+" vor, so besitzt A eine Gewinnstrategie. Jeder Pfeil, der von der Wurzel α zu einem mit „+" markierten Endpunkt führt, beschreibt eine solche Gewinnstrategie. Eine bestimmte Gewinnstrategie kann dadurch ausgewählt werden, daß man — wenn mehrere existieren — die wählt, die zu dem am weitesten links gezeichneten und durch „+" markierten Endpunkt gehört. Sind alle σ_i gleich „−", so besitzt B eine Gewinnstrategie, da jeder mögliche Zug von A zur Niederlage von A führt. In diesem Fall gibt es allerdings keine Pfeile, die eine Auswahl von B markieren, da ja im betrachteten Spiel der Spieler B niemals zum Zuge kommt.

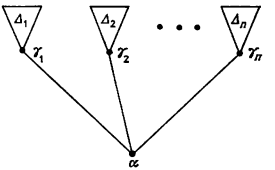


Abb. 6

Dieselbe Überlegung führt zum Ziel, wenn wir annehmen, daß α eine Situation bezeichnet, in der B am Zuge ist. In diesem Fall hat B eine Gewinnstrategie, falls unter $\sigma_1, \ldots, \sigma_n$ wenigstens einmal das Symbol „−" vorkommt, anderenfalls besitzt A eine solche.

Induktionsschritt. Wir nehmen an, der Satz sei schon für alle Spiele bewiesen, deren Rang kleiner als ν ist, und zeigen, daß er dann auch für alle Spiele gilt, deren Rang gleich ν ist. Der Baum eines derartigen Spiels sei durch Abb. 6 gegeben, wobei die Dreiecke die Teilbäume $\Delta_1, \ldots, \Delta_n$ des Baumes andeuteten, die als Wurzeln $\gamma_1, \ldots, \gamma_n$ die oberen Nachbarn der Wurzel α des betrachteten Baumes haben. Wir nehmen an, daß in der Situation α der Spieler A am Zuge ist. Dann stellen die Teilbäume $\Delta_1, \ldots, \Delta_n$ Spiele dar, in denen B den ersten Zug ausführt. Die Maximallänge aller dieser Spiele ist kleiner als ν, so daß wir auf Grund der Induktionsvoraussetzung annehmen können, daß in jedem dieser Spiele einer der beiden Spieler eine Gewinnstrategie besitzt. Hat nun in wenigstens einem dieser Teilspiele der Spieler A eine Gewinnstrategie, eine solche möge o. B. d. A. in dem durch Δ_1 beschriebenen Teilspiel vorhanden sein, so hat A auch eine Gewinnstrategie im Gesamtspiel. Man braucht dazu nur zu den in Δ_1 gelegenen Pfeilen einer solchen Gewinnstrategie den Pfeil von α zur Wurzel γ_1 von Δ_1 hinzuzufügen, d. h., A muß im Gesamtspiel als erstes den dem Ast von α nach γ_1 entsprechenden Zug ausführen und anschließend die Gewinnstrategie in Δ_1 befolgen. Hat dagegen in jedem der Teilspiele $\Delta_1, \ldots, \Delta_n$ der

Spieler B eine Gewinnstrategie, so hat B auch im Gesamtspiel eine Gewinnstrategie. Sie besteht in der Vereinigung aller Gewinnstrategien in den Teilspielen.

Damit ist der behauptete Satz bewiesen.

Wir illustrieren den geschilderten Algorithmus am Beispiel des Baumes für das Spiel „Sechs Gegenstände" (vgl. Abb. 7). Ausgehend von den Endknoten, d. h. den Knoten höchsten Ranges, versehen wir die Knoten mit dem Symbol „+" oder „−", je nachdem, ob im entsprechenden Teilspiel der Spieler A oder der Spieler B eine Gewinnstrategie hat. Die Verteilung der Symbole „+" und „−" in den Endknoten

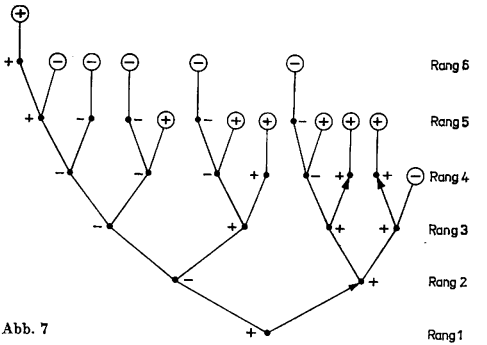


Abb. 7

ist durch die Regeln des Spiels determiniert. Unter den Knoten vom Rang 6 erhält offenbar nur einer das Symbol „+". Wir gehen nun von links nach rechts die Knoten des Ranges 5 durch, die Positionen entsprechen, in denen A am Zuge ist. Offenbar muß der erste (äußerste linke) das Zeichen „+" erhalten, weil man von ihm zu dem mit „+" versehenen Knoten vom Rang 6 gelangen kann, die übrigen müssen das Symbol „−" erhalten. Unter den Knoten von Rang 4, die Positionen entsprechen, in denen B am Zuge ist, erweisen sich drei als „+"-Knoten, nämlich die, aus denen kein Ast zu einem „−"-Knoten vom Rang 5 führt; die übrigen vier erhalten das Symbol „−". Indem wir diesen Prozeß bis zur Wurzel fortsetzen, finden wir, daß α ein „+"-Knoten ist, d. h., im Spiel „Sechs Gegenstände" besitzt Spieler A eine Gewinnstrategie. Die Konstruktion einer Gewinnstrategie, d. h. das Anbringen der ihr entsprechenden Pfeile, erfolgt in entgegengesetzter Richtung (von der Wurzel zu den Endknoten). Zunächst wird α mit einem „+"-Knoten zweiten Ranges verbunden (im vorliegenden Beispiel gibt es nur einen). Sodann wird in jedem mit dem ausgewählten Knoten zweiten Ranges verbundenen Knoten dritten Ranges genau ein Pfeil zu einem mit ihm verbundenen „+"-Knoten vierten Ranges gezeichnet (was im vorliegenden Beispiel ebenfalls nur auf eine Weise möglich ist). Im betrachteten

Beispiel ist die Konstruktion einer Gewinnstrategie für A damit bereits beendet; sie ist in Abb. 7 dargestellt. Die Konstruktion zeigt, daß es im Spiel „Sechs Gegenstände" für A nur eine einzige Gewinnstrategie gibt. Es sind bei dieser Strategie zwei verschiedene Partien möglich, die beide mit der Niederlage von B enden.

Wir bemerken, daß der im vorliegenden Abschnitt bewiesene Satz und der zu seinem Beweis konstruierte Algorithmus leicht auf Spiele ausgedehnt werden können, die nicht die Bedingungen 1 und 2 aus Abschnitt 1.2.3 erfüllen. Die Bedingungen 3 und 4 erweisen sich demgegenüber als wesentlich, auf sie kann nicht verzichtet werden. So kann man z. B. Spiele zweier Partner A und B betrachten, bei denen außer dem Sieg von A oder B ein Unentschieden (Remis) möglich ist. Hier kann es natürlich passieren, daß keiner der beiden Spieler eine Gewinnstrategie besitzt. Dann liefert der Algorithmus für jeden der beiden Spieler eine Strategie, deren Befolgen beiden Spielern ein Unentschieden garantiert, fehlerhaftes Verhalten des Gegners möglicherweise sogar einen Sieg. Da auch das Schachspiel zu den Spielen dieses Typs gehört, ermöglicht unser Algorithmus eine Entscheidung, welcher Fall hier vorliegt, d. h., ob es für Weiß (Spieler A) eine Gewinnstrategie gibt, oder ob Schwarz (Spieler B) eine Gewinnstrategie hat, oder ob es für beide Spieler eine Remis-Strategie gibt. Hierzu genügt es (!), den Baum des Schachspiels aufzustellen und aus ihm optimale Strategien für die Spieler herzuleiten. Wenn sich dabei herausstellt, daß A eine Gewinnstrategie besitzt, so ist der Ausgang einer Partie von vornherein zugunsten von A entschieden, wenn A nur die Gewinnstrategie befolgt. Analog ist der Ausgang einer Partie von vornherein klar, falls B eine Gewinnstrategie hat und diese befolgt oder wenn beide Partner Remis-Strategien besitzen und diese befolgen.

Die Anwendung des oben beschriebenen Algorithmus auf das Schachspiel müßte doch nun eigentlich zur Folge haben, daß dieses vollständig „entschleiert" und damit uninteressant wird, wie das z. B. bei den Streichholzspielen der Fall ist. Warum ist das, obwohl ein solcher Algorithmus existiert, beim Schachspiel bis jetzt nicht so, warum gehört das Schachspiel nach wie vor zu den Spielen, deren Beherrschung große Meisterschaft und großes Geschick erfordert?

Bei der Beantwortung dieser Frage kommen wir in naheliegender Weise zu der Feststellung, daß der Prozeß, der durch den Algorithmus beschrieben wird, praktisch nicht realisierbar ist. Aus der Wurzel des Baumes, der das Schachspiel beschreibt, führen bereits 20 Äste heraus, die den verschiedenen möglichen Eröffnungen von Weiß entsprechen. Aus jedem der 20 Knoten zweiten Ranges führen ebenfalls 20 Äste heraus. Die 400 Knoten dritten Ranges haben ihrerseits eine sehr große, zum Teil unterschiedliche Anzahl von Ästen usw. Und schließlich ist noch der Rang des Gesamtbaumes sehr groß. Uns steht weder genügend Zeit noch genügend Raum noch genügend Material zur Verfügung, um ein solches Vorhaben zu realisieren. Ungeachtet dessen müssen wir die Frage, ob wir eine genaue Beschreibung besitzen, nach der wir auf Grund eines einheitlichen Verfahrens für ein beliebiges Spiel des betrachteten Typs in endlich vielen Schritten eine optimale Strategie aufstellen können, vorbehaltlos mit „ja" beantworten. Hierfür wollen wir auch sagen, daß der durch diese Beschreibung bestimmte Prozeß *potentiell realisierbar* ist, d. h., das Ergebnis

kann nach einer endlichen (möglicherweise außerordentlich großen) Anzahl von Elementaroperationen erhalten werden.

Natürlich ist man in erster Linie an auch *praktisch realisierbaren* Prozessen interessiert. Jedoch gibt es keine formalen mathematischen Kriterien dafür, wann ein potentiell realisierbarer Prozeß auch praktisch realisierbar ist. Das liegt daran, daß die praktische Realisierbarkeit in entscheidendem Maße von den zur Verfügung stehenden Hilfsmitteln abhängt. Diese können sich aber, z. B. im Zuge der technischen Entwicklung, sehr rasch ändern. So wurden und werden mit der Entwicklung schnell arbeitender elektronischer Rechenautomaten Prozesse praktisch realisierbar, die noch vor kurzem nur potentiell realisierbar waren.

Trotz des großen Umfanges des betrachteten Algorithmus ist die Tatsache, daß ein solcher existiert, sehr wichtig. Denn beim 10. Hilbertschen Problem gibt es nach dem Theorem von Ju. V. Matijasevič nachweisbar keinen Algorithmus. Verfügt man erst einmal über einen Algorithmus, und mag er noch so umfangreich sein, so ist durchaus die Möglichkeit vorhanden, daß er verbessert werden kann oder ein bequemerer Algorithmus zur Lösung der gestellten Aufgabenklasse gefunden wird.

Im folgenden werden wir unter einem Prozeß, der durch einen Algorithmus beschrieben wird, stets einen *potentiell realisierbaren Prozeß* verstehen.

1.3. Algorithmen zur Suche eines Weges in einem Labyrinth

1.3.1. Labyrinthe

Die griechische Mythologie berichtet von dem legendären Helden Theseus, daß er sich in ein Labyrinth begab, um dort das Ungeheuer Minotaurus aufzuspüren und zu erschlagen. Bei diesem Vorhaben half ihm Ariadne, indem sie ihm ein Fadenknäuel mitgab, dessen Ende sie in der Hand behielt, damit Theseus den Rückweg finden konnte. Beim Eindringen in das Labyrinth wickelte Theseus das Knäuel ab; indem er den Faden wieder aufwickelte, gelangte er wohlbehalten zum Ausgang zurück.

An diese alte Legende erinnert das vor einigen Jahren entwickelte automatische Spielzeug „Maus im Labyrinth" des amerikanischen Mathematikers und Ingenieurs Claude Shannon. An einer Stelle eines speziellen Labyrinths befindet sich ein Stück „Speck" und an einer anderen die „Maus". Die Maus irrt planlos im Labyrinth umher, bis sie den „Speck" gefunden hat. Hierbei kommt sie an manchen Orten mehrmals vorbei, d. h., sie läuft in Schlaufen zum „Speck". Läßt man die „Maus" aber zum zweitenmal von derselben Stelle aus starten, so läuft sie direkt, ohne eine Schlaufe zu machen, zum „Futter". Wir wollen hier ein verwandtes Problem (genauer gesagt eine Schar von Aufgaben gleichen Typs) untersuchen:

Wie findet man einen Weg in einem Labyrinth?

Wir werden einen Algorithmus angeben, der beschreibt, wie man zur Erreichung des in der Aufgabe gestellten Ziels vorgehen kann.

Wir stellen uns ein *Labyrinth* als ein endliches System von *Plätzen* vor, von denen *Gänge* ausgehen, wobei jeder Gang zwei Plätze verbindet; solche Plätze sollen *benachbart* heißen. Es kann dabei Plätze geben, zu denen man nur längs eines einzigen Ganges gelangen kann; solche Gänge nennen wir *Sackgassen*. Geometrisch läßt sich ein Labyrinth auffassen als ein System von Punkten A, B, C, ... (den Bildern der Plätze) und eine Gesamtheit von Strecken AB, BC, ... (den Bildern der Gänge), die gewisse Paare von Punkten miteinander verbinden (vgl. Abb. 8). Eine solche Figur nennt man auch einen *Graphen*.

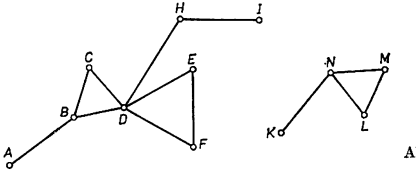


Abb. 8

Wir wollen sagen, daß ein Platz Y von einem Platz X aus *erreichbar* ist, wenn es einen Weg gibt, der längs gewisser Gänge und über gewisse Plätze von X nach Y führt. Genauer gesagt heißt das, daß X und Y entweder benachbarte Plätze sind oder daß eine Folge von Plätzen $X_1, X_2, X_3, \ldots, X_n$ existiert, so daß X und X_1, ferner X_1 und X_2, ferner X_2 und X_3, ..., und schließlich X_n und Y benachbart sind. So ist in Abb. 8 beispielsweise der Platz H vom Platz A aus längs des Weges AB, BC, CD, DE, EF, FD, DH erreichbar, während K von A aus nicht erreichbar ist. Wenn Y von X aus überhaupt erreichbar ist, so auch längs eines *einfachen Weges*, d. h. eines Weges, in dem jeder Platz (und erst recht auch jeder Gang) höchstens einmal vorkommt. Der Weg in unserem Beispiel war kein einfacher Weg. Schneidet man jedoch die Schlaufe DE, EF, FD ab, so ergibt sich der einfache Weg AB, BC, CD, DH.

Wir nehmen an, der Minotaurus halte sich auf einem Platz M des Labyrinths auf. Theseus, der sich zu Beginn auf dem Platz A befindet, auf dem Ariadne auf ihn wartet, hat die folgende Aufgabe zu lösen: Er hat festzustellen, ob M von A aus erreichbar ist oder nicht.[1]) Wenn M erreichbar ist, so soll Theseus auf irgendeinem Wege dorthin gelangen und anschließend auf einem einfachen Wege zu Ariadne zurückkehren. Ist M nicht erreichbar, so kehrt Theseus zu Ariadne zurück.

Es gibt unendlich viele verschiedene Labyrinthe, wobei auch die gegenseitige Lage der Plätze A und M noch variieren kann. Da Theseus im voraus nichts über den Aufbau des vorliegenden Labyrinths und über den Aufenthaltsort des Minotaurus weiß, ist die Aufgabe folgendermaßen zu verstehen: Es soll ein allgemeines Verfahren gefunden werden, welches für beliebige Labyrinthe und beliebige Anordnungen der Plätze A und M brauchbar ist. Mit anderen Worten: Es wird ein Algorithmus gesucht, der auf beliebige Aufgaben des gegebenen Typs anwendbar ist.

[1]) Natürlich soll M von A verschieden sein.

1.3.2. Der Suchalgorithmus

Statt des allgemeinen Algorithmus konstruieren wir ein etwas spezielleres Suchverfahren. Bei dem zu schildernden Verfahren wird vorausgesetzt, daß man in jedem Stadium des Suchprozesses feststellen kann, ob THESEUS einen Gang überhaupt noch nicht durchlaufen hat (einen solchen Gang wollen wir einen grünen Gang nennen), ob er ihn einmal durchlaufen hat (gelber Gang), oder ob er ihn zweimal durchlaufen hat (roter Gang). Befindet sich THESEUS auf einem beliebigen Platz des Labyrinths, so kann er auf folgende Arten zu einem benachbarten Platz gelangen:

1. *Abwickeln des Fadens.* THESEUS geht von dem vorgegebenen Platz durch einen grünen Gang zu dem benachbarten Platz. Hierbei wird der Faden der ARIADNE längs dieses Ganges, der nach dem Durchlaufen ein gelber Gang ist, abgewickelt.

2. *Aufwickeln des Fadens.* THESEUS kehrt von dem erreichten Platz auf dem zuletzt durchlaufenen gelben Gang zu dem benachbarten Platz zurück. Hierbei wird der Faden der ARIADNE, der vorher längs dieses Ganges abgewickelt wurde, wieder aufgewickelt. Aus dem gelben Gang ist dabei ein roter Gang geworden.

Es wird angenommen, daß THESEUS ein Merkmal hinterläßt, das es ihm gestattet, später einen roten von einem grünen Gang zu unterscheiden. Ein gelber Gang ist ja dadurch ausgezeichnet, daß durch ihn der Faden der ARIADNE läuft. Wie THESEUS sich in jedem einzelnen Stadium des Suchens zu verhalten hat, hängt von den Umständen ab, die er auf dem Platz vorfindet, auf dem er sich gerade aufhält. Diese Umstände lassen sich jeweils durch eines oder mehrere der folgenden Kennzeichen charakterisieren:

1. *Minotaurus.* Auf dem erreichten Platz befindet sich der MINOTAURUS.

2. *Schlaufe.* Über den erreichten Platz läuft schon der Faden der ARIADNE; mit anderen Worten, von dem erreichten Platz gehen wenigstens zwei gelbe Gänge ab.

3. *Grüner Gang.* Von dem erreichten Platz gibt es wenigstens einen Ausgang in einen grünen Gang.

4. *Ariadne.* Auf dem erreichten Platz befindet sich ARIADNE.

5. *Fünfter Fall.* Es fehlen die erwähnten Kennzeichen.

Unser Suchverfahren kann man jetzt durch folgendes Schema beschreiben:

Kennzeichen	Operation
1. *Minotaurus*	*Halt*
2. *Schlaufe*	*Faden aufwickeln*
3. *Grüner Gang*	*Faden abwickeln*
4. *Ariadne*	*Halt*
5. *Fünfter Fall*	*Faden aufwickeln.*

Ist THESEUS auf einem Platz angekommen, so wird er wie folgt verfahren: Er geht nacheinander die einzelnen Kennzeichen der linken Seite des Schemas in der vorgeschriebenen Reihenfolge durch und kontrolliert dabei, welches zutrifft. Wenn er das erste zutreffende Kennzeichen ermittelt hat (die anderen prüft er dann nicht mehr), führt er die in der rechten Spalte stehende Operation aus. Dieses Verfahren setzt er solange fort, bis er zur Operation *Halt* kommt.

Die Brauchbarkeit der vorgeschlagenen Methode ergibt sich unmittelbar aus den folgenden drei Behauptungen:

1. *Bei beliebiger Lage von A und M im Labyrinth kommt* THESEUS *nach endlich vielen Operationen entweder zum* MINOTAURUS *oder zu* ARIADNE, *was ja die Operation Halt zur Folge hat.*

2. *Tritt das Halt auf dem Platz des* MINOTAURUS *ein, dann ist dieser natürlich erreichbar. Der Faden der* ARIADNE *ist in diesem Fall längs eines von A nach M führenden einfachen Weges gezogen, und* THESEUS *kann auf diesem Weg, indem er den Faden aufwickelt, zu* ARIADNE *zurückkehren.*

3. *Tritt das Halt auf dem Platz der* ARIADNE *ein, so ist der* MINOTAURUS *unerreichbar.*

Bevor wir diese Behauptungen beweisen, illustrieren wir durch zwei Beispiele, wie die vorgeschlagene Methode gehandhabt wird.

Beispiel 1. Wir betrachten das in Abb. 8 dargestellte Labyrinth. Die Suche nach dem MINOTAURUS beginne auf dem Platz *A*. Der MINOTAURUS befinde sich auf dem Platz *F*. Man kann den Suchprozeß bequem in einer Tabelle wiedergeben (vgl. Tabelle 1). Es ist zu beachten, daß unser Suchverfahren nicht eindeutig ist, da es bei der Auswahl eines grünen Ganges mehrere Möglichkeiten geben kann.

Tabelle 1

Nummer der Operation	Vorgefundenes Kennzeichen	Operation	Zu durchlaufender Gang	Farbe des Ganges nach dem Durchlaufen
1	*Grüner Gang*	*Abwickeln*	*AB*	gelb
2	*Grüner Gang*	*Abwickeln*	*BC*	gelb
3	*Grüner Gang*	*Abwickeln*	*CD*	gelb
4	*Grüner Gang*	*Abwickeln*	*DH*	gelb
5	*Grüner Gang*	*Abwickeln*	*HI*	gelb
6	*Fünfter Fall*	*Aufwickeln*	*IH*	rot
7	*Fünfter Fall*	*Aufwickeln*	*HD*	rot
8	*Grüner Gang*	*Abwickeln*	*DB*	gelb
9	*Schlaufe*	*Aufwickeln*	*BD*	rot
10	*Grüner Gang*	*Abwickeln*	*DF*	gelb
11	*Minotaurus*	*Halt*	—	—

Wir sehen, daß der MINOTAURUS im vorliegenden Fall erreichbar ist. Wir suchen uns nun aus der vorletzten Spalte (unter Beachtung der Angaben in der letzten Spalte) diejenigen Gänge heraus, die gelb geblieben sind, und erhalten so einen einfachen Weg, der von A nach F führt:

$$AB, BD, CD, DF.$$

Beispiel 2. Das Suchen nach dem in F befindlichen MINOTAURUS beginne auf dem Platz K. Das Schema des Suchprozesses ist in Tabelle 2 dargestellt.

Tabelle 2

Nummer der Operation	Vorgefundenes Kennzeichen	Operation	Zu durchlaufender Gang	Farbe des Ganges nach dem Durchlaufen
1	*Grüner Gang*	*Abwickeln*	KN	gelb
2	*Grüner Gang*	*Abwickeln*	NL	gelb
3	*Grüner Gang*	*Abwickeln*	LM	gelb
4	*Grüner Gang*	*Abwickeln*	MN	gelb
5	*Schlaufe*	*Aufwickeln*	NM	rot
6	*Fünfter Fall*	*Aufwickeln*	ML	rot
7	*Fünfter Fall*	*Aufwickeln*	LN	rot
8	*Fünfter Fall*	*Aufwickeln*	NK	rot
9	*Ariadne*	*Halt*	—	—

In diesem Fall ist also der MINOTAURUS unerreichbar.

1.3.3. Rechtfertigung des Suchalgorithmus

Wir kommen nun zum Beweis der Behauptungen 1 bis 3 aus Abschnitt 1.3.2.

Beweis der Behauptung 1. Zuerst zeigen wir mittels vollständiger Induktion nach der Anzahl der Operationen des THESEUS, daß in jedem Stadium des Suchprozesses die folgende Alternative gilt (d. h., es gilt einer der beiden folgenden, sich gegenseitig ausschließenden Fälle):

a) Im Labyrinth gibt es keinen gelben Gang; dann befindet sich THESEUS auf dem Platz A (ARIADNE).

b) im Labyrinth gibt es gelbe Gänge, und diese bilden, wenn man sie in der Reihenfolge betrachtet, in der sie von THESEUS durchlaufen wurden, einen Weg, der von A zu dem Platz führt, auf dem sich THESEUS befindet.

Außerdem zeigt sich, daß THESEUS niemals durch einen roten Gang geht.

Die Behauptung stimmt zu Beginn des Prozesses, wenn sich THESEUS auf dem Platz A befindet und noch keinen Gang durchlaufen hat (alle Gänge sind dabei grün). Wir nehmen nun an, daß die angegebene Alternative nach Anwendung der $(n-1)$-ten

Operation richtig ist, und beweisen, daß die Alternative dann auch nach Anwendung der n-ten Operation gilt (selbstverständlich soll die $(n-1)$-te Operation nicht *Halt* sein, denn sonst gibt es keine n-te Operation mehr).

Es liege nach der $(n-1)$-ten Operation der Fall a) vor. Dann ist die nächste Operation entweder das Durchlaufen eines grünen Ganges von A zu einem benachbarten Platz K, und nach der n-ten Operation gilt der Fall b) mit dem gelben Gang AK, oder die nächste Operation ist *Halt* auf dem Platz A, und nach der n-ten Operation bleibt der Fall a) bestehen.

Wir wollen jetzt annehmen, daß nach der $(n-1)$-ten Operation der Fall b) mit s gelben Gängen vorliegt. Die gelben Gänge beschreiben dann einen Weg $AA_1, A_1A_2, \ldots, A_{s-1}K$. Je nachdem, welches Kennzeichen THESEUS am Platz K vorfindet, ergeben sich für die n-te Operation folgende Möglichkeiten:

1. *Minotaurus.* THESEUS hat die Operation *Halt* auf dem Platz K auszuführen, die früheren gelben Gänge bleiben erhalten, und nach der n-ten Operation liegt der Fall b) vor.

2. *Schlaufe.* THESEUS wickelt den Faden auf, d. h., er läuft den Gang KA_{s-1} zurück, und der Gang KA_{s-1} wird ein roter Gang. Der gelbe Weg verkürzt sich also um einen Gang. Wenn die Anzahl s der Gänge des früheren Weges größer als 1 war, liegt nach der n-ten Operation der Fall b) mit $s-1$ gelben Gängen vor. War jedoch $s=1$, so liegt der Fall a) vor.

3. *Grüner Gang.* THESEUS wickelt den Faden ab, d. h., er geht durch einen grünen Gang, der dabei gelb wird. Es tritt also sicher der Fall b) mit $s+1$ gelben Gängen ein.

4. *Ariadne.* Dieser Fall kann nicht zur Anwendung kommen. Denn wenn sich THESEUS längs des gelben Weges $AA_1, A_1A_2, \ldots, A_{s-1}K$ zu ARIADNE zurück bewegt hat (d. h. wenn $K=A$ ist), muß er sich entsprechend den Anweisungen des Suchalgorithmus nach dem vor dem Kennzeichen *Ariadne* stehenden Kennzeichen *Schlaufe* richten.

5. *Fünfter Fall.* Analog wie beim Kennzeichen *Schlaufe* wird der Faden aufgewickelt, und man kommt auf den Fall a) oder b), je nachdem, ob $s=1$ oder $s>1$ ist. Damit ist die oben angegebene Alternative gerechtfertigt. Zugleich ist klar, daß THESEUS keinen Gang mehr als zweimal durchläuft, d. h., er kommt nie durch einen roten Gang. Da aber die Anzahl aller Gänge des Labyrinths endlich ist, muß auch die Folge der Operationen endlich sein. Diese Folge kann jedoch nur bei *Halt* abbrechen, d. h., wenn THESEUS sich entweder auf dem Platz des MINOTAURUS oder auf dem der ARIADNE befindet.

Beweis der Behauptung 2. Tritt die Operation *Halt* auf dem Platz des MINOTAURUS auf, so ist dieser offensichtlich erreichbar. Hierbei ist der Faden der ARIADNE längs eines gelben Weges gezogen, wie oben nachgewiesen wurde. Der Faden kann keine Schlaufen haben, denn jedesmal, wenn THESEUS beim Umherirren das Kennzeichen *Schlaufe* findet, läuft er wieder zurück und beseitigt die Schlaufe.

Beweis der Behauptung 3. Damit die Operation *Halt* auf dem Platz der ARIADNE zur Anwendung kommt, muß folgendes vorliegen:

1. Für jeden Gang des Labyrinths gilt: Er wurde entweder zweimal (roter Gang) oder gar nicht durchlaufen (grüner Gang). Mit anderen Worten, das Knäuel ist vollständig aufgewickelt, es gibt im Labyrinth keinen gelben Gang.[1])

2. Alle Gänge, die von A ausgehen, sind rot; denn das Kennzeichen *Ariadne* wird nur dann berücksichtigt, wenn die im Suchalgorithmus früher auftretenden Kennzeichen *Schlaufe* und *Grüner Gang* nicht zutreffen.

Wir nehmen nun an, die Behauptung 3 wäre falsch, d. h., der auf dem Platz M befindliche MINOTAURUS wäre doch erreichbar. Es sei $AA_1, A_1A_2, \ldots, A_nM$ ein Weg von A nach M. Nach Abschluß des Suchverfahrens ist der erste Gang dieses Weges ein roter Gang, da von A dann nur rote Gänge ausgehen, und der letzte Gang A_nM ein grüner Gang, da der MINOTAURUS nicht gefunden wurde. Es sei A_iA_{i+1} der erste grüne Gang in der Folge $AA_1, A_1A_2, \ldots, A_nM$. Von A_i geht sowohl ein roter als auch ein grüner Gang aus (da nach Abschluß des Suchverfahrens keine gelben Gänge mehr auftreten). Daher muß der Platz A_i beim Suchverfahren erreicht worden sein. Wir betrachten den letzten Durchgang des THESEUS durch A_i. Offenbar muß er dabei A_i unter Aufwickeln des Fadens durch einen jetzt roten Gang verlassen haben. Er muß also beim letzten Betreten von A_i entweder das Kennzeichen *Schlaufe* oder das Kennzeichen *Fünfter Fall* vorgefunden haben, da nur in diesen Fällen die Operation *Aufwickeln* zur Anwendung kommt. Letzteres ist offensichtlich unmöglich, denn es geht ja von A_i noch der grün gebliebene Gang A_iA_{i+1} aus. Aber auch der andere Fall führt zu einem Widerspruch. Denn hätte THESEUS in A_i eine Schlaufe vorgefunden, so wäre nach dem Verlassen von A_i dort wenigstens noch ein gelber Gang verblieben. Da aber nach Abschluß des Suchverfahrens keine gelben Gänge mehr vorhanden sind, hätte dieser gelbe Gang bei einem späteren Durchgang in einen roten Gang verwandelt werden müssen, was unserer Annahme widerspricht, daß wir den letzten Durchgang des THESEUS durch den Platz A_i betrachtet haben. Damit ist auch die Behauptung 3 bewiesen.

Wir weisen nochmals darauf hin, daß das von uns angegebene Suchverfahren ein willkürliches Element enthält. Beim Kennzeichen *Grüner Gang* ist die anzuwendende Operation nämlich nicht eindeutig bestimmt, wenn von dem gegebenen Platz mehrere grüne Gänge wegführen. Unsere Vorschrift läßt offen, welchen man zu beschreiten hat. Sie gestattet, genauer ausgedrückt, die willkürliche Auswahl eines Ganges. Dadurch geht aber die Eigenschaft der Determiniertheit verloren, von der im vorangehenden Paragraphen gesagt wurde, daß sie für Algorithmen charakteristisch ist. Dieses zufällige Element kann man leicht durch eine Vereinbarung beseitigen (und dadurch die angegebene Vorschrift in einen Algorithmus umwandeln), nach der beim Vorhandensein mehrerer grüner Gänge der zu begehende Gang auf Grund einer Regel ausgewählt wird. Beispielsweise könnte man verlangen, daß THESEUS den

[1]) Der Beweis dieser Tatsache bleibe dem Leser überlassen.

betretenen Platz im Uhrzeigersinn bis zum ersten grünen Gang umläuft und in diesen Gang hineingeht. Genausogut könnte man natürlich irgendeine andere Vereinbarung treffen. Die Untersuchung von Vorschriften, in denen man das Auftreten willkürlicher Elemente zuläßt, ist von großem theoretischen und praktischen Interesse, insbesondere in der modernen Spieltheorie und ihren Anwendungen in der Ökonomie. Gerade solche Vorschriften liefern für eine umfangreiche Klasse von Spielen die (in einem bestimmten Sinne) „besten" Strategien, wobei also die Wahl der folgenden Schritte nicht nur durch Entscheidungen der Spieler, sondern auch durch einen Zufalls-Generator beeinflußt wird. Vorschriften dieser Art sind unmittelbare Verallgemeinerungen der im vorangehenden Paragraphen beschriebenen Algorithmen für Spiele ohne Zufallselemente. Wir werden uns mit derartigen Verallgemeinerungen nicht befassen und nur streng determinierte Prozesse betrachten.

Wir bemerken noch, daß man für spezielle Labyrinthe oft einfachere Suchalgorithmen angeben kann. Ein allgemeiner Algorithmus, der auf beliebige Labyrinthe anwendbar ist, wird kaum anders beschaffen sein können, als daß er in einer bestimmten Weise alle Möglichkeiten durchmustert. Man wird daher keinen Algorithmus erhoffen dürfen, der wesentlich einfacher als der von uns vorgeschlagene Algorithmus ist.

1.4. Das Wortproblem

Das Wortproblem ist eine weitgehende Verallgemeinerung des Suchproblems des THESEUS. Während das Theseusproblem das Suchen in beliebigen endlichen Labyrinthen betrifft, kann das Wortproblem als ein Suchproblem in bestimmten unendlichen Labyrinthen interpretiert werden (vgl. Abschnitt 1.4.3). Das Wortproblem entstand in speziellen Teilgebieten der modernen Algebra, der sogenannten Theorie der Halbgruppen (der assoziativen Systeme) und der Gruppentheorie. Seine Bedeutung geht jedoch weit über den Rahmen dieser speziellen Theorien hinaus. Verschiedene Varianten dieses Problems wurden mit Erfolg von den hervorragenden sowjetischen Mathematikern ANDREJ ANDREEVIČ MARKOV und PETR SERGEEVIČ NOVIKOV sowie ihren Schülern bearbeitet.

1.4.1. Assoziative Kalküle

Wir beginnen mit einigen vorbereitenden Betrachtungen:

Unter einem *Alphabet* verstehen wir ein beliebiges endliches System aus paarweise verschiedenen Zeichen, die wir die *Buchstaben* dieses Alphabets nennen. So kann man beispielsweise das Alphabet $\{\alpha, \text{п}, z, ?\}$ betrachten, welches aus dem griechischen Buchstaben α, dem kyrillischen Buchstaben п, dem lateinischen Buchstaben z und dem Fragezeichen besteht. Ein *Wort* in einem gegebenen Alphabet ist eine beliebige endliche Folge von Buchstaben dieses Alphabets. Zum Beispiel

sind *abaa* und *bbac* Wörter im Alphabet $\{a, b, c\}$. Kommt ein Wort L als zusammenhängende Buchstabengruppe innerhalb eines gewissen anderen Wortes M vor, so nennen wir L ein *Teilwort* des Wortes M und sagen auch, L sei (als Teilwort) in M *eingebettet*. So ist z. B. das Wort *bcb* an zwei Stellen in das Wort *abcbcbab* eingebettet; die erste Einbettung beginnt beim zweiten Buchstaben, die zweite beim vierten.

Wir werden im folgenden Worttransformationen mittels gewisser zugelassener Substitutionen (Ersetzungen) betrachten. Eine zugelassene Substitution werden wir in der Form

$$P \to Q \quad \text{oder} \quad P - Q$$

schreiben. Dabei sind P und Q gegebene Wörter in dem betrachteten Alphabet. Die Anwendung der *gerichteten Substitution* $P \to Q$ auf ein Wort R ist genau dann möglich, wenn P als Teilwort in R eingebettet ist, und sie besteht darin, daß genau eines der (eventuell mehreren) Vorkommen von P in R durch Q ersetzt wird. Das im Ergebnis dieser Ersetzung entstehende Wort S nennen wir das *Resultat* der betreffenden gerichteten Substitution. Bei einer *ungerichteten Substitution* $P - Q$ kann demgegenüber entweder die linke Seite P durch die rechte Seite Q oder die rechte Seite Q durch die linke Seite P ersetzt werden; sie ist also auf ein Wort R genau dann anwendbar, wenn P oder Q Teilwort von R ist. Im folgenden werden wir zunächst meistens ungerichtete Substitutionen betrachten und diese einfach *Substitutionen* nennen.

Beispiel 1. Die Substitution $ab - bcb$ ist in vierfacher Weise auf das Wort *abcbcbab* anwendbar. Indem wir eine der beiden Buchstabengruppen *bcb* durch *ab* ersetzen, erhalten wir als Resultate die Wörter

aabcbab und *abcabab*;

die Substitution einer der beiden eingebetteten Buchstabengruppen *ab* durch *bcb* liefert die Wörter

bcbcbcbab und *abcbcbbcb*.

Auf das Wort *bacb* ist diese Substitution nicht anwendbar.

Die Menge aller Wörter in einem gegebenen Alphabet zusammen mit einem endlichen System von ungerichteten oder gerichteten Substitutionen nennen wir einen *assoziativen Kalkül*. Die zugelassenen Substitutionen bezeichnet man auch als die *Umformungsregeln* oder kurz *Regeln* des betreffenden assoziativen Kalküls.[1]) Ein assoziativer Kalkül ist also durch Angabe seines Alphabets und seiner Regeln festgelegt.

[1]) Assoziative Kalküle mit ungerichteten Substitutionen heißen (zu Ehren des norwegischen Mathematikers A. THUE, der sie erstmalig studierte) auch *Thue-Systeme*. Für assoziative Kalküle mit gerichteten Substitutionen ist die Bezeichnung *Semi-Thue-System* verbreitet. Ein Thue-System kann aufgefaßt werden als Semi-Thue-System, in dem mit einer Regel $P \to Q$ stets auch die inverse Regel $Q \to P$ auftritt. Bezüglich der Zusammenhänge der Thue-Systeme mit Halbgruppen verweisen wir auf das Ende von Abschnitt 1.4.2. [*Anm. d. Übers.*]

1.4.2. Das Äquivalenzproblem für Wörter

Kann man das Wort R durch einmalige Anwendung einer zugelassenen (ungerichteten!) Substitution in das Wort S überführen, so kann man offenbar durch einmalige Anwendung derselben Substitution in umgekehrter Richtung aus dem Wort S das Wort R erhalten. Wörter R und S, für die das der Fall ist, wollen wir *benachbart* nennen. Eine Folge

$$R_1, R_2, \ldots, R_{n-1}, R_n$$

von Wörtern, bei der jeweils R_i und R_{i+1} (bezüglich der Regeln eines gegebenen assoziativen Kalküls) benachbart sind, heißt eine vom Wort R_1 zum Wort R_n führende *Ableitungskette*. Wenn es in einem assoziativen Kalkül eine Ableitungskette gibt, die von R nach S führt, so gibt es auch eine Ableitungskette, die von S nach R führt. Wörter R und S, die im angegebenen Sinne durch eine Ableitungskette verbunden werden können, heißen im betrachteten Kalkül *äquivalent* (in Zeichen: $R \sim S$). Es ist klar, daß aus $R \sim S$ und $S \sim T$ stets $R \sim T$ folgt. Wir wollen ferner vereinbaren, daß jedes Wort zu sich selbst äquivalent ist (d. h., stets gilt $R \sim R$).

Für die weiteren Überlegungen benötigen wir das folgende Lemma:

Lemma. *Es sei $P \sim Q$, R sei ein Wort, das P als Teilwort enthält, und das Wort S entstehe dadurch aus dem Wort R, daß man ein Vorkommen von P in R durch Q ersetzt. Dann gilt auch $R \sim S$.*

Beweis. Da P Teilwort von R ist, läßt sich R in der Form UPV darstellen, wobei U der Teil des Wortes R ist, der vor P steht und V der Teil von R, der hinter P steht, und zwar betrachten wir das Vorkommen von P in R, das beim Übergang zu S durch Q ersetzt wird, so daß S das Wort UQV ist. Dabei lassen wir zu, daß U oder V „leer" ist. Wenn nun $P, P_1, \ldots, P_m, Q$ eine Ableitungskette ist, die von P zu Q führt, eine solche existiert wegen der vorausgesetzten Äquivalenz $P \sim Q$, so ist

$$UPV, UP_1V, \ldots, UP_mV, UQV$$

offenbar eine Ableitungskette, die von UPV (d. h. also R) zu UQV (d. h. also S) führt. Damit ist das behauptete Lemma bewiesen.

Beispiel 2. Wir betrachten einen assoziativen Kalkül, der zuerst von G. S. Cejtin untersucht wurde. Sein Alphabet ist die Menge $\{a, b, c, d, e\}$, und das System seiner zugelassenen Substitutionen (Regeln) lautet

$$\begin{aligned}
&ac - ca,\\
&ad - da,\\
&bc - cb,\\
&bd - db,\\
&abac - abace,\\
&eca - ae,\\
&edb - be.
\end{aligned}$$

In diesem Kalkül ist auf das Wort *abcde* nur die dritte Substitution anwendbar, und es gibt nur ein benachbartes Wort *acbde*. Es gilt ferner die Äquivalenz $abcde \sim cadedb$, wie aus der folgenden Ableitungskette ersichtlich ist:

$$\underline{abc}de,\ \underline{acb}de,\ ca\underline{bd}e,\ cad\underline{be},\ cadedb.$$

Auf das Wort *aaabb* kann keine der zugelassenen Substitutionen angewandt werden, daher existieren keine ihm benachbarten Wörter, und mithin gibt es kein von *aaabb* verschiedenes Wort, das zu ihm äquivalent ist.

Für jeden assoziativen Kalkül entsteht in natürlicher Weise ein spezielles *Äquivalenz-* oder *Wortproblem*. Es läßt sich folgendermaßen formulieren:

Von je zwei Wörtern des zugrunde liegenden Alphabets ist festzustellen, ob sie im betrachteten Kalkül äquivalent sind oder nicht.

In jedem Alphabet gibt es unendlich viele verschiedene Wörter. Folglich handelt es sich hier faktisch um unendlich viele Aufgaben gleichen Typs, zu deren Lösung ein Algorithmus gesucht wird, der die Äquivalenz oder die Nichtäquivalenz beliebiger Wortpaare festzustellen gestattet.

Es kann hier der Eindruck entstehen, daß das Äquivalenzproblem ein künstlich erdachtes Puzzlespiel wäre, dessen Lösung mittels eines Algorithmus keinerlei praktischen oder theoretischen Wert hätte. Das ist durchaus nicht der Fall. Es handelt sich sogar um ein ganz natürliches Problem von großer theoretischer und praktischer Bedeutung, welche die Anstrengungen völlig rechtfertigt, die zur Konstruktion eines entsprechenden Algorithmus aufgewandt werden. Im jetzigen Stadium unserer Ausführungen können wir diese Frage noch nicht ausführlich genug erörtern und gehen daher zunächst zur Untersuchung konkreter Tatsachen über.

[*Anm. d. Übers.:* Geht man bei einem assoziativen Kalkül mit ungerichteten Substitutionen von den Wörtern P zu ihren Äquivalenzklassen $[P]$ nach der Relation „$\sim$" über (es sei also $[P]$ die Menge aller Wörter Q, die der Bedingung $Q \sim P$ genügen) und erklärt man das Produkt $[P] \cdot [Q]$ zweier solcher Äquivalenzklassen als die Äquivalenzklasse $[PQ]$, der das durch Hintereinanderschreiben von P und Q entstehende Wort PQ angehört (man zeigt leicht, daß diese Definition korrekt ist), so erhält man eine sogenannte *Halbgruppe*. Das ist eine mit einer zweistelligen Operation „$\cdot$" versehene Menge $\mathfrak{M}$, in der das Assoziativgesetz für die Relation „$\cdot$" gilt, d. h. für beliebige x, y, z aus $\mathfrak{M}$ stets $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ ist. Aus diesem Grunde werden die Halbgruppen gelegentlich auch als *assoziative Systeme* oder *Assoziative* bezeichnet. Die den endlich vielen Buchstaben des betrachteten Alphabets entsprechenden Äquivalenzklassen haben die Eigenschaft, daß jedes Element der Halbgruppe Produkt dieser endlich vielen Elemente ist. Eine Halbgruppe, in der jedes Element Produkt von Elementen einer gegebenen endlichen Menge $\mathfrak{E}$ ist, nennt man *endlich erzeugbar* und die Menge $\mathfrak{E}$ ein (endliches) *Erzeugendensystem* für die Halbgruppe.

Wir bemerken, daß aus den Regeln $P - Q$ des assoziativen Kalküls in der Halbgruppe Gleichungen $[P] = [Q]$ werden, wobei man die Äquivalenzklassen $[P]$ und $[Q]$ noch entsprechend dem Aufbau der Wörter P und Q als Produkte von Erzeugen-

den schreiben kann. Die den endlich vielen Regeln des assoziativen Kalküls entsprechenden endlich vielen Gleichungen bezeichnet man als die *definierenden Relationen* der Halbgruppe. Sie legen in einem ganz bestimmten Sinne sämtliche in der Halbgruppe gültigen Gleichungen zwischen Produkten von Erzeugenden fest. Die Frage, ob Wörter R und S im assoziativen Kalkül äquivalent sind oder nicht, spiegelt sich in der Halbgruppe in der Frage wieder, ob zwei endliche Produkte von Erzeugenden dasselbe Element der Halbgruppe darstellen oder nicht. Dieses Problem bezeichnet man auch als das *Wortproblem* für die entsprechende Halbgruppe. Das Äquivalenzproblem für assoziative Kalküle entspricht also dem Wortproblem für endlich erzeugte Halbgruppen mit endlich vielen definierenden Relationen. Ein Beispiel für diesen Zusammenhang zwischen assoziativen Kalkülen und Halbgruppen wird in Abschnitt 1.4.5 behandelt.

Wir bemerken, daß bei Semi-Thue-Systemen, d. h. bei assoziativen Kalkülen mit gerichteten Substitutionen, die der Relation „$\sim$" entsprechende Relation $R \Rightarrow S$ (es gibt eine *gerichtete Ableitungskette* von R nach S) nicht mehr symmetrisch ist.]

1.4.3. Das Wortproblem und Labyrinthe

Zunächst wollen wir das Äquivalenzproblem für Wörter mit dem Problem des THESEUS in Verbindung bringen. Ordnet man jedem Wort einen „Platz" und jedem Paar benachbarter Wörter einen „Gang" zu, der die entsprechenden Plätze miteinander verbindet, so erscheint der assoziative Kalkül als ein Labyrinth mit unendlich vielen Plätzen und Gängen, wobei von jedem Platz nur endlich viele Gänge abgehen. Es kann dabei selbstverständlich auch Plätze geben, von denen kein Gang ausgeht (z. B. das Wort *aaabb* im Beispiel 2). Bei dieser Betrachtungsweise wird eine Ableitungskette, die von irgendeinem Wort R zu irgendeinem Wort Q führt, ein Weg in dem Labyrinth sein, der von einem Platz zu einem anderen geht; der Äquivalenz von Wörtern entspricht damit die gegenseitige Erreichbarkeit der betreffenden Plätze. Das Wortproblem wird auf diese Weise zu einem Suchproblem in einem unendlichen Labyrinth.

Damit die Besonderheiten der hier auftretenden Schwierigkeiten besser und klarer hervortreten, wollen wir vorher das *eingeschränkte Wortproblem* erörtern, welches folgendermaßen lautet:

Von je zwei Wörtern R und T des zugrunde liegenden Alphabets ist festzustellen, ob sie im betrachteten Kalkül durch höchstens k-malige Anwendung zugelassener Substitutionen ineinander übergeführt werden können oder nicht (k ist eine beliebige, aber ein für allemal fest gewählte natürliche Zahl).

Für diese Problemstellung kann man den geforderten Algorithmus leicht konstruieren. Es wird eine Liste aufgestellt, die mit dem Wort R beginnt, dann alle zu R benachbarten Wörter enthält, ferner die benachbarten dieser benachbarten usw., insgesamt k mal. Die Antwort lautet „ja" oder „nein", je nachdem, ob das Wort T in dieser Liste erscheint oder nicht.

Beim uneingeschränkten Wortproblem ist die Lage dagegen wesentlich anders. Da die Länge einer Ableitungskette, die von R nach T' führt, groß sein kann (wenn eine solche existiert), ist es im allgemeinen ungewiß, von welcher Stelle k ab man den Durchmusterungsprozeß als abgeschlossen betrachten darf. So habe man beispielsweise den Durchmusterungsprozeß schon bis zur Stelle $10^{20} = 100000000000000000000$ fortgesetzt und die Liste aller Wörter vor sich liegen, die man aus R durch mehrfache Anwendung der zugelassenen Substitutionen erhalten kann, deren Anzahl jedoch nicht größer als 10^{20} ist, und das Wort T' sei in dieser Liste nicht enthalten. Gibt es einen Grund anzunehmen, daß die Wörter R und T nicht äquivalent sind? Selbstverständlich nicht; denn es besteht ja durchaus die Möglichkeit, daß sie äquivalent sind, aber die kleinste Ableitungskette, die R und T' verbindet, länger ist.

1.4.4. Die Konstruktion von Algorithmen

Das einfache Durchmustern führt also beim uneingeschränkten Äquivalenzproblem nicht zum Ziel. Um hier zum gewünschten Ergebnis zu gelangen, muß man andere Methoden heranziehen, die den Mechanismus der Transformation von Wörtern in andere mittels der zugelassenen Substitutionen näher analysieren. Wir wollen etwa untersuchen, ob im assoziativen Kalkül von Cejtin (vgl. Beispiel 2) die Wörter *abaacd* und *acbdad* äquivalent sind. Die negative Antwort auf diese Frage ergibt sich aus den folgenden Überlegungen: In jeder der zugelassenen Substitutionen enthalten die linke und die rechte Seite den Buchstaben a gleich oft. Daher müssen alle Wörter einer Ableitungskette den Buchstaben a gleich oft enthalten. Da in den vorgegebenen beiden Wörtern der Buchstabe a verschieden oft auftritt, können sie nicht äquivalent sein, denn es kann keine Ableitungskette zwischen ihnen geben.

Mit Hilfe geeigneter *deduktiver Invarianten*, d. h. Eigenschaften, die für alle Wörter einer Ableitungskette gelten, kann man in einigen Fällen einen geforderten Algorithmus konstruieren.

Beispiel 3. Das Alphabet sei die Menge $\{a, b, c, d, e\}$, und das System der zugelassenen Substitutionen habe die Form

$$
\begin{aligned}
&ab - ba,\\
&ac - ca,\\
&ad - da,\\
&ae - ea,\\
&bc - cb,\\
&bd - db,\\
&be - eb,\\
&cd - dc,\\
&ce - ec,\\
&de - ed.
\end{aligned}
$$

Diese zugelassenen Substitutionen ändern die Anzahl der in einem Wort enthaltenen Buchstaben nicht, sondern nur ihre Stellung im Wort. Es ergibt sich unmittelbar, daß Wörter dann und nur dann äquivalent sind, wenn jeder Buchstabe in ihnen gleich oft auftritt. Daher ist ein Algorithmus zur Feststellung der Äquivalenz äußerst einfach anzugeben. Man hat nur nachzuzählen, wie oft jeder Buchstabe in den auf Äquivalenz zu untersuchenden Wörtern vorkommt, und die ermittelten Zahlen miteinander zu vergleichen.

Weiter unten werden wir ein etwas komplizierteres Beispiel betrachten. Zuvor wollen wir aber noch die Begriffe „Wort" und „zugelassene Substitution" etwas verallgemeinern. Neben den gewöhnlichen Wörtern des vorgegebenen Alphabets wollen wir ausdrücklich auch das *leere Wort*, das keinen Buchstaben enthält, als ein Wort unseres Alphabets betrachten; es werde durch das große griechische Lambda (Λ) bezeichnet. Sodann sollen auch Substitutionen der Form

$$P - \Lambda$$

zugelassen werden. Das Ersetzen des Wortes P durch das leere Wort bedeutet einfach, daß aus dem zu transformierenden Wort ein darin eingebettetes Vorkommen von P gestrichen wird. Das Ersetzen der rechten Seite durch die linke bedeutet, daß zwischen zwei beliebigen Buchstaben des zu transformierenden Wortes oder vor oder hinter dieses Wort das Wort P zu setzen ist.

Beispiel 4. Wir betrachten den im Alphabet $\{a, b, c\}$ durch die zugelassenen Substitutionen

$$\begin{aligned} b &- acc, \\ ca &- accc, \\ aa &- \Lambda, \\ bb &- \Lambda, \\ cccc &- \Lambda \end{aligned}$$

definierten assoziativen Kalkül. Es wird ein Algorithmus zur Lösung des Äquivalenzproblems für diesen Kalkül gesucht.

Wir konstruieren zunächst einen Hilfsalgorithmus, einen sogenannten *Reduktionsalgorithmus*, der zu jedem Wort ein ihm äquivalentes Wort spezieller Bauart (das zum gegebenen Wort gehörende reduzierte Wort) liefert. Dazu betrachten wir das folgende geordnete System von gerichteten Substitutionen

$$\begin{aligned} b &\to acc, \\ ca &\to accc, \\ aa &\to \Lambda, \\ cccc &\to \Lambda \end{aligned}$$

und verabreden die folgende Verfahrensweise bei der Anwendung des Reduktionsalgorithmus auf ein beliebiges Wort R. Es wird die erste auf das Wort R anwendbare

(gerichtete) Substitution dieser Liste genommen und ausgeführt. Sollte die zu substituierende Buchstabengruppe mehrmals in R eingebettet sein, so wird die erste, am weitesten links stehende Buchstabengruppe ersetzt. Auf das so erhaltene Wort R' wird wieder die in der Liste als erste auftretende Substitution, und zwar ebenfalls auf die am weitesten links stehende Buchstabengruppe angewandt. Hat man nach endlich vielen Schritten ein Wort S erhalten, auf das keine der Substitutionen mehr anwendbar ist, so sagen wir, daß der Algorithmus auf das Wort R angewandt wurde und es in das Wort S verwandelt hat.[1])

Wir zeigen, daß jedes Wort R durch Anwendung des Reduktionsalgorithmus in genau eines der folgenden acht Wörter umgewandelt wird:

$$\Lambda, c, cc, ccc, a, ac, acc, accc.$$

In der Tat: Tritt in R der Buchstabe b auf, so läßt er sich durch die erste Substitution $b \to acc$ beseitigen. Das wird gemäß der Vorschrift so lange fortgeführt, bis es kein b mehr gibt. Dann führt man die zweite Substitution aus, indem man alle a mit einem links stehenden c durch $accc$ ersetzt und zwar so lange, bis kein einziges a mehr vorhanden ist, welches unmittelbar rechter Nachbar von einem c ist, d. h., solange nicht alle a vor allen c stehen. Schließlich beseitigt man alle Paare benachbarter a und alle Quadrupel benachbarter c, bis nicht mehr als ein a und nicht mehr als drei c in dem Wort enthalten sind. Wörter dieser Form gibt es insgesamt acht, die oben angegeben wurden.

Offenbar entspricht jedem Wort ein reduziertes, dem es äquivalent ist. Daher sind Wörter dann und nur dann äquivalent, wenn ihnen dasselbe reduzierte Wort entspricht, oder wenn die ihnen entsprechenden reduzierten Wörter äquivalent sind. Wir werden später beweisen, daß keine zwei der reduzierten Wörter äquivalent sind. Hieraus folgt, daß Wörter dann und nur dann äquivalent sind, wenn ihnen dasselbe reduzierte Wort entspricht. Damit können wir leicht einen Algorithmus für das gestellte Wortproblem angeben. Er besteht darin, daß der Reduktionsalgorithmus auf jedes der beiden auf Äquivalenz zu untersuchenden Wörter angewandt wird und die erhaltenen reduzierten Wörter miteinander verglichen werden.

Es seien beispielsweise die Wörter $cacb$ und bb gegeben. Wir bestimmen die zu ihnen gehörigen reduzierten Wörter:

1. $cac\underline{b}$, $\underline{ca}cacc$, $acc\underline{ca}ccc$, $acc\underline{ca}cccc$, $acc\underline{a}ccccc$, $ac\underline{a}cccccccc$, $ac\underline{a}ccccccccc$, $\underline{aa}ccccccccccc$, $\underline{cc}ccccccccccc$, $\underline{cc}cccccccc$, $\underline{cc}ccccc$, cc;

2. $\underline{bb}$, $acc\underline{b}$, $acc\underline{a}cc$, $ac\underline{a}cccc$, $\underline{aa}ccccccc$, $\underline{cc}ccccc$, $\underline{cc}cc$, Λ.

Schlußfolgerung: Die Wörter $cacb$ und bb sind nicht äquivalent, weil wir die beiden verschiedenen reduzierten Wörter cc und Λ erhielten.

[1]) Algorithmen zur Umwandlung der Wörter eines gewissen Alphabets, die in dieser Weise mittels eines geordneten Systems gerichteter Substitutionen arbeiten, werden als *normal* bezeichnet. Die Theorie der normalen Algorithmen, die von großem praktischen und theoretischen Interesse ist, wurde von A. A. Markov geschaffen und in seinem 1954 erschienenen Buch „Теория алгорифмов" dargelegt.

Wir wollen nun zeigen, daß keine zwei der reduzierten Wörter zueinander äquivalent sind. Dazu stellen wir zunächst folgendes fest: Wenn es eine Ableitungskette von dem nicht den Buchstaben b enthaltenden Wort R zu dem ebenfalls nicht den Buchstaben b enthaltenden Wort S gibt, so gibt es eine Ableitungskette von R nach S, deren sämtliche Glieder den Buchstaben b nicht enthalten. Ersetzt man nämlich in allen Wörtern einer Ableitungskette von R nach S alle darin eventuell auftretenden Vorkommen von b durch acc, so erhält man eine Folge von Wörtern, in der zwei nebeneinanderstehende Wörter entweder (im Sinne des betrachteten assoziativen Kalküls) benachbart oder identisch sind. Läßt man die nebeneinanderstehenden Wiederholungen von Wörtern fort, so erhält man eine Ableitungskette der gewünschten Art. Wir bemerken, daß bei Ableitungsketten dieses Typs die Substitution $b - acc$ nicht benutzt wird. Bei allen anderen zugelassenen Substitutionen tritt der Buchstabe a entweder auf beiden Seiten in gerader oder auf beiden Seiten in ungerader Anzahl auf, und dasselbe ist für den Buchstaben c der Fall. Aus alldem folgt, daß keines der vier reduzierten Wörter, das den Buchstaben a enthält, einem der vier reduzierten Wörter äquivalent sein kann, das den Buchstaben a nicht enthält. Genauso kann keines der reduzierten Wörter, das den Buchstaben c einmal oder dreimal enthält, einem der reduzierten Wörter äquivalent sein, das den Buchstaben c keinmal oder zweimal enthält. Folglich brauchen wir nur noch zu zeigen, daß die folgenden vier Paare von reduzierten Wörtern nicht äquivalent sind:

$$\Lambda, cc; \quad c, ccc; \quad a, acc; \quad ac, accc.$$

Auf Grund des Lemmas aus Abschnitt 4.2 ist nun unmittelbar klar, daß die Äquivalenz eines der ersten drei Paare die Äquivalenz des vierten Paares zur Folge hätte. Daher genügt es zu zeigen, daß die Wörter ac und $accc$ nicht äquivalent sind. Das wollen wir nun tun:

Unter dem *Index* eines an einer bestimmten Stelle eines Wortes R stehenden Buchstabens a verstehen wir die Anzahl der c, die in R rechts von diesem a stehen. Die Summe aller Indizes der in einem Wort R enthaltenen a nennen wir den Index des Wortes R.[1]) Jede der Substitutionen $aa - \Lambda$ und $cccc - \Lambda$ ändert den Index des Wortes um eine gerade Zahl. In der Tat: Substituiert man aa anstelle des leeren Wortes, so erhöht sich der Index um die Summe der Indizes der beiden a, da beide Indizes gleich sind also um eine gerade Zahl. Entsprechend verkleinert sich der Index des Wortes um eine gerade Zahl, wenn man aa durch das leere Wort ersetzt. Setzt man $cccc$ für das leere Wort ein, so vergrößern sich die Indizes gewisser a um 4, während die Indizes der anderen a unverändert bleiben. Insgesamt vergrößert sich also der Index des Wortes um eine gerade Zahl. Analog wird der Index um eine gerade Zahl verkleinert, wenn man $cccc$ streicht. Die Substitution $b - acc$ ändert offenbar den Index des Wortes ebenfalls nur um eine gerade Zahl.

[1]) Im Wort $acbca$ beispielsweise hat der erste Buchstabe a den Index 2, während das zweite a den Index 0 hat. Also hat das Wort den Index 2.

Jetzt zeigen wir, daß die Substitution $ca \to accc$ den Index des Wortes um eine ungerade Zahl ändert. Dazu vergleichen wir die Indizes der Wörter $PcaQ$ und $PacccQ$. Der Index jedes in P enthaltenen a ist in $PacccQ$ um 2 größer als in $PcaQ$, die Indizes der a in Q sind in beiden Wörtern dieselben, der Index des a zwischen P und Q ist in $PacccQ$ um 3 größer als in $PcaQ$. Also ändert diese Substitution den Index in der Tat um eine ungerade Zahl. Die Wörter ac und $accc$ haben nun beide einen ungeraden Index. Wären sie also äquivalent, so müßte in der Ableitungskette, die ac in $accc$ überführt und von der wir annehmen können, daß in ihr der Buchstabe b nicht auftritt, eine gerade Anzahl von Substitutionen $ca \to accc$ auftreten. Das führt jedoch, wie wir sofort sehen werden, auf einen Widerspruch: Bei jeder Substitution $ca \to accc$ ändert sich die Anzahl der c um 2. Daher muß eine gerade Anzahl von derartigen Substitutionen die Anzahl der vorkommenden c um ein Vielfaches von 4 ändern. Die Substitution $cccc \to \Lambda$ verändert die Anzahl der c um genau 4, während $aa \to \Lambda$ die Anzahl der c nicht ändert. Wäre $ac \sim accc$, so müßte sich die Anzahl der vorkommenden Buchstaben c um ein Vielfaches von 4 unterscheiden; das ist aber offenbar nicht der Fall. Also können die Wörter ac und $accc$ nicht äquivalent sein. Damit ist der vorgeschlagene Algorithmus als zweckentsprechend nachgewiesen.

1.4.5. Decktransformationen eines Quadrats

Die in Abschnitt 1.4.4 vorgeführte Lösung des Wortproblems für einen konkreten assoziativen Kalkül charakterisiert in vielem die Begriffe und Methoden, die auch bei der Untersuchung des Wortproblems bei anderen assoziativen Kalkülen auftreten. Wir wollen uns jetzt noch über die inhaltliche Bedeutung des Wortproblems und seine Wichtigkeit für die moderne Algebra klar werden. Das läßt sich an einem konkreten Beispiel zeigen.

Wir betrachten ein Quadrat (vgl. Abb. 9, I) und drei seiner *Decktransformationen*, d. h. Transformationen, die das Quadrat deckungsgleich in sich überführen:

1. Spiegelung an der vertikalen Achse durch den Mittelpunkt O des Quadrates;
2. Spiegelung an der horizontalen Achse durch den Mittelpunkt O des Quadrates;
3. Drehung um 90° im Uhrzeigersinn um den Mittelpunkt O.

Diese Transformationen wollen wir *elementar* nennen und mit den Buchstaben a, b, c bezeichnen. Abb. 9, III, IV, V zeigt, wie sich die Lage der Ecken des Quadrats II bei jeder der elementaren Transformationen ändert.

Man sieht sofort, daß die Hintereinanderausführung zweier (und damit beliebig vieler) Decktransformationen wieder eine Decktransformation ist. In der Mathematik ist es üblich, die Hintereinanderausführung von Transformationen (Abbildungen) als *Produkt* zu bezeichnen und die bei der Multiplikation von Zahlen übliche Symbolik und Terminologie zu verwenden. Demgemäß ist das Produkt fg zweier Transformationen f und g diejenige Transformation, die man erhält, wenn man zunächst

die Transformation f und anschließend die Transformation g anwendet.[1]) So ist beispielsweise das Produkt cc (vgl. Abb. 9, VI) das Ergebnis einer zweimaligen Drehung um 90°, d. h. die Drehung um 180°. Das Produkt ac (vgl. Abb. 9, VIII) ist Resultat der Spiegelung an der vertikalen Achse mit nachfolgender Drehung um 90°, was der Spiegelung an der rechten Diagonalen gleichwertig ist (vgl. Abb. 9, I).

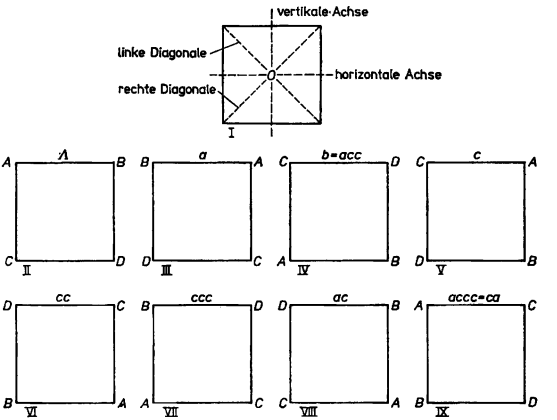


Abb. 9

Das Produkt $(ac)\,(cc)$ der beiden vorigen Produkte entspricht der Spiegelung an der linken Diagonalen (vgl. Abb. 9, I). Die hier eingeführte Multiplikation ist *nicht kommutativ*, d. h., die Faktoren eines Produkts dürfen im allgemeinen nicht miteinander vertauscht werden. In Abb. 9, VIII und IX sind die verschiedenen Lagen der Ecken des Quadrats dargestellt, die sich bei den Deckransformationen ac bzw. ca des Ausgangsquadrats II ergeben. Von den Eigenschaften der Multiplikation von Zahlen bleibt aber die *Assoziativität* erhalten: Für beliebige Transformationen p, q, r gilt die Identität

$$(pq)\,r = p(qr).$$

[1]) Weiter verbreitet ist allerdings die Festsetzung, daß fg die Transformation ist, die man erhält, wenn man zuerst g und dann f anwendet, d. h., die durch $(fg)\,(x) = f(g(x))$ gegeben wird. [*Anm. d. Übers.*]

Also kann man die Stellung der Klammern in den Produkten beliebig ändern. So ergeben zum Beispiel $(ac)(cc)$ und $\big((ac)\,c)\,c)\big)$ die gleiche Decktransformation des Quadrats, und zwar die Spiegelung an der linken Diagonalen.

Gegenstand unserer nachfolgenden Betrachtungen ist die Gesamtheit Ω der elementaren Transformationen a, b, c und aller jener Decktransformationen des Quadrats, die sich als Produkt endlich vieler elementarer Transformationen darstellen lassen. Da die Multiplikation assoziativ ist, kann man bei der symbolischen Schreibweise der Elemente aus Ω die Klammern weglassen, also *abb*, *cabb*, *accc* usw. schreiben. Man muß nur auf die richtige Reihenfolge beim Ausführen der Transformationen achten. Somit kann man also jedes Produkt als Wort im Alphabet $\{a, b, c\}$ schreiben.[1])

Wegen der Assoziativität der Multiplikation von Transformationen entspricht dem Wort PQ das Produkt der Decktransformationen, welche den Wörtern P und Q entsprechen. So ist z. B. die zum Wort *abccab* gehörige Decktransformation das Produkt der zu den Wörtern *abc* und *cab* gehörenden Decktransformationen.

Im Alphabet $\{a, b, c\}$ gibt es nun unendlich viele graphisch verschiedene Wörter. Dabei können graphisch verschiedene Wörter P und Q, wie wir bereits an Beispielen gesehen haben, dieselbe Decktransformation aus Ω darstellen. Diesen Fakt wollen wir durch $P = Q$ wiedergeben. Man verifiziert leicht, daß die folgenden Gleichungen gelten:

$$b = acc, \tag{1}$$

$$ca = accc. \tag{2}$$

Zu diesem Zweck braucht man nur die durch die Transformationen auf den linken bzw. rechten Seiten dieser Gleichungen entstehende Lage der Ecken des Quadrats zu vergleichen. Ferner sieht man ohne weiteres ein, daß jedes der Wörter *aa*, *bb*, *cccc* dieselbe Decktransformation ergibt, und zwar die sogenannte *identische Transformation*, bei der alle Ecken in ihrer Ausgangslage bleiben. Diese Transformation wollen wir mit dem Symbol Λ bezeichnen. Somit gelten die Gleichungen

$$aa = \Lambda, \tag{3}$$

$$bb = \Lambda, \tag{4}$$

$$cccc = \Lambda. \tag{5}$$

Ein Vergleich von (1) bis (5) mit den zugelassenen Substitutionen des assoziativen Kalküls aus Beispiel 4 gestattet die folgende Aussage, die den engen Zusammenhang zwischen dem Kalkül und dem betrachteten System von Decktransformationen des Quadrats zeigt:

Zwei Produkte elementarer Decktransformationen eines Quadrats ergeben dann und nur dann die gleiche Transformation, wenn die ihnen entsprechenden Wörter im assoziativen Kalkül des Beispiels 4 *äquivalent sind.*

[1]) Die Menge Ω bildet also bezüglich der Multiplikation „$\cdot$" eine Halbgruppe, in der die Menge $\{a, b, c\}$ der elementaren Transformationen ein Erzeugendensystem ist. Die folgenden Gleichungen (1) bis (5) sind die definierenden Relationen dieser Halbgruppe. [*Anm. d. Übers.*]

Aus den Gleichungen (1) bis (5) folgt nämlich, daß bei jeder Anwendung einer zugelassenen Substitution auf ein beliebiges Wort S dieses in ein Wort übergeführt wird, das dieselbe Transformation wie das Ausgangswort darstellt. Wenn wir beispielsweise die Substitution $ca \to accc$ auf das Wort $bcac$ anwenden, erhalten wir das Wort $bacccc$; wegen der Assoziativität der Multiplikation können wir aber

$$bcac = b(ca)\,c \quad \text{und} \quad bacccc = b(accc)\,c$$

schreiben. Die rechten Seiten dieser Gleichungen sind als Produkte jeweils gleicher Faktoren ebenfalls gleich; dann müssen aber auch die linken Seiten übereinstimmen. Je zwei im Kalkül benachbarte Wörter stellen also dieselbe Decktransformation dar. Hieraus folgt, daß die Äquivalenz zweier Wörter die Gleichheit der zugehörigen Decktransformationen zur Folge hat. Aber auch die Umkehrung ist richtig: Denn stellen die Wörter S und T dieselbe Decktransformation dar, so gilt das auch für die zu S und T gehörigen reduzierten Wörter (denn diese sind zu S und T äquivalent und stellen daher nach dem bereits Bewiesenen dieselbe Decktransformation wie S bzw. T dar). Man verifiziert nun leicht direkt (vgl. Abb. 9, II bis IX), daß die acht reduzierten Wörter paarweise verschiedene Decktransformationen ergeben. Folglich müssen S und T das gleiche reduzierte Wort haben. Dann sind sie aber nach dem in Abschnitt 1.4.4 Bewiesenen äquivalent.

Damit hat die formale Äquivalenz zweier Wörter im betrachteten assoziativen Kalkül eine ganz konkrete geometrische Bedeutung erhalten, wobei das Erkennen der Äquivalenz zweier Wörter der Lösung einer bestimmten geometrischen Aufgabe entspricht (nämlich der Aufgabe, festzustellen, ob die den Wörtern entsprechenden Produkte von elementaren Decktransformationen dieselbe Decktransformation ergeben). Zugleich liefert der von uns beschriebene Algorithmus eine allgemeine Methode zur Lösung beliebiger geometrischer Aufgaben dieses Typs.

Analog kann man dem Äquivalenzproblem vieler anderer assoziativer Kalküle eine konkrete geometrische, algebraische oder auch andere Deutung geben. Ohne Übertreibung kann man sagen, daß es in jedem Gebiet der Mathematik Sätze gibt, die nach einigen Umformulierungen als Aussagen über die Äquivalenz von Wörtern in einem gewissen assoziativen Kalkül formuliert werden können. Dieser Problemkreis kann natürlich aus Platzmangel nicht ausführlich behandelt werden; einige Ergänzungen werden wir noch im Verlauf der weiteren Darlegungen geben (vgl. Abschnitt 2.1.2).

Wir wollen noch hervorheben, daß wir, ausgehend von der geometrischen Deutung, die wir dem Wortproblem des assoziativen Kalküls aus Beispiel 4 gaben, einen noch einfacheren Algorithmus konstruieren können: Man braucht nur für jedes von zwei vorgelegten Produkten die Folge der entsprechenden Decktransformationen auszuführen (etwa anhand einer Zeichnung) und die erhaltenen Ergebnisse zu vergleichen.

Übungsaufgabe. Man löse das Wortproblem für den assoziativen Kalkül, der durch das Alphabet $\{a, b\}$ und die zugelassenen Substitutionen

$$aaa - bb,$$
$$bbbb - \Lambda$$

definiert wird.

1.4.6.* Wortgleichungen

Wir wollen noch ein mit der Untersuchung assoziativer Kalküle zusammenhängendes Problem beschreiben, für das bislang kein Entscheidungsalgorithmus gefunden werden konnte.

Dazu fixieren wir ein der Einfachheit halber zweibuchstabiges Alphabet $\{a, b\}$. Im folgenden sollen unter Wörtern stets nichtleere Wörter in diesem Alphabet verstanden werden. Unter dem Produkt $P_1P_2 \cdots P_m$ der Wörter $P_1, P_2, \ldots, P_m$ (in dieser Reihenfolge) verstehen wir das Wort, das wir erhalten, wenn wir rechts an das Wort P_1 das Wort P_2 anfügen, rechts an das so erhaltene Wort das Wort P_3 und so weiter bis zum Wort P_m. Es sei z. B.

$$P = ab, \quad R = baba, \quad S = ba, \quad T = abba.$$

Das Wort $abbabababa$ kann dann sowohl als Produkt PRR als auch als Produkt TRS aufgefaßt werden. In diesem Sinne hat die Wortgleichung $xyy = uyz$ (man kann sie auch kürzer in der Form $xy^2 = uyz$ schreiben) die Lösung $x = P, y = R, u = T, z = S$. Offenbar hat diese Gleichung auch viele andere Lösungen. Die Wortgleichung $xy = xy^2$ hat keine Lösung. Allgemein können in Wortgleichungen auch Wörter im Alphabet $\{a, b\}$ als Koeffizienten auftreten. So ist $xbabay = uyba$ beispielsweise eine Wortgleichung mit den Koeffizienten $baba$ (links) und ba (rechts) und den Unbekannten x, y und u. Eine mögliche Lösung ist

$$x = ab, \quad y = baba, \quad u = abba.$$

Unter einem System von Wortgleichungen verstehen wir ein System, in dem jede Gleichung, wie in den gerade analysierten Beispielen, eine Verknüpfung zweier Produkte von Unbekannten und Koeffizienten mittels des Gleichheitszeichens ist. Wie üblich heißt ein Tupel von Wörtern, die, für die Unbekannten eingesetzt, jede Gleichung des Systems erfüllen, eine Lösung des Systems.

Wir betonen, daß wir hier unter der Gleichheit von Wörtern ihre graphische Gleichheit verstehen. Das System

$$xy^2 = uyz, \quad xy = yz$$

wird z. B. durch $x = U$, $y = U$, $u = U$, $z = U$ mit einem beliebigen Wort U ge-

löst. Dagegen ist $x = P$, $y = R$, $u = I$ und $z = S$ keine Lösung mehr, denn es gilt

$$PR = abbaba \neq babaab = RP.$$

Das uns interessierende Problem besteht nun in folgendem:

Gesucht ist ein Algorithmus, der für ein beliebiges System von Wortgleichungen entscheidet, ob es lösbar ist oder nicht.

1.5. Rechenmaschinen mit automatischer Steuerung

1.5.1. Der Mensch als Rechner

Unsere nächste Aufgabe besteht darin, die Grundprinzipien der Konstruktion und der Arbeitsweise von Maschinen mit automatischer Steuerung zu erläutern. Zu diesem Zweck wenden wir uns noch einmal vorbereitend der Betrachtung eines algorithmischen Prozesses zu, der von einem Menschen (Rechner) ausgeführt wird.

Durch den Algorithmus angeleitet, führt der rechnende Mensch einen Prozeß aus, bei dem einerseits das Verfahren und andererseits die Aufbewahrung (Speicherung), die Verarbeitung und die Ausgabe gewisser Daten (gewisser *Informationen*) von Bedeutung sind. Diese Daten schreibt (speichert) der Rechner gewöhnlich auf einem Bogen Papier mit Hilfe von Ziffern, Buchstaben und anderen Symbolen auf. Die Gesamtheit aller dieser Symbole bildet das sogenannte *Arbeitsalphabet*. So wird beispielsweise in der Algebra ein Alphabet benutzt, das außer den gewöhnlichen Buchstaben und Ziffern auch Zeichen für die algebraischen Operationen, Klammern usw. enthält.

Für einen Rechenprozeß, der von einem Menschen (dem Rechner) durchgeführt werden soll, ist das Auftreten folgender drei Faktoren charakteristisch (vgl. Abb. 10a).

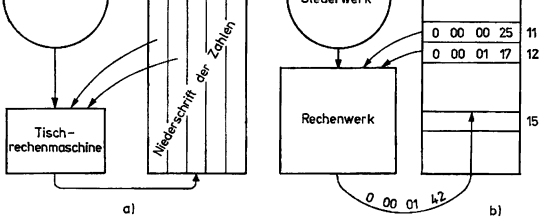


Abb. 10

1. Aufbewahrung von Informationen. Sie wird gewöhnlich durch Aufschreiben aller Daten auf einen Bogen Papier realisiert, wobei zu den Daten auch die Anweisung (das Schema des Algorithmus) zur Lösung der Aufgaben gehören möge. Wir wollen nicht unerwähnt lassen, daß der Rechner faktisch nicht alles zu Papier bringt; einiges merkt er sich (bewahrt es nicht auf dem Bogen Papier, sondern im Gedächtnis auf), und gewisse Daten entnimmt er verschiedenen Nachschlagewerken und Tabellen. Dadurch darf jedoch nicht die Grundtatsache verschleiert werden, daß der Rechenprozeß das Vorhandensein eines Mediums voraussetzt, das die Aufbewahrung aller notwendigen Daten ermöglicht. In unserem Schema hat man also unter dem Bogen Papier die Gesamtheit aller Mittel zu verstehen, durch die die Aufbewahrung aller Daten gewährleistet wird.

2. Bearbeitung von Informationen. Sie setzt die Fähigkeit des Rechners voraus, die im Algorithmus vorgesehenen einzelnen elementaren Operationen auszuführen. Eventuell müssen dem Rechner zu diesem Zweck spezielle Mechanismen zur Verfügung gestellt werden, z. B. können die gewöhnlichen arithmetischen Operationen auf einer Tischrechenmaschine ausgeführt werden. Jede einzelne Operation besteht im Prinzip in folgendem: Der Rechner entnimmt gemäß den Anweisungen dem Bogen Papier gewisse Daten (z. B. Zahlen) und überträgt sie auf das Arbeitsgerät (Tischrechenmaschine). Das Ergebnis wird dann wieder an einem wohlbestimmten Platz des Bogens notiert.

3. Steuerung des Prozesses. Die Vorbereitung und die Durchführung der einzelnen Operationen in jeder Etappe des Prozesses wird vom Rechner gemäß den Anweisungen besorgt.

1.5.2. Rechenautomaten

Aus welchen Teilen muß sich nun eine Maschine aufbauen, die die geschilderte Funktion eines Rechners übernehmen kann, und wie müssen diese zusammenwirken? Die Antwort auf diese Frage kann man vorwegnehmen. Die Maschine muß den eben beschriebenen Prozeß ebenfalls ausführen können, aber ohne Eingreifen eines Rechners.

Die Maschine muß also erstens zur Niederschrift der Informationen ein bestimmtes *Arbeitsalphabet* haben. Statt durch die gewöhnlichen graphischen Darstellungen, die sich durch ihre Gestalt voneinander unterscheiden, werden in der Maschine die verschiedenen Symbole des Alphabets durch voneinander verschiedene physikalische Zustände nachgebildet, z. B. durch verschiedene elektrische Potentiale oder verschiedene Magnetisierungszustände.

Allgemeine Überlegungen rechtfertigen die bevorzugte Anwendung eines Alphabets mit zwei Symbolen (*duales Alphabet*); man schreibt gewöhnlich 0 und 1. Dieses Alphabet läßt sich einfach in Gestalt zweier Zustände physikalisch realisieren: hohes Potential (oder Strom fließt) und niedriges Potential (oder kein Strom fließt). Dabei kommt hinzu, daß die einfachsten logischen Operationen an Veränderlichen aus-

geführt werden, die nur zwei Werte annehmen können: „wahr" und „falsch". Die Wahl eines bestimmten Alphabets und die Methoden zur Darstellung der benötigten Daten in diesem Alphabet sind jedoch für das Verständnis des Aufbaus und der Arbeitsweise der Maschine nicht entscheidend. Daher wollen wir uns auf den Hinweis beschränken, daß in den modernen Maschinen vorzugsweise das duale Alphabet und das Dualsystem (anstelle des üblichen Dezimalsystems) verwendet werden. Auf diese Einzelheiten werden wir aber im folgenden nicht näher eingehen.

Die in die Maschine eingegebenen Informationen sowie diejenigen, die im Verlauf des Prozesses erarbeitet werden, erscheinen in Form gewisser physikalischer Parameter. In den uns interessierenden Fällen werden alle Daten durch Zahlen wiedergegeben. Insbesondere wird auch der Algorithmus, der die Maschine bei ihrer Arbeit „leitet", durch den Satz von Zahlen chiffriert. Algorithmen, die speziell für Maschinen entwickelt worden sind, nennt man gewöhnlich *Programme*. Das Programm ist der wichtigste Teil der Information, mit der die Maschine operiert.

Ferner gibt es in Übereinstimmung mit Abb. 10a in der Maschine Baugruppen, welche die Funktion der Speicherung, der Weitergabe, der Bearbeitung und der Steuerung des Prozesses übernehmen (vgl. Abb. 10b).

1. Ein Speicher übernimmt die Funktion des Bogens Papier. In der verabredeten Sprache fixiert die Maschine in ihm alle notwendigen Daten einschließlich des Programms. Es dürfte keinem Zweifel unterliegen, daß es möglich ist, eine Baugruppe zu konstrujeren, die diese Funktion erfüllt. Diese Tätigkeit kann z. B. ein Magnetband übernehmen, auf dem die kodierten Daten festgehalten und von dem sie wieder abgenommen werden können wie beim normalen Tonbandgerät. Der Speicher (das Gedächtnis der Maschine) besteht aus einer Menge von *Zellen*, die mit Hilfe der natürlichen Zahlen 1, 2, 3, ... numeriert sind. Diese Zahlen nennt man die *Adressen* der Zellen. In jeder Zelle kann eine kodierte Nachricht gespeichert werden. In den Rechenautomaten wird jede solche Nachricht, wie schon erwähnt, in Form einer Zahl dargestellt.

In der Praxis werden in den elektronischen Automaten neben dem Magnetband auch andere Arten von Speichern verwendet, z. B. Elektronenstrahlröhren, deren Arbeitsweise etwas an die Fernsehröhren erinnert, oder Magnettrommelspeicher; alle diese Bauelemente werden zweckentsprechend zum Speichern eingesetzt. Wir werden die einzelnen Speicherarten nicht weiter unterscheiden. Ohne Nachteil für das Verständnis der Sache können wir uns einen bestimmten Speicher vorstellen, etwa ein Magnetband.

2. Ein Rechenwerk übernimmt die Aufgabe der gewöhnlichen Tischrechenmaschine, obwohl die seiner Konstruktion zugrunde liegenden physikalischen Prinzipien völlig andersgeartet sind. Die Verarbeitung der Eingabewerte zum gewünschten Ergebnis (z. B. die Addition von Zahlen) geschieht dadurch, daß elektrische Eingangssignale (die den Eingabewerten entsprechen) durch elektronische Geräte in elektrische Ausgangssignale (die den Ausgabedaten entsprechen) verwandelt werden. Die Eingangssignale kommen aus Speicherzellen in das Rechenwerk, die Ausgangs-

signale verlassen das Rechenwerk und werden wieder in einer Speicherzelle untergebracht. Schematisch ist das in Abb. 10b dargestellt, wo die Zahlen aus den Zellen 11 und 12 addiert werden und das Ergebnis in die Zelle 15 gebracht wird. Damit nun diese Operationen in der Maschine in einem gewissen Rhythmus ausgeführt werden, müssen zu Beginn des Taktes die Zellen 11 und 12 sowie die Zelle 15 mit dem Rechenwerk verbunden werden. Ferner muß auch das Rechenwerk auf die richtige Operation (im vorliegenden Fall die Addition) eingestellt sein. Alles das fällt in die „Kompetenzen" des Steuerwerks.

3. Ein Steuerwerk hat die Funktion zu erfüllen, die im Schema der Abb. 10a dem Rechner zukommt. In jedem Arbeitsstadium muß das Steuerwerk die Bedingungen zur Realisierung der nachfolgenden Operation herstellen. Dabei arbeitet es wie eine im Fernsprechbetrieb verwendete Selbstwählanlage, die diejenigen „Teilnehmer" (Verteiler und Zellen des Speichers der Maschine) miteinander verbindet, die an der jeweiligen Operation beteiligt sind. Bildlich gesprochen schlägt das Steuerwerk im Programm nach, was zu machen ist, und stellt dann die entsprechenden Verbindungen in der Maschine her, damit die nächste Operation ungestört ablaufen kann.

1.5.3. Maschinenbefehle

Bevor wir die hier auftretenden Besonderheiten genauer beschreiben, bemerken wir, daß eine Maschine im wesentlichen durch ihr Befehlssystem, d. h. die in den Programmen zu verwendenden Elementaroperationen (Befehle), charakterisiert ist. Jedes Programm, welches die Maschine verarbeiten kann, ist eine bestimmte Kombination möglicher Befehle mit gewissen Hilfszahlen (Parameter). Je nach dem Aufbau der Befehle unterscheidet man zwischen *Ein-*, *Zwei-* und *Dreiadreßmaschinen*. Die Maschine M 20 ist ein Beispiel für eine Dreiadreßmaschine, die Maschinen der Serie „Minsk" sind Zweiadreßmaschinen, die Maschine BESM 6 ist eine Einadreßmaschine.

Bei einer Dreiadreßmaschine, an deren Beispiel wir die Struktur der Befehle genauer erläutern wollen, ist jeder *Befehl* ein Quadrupel

$$\alpha\beta\gamma\delta$$

von Zahlen. Die Zahl α gibt die Nummer der auszuführenden Operation, die Zahlen β und γ sind die Adressen der Zellen, aus denen die durch die Operation mit der Nummer α zu verknüpfenden Zahlen (die sogenannten *Operanden*) zu entnehmen sind, die Zahl δ ist die Adresse der Zelle, in der das Resultat abgespeichert werden soll.

Die Befehle werden ihrerseits in gewissen Zellen des Speichers untergebracht. Zu ihrem Verständnis müssen die Ziffern einer Zahl, die einen Befehl kodiert, in vier Gruppen aufgeschlüsselt werden, die die angegebene Bedeutung haben. So soll beispielsweise die in Zelle 1 des Speichers aus Abb. 10b stehende Zahl 1 11 12 15 den folgenden Befehl chiffrieren: *Die Zahlen aus den Zellen* 11 *und* 12 *sind der Operation*

mit der Nummer 1 (das möge die Addition sein) *zu unterwerfen, und das Resultat ist in Zelle* 15 *zu speichern* (wir haben also eine Zerlegung der jeweiligen Zahl von rechts nach links in Gruppen zu je zwei Ziffern angenommen, was wir auch in den folgenden Beispielen tun wollen).

Die heute üblichen Befehlssysteme umfassen einige Dutzend Befehlstypen. Es ist jedoch der Trend zu Befehlssystemen mit einigen hundert oder sogar tausend Befehlstypen zu beobachten. Es seien hier einige Beispiele von besonders wichtigen Befehlstypen genannt. Wir weisen darauf hin, daß insbesondere die von uns gewählte Numerierung der Operationen Beispielcharakter hat, d. h. in realen Maschinen im allgemeinen ganz anders ist.

1. Arithmetische Befehle:

a) 1 β γ δ Addiere die Zahl aus β zu der Zahl aus γ und speichere die Summe in δ;

b) 2 β γ δ Subtrahiere von der in β stehenden Zahl die Zahl aus γ und speichere die Differenz in δ;

c) 3 β γ δ Multipliziere die Zahl aus β mit der Zahl aus γ und speichere das Produkt in δ;

d) 4 β γ δ Dividiere die Zahl aus β durch die Zahl aus γ und speichere den Quotienten in δ.

2. Sprungbefehle:

e) 5 00 00 δ Gehe zu dem in δ stehenden Befehl über (*unbedingter Sprung*);

f) 5 01 γ δ Gehe zu dem in δ stehenden Befehl über unter der Bedingung, daß in der Zelle γ eine positive Zahl gespeichert ist;

g) 5 02 γ δ Gehe zu dem in δ stehenden Befehl über unter der Bedingung, daß in der Zelle γ eine negative Zahl gespeichert ist.

3. Stopbefehl: 0 00 00 00.

Außer den aufgezählten Befehlen gibt es in modernen Rechenautomaten noch Befehle für die sogenannten *logischen Operationen* und auch andere, mit denen wir uns hier jedoch nicht weiter aufhalten wollen. Die aufgezählten Befehle genügen schon völlig zur Aufstellung der verschiedenartigsten Programme. Einige Beispiele werden in Abschnitt 1.6 behandelt.

Die Befehle f) und g) nennt man *bedingte Sprungbefehle*. Sie werden nur dann ausgeführt, wenn die betreffende Bedingung erfüllt ist, andernfalls haben diese Befehle keinerlei Auswirkungen. Im Regelfall werden die Befehle von der Maschine in der Reihenfolge ausgeführt, in der sie gespeichert sind. Von dieser Reihenfolge wird nur abgegangen, wenn ein entsprechender (unbedingter oder bedingter) Sprungbefehl erteilt wird und zur Ausführung gelangt.

Die Arbeit der Maschine erfolgt in sogenannten *Takten*, wobei in jedem Takt ein einziger Befehl ausgeführt wird. Zu Beginn jedes Taktes kommt aus einer Speicherzelle die in ihr enthaltene Zahl (die also einen Befehl darstellt) in das Steuerwerk. Wenn der Befehl dort ist, werden die entsprechenden Verbindungen hergestellt, und dadurch wird die Ausführung der nächsten Operation des Prozesses sichergestellt.

Danach läuft der folgende Befehl in das Steuerwerk ein, und die Maschine führt die entsprechende Operation aus usw., bis der Befehl zum Stoppen der Maschine kommt.

Die technische Realisierung eines solchen Steuerwerks bietet keinerlei prinzipielle Schwierigkeiten. Es wird von diesem Werk nicht mehr gefordert, als was auch jede Selbstwählanlage zu leisten hat, in der die Nummer mittels eines elektrischen Signals gewählt wird.

Steuerwerk, Speicher und Rechenwerk sind die drei wichtigsten Grundbausteine eines Rechenautomaten. Zwar gibt es noch eine Reihe weiterer wichtiger Baugruppen, insbesondere für die Eingabe in die Maschine, für die Ausgabe der in ihr erarbeiteten Resultate usw.; diese Baugruppen haben jedoch auf die Arbeitsprinzipien und die logischen und mathematischen Möglichkeiten einer Maschine keinen Einfluß. Daher brauchen wir bei den folgenden Betrachtungen nicht auf sie einzugehen. Wir wollen also stets annehmen, daß sich die einzugebenden und die auszugebenden Informationen bereits im Speicher befinden und auch dort verbleiben.

1.6. Programme (Maschinenalgorithmen)

Im vorliegenden Abschnitt wollen wir einige Beispiele für Programme für elektronische Rechenautomaten mit Dreiadreßsystem betrachten. Diese Programme werden aus früher untersuchten Algorithmen abgeleitet, wobei berücksichtigt wird, daß nicht mehr ein Mensch, sondern ein Rechenautomat, der die im vorigen Paragraphen beschriebenen Befehle ausführen kann, nach diesem Algorithmus arbeitet. Die Analyse der behandelten Beispiele klärt den realen Sinn der Redeweise *„Die Maschine wird von dem ihr eingegebenen Programm gesteuert"*. Es wird dargelegt, auf welche Weise die Reihenfolge der ins Steuerwerk einlaufenden Befehle reguliert wird und wie man mit einer relativ kleinen Anzahl von Befehlen recht umfangreiche Rechnungen ausführen lassen kann, auch wenn beispielsweise sehr viele Operationen notwendig sind, um irgendeine Variante des Aufgabentyps zu lösen.

Im folgenden wollen wir annehmen, daß die arithmetischen Operationen Addition, Subtraktion, Multiplikation und Division in den Befehlen durch die Nummern 1, 2, 3, 4 bezeichnet werden (vgl. auch Abschnitt 1.5).

1.6.1. Ein Programm zur Lösung eines linearen Gleichungssystems

Beispiel 1. Es ist die Auflösung des Gleichungssystems

$$ax + by = c,$$
$$dx + ey = f$$

zu programmieren. Um etwas Bestimmtes vor Augen zu haben, nehmen wir an,

die Koeffizienten a, b, c, d, e, f seien der Reihe nach in den Speicherzellen mit den Adressen von Nummer 51 an untergebracht.

Adresse	Inhalt	Adresse	Inhalt
51	a,	54	d,
52	b,	55	e,
53	c,	56	f.

Ferner wollen wir für die Zwischen- und Endresultate der Rechnung die Zellen 31 bis 50 benutzen. Wie aus den Formeln

$$x = \frac{ce - fb}{ae - bd}, \quad y = \frac{af - cd}{ae - bd}$$

zu ersehen ist, muß man, um das Ergebnis zu erhalten, sechs Multiplikationen, drei Subtraktionen und zwei Divisionen ausführen. (Der Nenner ist für beide Brüche derselbe. Er wird natürlich als von Null verschieden vorausgesetzt [*Anm. d. Übers.*].) Dementsprechend besteht das Programm aus 12 Befehlen, die in den Zellen 1 bis 12 stehen:

Adresse	Inhalt (Befehl)	Adresse	Inhalt (Befehl)
1	3 53 55 31,	7	2 31 32 37,
2	3 56 52 32,	8	2 33 34 38,
3	3 51 56 33,	9	2 35 36 39,
4	3 53 54 34,	10	4 37 39 40,
5	3 51 55 35,	11	4 38 39 41,
6	3 52 54 36,	12	0 00 00 00.

Die Befehle laufen in der Reihenfolge wachsender Adressen in das Steuerwerk und werden in dieser Reihenfolge ausgeführt. Nachdem der letzte Befehl ausgeführt ist, sind in den Zellen 31 bis 41 die folgenden Zahlen gespeichert:

Adresse	Inhalt	Adresse	Inhalt
31	ce,	38	$af - cd$,
32	fb,	39	$ae - bd$,
33	af,	40	$\frac{ce - fb}{ae - bd}$,
34	cd,		
35	ae,	41	$\frac{af - cd}{ae - bd}$.
36	bd,		
37	$ce - fb$,		

Das sind die Zwischenergebnisse und das Endresultat (in den Zellen 40 und 41) der Rechnung.

1.6.2. Zyklen

Beispiel 2. Gesucht seien die Lösungen von n vorgegebenen Gleichungssystemen

$$\begin{aligned} a_i x + b_i y &= c_i, \\ d_i x + e_i y &= f_i, \end{aligned} \qquad (i = 1,2, \ldots, n).$$

Der lösende Algorithmus ist eine n-fache Wiederholung des Algorithmus zur Auflösung eines einzelnen Systems dieser Art. Man könnte leicht ein entsprechendes Programm für die Maschine aus dem oben angegebenen Programm zusammenstellen. Die $6n$ Koeffizienten müßten in $6n$ Zellen untergebracht werden, und das Programm bestünde aus $11n + 1$ Befehlen. Der erste Zyklus von 11 Befehlen würde die Lösung des ersten Systems ermitteln, der zweite Zyklus von 11 Befehlen die Lösung des zweiten Systems usw., insgesamt n mal. Der $(11n + 1)$-te Befehl wäre schließlich der Stopbefehl.

Eine derartige beträchtliche Vergrößerung des Programmaufwandes ist jedoch denkbar unzweckmäßig. Man kann sie umgehen. Wir brauchen nur daran zu denken, daß jeder folgende Zyklus von 11 Befehlen aus dem vorhergehenden dadurch erhalten werden kann, daß man die in den Befehlen enthaltenen Adressen ändert. Wenn nämlich die $6n$ Koeffizienten in aufeinanderfolgenden Zellen beispielsweise mit der Adresse 51 beginnend angeordnet sind, so brauchen in den ersten sechs Befehlen nur die Adressen der Faktoren um jeweils sechs vergrößert zu werden. Auf diese Weise erhält man die Befehle für den folgenden Zyklus. Damit die Lösungen der einzelnen Systeme, die jeweils zwei Zellen beanspruchen, ebenfalls aufeinanderfolgen, hat man jede der letzten Adressen im zehnten und elften Befehl um 2 zu erhöhen. Derartige Adressenänderungen können mittels sogenannter *Adressenänderungsbefehle* durchgeführt werden. In unserem Fall benötigen wir acht dieser Befehle. Zu diesem Zweck setzen wir in die Zellen 25 und 26 folgende Parameter (Konstanten)

Adresse	Inhalt
25	0 06 06 00,
26	0 00 00 02.

In die Zellen 12 bis 18 kommen die folgenden acht Adressenänderungsbefehle

Adresse	Inhalt
12	1 01 25 01,
13	1 02 25 02,
14	1 03 25 03,
15	1 04 25 04,
16	1 05 25 05,
17	1 06 25 06,
18	1 10 26 10,
19	1 11 26 11.

Nachdem die Befehle aus den Zellen 1 bis 19 ausgeführt sind, steht in den Zellen 40 und 41, genau wie früher, die Lösung des ersten Gleichungssystems, während in den Zellen 1 bis 6 und 10 bis 11 jetzt schon die abgeänderten Befehle stehen:

Adresse	Inhalt
1	3 59 61 31,
2	3 62 58 32,
3	3 57 62 33,
4	3 59 60 34,
5	3 57 61 35,
6	3 58 60 36,
10	4 37 39 42,
11	4 38 39 43.

Werden jetzt die Befehle aus den Zellen 1 bis 19 noch einmal ausgeführt, so liefert dieser zweite Zyklus die Lösung des zweiten Gleichungssystems, welche in den Zellen 42 und 43 gespeichert wird. Außerdem sind auch bereits wieder die Befehle 1 bis 6 sowie 10 und 11 umadressiert worden, womit die Bedingungen für einen dritten Arbeitszyklus geschaffen worden sind usw.

Wie kann man aber nun erreichen, daß die Maschine diesen Zyklus von 19 Befehlen so oft ausführt, wie Gleichungssysteme vorgegeben sind, und daß dann, wenn die Lösungen aller Gleichungssysteme gefunden sind, die Maschine gestoppt wird? Zu diesem Zweck werden die Zellen 27 und 28 mit den Parametern 0 00 00 01 und 0 00 00 0n besetzt, wobei n gleich der Anzahl der zu lösenden Gleichungssysteme ist. Außerdem treten zu den vorhandenen 19 Befehlen noch die folgenden drei hinzu:

Adresse	Inhalt
20	2 28 27 28,
21	5 01 28 01,
22	0 00 00 00.

Der Befehl 20 verkleinert den Inhalt der Zelle 28 nach jedem ausgeführten Befehlszyklus (1 bis 19) um 1. Der Befehl 21 ist ein bedingter Sprung zum Befehl 1: Er wird so lange ausgeführt, wie in der Zelle 28 noch eine positive Zahl steht. Durch diesen Befehl wird also der Übergang zum folgenden Befehlszyklus (1 bis 19) sichergestellt. Steht aber in der Zelle 28 die Zahl 0, und das ist nach genau n Befehlszyklen der Fall, so wird der bedingte Sprung nicht ausgeführt; statt dessen bringt der folgende Befehl 22 die Maschine zum Stillstand.

Es dürfte klar sein, daß das Programm aus den oben angegebenen Befehlen 1 bis 22 und den Parametern in den Zellen 25 bis 28 zur Lösung der gestellten Aufgabe brauchbar ist. Die Struktur dieses Programms läßt sich übersichtlich in einem Schema darstellen:

Befehle	
1 2 3 4 5 6 7 8 9 10 11	Arithmetische Operationen
12 13 14 15 16 17 18 19	Adressen-änderungen
20	Zählung der Zyklen
21	Bedingter Sprung
22	Stop

Parameter	
25	0 06 06 00
26	0 00 00 02
27	0 00 00 01
28	0 00 00 0n

1.6.3. Ein Programm für den Euklidischen Algorithmus

Beispiel 3. Wir stellen jetzt ein Programm zum Aufsuchen des größten gemeinsamen Teilers zweier Zahlen a und b auf. In den Zellen 12 und 13 sollen die Ausgangswerte a und b und in den Zellen 14 und 15 die Zwischenresultate untergebracht werden, wobei das endgültige Ergebnis in der Zelle 15 verbleiben soll. Das folgende

Programm wurde gemäß dem in Abschnitt 1.1.1 beschriebenen Euklidischen Algorithmus aufgestellt:

Adresse	Inhalt	Erklärung
01	1 12 05 15	Transportiere die Zahl aus Zelle 12 in Zelle 15;
02	2 12 13 14	Speichere die Differenz der Zahlen aus Zelle 12 und Zelle 13 in Zelle 14;
03	5 02 14 06	Springe nach Zelle 06, wenn in Zelle 14 eine negative Zahl steht;
04	5 01 14 09	Springe nach Zelle 09, wenn in Zelle 14 eine positive Zahl steht;
05	0 00 00 00	Stop;
06	1 13 05 12	Transportiere die Zahl aus Zelle 13 in Zelle 12;
07	1 15 05 13	Transportiere die Zahl aus Zelle 15 in Zelle 13;
08	5 00 00 01	Springe unbedingt nach Zelle 01;
09	1 13 05 12	Transportiere die Zahl aus Zelle 13 in Zelle 12;
10	1 14 05 13	Transportiere die Zahl aus Zelle 14 in Zelle 13;
11	5 00 00 01	Springe unbedingt nach Zelle 01.

Nach den ersten beiden Takten stehen in den Zellen 12 bis 15 folgende Zahlen:

Adresse	Inhalt	Adresse	Inhalt
12	a,	14	$a - b$,
13	b,	15	a.

Ist $a - b = 0$ (d. h. $a = b$), so sind die bedingten Sprungbefehle ohne Auswirkungen, und der Befehl 05, der die Maschine anhält, kommt zur Ausführung. In diesem Moment befindet sich in der Zelle 15 tatsächlich das endgültige Ergebnis (vergleiche mit der Anweisung 3 aus Abschnitt 1.1.1).

Ist $a - b > 0$ (d. h. $a > b$), so folgt auf den Befehl 03 der Befehl 06, der zusammen mit dem Befehl 07 die Plätze der Zahlen a und b in den Zellen 12 und 13 vertauscht (vgl. Anweisung 4). Der nachfolgende Befehl 08 verlangt einen unbedingten Sprung nach 01, und dort beginnt der zweite Arbeitszyklus der Maschine.

Ist $a - b < 0$ (d. h. $a < b$), so wird der bedingte Sprung 03 nicht ausgeführt, und der Befehl 04 zieht den Befehl 09 nach sich, der zusammen mit dem folgenden Befehl 10 in die Zellen 12 und 13 den früheren Subtrahenden und die Differenz, also b und $a - b$ transportiert (vgl. Anweisung 4). Sodann verlangt der Befehl 11 den unbedingten Sprung zum Befehl 01. Dort beginnt dann der zweite Arbeitszyklus der Maschine.

Die aufeinanderfolgenden Arbeitszyklen der Maschine erzeugen in den Zellen 12 und 13 eine Folge von Zahlenpaaren

$$(a_1, b_1), (a_2, b_2), \ldots, (a_i, b_i), (a_{i+1}, b_{i+1}), \ldots$$

und in der Zelle 15 eine Zahlenfolge

$$a_1, a_2, a_3, \ldots, a_i, a_{i+1}, \ldots$$

bis erstmalig ein Paar gleicher Zahlen (a_k, b_k) auftritt. Dann bringt der Befehl 05 die Maschine zum Stehen, und in der Zelle 15 ist das gesuchte Resultat gespeichert.

1.6.4. Die Arbeitsweise eines Rechenautomaten

In den analysierten Beispielen treten mit hinreichender Klarheit folgende zwei grundlegende Arbeitsprinzipien der elektronischen Rechenautomaten zutage:

1. In der Regel werden die Befehle des Programms durch die Maschine in der Reihenfolge ausgeführt, in der sie in den Speicherzellen stehen. Die Maschine kann allerdings auch den Gang des Rechenprozesses in Abhängigkeit von den laufend erhaltenen Rechenergebnissen automatisch ändern. Das wird gerade durch die bedingten Sprungbefehle ermöglicht.

2. Bei einem verhältnismäßig umfangreichen Programm besteht unter Umständen die Möglichkeit, es dadurch zu verkürzen, daß die Maschine einzelne Teile des Programms oder sogar das ganze Programm mehrfach wiederholt, wobei sie selbständig Adressen ändert. Das durch Ziffern verschlüsselte Programm wird nämlich in demselben Speicher wie die gewöhnlichen Zahlen aufbewahrt. Daher ist es der Maschine möglich, auch mit den bedingten Zahlen, d. h. mit den verschlüsselten Befehlen, zu rechnen. Auf diese Weise kann die Maschine beispielsweise Adressenänderungen in den Befehlen vornehmen.

Die Maschine ist aufgrund der für sie charakteristischen Arbeitsprinzipien in der Lage, auch Aufgaben zu lösen, die keine ausgesprochenen Rechenaufgaben sind. So kann man z. B. den Algorithmus des THESEUS (das Suchen in einem Labyrinth) oder Algorithmen für Wortprobleme in gewissen assoziativen Kalkülen programmieren und die entsprechenden Prozesse in der Maschine realisieren. Dazu ist es allerdings notwendig, daß die Maschine außer den arithmetischen noch einige andere Operationen ausführen kann. Es kommen auch noch einige Steuerbefehle hinzu, die für die oben betrachteten arithmetischen Aufgaben nicht erforderlich waren. In den real arbeitenden elektronischen Rechenautomaten können diese einfachen Operationen ausgeführt werden (d. h. sind die entsprechenden Befehle vorgesehen). Damit lassen sich dann in der Tat Programme aufstellen, die Aufgaben der genannten Art lösen.

1.6.5. Andere Anwendungen von Rechenautomaten

Wir haben schon in der Einleitung bemerkt, daß nicht nur in der Mathematik, sondern auch in vielen anderen Bereichen der menschlichen Tätigkeit Prozesse auftreten, die nach streng festgelegten formalen Vorschriften (d. h. Algorithmen)

ablaufen. Diese kann man ebenfalls programmieren. Hierzu gehört z. B. das Übersetzen von einer Sprache in eine andere. Durch eine hinreichend tiefgehende Analyse und entsprechende Klassifikation der grundlegenden grammatikalischen und stilistischen Regeln und der Verfahren, wie man Wörterbücher verwendet, kann man einen recht befriedigenden Übersetzungsalgorithmus aufstellen, mit dessen Hilfe etwa wissenschaftliche und geschäftliche Texte übertragen werden können (für einige Sprachen wurden solche Algorithmen bereits entwickelt).

Wir erörtern noch kurz die Möglichkeit einer erfolgreichen Spielführung mit Hilfe von Maschinen. Zunächst kann man das Aufsuchen einer besten Strategie nach dem Algorithmus aus Abschnitt 1.2.4 im Prinzip einer Maschine übertragen. Praktisch ist das aber natürlich nur bei Spielen mit nicht zu großem Baum möglich, z. B. bei Spielen vom Typ der Streichholzspiele mit relativ kleiner Anzahl von Streichhölzern. Man kann weiter für ein spezielles Spiel, für das eine optimale Strategie (in Form eines Algorithmus) bekannt ist, diese optimale Strategie programmieren. Bei komplizierteren Spielen, für die eine solche Strategie noch nicht aufgestellt werden konnte oder für die äußerst umfangreiche Strategien vorliegen, muß man sich auf Verfahren beschränken, die auf einer speziellen Analyse und Bewertung jedes Zuges des Spiels basieren und sogenannte *Taktiken* für das Spiel liefern. So kann man beispielsweise beim Schach für die Figuren ein Bewertungssystem einführen. Hierbei muß der König sehr hoch bewertet werden, die Dame etwas weniger, der Turm noch weniger usw., einem Bauern kommt der geringste Wert zu. Außerdem müssen aber auch die Stellungen in bestimmter Weise bewertet werden (Position der Figuren auf dem Brett, ihre Beweglichkeit usw.). Die Differenz der Summe der Bewertungen für die weißen Figuren und der Summe der Bewertungen für die schwarzen Figuren charakterisiert dann (im Sinne der vorgegebenen Taktik) die materiellen und positionellen Vorteile der weißen über die schwarzen Figuren im vorliegenden Spielstadium. Der einfachste Algorithmus besteht in der Durchmusterung aller Züge, die im Moment möglich sind, und der Auswahl eines Zuges, der vom Standpunkt des vorgegebenen Bewertungssystems eine maximale Überlegenheit sichert. Besser, aber auch wesentlich komplizierter ist ein Algorithmus, der alle möglichen Kombinationen der nächsten drei oder sogar fünf Züge berücksichtigt und nach dem vorgegebenen Auswahlprinzip den optimalen Zug ermittelt.

Aus den genannten Beispielen geht klar hervor, wie vielfältig geistige Arbeit sein kann, die nach bestimmten Algorithmen ausgeführt wird und ausgeführt werden kann. In allen Fällen kann man diese Algorithmen prinzipiell auch programmieren und die entsprechenden Arbeiten von programmgesteuerten Automaten ausführen lassen. Es existieren insbesondere schon heute Programme zur Übersetzung aus einer Sprache in eine andere und zum Schachspielen, die auf modernen Rechenanlagen realisiert werden können.

2. Turing-Maschinen

2.1. Die Notwendigkeit einer Präzisierung des Algorithmenbegriffs

2.1.1. Die Existenz von Algorithmen

Aus unseren bisherigen Ausführungen ist bereits zu ersehen, wie eng die Algorithmen mit den programmgesteuerten Rechenautomaten zusammenhängen.

Offenbar läßt sich jeder Prozeß, dessen einzelne Schritte nacheinander auf einer automatisch arbeitenden Maschine verwirklicht werden können, durch einen Algorithmus beschreiben. Andererseits lassen sich alle bis jetzt bekannten Algorithmen sowie diejenigen, die man beim heutigen Stand der Wissenschaften erwarten kann, prinzipiell in programmgesteuerten Rechenautomaten realisieren.

Die letzte Behauptung erfordert allerdings noch einige Erläuterungen. Wie schon erwähnt, kann der algorithmische Prozeß zur Lösung einer Aufgabe bestimmten Typs beliebig lang sein und die Menge der Daten, die mit dem Algorithmus bearbeitet wird, alle Vorstellungen übersteigen. Andererseits hat aber der Speicher eines realen Automaten nur ein beschränktes Volumen (da erstens die Anzahl der Zellen und zweitens auch das Fassungsvermögen jeder Zelle beschränkt ist). Deshalb kann es vorkommen, daß man einen Algorithmus unter den bestehenden technischen Voraussetzungen praktisch nicht realisieren kann.

Das läßt sich schon am Beispiel des Euklidischen Algorithmus illustrieren. Die einfache Aufgabe des Bestimmens des größten gemeinsamen Teilers zweier Zahlen wird für einen Rechner praktisch undurchführbar, wenn er zur Lösung dieser Aufgabe mehr Papier und Tinte benötigt, als er sich jemals beschaffen kann. Genauso, wie es dem Rechner mit dem Euklidischen Algorithmus gehen kann, wird für eine Maschine eine gegebene konkrete Aufgabe undurchführbar, wenn zu ihrer Lösung mehr Speicherraum erforderlich ist, als der Maschine zur Verfügung steht.

In solchen Fällen muß man den algorithmischen Prozeß, wie schon betont, als *potentiell durchführbaren Prozeß* ansehen, der stets nach endlich vielen Schritten (wobei die Anzahl dieser Schritte sehr groß sein kann) zum gesuchten Ergebnis führt. Wenn von der Möglichkeit die Rede ist, einen Algorithmus in einer Maschine zu verwirklichen, denke man analog stets an die potentielle Möglichkeit, das Speichervolumen unbegrenzt zu vergrößern.

Der erwähnte Zusammenhang zwischen dem Begriff „Algorithmus" und dem Begriff „programmgesteuerter Rechenautomat mit potentiell unbeschränktem Speichervolumen" läßt ihre wesentlichen Züge deutlich hervortreten. Jede Präzisierung des einen Begriffs ist gleichzeitig eine Präzisierung des anderen.

Trotz aller Betonung der Wichtigkeit dieser Begriffe ist keiner bisher von uns exakt definiert worden. Eine exakte mathematische Definition des Begriffs des Algorithmus (und gleichzeitig damit eine genaue Definition des artverwandten Begriffs des programmgesteuerten Rechenautomaten) wurde erst in den dreißiger Jahren unseres Jahrhunderts von den Wissenschaftlern erarbeitet. Warum haben die Mathematiker vieler Jahrhunderte, ohne besonders beunruhigt zu sein, sich mit einem verschwommenen Begriff des Algorithmus zufriedengegeben? Warum entstand vor relativ kurzer Zeit ein dringendes Bedürfnis nach einer streng mathematischen Definition dieses Begriffs, damit er als Gegenstand strenger mathematischer Untersuchungen dienen könne?

Bis vor kurzem trat der Begriff des Algorithmus in der Mathematik nur im Zusammenhang mit der Konstruktion (Entwicklung) konkreter Algorithmen auf; die Aussage, daß für Aufgaben eines bestimmten Typs ein Algorithmus existiere, war praktisch stets mit einer ausführlichen Beschreibung eines solchen Algorithmus verbunden. Unter diesen Umständen brauchte jemand, dem ein System formaler Regeln mitgeteilt wurde, nur Daten einzusetzen und sich bei der Anwendung der Regeln davon zu überzeugen, daß er automatisch auf das Ergebnis geführt wurde. Daher tauchte das Problem einer streng mathematischen Definition des Begriffs „Algorithmus" gar nicht auf, und man konnte sich mit dem verschwommenen, von jedem Mathematiker mit der richtigen Vorstellung verknüpften Begriff des Algorithmus begnügen.

Im Zuge der neueren Entwicklung häuften sich in der Mathematik jedoch Tatsachen an, die diese Situation von Grund auf änderten. Die Ursachen dafür lagen in dem natürlichen Streben der Mathematiker begründet, immer mächtigere Algorithmen zu schaffen, die nach Möglichkeit umfangreichere Aufgabenklassen (Aufgaben allgemeineren Typs) lösen können. Der Betrachtung solcher Tatsachen wollen wir uns nun zuwenden.

Wir erinnern an den Algorithmus zur Berechnung von Quadratwurzeln, der in allen Schulbüchern beschrieben wird. Man kann sich ein allgemeineres Ziel stellen:

Es ist ein Algorithmus zu konstruieren, mit dem man die Wurzel beliebigen Grades aus jeder vorgegebenen Zahl ziehen kann. Es ist natürlich zu erwarten, daß ein solcher Algorithmus etwas schwieriger zu konstruieren sein wird. Die Perspektiven, die sich aus dem Besitz eines solchen Algorithmus ergeben, sind schon sehr verlockend. Man kann jedoch noch weiter gehen. Das Ziehen der Wurzel n-ten Grades aus einer Zahl a ist gleichbedeutend mit dem Auflösen der Gleichung $x^n - a = 0$ (dem Bestimmen einer Wurzel dieser Gleichung). Jetzt kann man sofort eine noch allgemeinere Aufgabe formulieren:

Es ist ein Algorithmus zu konstruieren, mit dessen Hilfe man für jede Gleichung der

Form

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \tag{1}$$

(n eine beliebige natürliche Zahl) *alle Wurzeln bestimmen kann*[1]).

Die Konstruktion eines solchen Algorithmus ist natürlich noch schwieriger. Dem Wesen nach gehört diese Aufgabe in ein Teilgebiet der höheren Algebra, in die sogenannte Algebra der Polynome; sie verlangt die Konstruktion und die Begründung dieses Algorithmus, der natürlich von großer praktischer Bedeutung ist.

2.1.2. Das Entscheidungsproblem

Die ausgeführten Beispiele charakterisieren bereits in ausreichendem Maße das natürliche Streben der Mathematiker nach immer mächtigeren Algorithmen, mit denen man immer umfangreichere Klassen von Aufgaben (Aufgaben immer allgemeineren Typs) lösen kann. Selbstverständlich ist das Lösen von Gleichungen der Form (1) noch lange nicht die Grenze, die man überhaupt erreichen kann. Wenn nun ein Mathematiker in seinem Streben nach Erkenntnis die Aufgabenklasse, für die ein universeller lösender Algorithmus gesucht wird, immer weiter ausdehnt, so gelangt er unweigerlich zu der folgenden Problemstellung:

Es ist ein Algorithmus zu konstruieren, der es gestattet, jede beliebige mathematische Aufgabe zu lösen.

Diese Problemstellung ist natürlich so allgemein, daß sie wahrlich als eine verwegene Herausforderung der gesamten Mathematik empfunden wird; außerdem könnte man an ihrer Formulierung bemängeln, daß keinesfalls klar ist, was eigentlich „jede beliebige mathematische Aufgabe" bedeutet. Übrigens braucht man derartige Probleme wegen ihrer großen Anziehungskraft und ihres Reizes nicht besonders zu propagieren.

Das Problem hat seine Geschichte. Schon der große deutsche Mathematiker und Philosoph G. W. Leibniz (1646–1716) träumte von der Entwicklung einer allumfassenden Methode, die eine effektive Lösung jeder Aufgabe gestatten würde. Obwohl Leibniz einen solchen allumfassenden Algorithmus nicht finden konnte, nahm er an, daß man einmal einen finden würde und daß dann jede Meinungsverschiedenheit zwischen Mathematikern automatisch mit Bleistift und Papier nach diesem Universal-Algorithmus entschieden werden könne.

In der weiteren Entwicklung wurde diese Fragestellung zu einem der wichtigsten Probleme der mathematischen Logik präzisiert, dem sogenannten *Entscheidungsproblem*. Da wir hier nicht die Möglichkeit haben, eine erschöpfende und exakte Darstellung dieses Problemkreises zu geben, beschränken wir uns auf die Darlegung seiner wichtigsten Züge.

[1]) Genauer gesagt: Zu jeder natürlichen Zahl k soll man Näherungswerte für die Wurzeln angeben können, die sich von dem genauen Wert um weniger als 10^{-k} unterscheiden.

Bekanntlich besteht die axiomatische Methode in der Mathematik darin, daß alle Aussagen (Sätze) einer vorliegenden Theorie durch formal-logische Schlußfolgerungen aus gewissen Aussagen (Axiomen), die man in dieser Theorie ohne Beweis als gültig annimmt, abgeleitet werden. Die Geometrie war die erste mathematische Disziplin, in der eine solche Axiomatisierung verwirklicht werden konnte. Heute sind bereits fast alle mathematischen Theorien axiomatisiert. In der mathematischen Logik wurde nun eine spezielle Formelsprache entwickelt, mit der man jede Behauptung einer mathematischen Theorie in Gestalt einer wohlbestimmten *Formel* schreiben kann. In der Terminologie des Abschnitts 1.4.1 können wir sagen, daß jede solche Formel ein Wort in einem speziellen Alphabet ist, das neben den üblichen mathematischen Zeichen (Ziffern, Buchstaben, Klammern, Formelzeichen usw.) auch Zeichen für die logischen Operationen (Negation (Verknüpfung mit „nicht"), Konjunktion (Verknüpfung mit „und"), Alternative (Verknüpfung mit „oder"), Implikation (Verknüpfung mit „wenn — so"), Generalisierung (Verknüpfung mit „für alle") und Partikularisierung (Verknüpfung mit „es gibt ein")) enthält. Die wesentliche Analogie zu den assoziativen Kalkülen besteht jedoch darin, daß in solchen logischen Kalkülen der Prozeß der *logischen Ableitung* einer Aussage S aus einer Aussage R zu einer formalen Transformation der entsprechenden Wörter nach bestimmten *logischen Substitutionen* wird. Ein logischer Kalkül kann also durch ein bestimmtes System von zugelassenen Wortsubstitutionen (den *Schlußregeln dieses Kalküls*) charakterisiert werden. Mit diesen zugelassenen Substitutionen lassen sich alle formal-logischen Ableitungen aufbauen. Ein Beispiel für eine zugelassene Substitution ist die Streichung zweier hintereinanderstehender Negationszeichen (etwa der Übergang von „nicht nicht schön" zu „schön"), vergleichbar der Substitution $aa \to A$ im assoziativen Kalkül des Abschnitts 1.4.4.

Die Frage, ob eine Behauptung S aus einer Aussage R innerhalb des logischen Kalküls abgeleitet werden kann, ist die Frage nach der Existenz einer Ableitungskette, die von dem Wort, welches die Aussage R repräsentiert, zu dem Wort führt, welches die Behauptung S darstellt.[1]) Das Entscheidungsproblem läßt sich jetzt so formulieren:

Von je zwei Wörtern (Formeln) R und S eines logischen Kalküls soll entschieden werden, ob eine Ableitungskette existiert, die von R nach S führt, oder nicht.

Unter einer Lösung dieses Problems ist die Angabe eines Algorithmus zu verstehen, der auf alle Fragen dieses Typs (d. h. für beliebiges R und S) die Antwort liefert.

Ein solcher Algorithmus wäre offenbar eine allgemeine Methode zur automatischen Lösung der verschiedenartigsten Probleme aus beliebigen axiomatisch aufgebauten mathematischen Theorien. Denn die Gültigkeit einer Aussage S (d. h. der Formulierung eines Satzes) in einer axiomatischen Theorie T bedeutet ihre formal-logische Beweisbarkeit aus dem Axiomensystem der Theorie T, das in vielen Fällen durch eine

[1]) Die logischen Kalküle sind wichtige Beispiele für assoziative Kalküle mit gerichteten Substitutionen. [*Anm. d. Übers.*]

einzelne Aussage R (nämlich die Konjunktion der Axiome) ersetzt werden kann. Unter Verwendung eines Entscheidungsalgorithmus könnte man also feststellen, ob die Aussage S in der betrachteten Theorie T gültig ist oder nicht. Und im Fall einer positiven Antwort würde der Algorithmus vielleicht sogar eine Ableitungskette liefern, die einem Beweis der Aussage S entspräche. Der Entscheidungsalgorithmus wäre also eine einheitliche, effektive Methode, mit der man alle in axiomatischen Theorien formulierbaren und bis heute ungelösten mathematischen Probleme entscheiden könnte. Das erklärt nicht nur die große Anziehungskraft, die der Versuch der Konstruktion eines solchen „allumfassenden" Algorithmus ausstrahlte, sondern auch die Schwierigkeiten bei der Realisierung dieser Konstruktion.

Trotz langer und hartnäckiger Bemühungen vieler berühmter Spezialisten waren die mit der Konstruktion eines allgemeinen Entscheidungsalgorithmus verbundenen Schwierigkeiten unüberwindlich. Und ähnliche Schwierigkeiten traten schon bei mathematischen Problemen sehr speziellen Typs auf. Zu diesen Problemen gehörten z. B. das 10. Hilbertsche Problem (vgl. Abschnitt 1.1.3) und eine Reihe weiterer Probleme, auf die wir später zu sprechen kommen.

Im Resultat der zahlreichen vergeblichen Bemühungen kam man zu der Vermutung, daß hier Schwierigkeiten prinzipieller Natur vorliegen müssen und vielleicht Klassen von Aufgaben existieren, für die man keinen Lösungsalgorithmus konstruieren kann.

2.1.3. Die Problematik einer exakten Definition des Algorithmenbegriffs

Die Behauptung der algorithmischen Unlösbarkeit einer bestimmten Aufgabenklasse, d. h., der Unmöglichkeit, für sie einen Lösungsalgorithmus anzugeben, ist nicht einfach eine Anerkennung der Tatsache, daß uns kein solcher Algorithmus bekannt ist, oder daß ihn noch keiner gefunden hat. Diese Behauptung ist gleichzeitig eine Prognose. Sie besagt, daß ein solcher Algorithmus auch niemals gefunden werden kann (mit anderen Worten, daß es ihn nicht gibt). Eine derartige Behauptung bedarf natürlich eines strengen mathematischen Beweises. Über einen Beweis nachzudenken, hat aber erst dann einen Sinn, wenn man über eine exakte Definition des Begriffs des Algorithmus verfügt. Ohne diese ist es natürlich völlig schleierhaft, wie man die Nichtexistenz überhaupt beweisen sollte.

Wir möchten daran erinnern, daß es auch in früheren Etappen der Entwicklung der Mathematik immer wieder Probleme gegeben hat, die sich nicht lösen ließen und von denen man später zeigen konnte, daß sie unlösbar waren. Historisch markante Beispiele sind unter anderem das Problem der Dreiteilung eines beliebigen Winkels und das Problem der Auflösung einer beliebigen Gleichung durch Radikale.

Aus der Schule ist bekannt, daß man jeden Winkel mit Zirkel und Lineal halbieren kann. Bereits im alten Griechenland beschäftigte man sich mit der analogen Aufgabe, einen beliebigen Winkel mit Zirkel und Lineal in drei gleichgroße Teile zu zerlegen. Die Lösung dieser Aufgabe wollte nicht gelingen, und inzwischen ist bewiesen worden,

daß sie gar nicht gelingen konnte, daß es nämlich unmöglich ist, einen beliebigen Winkel mit Zirkel und Lineal in drei gleichgroße Teile zu zerlegen (es gibt natürlich spezielle Winkel, für die das kein Problem ist, und es gibt auch zahlreiche Näherungskonstruktionen; d. h., es geht um beliebige Winkel und gleichgroße Teile).

Aus der Schule ist weiter die Formel bekannt, nach der man die Wurzeln einer quadratischen Gleichung durch deren Koeffizienten ausdrücken kann. In dieser Formel tritt außer den arithmetischen Operationen das Zeichen für die Quadratwurzel auf. Bereits im 16. Jahrhundert waren analoge Formeln für Gleichungen dritten und vierten Grades bekannt, durch die die Wurzeln einer solchen Gleichung in Form von Wurzelausdrücken in den Koeffizienten (sogenannten *Radikalen*) dargestellt werden, die natürlich komplizierter als bei quadratischen Gleichungen sind, nämlich auch eingeschachtelte Radikale enthalten. Dagegen blieb die Suche nach ähnlichen Formeln für Gleichungen höheren als vierten Grades bis zum Anfang des 19. Jahrhunderts erfolglos und wurde dann eingestellt, als nämlich der Beweis des folgenden bemerkenswerten Resultats gelang:

Für kein $n \geqq 5$ existiert eine Formel, die die Wurzeln einer beliebigen Gleichung n-ten Grades durch Radikale in den Koeffizienten der Gleichung ausdrückt.

In beiden Fällen war der Unmöglichkeitsbeweis erst zu erbringen, nachdem man eine präzise Definition hatte, die auf folgende Frage eine Antwort gab: „Was bedeutet Konstruktion mit Zirkel und Lineal?" bzw. „Was bedeutet Auflösung einer Gleichung durch Radikale?" Beide Definitionen präzisieren übrigens den Sinn gewisser spezieller Algorithmen, nämlich eines Algorithmus zur Lösung von Gleichungen durch Radikale (statt eines allgemeinen Algorithmus zur Lösung algebraischer Gleichungen) bzw. eines Algorithmus zur Dreiteilung von Winkeln mittels Zirkel und Lineal (statt eines beliebigen Algorithmus zur Winkeldreiteilung).

Eine exakte Definition für den allgemeinen Begriff des Algorithmus gab es bis Anfang der dreißiger Jahre unseres Jahrhunderts nicht. Die Erarbeitung einer solchen Definition war eine der wichtigsten Aufgaben der modernen Mathematik. Dabei muß hervorgehoben werden, daß es bei dieser Definition (wie übrigens auch bei vielen anderen mathematischen Definitionen) um mehr als eine Vereinbarung der Mathematiker darüber ging, was man unter dem Terminus „Algorithmus" hinfort verstehen will. Bis zu ihrer endgültigen Formulierung waren große Schwierigkeiten zu überwinden. Sie bestanden vor allem darin, daß die vorgeschlagene Definition das Wesen eines Begriffs richtig wiedergeben mußte, der — wenn auch verschwommen — durchaus schon vorhanden war und den wir auch in diesem Buch schon an vielen Beispielen demonstriert haben. Dazu wurden Anfang der dreißiger Jahre in vielen Untersuchungen die Mittel gesichtet, die bei der Konstruktion von Algorithmen bislang benutzt wurden. Auf dieser Grundlage wurde sodann eine Definition des Begriffs „Algorithmus" gefunden, die nicht nur aus der Sicht der formalen Genauigkeit vollkommen war, sondern sich mit dem Wesen des zu definierenden Begriffs deckte. Interessant ist, daß die verschiedenen Forscher von unterschiedlichen technischen und logischen Gesichtspunkten aus an das Problem herangingen, so daß

zunächst sehr unterschiedliche Definitionen entstanden. Recht schnell zeigte sich jedoch, daß alle diese Definitionen äquivalent waren, d. h. denselben präzisen Begriff erfaßten, eben den heutigen exakten Begriff des Algorithmus. Die Tatsache, daß alle Versuche einer Präzisierung des Algorithmenbegriffs, ungeachtet ihrer Vielschichtigkeit, bislang stets zum gleichen Resultat führten und vermutlich auch immer führen werden, hat erkenntnistheoretische Bedeutung. Sie bezeugt, daß die erarbeitete Definition gut das Wesen der Sache erfaßt.

Aus der Sicht der modernen Rechentechnik verdient diejenige Definition besonderes Interesse, bei der das Wesen des Algorithmenbegriffs durch Analyse der Prozesse offenbar wird, die in einem Rechenautomaten ablaufen. Bei ihnen wird die Arbeitsweise realer Maschinen durch ein gewisses Standardschema idealisiert, das seiner logischen Struktur nach hinreichend einfach und andererseits so präzise ist, daß es Gegenstand mathematischer Untersuchungen sein kann. Ein erstes derartiges Schema wurde 1936 von dem englischen Mathematiker A. M. TURING vorgeschlagen, d. h. zu einer Zeit, als es noch keine programmgesteuerten Rechenautomaten gab. Bei seinem sehr allgemeinen und trotzdem einfachen Konzept ging TURING von der Idee aus, die Arbeit eines nach gewissen strengen Vorschriften operierenden Rechners durch eine Maschine zu modellieren. Wir werden im folgenden im wesentlichen den Ideen von TURING folgen, uns aber von den Analogien zu realen elektronischen Rechenautomaten leiten lassen.

2.2. Die Turing-Maschine

Die Turing-Maschinen sind gegenüber dem in den Abschnitten 1.5 und 1.6 beschriebenen elektronischen Rechenautomaten durch folgende Besonderheiten gekennzeichnet:

1. In einer Turing-Maschine ist die Zerlegung der algorithmischen Prozesse in einfache, elementare Operationen in gewissem Sinne bis an die Grenze des Möglichen getrieben. So wird beispielsweise die Addition, die in elektronischen Rechenautomaten als einheitliche Elementaroperation behandelt wird, in eine Kette noch einfacherer Operationen zerlegt. Das verlängert natürlich die Prozesse in einer Turing-Maschine wesentlich. Dadurch wird jedoch die logische Struktur stark vereinfacht und in eine für theoretische Untersuchungen geeignete Standardform gebracht.

2. In einer Turing-Maschine hat man sich einen Teil des Speichers[1]) als ein nach beiden Seiten unbeschränktes Band vorzustellen, welches in einzelne Zellen unterteilt ist. Offenbar kann es in einer real existierenden Maschine keinen unendlichen Speicher (kein unendliches Band) geben. In diesem Sinne ist eine Turing-Maschine nur ein idealisiertes Schema, bei dem das Speichervolumen potentiell unbeschränkt

[1]) Den sogenannten *äußeren Speicher*.

ist. Diese Idealisierung wird durch den schon früher erwähnten Zusammenhang zwischen dem Begriff des Algorithmus und dem Begriff der Maschine mit potentiell unbeschränktem Speicher gerechtfertigt.

2.2.1. Definition der Turing-Maschine

Wir kommen jetzt zur Beschreibung einer Turing-Maschine.

1. Die Maschine verfügt über endlich viele Zeichen (Symbole)

$$s_1, s_2, \ldots, s_k,$$

die das sogenannte *äußere Alphabet* bilden. In diesem Alphabet werden die Eingabedaten sowie die in der Maschine erarbeiteten Daten geschrieben. Unter den Symbolen befinde sich ein *Leerzeichen*, und zwar sei es im allgemeinen Fall durch s_1 und in speziellen Beispielen durch Λ angedeutet. Wird das Leerzeichen in eine Zelle des Bandes (Speichers) eingegeben, so wird das vorher in der Zelle vorhandene Zeichen *gelöscht*, und die Zelle wird leer. Umgekehrt nehmen wir von einer leeren Zelle an, daß in ihr das Leerzeichen steht.

In keinem Stadium der Arbeit der Maschine kann sich in einer Zelle des Bandes mehr als ein Zeichen befinden. Alle Daten, die auf dem Band gespeichert sind, werden mittels endlich vieler Zeichen des äußeren Alphabets dargestellt, die vom Leerzeichen verschieden und einzeln in gewissen Zellen des Bandes untergebracht sind. Zu Arbeitsbeginn befinden sich auf dem Band die *Anfangsdaten* (die *Anfangsinformation*).

Die Maschine arbeitet in aufeinanderfolgenden *Takten*. Jedem Takt entspricht eine Transformation der Anfangsinformation in eine gewisse Zwischeninformation (die Gesamtheit der am Ende eines Taktes auf dem Bande befindlichen Zeichen bildet die entsprechende Zwischeninformation). Als Anfangsinformation kann sich auf dem Band ein beliebiges endliches System von nichtleeren Zeichen des äußeren Alphabets befinden, die in beliebiger Weise auf die Zellen verteilt sind (z. B. ein beliebiges Wort dieses Alphabets). Je nach der Anfangsinformation A sind zwei Fälle möglich:

a) Nach endlich vielen Takten hält die Maschine an, weil sie ein Stopsignal erhielt. Bis zu diesem Zeitpunkt hat sich auf dem Band eine gewisse Information B herausgebildet. Wir wollen dann sagen, daß *die Maschine auf die Anfangsinformation A anwendbar ist und diese in die Endinformation B umformt.*

b) Die Maschine hält nicht an, ein Stopsignal tritt nicht auf. Wir sagen dann, daß *die Maschine auf die Anfangsinformation A nicht anwendbar ist.*

Man sagt, daß *eine Maschine eine gewisse Klasse von Aufgaben lösen kann,* wenn sie auf jede Information anwendbar ist, die in einem bestimmten Kode die Bedingungen für eine einzelne Aufgabe dieses Typs wiedergibt und diese in eine Information umformt, die in demselben Kode die Lösung dieser Aufgabe darstellt.

2. Das Dreiadreßsystem, welches in einigen programmgesteuerten Rechenautomaten angewandt wird, verlangt das Vorhandensein elementarer Operationen, durch die der Inhalt dreier Speicherzellen in Beziehung gesetzt wird. In neueren programmgesteuerten Rechenautomaten wird meistens das sogenannte Einadreßsystem benutzt. Hier ist an jedem Arbeitstakt nur eine Speicherzelle beteiligt. (Diese wollen wir als die *aufgerufene Zelle* bezeichnen.) So kann z. B. die Addition der Zahlen aus den Zellen β und γ mit anschließendem Transport des Resultats in die Zelle δ des Dreiadreßsystems durch drei aufeinanderfolgende Befehle im Einadreßsystem ersetzt werden:

a) Transport der Zahl aus der Zelle β in den Addiator,

b) Transport der Zahl aus der Zelle γ in den Addiator,

c) Transport des Resultats in die Zelle δ.

In einer Turing-Maschine ist das System der elementaren Operationen und damit auch das Einadreßsystem noch weiter vereinfacht: In jedem Einzeltakt wird durch den Befehl nur das Zeichen s_i, welches sich in der aufgerufenen Zelle befindet, durch ein Zeichen s_j ersetzt ($j = i$ bedeutet, daß der Inhalt der aufgerufenen Zelle ungeändert bleibt; $j = 1$ bedeutet, daß das in der aufgerufenen Zelle gespeicherte Zeichen gelöscht wird). Eine weitere Vereinfachung besteht darin, daß von einem Takt zum nächsten Takt die Adresse der aufgerufenen Zelle sich höchstens um 1 ändert, d. h., die im nächsten Takt aufgerufene Zelle ist entweder der unmittelbare linke oder rechte Nachbar der zuletzt aufgerufenen Zelle oder aber sie selbst.

Dieser Vereinfachung liegt die Idee zugrunde, daß die im Ablauf eines Rechenprozesses eventuell benötigten Inhalte entfernter liegender Zellen von der Maschine durch schrittweises Abtasten jeweils benachbarter Zellen aufgesucht werden. Dadurch wird natürlich der Rechenprozeß erheblich verlängert. Es bringt aber folgende Annehmlichkeit mit sich: In den Befehlen eines Programms benötigt man anstelle beliebiger Adressen zur Kennzeichnung der aufgerufenen Zelle nur drei (relative) Standardadressen, die wir mit R, L, M bezeichnen wollen, wobei diese Symbole folgende Bedeutung haben:

R — die rechts benachbarte Zelle ist aufgerufen,
L — die links benachbarte Zelle ist aufgerufen,
M — es ist die gleiche Zelle wie im vorangehenden Takt aufgerufen.

3. Zur Verarbeitung der im Speicher aufbewahrten numerischen Informationen besitzt ein programmgesteuerter Rechenautomat, wie in den Abschnitten 1.5 und 1.6 beschrieben, ein Rechenwerk $\mathfrak{L}$, welches endlich viele Zustände annehmen kann, die der Addition, der Subtraktion usw. entsprechen. Damit im Rechenwerk eine bestimmte Operation ausgeführt wird, müssen über bestimmte Kanäle nicht nur die Zahlen, an denen sie auszuführen ist, sondern auch ein Signal kommen, welches das Rechenwerk auf die betreffende Operation einstellt, d. h. in den entsprechenden Zustand überführt (vgl. Abb. 10b). Eine Turing-Maschine verarbeitet die Informa-

tionen im sogenannten *logischen Block*, der ebenfalls endlich viele *Zustände* annehmen kann, die mit

$$q_1, q_2, \ldots, q_m$$

bezeichnet seien. Der Block besitzt zwei Eingänge. Durch den einen läuft in jedem Arbeitsstadium der Maschine (in jedem Takt) das Zeichen s_i aus der aufgerufenen Zelle ein, durch den anderen das Zeichen q_n des Zustandes, der für den Block im gegebenen Takt vorgeschrieben ist. Das von ihm „erarbeitete" Zeichen s_j, welches eine eindeutige Funktion der Signale s_i und q_n ist, gelangt durch den Ausgang des Blocks in die aufgerufene Zelle. Die Befehle, die die Arbeit der Maschine in jedem Einzeltakt bestimmten, haben die Form

$$Rq_l, Lq_l, Mq_l \qquad (l = 1, 2, \ldots, m),$$

wobei das erste Zeichen die aufzurufende Zelle bezeichnet und das zweite den neuen Zustand des logischen Blocks angibt. Die Zeichen $R, L, M, q_1, \ldots, q_m$ bilden das sogenannte *innere Alphabet* der Maschine.

Der logische Block einer Turing-Maschine weist nun noch eine spezifische Besonderheit auf. In ihm werden in jedem Takt auch die Befehle erarbeitet, die zu Beginn des folgenden Taktes in das Steuerwerk gegeben werden. Daher hat der logische Block außer dem Ausgangskanal für das Zeichen s_j noch zwei Ausgangskanäle für die Zeichen des folgenden Befehls. Abb. 11 zeigt eine schematische Darstellung.

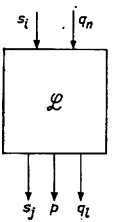


Abb. 11

Hierbei ist wesentlich, daß das *Ausgangstripel*[1]) s_jPq_l ausschließlich davon abhängt, welches *Eingangspaar* s_iq_n in diesem Takt in den Block einlief. Das bedeutet, daß der logische Block eine Funktion realisiert, die jedem Paar s_iq_n von Zeichen (es gibt insgesamt km derartige Paare) ein Tripel s_jPq_l von Zeichen zuordnet. Diese Funktion, die wir die *logische Funktion* der Maschine nennen wollen, läßt sich bequem in einer Tabelle darstellen, deren Spalten durch die Zeichen der Zustände und deren Zeilen durch die Zeichen des äußeren Alphabets markiert sind. In jedem Feld dieser Tabelle

[1]) Unter P wird hier eines der Zeichen R, L, M verstanden.

steht das entsprechende Ausgangstripel. Diese Tabelle nennen wir das *Funktionsschema* der Maschine; Abb. 12 zeigt ein Beispiel.

Aus unserer Beschreibung ergibt sich, daß die *Arbeit* einer Turing-Maschine völlig bestimmt ist durch ihre logische Funktion, die durch den logischen Block realisiert wird. Mit anderen Worten, zwei Turing-Maschinen mit gleichem Funktionsschema sind nicht voneinander zu unterscheiden (solange wir uns nur dafür interessieren, wie sie arbeiten). Andererseits kann man die Struktur einer Maschine in Form eines

	q_1	q_2	q_3	q_4	q_5
Λ	$\Lambda R q_4$	$\Lambda L q_3$	$\Lambda R q_1$	$\Lambda M q_5$	$\Lambda M q_5$
1	$\alpha M q_2$	$\beta M q_1$	$1 R q_1$	$1 L q_1$	$1 M q_5$
α	$\alpha L q_1$	$\alpha R q_2$	$1 L q_3$	$\Lambda R q_4$	$\alpha M q_5$
β	$\beta L q_1$	$\beta R q_2$	$\Lambda L q_3$	$1 R q_4$	$\beta M q_5$

Abb. 12

Strukturschemas angeben. Daraus ist zu ersehen, aus welchen Organen die Maschine besteht und wie diese ineinandergreifen. Alle Turing-Maschinen haben das gleiche Strukturschema (vgl. Abb. 13). In diesem Schema ist der Speicher in einen *inneren* und einen *äußeren Speicher* unterteilt. Der äußere Speicher wird von den *Zellen des unendlichen Bandes* gebildet, die zur Speicherung von Informationen in Symbolen des äußeren Alphabets bestimmt sind. Der innere Speicher besteht aus zwei Zellen zur Speicherung des nächsten Befehls: Die Q-Zelle speichert das Symbol für den Zu-

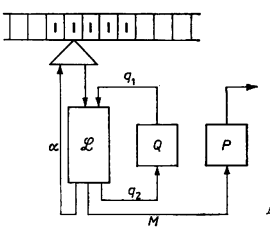


Abb. 13

stand und die P-Zelle das Verschiebungssymbol. In diesen beiden Zellen werden die Symbole P und q_l, die am Ausgang des logischen Blocks im aktuellen Arbeitstakt erhalten wurden, bis zum Beginn des nächsten Taktes gespeichert. Das in einem bestimmten Takt erarbeitete Zustandssymbol q_l wird aus der Q-Zelle über eine sogenannte *Rückkopplung* im nächsten Takt dem logischen Block $\mathfrak{L}$ wieder als Eingangssignal zur Verfügung gestellt. Das in der P-Zelle enthaltene *Verschiebungssymbol* steuert einen *Transportmechanismus* für das Band. Die Funktion des Steuerwerks reduziert sich also bei einer Turing-Maschine im wesentlichen auf eine Ver-

schiebung des Bandes um höchstens eine Zelle, entsprechend dem in der P-Zelle gespeicherten Symbol. Der logische Block $\mathfrak{L}$ ist mit dem äußeren Speicher über einen *Lese- und Schreibkopf* (im folgenden kurz „Kopf" genannt) verbunden, in den der Eingangskanal (Lesen) und der Ausgangskanal (Schreiben) von $\mathfrak{L}$ für die Symbole des äußeren Alphabets münden. In Abb. 13 ist dieser Kopf durch ein Dreieck dargestellt, das auf die eingestellte Zelle des Bandes weist.

2.2.2. Die Arbeitsweise einer Turing-Maschine

Die Arbeit einer Turing-Maschine läuft auf folgende Weise ab. Zu Beginn der Arbeit befindet sich auf dem Band eine bestimmte Anfangsinformation (in Abb. 13 sind es fünf aufeinanderfolgende Striche). Der Kopf ist auf eine bestimmte Anfangszelle des Bandes eingestellt (in Abb. 13 auf die zweite mit einem Strich belegte Zelle von links). In den Zellen P und Q sind eine bestimmte Anfangsverschiebung (z. B. M) und ein bestimmter Anfangszustand (z. B. q_1) vorgegeben.[1]) Dann läuft in der Maschine automatisch ein Prozeß ab, der eindeutig durch das Funktionsschema der Maschine und die Anfangswerte bestimmt ist. Wir wollen uns das am Beispiel des Funktionsschemas aus Abb. 12 ansehen:

Erster Takt. Aufgerufen wird die Anfangszelle des Bandes (Anfangsverschiebung sollte M sein) im Anfangszustand q_1. Aus der aufgerufenen Zelle wird das Zeichen | (Strich) entnommen. Aufgrund des Funktionsschemas aus Abb. 12 entspricht dem Eingangspaar $|q_1$ das Ausgangstripel $\alpha M q_2$. Es werden also in der aufgerufenen Zelle des Bandes das Symbol α und in den Zellen P und Q die den nächsten Befehl charakterisierenden Symbole M und q_2 gespeichert.

Zweiter Takt. Es wird die gleiche Zelle wie im vorangehenden Takt aufgerufen (da der Inhalt von P nach dem ersten Takt das Zeichen M ist), und es wird aus ihr das Zeichen α entnommen. Aus der Zelle Q kommt der Zustand q_2. Das Funktionsschema (3. Zeile, 2. Spalte) liefert das Ausgangstripel $\alpha R q_2$, d. h., der Inhalt α der aufgerufenen Zelle des Bandes wird nicht geändert, und der nächste Befehl lautet Rq_2.

Dritter Takt. Es wird die zur im vorangehenden Takt betrachteten Zelle rechts benachbarte Zelle aufgerufen (da der Inhalt von P das Zeichen R ist). Das in ihr enthaltene Zeichen | wird durch das Symbol β ersetzt (2. Zeile, 2. Spalte des Funktionsschemas), und es wird der neue Befehl Mq_1 gespeichert, usw.

Aus dem Funktionsschema der Abb. 12 ist zu entnehmen, daß der durch die Maschine realisierte Prozeß in einem gewissen Sinne abbricht, wenn die Maschine im Verlauf eines Rechenprozesses einmal in den Zustand q_5 gekommen ist; denn dann wird das in der aufgerufenen Zelle stehende Zeichen nicht mehr geändert, es wird (da in der Spalte unter q_5 überall die Verschiebung M notiert ist) zu keiner Nachbarzelle mehr

[1]) Wenn nichts anderes gesagt wird, soll die Anfangsverschiebung stets M sein.

übergegangen, und auch der Zustand q_5 bleibt dauernd erhalten. Die Maschine hält zwar im eigentlichen Sinne des Wortes nicht an, jedoch wird durch die Maschine ein Ende des Prozesses signalisiert, so daß wir den Zustand q_5 als einen *Endzustand* (*Stopzustand*) interpretieren können (sogenannter *dynamischer Stop*). Die Maschine mit dem Funktionsschema aus Abb. 12 ist also auf genau die Anfangsinformationen *anwendbar*, bei denen die Maschine einmal in den Zustand q_5 gelangt.

Natürlich kann das Funktionsschema einer Turing-Maschine auch von einem Rechner als ein Schema zum Umformen einer im äußeren Alphabet geschriebenen Information verstanden werden. Faktisch sind wir bei der Beschreibung der Anwendung des Funktionsschemas aus Abb. 12 auf das Wort aus fünf Strichen sogar so vorgegangen und haben nur festgestellt, daß die gedachte Turing-Maschine genauso operieren würde. Für einen Rechner wäre das Funktionsschema einer Turing-Maschine ein bestimmtes Standardverfahren zur Mitteilung eines Algorithmus, den man etwa als *Turing-Algorithmus* bezeichnen könnte. Ein Turing-Algorithmus kann auch durch eine Liste von Anweisungen (sogenannte *Turing-Befehle*) der Form

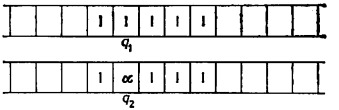
$$xq \to x'Pq'$$

charakterisiert werden, wobei x und q die Zeile und Spalte des Funktionsschemas bezeichnen und $x'Pq'$ das an der entsprechenden Stelle des Schemas stehende Tripel ist. Dieser Befehl ist wie folgt auszuführen: Wird im aktuellen Takt an der betrachteten Stelle der Information das Symbol x im Zustand q gelesen, so ist x durch x' zu ersetzen und anschließend zu der in Richtung P (also z. B. „rechts" im Fall $P = R$) benachbarten Stelle und in den Zustand q' überzugehen.

Da uns im folgenden nur der durch eine Turing-Maschine realisierte Prozeß interessiert, ist es gleichgültig, ob wir das Funktionsschema als Charakterisierung des logischen Blocks einer Turing-Maschine oder als Niederschrift eines Turing-Algorithmus (als ein *Turing-Programm*) für einen Rechner interpretieren. Wir werden übrigens in Abschnitt 2.7 noch ein wichtiges Prinzip darlegen, das die Möglichkeit dieser doppelten Interpretation rechtfertigt. Im folgenden werden also die Begriffe „Funktionsschema einer Turing-Maschine" und „Turing-Programm", aber z. B. auch „Turing-Maschine" und „Turing-Algorithmus" usw. synonym gebraucht.

Beim Studium der Arbeit einer Turing-Maschine bzw. eines Turing-Algorithmus empfiehlt es sich manchmal, den Begriff der *Turing-Konfiguration* (oder kurz *Konfiguration*) zu benutzen. Unter der k-ten Konfiguration verstehen wir eine graphische Darstellung der Information, die sich zu Beginn des k-ten Taktes auf dem Band befindet, und wo die aufgerufene Zelle dadurch gekennzeichnet ist, daß unter ihr der Zustand notiert ist, in dem sich die Maschine zu Beginn des k-ten Taktes befindet (d. h., der in der Zelle Q gespeichert ist). Damit ist aus der k-ten Konfiguration insbesondere das Eingangspaar für den k-ten Takt ablesbar. Aus dem Funktionsschema kann man sodann das zugehörige Ausgangstripel ermitteln und damit die $(k + 1)$-Konfiguration bestimmen.

Im analysierten Beispiel haben die erste und zweite Konfiguration das folgende Aussehen:



Eine Konfiguration heißt *endlich*, wenn bei ihr nur endlich viele Zellen des Bandes ein vom Leerzeichen verschiedenes Zeichen enthalten. Es ist klar, daß eine Turing-Maschine, wenn sie auf eine endliche Konfiguration angesetzt wird (und nur dieser Fall ist von Interesse), auch nur endliche Konfigurationen produziert, wobei sich natürlich die Anzahl der nichtleeren Zellen im Verlauf der Arbeit beliebig vergrößern kann. Somit ist also bei einer Turing-Maschine, wenn sie auch formal über ein unendliches Band verfügt, in jedem Takt immer nur ein endliches Stück dieses unendlichen Bandes von Bedeutung. Natürlich muß dieses endliche Stück jedesmal dann, wenn der Kopf der Maschine eines seiner Enden erreicht hat und es verlassen würde, verlängert werden. Daher kann man die Turing-Maschine statt als Maschine mit unendlichem Speicher besser als Maschine mit stets endlichem, aber potentiell unbeschränkt vergrößerbarem Speicher auffassen.

Bei der Behandlung von Beispielen wollen wir das Funktionsschema etwas einfacher notieren: Wir wollen im Ausgangstripel $s_i P q_l$ die Symbole s_i und q_l nur dann aufschreiben, wenn sie von den entsprechenden Symbolen des zugehörigen Eingangspaars verschieden sind. Ferner wollen wir das Verschiebungssymbol M, das ja angibt, daß keine Verschiebung stattfindet, in Zukunft fortlassen. Damit werden die Spalten, die einem Stopzustand entsprechen, „leer". Diese Spalten lassen wir weg und notieren in den übrigen die Stopzustände (die zu einem einzigen zusammengefaßt werden können) durch das Symbol „!". Damit nimmt z. B. das Funktionsschema aus Abb. 12 die folgende einfachere Gestalt an:

	q_1	q_2	q_3	q_4
Λ	Rq_4	Lq_3	Rq_1	!
1	αq_2	βq_1	Rq_1	Lq_1
α	L	R	$1L$	ΛR
β	L	R	ΛL	$1R$

Abb. 14

Aus diesem Schema ist u. a. zu entnehmen, daß nach einem α im Zustand q_1 die Maschine eine ganze Serie von Linksverschiebungen über alle hintereinanderstehenden α und β ausführt, ohne den Zustand q_1 und den Inhalt der aufgerufenen Zelle zu verändern. Erst der erste Strich oder die erste leere Zeile unterbrechen diese Arbeitsweise, und nur unter diesen Voraussetzungen verläßt die Maschine den Zustand q_1.

2.3. Die Realisierung eines Algorithmus durch eine Turing-Maschine (Turing-Programmierung)

Wir wollen nun an einer Reihe von Beispielen zeigen, wie Turing-Maschinen aufgebaut sind, die einfache arithmetische Operationen realisieren. In Übereinstimmung mit den Ausführungen des vorangehenden Abschnitts werden wir die Maschinen durch Angabe des Funktionsschemas ihres logischen Blocks beschreiben. Wir werden dabei auch einige methodische Überlegungen darüber anstellen, wie man zu Turing-Maschinen (Funktionsschemata, Turing-Programmen) gelangt, die einen bestimmten Algorithmus realisieren. In einer Reihe von Fällen werden wir abschätzen, wie lange der in der Maschine ablaufende Prozeß dauert (aus wieviel Takten er besteht).

2.3.1. Der Übergang von n zu $n + 1$ im Dezimalsystem

Gesucht ist ein Turing-Programm, das alle Aufgaben folgenden Typs löst:

Aus der Dezimaldarstellung einer beliebigen natürlichen Zahl n ist die Dezimaldarstellung der Zahl $n + 1$ herzustellen.

Als äußeres Alphabet verwenden wir das Alphabet aus den zehn Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 und dem Leerzeichen Λ. Die Maschine habe die beiden Zustände q_0 (Arbeitszustand) und ! (Stopzustand). Die Dezimaldarstellung der natürlichen Zahl n und der Zahl $n + 1$ sollen lückenlos und ziffernweise in üblicher Art auf dem Band stehen, wobei also jede Ziffer in einer Zelle untergebracht ist. Das Funktionsschema der zu untersuchenden Maschine ist in Abb. 15 dargestellt, wobei wir vorläufig die letzte Zeile und die letzte Spalte nicht berücksichtigen wollen (ihr Sinn wird später

	q_0	q_1
0	1 !	
1	2 !	
2	3 !	
3	4 !	
4	5 !	
5	6 !	
6	7 !	
7	8 !	
8	9 !	
9	0 L	
Λ	1 !	
I	L	$\Lambda L q_0$

Abb. 15

erklärt). Zu Beginn der Arbeit möge der Kopf der Maschine auf die Einerstelle der Zahl n eingestellt sein und sich die Maschine im Zustand q_0 befinden. Wenn die eingestellte Ziffer von 9 verschieden ist, so stoppt die Maschine schon nach dem ersten Arbeitstakt, nachdem diese Ziffer durch die ihr gemäß dem Schema entsprechende Ziffer ersetzt wurde. War aber die letzte Ziffer eine 9, so ersetzt sie die Maschine durch eine 0, führt eine Linksverschiebung aus (geht zur benachbarten nächsthöheren Stelle über) und setzt ihre Arbeit im Zustand q_0 fort (auf diese Weise erfolgt die Zehnerübertragung). Endet die Zahl auf k Neunen, so stoppt die Maschine nach genau $k + 1$ Takten.

Die Maschine arbeitet am längsten, wenn die Dezimaldarstellung der Zahl n nur aus Neunen besteht. Dann ist die Zahl der Takte offenbar $]\log_{10} n[+1$, wobei allgemein $]x[$ die kleinste ganze Zahl größer oder gleich x ist. Für alle anderen natürlichen Zahlen n ist die Anzahl der Takte kleiner als $]\log_{10} n[+1$.

In Abb. 16 sind die Konfigurationen für $n = 389$ dargestellt.

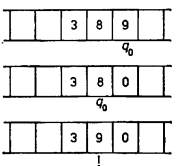


Abb. 16

Wir wollen uns jetzt den Sinn der erweiterten Tabelle aus Abb. 15 klarmachen. Sie gibt das Funktionsschema einer Maschine wieder, in der es noch einen weiteren Zustand q_1 gibt. Außerdem ist auch das äußere Alphabet um ein Symbol, den „Strich", erweitert worden. Befindet sich die Maschine zu Beginn der Arbeit im Zustand q_0 und ist auf dem Band kein Strich vorhanden, so ist die Arbeitsweise dieselbe, wie bei der Maschine des vorhergehenden Beispiels. Das folgt unmittelbar daraus, daß die letzte Spalte und die letzte Zeile der Tabelle dann gar nicht zur Anwendung kommen. Also kann diese Maschine ebenfalls zur Realisierung des vorhergehenden Algorithmus verwendet werden. Da die Maschine aber noch einiges mehr leistet, wollen wir sie etwas näher untersuchen.

Es sei auf das Band die Zahl n im Dezimalsystem aufgetragen. Außerdem mögen in einigen unmittelbar rechts benachbarten Zellen Striche gespeichert sein, je einer in einer Zelle. Wir wollen untersuchen, wie die Maschine arbeitet, wenn zu Beginn der Arbeit der Kopf auf den am weitesten rechts stehenden Strich eingestellt ist und sie selbst sich im Zustand q_1 befindet. Im ersten Takt (Eingangspaar $|q_1$) wird dieser Strich gelöscht, und es erfolgt eine Verschiebung nach links mit Übergang in den Zustand q_0 (Ausgangstripel $\Lambda L q_0$). In den folgenden Takten setzt die Maschine die Verschiebung nach links im Zustand q_0 fort, und zwar so lange, bis sie auf die Einer-

stelle der Zahl n stößt. Von diesem Moment an läuft der vorher behandelte Algorithmus ab, d. h., die Darstellung der Zahl n geht in die Darstellung der Zahl $n + 1$ über, und der Prozeß ist abgeschlossen. Kürzer ausgedrückt: Die Maschine verkleinert die Anzahl der Striche um 1 und führt im Dezimalsystem den Übergang von der Zahl n zur Zahl $n + 1$ aus. Diesen Prozeß wollen wir als *kontrollierten Übergang* von der Dezimaldarstellung von n zur Dezimaldarstellung von $n + 1$ bezeichnen.

In Abb. 17 sind die Konfigurationen für einen Satz von fünf Strichen und für $n = 389$ dargestellt.

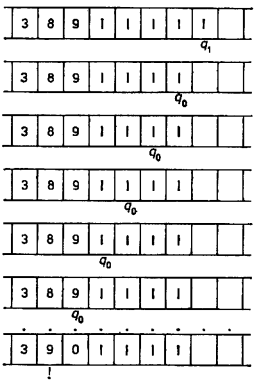


Abb. 17

Wenn in der Anfangskonfiguration rechts von der Dezimaldarstellung der Zahl n noch ν Striche stehen, so können wir die Anzahl t der Takte der Maschine bis zum Erreichen der Endkonfiguration wie folgt abschätzen: Zunächst braucht die Maschine $t_1 = \nu$ Takte, um den letzten Strich zu löschen und die übrigen $\nu - 1$ Striche zu überqueren, d. h. die Einerstelle der Dezimaldarstellung von n aufzusuchen, und anschließend braucht sie noch eine bestimmte Anzahl t_2 von Takten, um aus der Dezimaldarstellung von n die Dezimaldarstellung von $n + 1$ zu erhalten, wobei für t_2 die Ungleichung

$$1 \leqq t_2 \leqq]\log_{10} n[+ 1$$

gilt. Damit erhalten wir

$$\nu + 1 \leqq t_1 + t_2 = t \leqq \nu +]\log_{10} n[+ 1.$$

2.3.2. Der Übergang von der unären Darstellung zur Dezimaldarstellung

Wir konstruieren ein Turing-Programm, das alle Aufgaben folgenden Typs löst:

Eine beliebige endliche Anzahl k von Strichen, die lückenlos in aufeinanderfolgenden Zellen des Bandes stehen (wir wollen das die *unäre Darstellung* der Zahl k nennen), *ist in die Dezimaldarstellung der Zahl k zu transformieren.*

In Abb. 18 ist ein Funktionsschema für eine Turing-Maschine dargestellt, von der wir zeigen wollen, daß sie das Verlangte leistet. Dazu vergleichen wir dieses Schema mit dem erweiterten Schema aus Abb. 15. Die Spalte q_0 im Schema der Abb. 18 unterscheidet sich von der Spalte q_0 im Schema der Abb. 15 nur dadurch, daß in ihr

	q_0	q_1	q_2
0	$1q_2$	!	R
1	$2q_2$	!	R
2	$3q_2$	!	R
3	$4q_2$	!	R
4	$5q_2$	!	R
5	$6q_2$	!	R
6	$7q_2$	!	R
7	$8q_2$	!	R
8	$9q_2$	!	R
9	$0\dot{L}$	!	R
Λ	$1q_2$	0!	Lq_1
!	L	ΛLq_0	R

Abb. 18

überall anstelle des Stopzustandes „!" der neue Zustand q_2 auftritt. Die Unterschiede in den Spalten q_1 haben keinen wesentlichen Einfluß auf die Arbeitsweise des Schemas aus Abb. 15. Wenn sich also auf dem Band die Zahl n im Dezimalsystem befindet und rechts von ihr ein Satz Striche steht, wenn ferner der Kopf der Maschine, wie früher, auf den am weitesten rechts stehenden Strich eingestellt ist und die Maschine sich im Zustand q_1 befindet, dann läuft in der Maschine derselbe Prozeß wie beim Schema der Abb. 15 ab. Es wird ein Strich des Satzes gelöscht und die Dezimaldarstellung der Zahl n durch die von $n + 1$ ersetzt. Während aber im Schema aus Abb. 15 jetzt der Stopzustand „!" eintritt, d. h. der Prozeß abbricht, tritt beim Schema aus Abb. 18 der Zustand q_2 ein, und der Prozeß wird fortgesetzt.

Nimmt man z. B. als erste Konfiguration die aus Abb. 17, so ergibt sich als achte Konfiguration die in Abb. 19 dargestellte. Wie der Prozeß dann fortgesetzt wird, ist aus der Zustandsspalte q_2 zu ersehen: Es erfolgt zuerst eine Serie Rechtsverschie-

bungen über alle Ziffern und Striche hinweg bis zur ersten leeren Zelle. Dann liegt das Eingangspaar Aq_2 (Konfiguration 14 aus Abb. 19) vor, welches eine Verschiebung nach links mit gleichzeitigem Zustandswechsel nach q_1 (Konfiguration 15 aus Abb. 19) bewirkt. Damit steht der Kopf der Maschine im Zustand q_1 wiederum unter dem am weitesten rechts stehenden Strich, womit ein Arbeitszyklus abgeschlossen ist, und ein zweiter analoger Zyklus beginnt. Beim zweiten Zyklus wird ein weiterer Strich gelöscht und die Dezimaldarstellung der Zahl $n + 1$ durch die der Zahl $n + 2$ ersetzt. Wenn ursprünglich ein Satz von k Strichen vorhanden war, so werden nach k Arbeitszyklen alle Striche gelöscht sein, und anstelle der ursprünglich vorhandenen Darstellung der Zahl n wird die der Zahl $n + k$ auftreten. Nach Beendigung des k-ten Zyklus geht die Maschine wiederum in den Zustand q_1 über. Über dem Kopf der Maschine taucht jetzt aber kein Strich mehr auf (denn die Striche sind alle gelöscht).

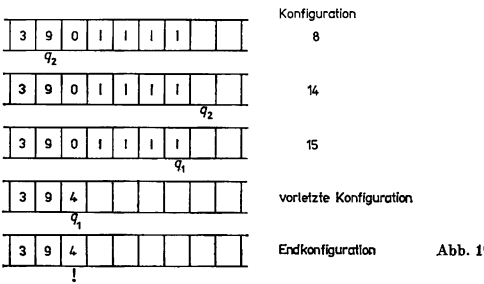


Abb. 19

sondern die letzte Ziffer, d. h. die Einerstelle der Zahl $n + k$ (vorletzte Konfiguration in Abb. 19). Wie aus dem Schema der Abb. 18 zu ersehen ist, wird in diesem Fall der Stopbefehl ausgeführt (letzte Konfiguration aus Abb. 19). War bei Beginn der Arbeit der Maschine auf dem Band die Ziffer 0 mit einem anschließenden Satz von k Strichen vorhanden, so sind also am Ende alle Striche gelöscht und anstelle der Ziffer 0 steht die Dezimaldarstellung der Zahl $0 + k$, d. h. also k. Jedoch kann man zu Beginn auch ohne die Ziffer 0 auskommen, da im Zustand q_0 die Maschine sowohl die Ziffer 0 als auch das Leerzeichen durch die Ziffer 1 ersetzt und in den Zustand q_2 übergeht. Also beschreibt das Schema der Abb. 18 tatsächlich eine Turing-Maschine, die einen Satz von k Strichen in die Dezimaldarstellung der Zahl k umwandelt, wenn man sie im Zustand q_1 auf den am weitesten rechts stehenden Strich des gegebenen Satzes ansetzt.

Wir wollen auch hier die Anzahl der benötigten Takte abschätzen. Da sich der Gesamtprozeß aus k Zyklen zusammensetzt, von denen jeder einen kontrollierten Übergang von der Dezimaldarstellung einer gewissen Zahl n zu der von $n + 1$ enthält, können wir die Abschätzung aus dem vorangehenden Abschnitt anwenden.

Allerdings müssen wir berücksichtigen, daß die Maschine in jedem Zyklus von der höchsten umgewandelten Dezimalstelle wieder zum letzten Strich zurücklaufen muß, bevor sie den nächsten Zyklus beginnen kann. Damit erhalten wir für die Gesamtzahl t der benötigten Takte die folgende Abschätzung:

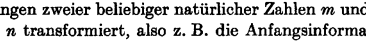
$$\begin{aligned} k(k+3) &= 2\big((k+1)+k+\cdots+2\big) \leqq t \\ &\leqq 2\big((k+1)+k+\cdots+2\big)+2(]\log_{10}1[+]\log_{10}2[+\cdots+]\log_{10}k[) \\ &\leqq k(k+3)+2k]\log_{10}k[\,. \end{aligned}$$

Übungsaufgaben. Man konstruiere nach dem Muster aus Abschnitt 2.3.1 das Funktionsschema einer Turing-Maschine, die für eine beliebige natürliche Zahl $n \geqq 1$ die Dezimaldarstellung von n in die Dezimaldarstellung von $n-1$ umwandelt. Man konstruiere in Analogie zum Abschnitt 2.3.2 das Funktionsschema einer Turing-Maschine, die die Dezimaldarstellung einer beliebigen natürlichen Zahl k in die unäre Darstellung von k (d. h. in einen Satz von k Strichen) umwandelt. Man schätze die Taktzahl dieser Algorithmen ab.

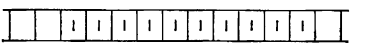
Wir behandeln nun noch eine Reihe von Beispielen für Turing-Maschinen zur Lösung arithmetischer Aufgaben. In diesen Aufgaben sollen sowohl die Anfangswerte als auch die Resultate natürliche Zahlen in unärer Darstellung sein. Kommen in einer Aufgabe mehrere natürliche Zahlen vor, so denken wir uns ihre unären Darstellungen auf dem Band durch ein *Trennzeichen* (z. B. einen Stern „*") voneinander getrennt aufgeführt. Dieses Zeichen gehört dann ebenfalls zum äußeren Alphabet.

2.3.3. Die Addition

Gesucht ist ein Turing-Programm, das die durch einen Stern getrennten unären Darstellungen zweier beliebiger natürlicher Zahlen m und n in die unäre Darstellung von $m+n$ transformiert, also z. B. die Anfangsinformation



in die Endinformation



überführt.

Wir bemerken, daß man diesen Prozeß nicht einfach durch Löschen des Sterns realisieren kann, weil dann zwischen den Strichen eine Leerzelle auftreten würde, folglich der Satz übrigbleibender Striche (wegen der Lücke) nicht als unäre Darstellung einer natürlichen Zahl angesehen werden kann.

Wir behaupten, daß die durch das Funktionsschema in Abb. 20 definierte Turing-Maschine das Verlangte leistet, wenn man den Zustand q_0 als Anfangszustand nimmt und den Kopf der Maschine zu Beginn der Arbeit auf den am weitesten links stehenden Strich einstellt (Konfiguration 1 in Abb. 21).

	q_0	q_1	q_2
I	$\Lambda R q_2$	L	R
Λ	R	$R q_0$	$\mathrm{I} q_1$
*	$\Lambda!$	L	R

Abb. 20

Im ersten Takt wird der Strich in der aufgerufenen Zelle gelöscht, es erfolgt eine Verschiebung um ein Feld nach rechts und Übergang in den Zustand q_2 (Konfiguration 2). In den folgenden Takten erfolgt eine Rechtsverschiebung im Zustand q_2 über alle Striche und den Stern hinweg bis zur ersten Leerzelle (Konfiguration 12). Im anschließenden Takt (Eingangspaar Λq_2) wird in diese Leerzelle ein Strich geschrieben und ohne Verschiebung in den Zustand q_1 übergegangen. Der Zustand q_1 bewirkt eine Linksverschiebung über alle Striche und den Stern hinweg bis zur ersten linken Leerzelle (Konfiguration 24). Sodann (Eingangspaar Λq_1) kommt es zu einer Rechtsverschiebung und Übergang in den Zustand q_0 (Konfiguration 25).

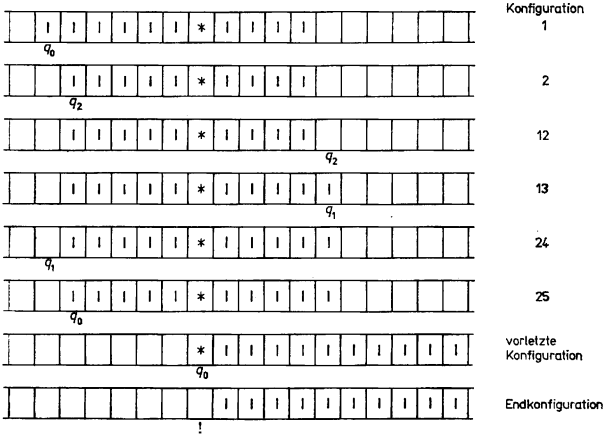


Abb. 21

Über dem Kopf steht die Zelle mit dem ersten stehengebliebenen Strich links vom Stern. Damit beginnt ein neuer Zyklus. In jedem Zyklus wird ein Strich des ersten Summanden vorn weggelassen und dafür ein Strich am zweiten Summanden hinten angehängt. Befanden sich zu Beginn der Arbeit links vom Stern m Striche und rechts vom Stern n Striche, so steht nach m Zyklen links vom Stern kein Strich mehr, während rechts vom Stern $m + n$ Striche stehen. Im letzten Takt des m-ten Zyklus, dem Schritt nach rechts mit Übergang in den Zustand q_0 (vorletzte Konfiguration), erscheint in der Zelle über dem Kopf anstelle eines Strichs der Stern. Im folgenden Takt (Eingangspaar $* q_0$) wird der Stern gelöscht, und die Maschine stoppt (letzte Konfiguration).

Also realisiert das Funktionsschema der Abb. 20 in der Tat die Addition von in unärer Darstellung gegebenen natürlichen Zahlen. Bei der Addition der Zahlen m und n (d. h. bei m Strichen links und n Strichen rechts vom Stern) benötigt die Maschine in jedem Zyklus $2(m + n + 1)$ Takte, so daß für die Bewältigung der m Zyklen insgesamt $2m(m + n + 1)$ Takte erforderlich sind.

2.3.4. Die iterierte Addition und die Multiplikation

Wir untersuchen, wie man das Schema in Abb. 20 abändern muß, damit ein endloser Prozeß entsteht. Dabei wird vorausgesetzt, daß das Zahlenpaar m, n so auf das Band geschrieben ist, wie es bei der in Abschnitt 2.3.3 behandelten Addition der Fall war. der Prozeß soll so verlaufen, daß zunächst die linke Zahl m zur rechten Zahl n, dann Die linke Zahl m zur Summe $n + m$, danach zur Summe $n + 2m$ usw. hinzugefügt wird, ohne daß der Prozeß jemals abbricht. Dabei darf offenbar der linke Summand

	q_0	q_1	q_2	q_3
|	$\alpha R q_2$	L	R	
Λ	R	Rq_0	$\|q_1$	Rq_0
$*$	q_3	L	R	L
α	R	Rq_0	$\|q_1$	$\|L$

Abb. 22

nach der ersten Addition nicht verschwinden, sondern er muß nach jeder Addition wiederhergestellt werden, damit man ihn wieder zum rechts vom Stern stehenden Summanden hinzufügen kann. Das kann man beispielsweise dadurch erreichen, daß die Striche des linken Satzes nicht gelöscht, sondern durch ein neues Zeichen ersetzt werden. In Abb. 22 ist ein Schema dargestellt, in dem der Buchstabe α die Rolle dieses *Merkzeichens* übernimmt. Aus den angeführten Gründen wird in der ersten Zeile des Schemas in Abb. 20 an die Stelle von Λ das Zeichen α gesetzt. Außerdem gibt es im Schema der Abb. 22 noch eine vierte Zeile für den Buchstaben α. Die ersten drei Fächer dieser Zeile stimmen mit den entsprechenden Fächern im Schema der Abb. 20 in der Zeile für Λ überein. Damit der Prozeß nicht nach der ersten

Addition abbricht, muß im Schema der Abb. 22 anstelle des Zeichens „!", welches in Abb. 20 beim Eingangspaar $* q_0$ erscheint, ein anderer Zustand auftreten, der die Fortsetzung des Prozesses sichert. In unserem Fall ist es der neue Zustand q_3. Die zu ihm gehörige Spalte muß so ausgefüllt werden, daß der Gesamtprozeß in gewünschter Weise abläuft. Da im Schema der Abb. 22 das Zeichen „!" (Stop) nicht auftritt, wird der Prozeß endlos laufen. Der Leser verifiziert nun ohne Mühe, daß die Maschine auch wirklich die links vom Stern stehende Zahl zu der rechts vom Stern stehenden unbeschränkt oft hinzufügt. Standen bei Arbeitsbeginn rechts vom Stern keine Striche (d. h. war die zweite Zahl 0), so erscheinen rechts vom Stern erst m, dann $2m$, osdann $3m$ Striche usw. ohne Ende.

Man kann das angegebene Schema leicht so abändern, daß der iterierte Additionsprozeß nicht unbeschränkt oft wiederholt wird, sondern nur so oft, wie Striche in der unären Darstellung von m (d. h. links vom Stern) vorhanden sind. Dazu muß in jedem Additionszyklus ein Strich vor dem Stern gelöscht werden und der Prozeß unter Löschung des Sterns abgebrochen werden, wenn die Striche vor dem Stern aufgebraucht sind.

Übungsaufgabe: Es ist ein Funktionsschema anzugeben, das die Multiplikation von in unärer Darstellung gegebenen Zahlen realisiert, und dessen Arbeitszeit abzuschätzen.

2.3.5. Der Euklidische Algorithmus

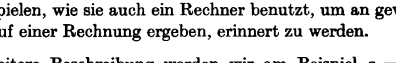
Wir wollen jetzt eine Turing-Maschine für den Euklidischen Algorithmus zum Aufsuchen des größten gemeinsamen Teilers zweier natürlicher Zahlen a und b angeben. Dieser Algorithmus ist bereits zweimal von uns beschrieben worden: Das erste Mal in Worten, das zweite Mal in Form eines Programms für einen elektronischen Rechenautomaten. Dieses Mal geben wir den Algorithmus in Form eines Funktionsschemas einer Turing-Maschine an und verfolgen den Berechnungsprozeß in der Maschine. Der Prozeß setzt sich aus abwechselnden Vergleichs- und Subtraktionszyklen zusammen, die den elementaren Operationen Vergleich und Subtraktion in einem programmgesteuerten Automaten entsprechen. Das Funktionsschema ist in Abb. 14 (Abschnitt 2.2.2) dargestellt. Das äußere Alphabet besteht aus den vier Zeichen

$$\Lambda, \mathrm{I}, \alpha, \beta.$$

Die natürlichen Zahlen werden auch hier durch Sätze von Strichen, d. h. in unärer Form, dargestellt. Um uns nicht in Einzelheiten zu verlieren, die nichts mit dem Wesen der Sache zu tun haben und unsere Untersuchungen nur unnötig komplizieren würden, wollen wir die unären Darstellungen der gegebenen Zahlen ohne Lücke oder Trennzeichen aufeinanderfolgen lassen. Zu Beginn des Prozesses sei der Kopf der Maschine auf die Zelle mit dem am weitesten rechts stehenden Strich der ersten Zahl eingestellt (d. h., die Trennung der beiden Zahlen erfolgt zu Beginn der Rechnung und auch in

einigen späteren Stadien durch die Stellung des Kopfes). Nach der Analyse des Algorithmus wird der Leser ohne besondere Schwierigkeiten das verwendete Funktionsschema so abändern können, daß die Maschine bei anderer Anfangsbedingung die Aufgabe einwandfrei löst (z. B., wenn die unären Darstellungen durch einen Stern getrennt sind und der Kopf der Maschine am Anfang unter einer Leerzelle steht). Wir wollen noch darauf hinweisen, daß die Buchstaben α und β die Rolle von *Merkzeichen* spielen, wie sie auch ein Rechner benutzt, um an gewisse Umstände, die sich im Verlauf einer Rechnung ergeben, erinnert zu werden.

Die weitere Beschreibung werden wir am Beispiel $a = 4$, $b = 6$ mittels Konfigurationen verdeutlichen. Die Anfangskonfiguration sieht hier so aus:



Im Vergleichszyklus treten nur die Zustände q_1, q_2 und im Subtraktionszyklus nur die Zustände q_3, q_4 auf.

Wir verfolgen jetzt den Prozeß im einzelnen. Zuerst vergleicht die Maschine die Zahlen, die auf dem Band dargestellt sind, um die größere von ihnen zu ermitteln. Hierbei geht sie genauso vor, wie es auch ein Mensch machen würde, der zwei lange, nicht sofort zu übersehende Folgen von Strichen vergleichen soll. Er würde abwechselnd in beiden Folgen je einen Strich markieren (indem er ihn z. B. durchstreicht). Ist eine der Folgen ausgeschöpft, so ist geklärt, welche der beiden Folgen länger ist. Die Maschine ersetzt einen Strich der ersten Zahl durch das Symbol α und danach einen Strich der zweiten durch das Symbol β. Dann kehrt sie wiederum zu den Strichen der ersten Zahl zurück und ersetzt einen weiteren Strich durch α und danach wieder einen Strich der zweiten Zahl durch β usw.

Die Konfigurationen der ersten vier Takte für unser Beispiel sind in Abb. 23 dargestellt. Am Ende des vierten Taktes hat die Maschine einen Strich jeder Zahl markiert und begibt sich durch Linksverschiebungen auf die Suche nach dem letzten noch nicht markierten Strich der linken Zahl. Nach einigen Takten liegt die Kon-

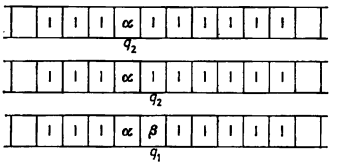


Abb. 23

figuration I aus Abb. 24 vor, d. h., die erste Zahl ist ausgeschöpft, die zweite noch nicht. Da das Suchen nach einem nicht markierten Strich der linken Zahl erfolglos ist, entsteht die Konfiguration II, und der Vergleichszyklus ist abgeschlossen, wobei nur die Zustände q_1 und q_2 benötigt wurden. Der folgende Takt liefert die Konfiguration III.

Wie aus der Spalte q_4 der Abb. 14 zu ersehen ist, werden bei den anschließenden Rechtsverschiebungen alle α durch das Leerzeichen und alle β durch Striche ersetzt (d. h., alle α werden gelöscht). Nachdem das letzte β durch einen Strich ersetzt ist, befindet sich die Konfiguration IV aus Abb. 24 auf dem Band. Der folgende Takt liefert die Konfiguration V. Dem Vergleichszyklus hat sich also ein Zyklus der Subtraktion der ersten von der zweiten Zahl angeschlossen. Hierbei ist die kleinere

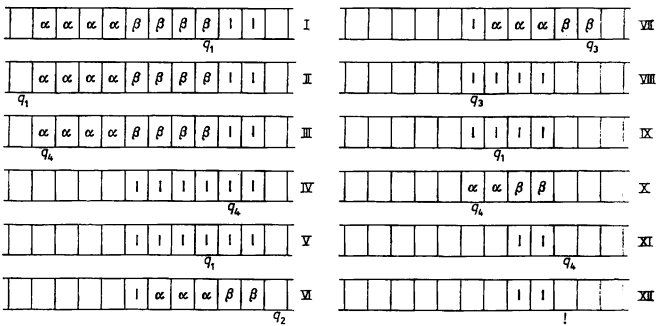


Abb. 24

Zahl a gelöscht und die größere b in a und $b - a$ aufgespalten worden. Die Zelle, in der der letzte Strich der ersten Zahl steht, ist aufgerufen, und die Maschine befindet sich wiederum im Zustand q_1. Das bedeutet, daß die Aufgabe für die Zahlen a und b auf die gleiche Aufgabe für die Zahlen a und $b - a$ zurückgeführt ist. Hierauf basiert aber gerade, wie wir schon wissen, der Euklidische Algorithmus.

Nun kommt wieder ein Vergleichszyklus. Diesmal ist jedoch (im Beispiel) die rechte Zahl die kleinere. Das stellt sich heraus, nachdem die Maschine drei Striche der ersten Zahl ersetzt hat und es in der zweiten keinen Strich mehr gibt, d. h., es liegt die Konfiguration VI vor. Der folgende Takt erzeugt die Konfiguration VII, und mit ihr beginnt der Zyklus der Subtraktion der zweiten Zahl von der ersten, d. h. das Löschen aller β und das Ersetzen der α durch Striche. Nach dem Ersetzen des am weitesten links stehenden α durch einen Strich befindet sich auf dem Band die Konfiguration VIII und danach die Konfiguration IX, womit der Subtraktionszyklus

abgeschlossen ist und der folgende Vergleichszyklus beginnt usw. Der Prozeß wird so lange fortgesetzt, bis die Aufgabe auf zwei gleiche Zahlen zurückgeführt ist (in unserem Beispiel ist das bei Konfiguration IX erreicht). Dann beginnt der letzte Vergleichszyklus, der zu dem Resultat führt. In der Tat: Nachdem die Konfiguration X entstanden ist, erzeugt die Subtraktion die Konfiguration XI, und diese geht direkt in die Endkonfiguration XII über.

2.3.6. Die Standard-Darstellung eines Wortes in einer Turing-Maschine

Für das Weitere wollen wir eine *Standard-Darstellung* für die Anfangs- und die Endinformation bei Berechnungen auf Turing-Maschinen vereinbaren. In der Regel werden diese Informationen durch ein Anfangs- und ein Resultatwort P bzw. R dargestellt, die einem vorher festgelegten Teilalphabet[1]) des äußeren Alphabets angehören (die übrigen Zeichen des äußeren Alphabets spielen die Rolle von Hilfs- oder Merkzeichen). Jedoch haben wir bisher keine feste Vereinbarung darüber getroffen, auf welche Zelle des Bandes der Kopf der Maschine am Anfang und am Ende der Rechnung eingestellt sein soll. So haben wir in den Beispielen in den Abschnitten 2.3.1 und 2.3.2 angenommen, daß sich der Kopf der Maschine in der Anfangskonfiguration unter der Zelle mit dem am weitesten rechts stehenden Symbol des Anfangswortes P befindet. In den Beispielen der Abschnitte 2.3.3 und 2.3.4 war es dagegen das am weitesten links stehende Symbol des Anfangswortes P. Beim Euklidischen Algorithmus in Abschnitt 2.3.5 schließlich befand sich der Kopf der Maschine in der Anfangskonfiguration an einer ausgezeichneten Stelle im Innern des Anfangswortes P. Analog haben wir bei der Endkonfiguration keine feste Vereinbarung über die Stellung des Kopfes getroffen.

Von nun an vereinbaren wir, daß sich (falls nicht ausdrücklich etwas anderes gesagt wird) in der Anfangs- und Endkonfiguration der Kopf der Maschine unter dem ersten (d. h. am weitesten links stehenden) Buchstaben des Anfangs- bzw. Resultatwortes P bzw. R befinden soll. Wir wollen das die Standard-Darstellung der Wörter P und R durch eine Konfiguration nennen. Führt die Turing-Maschine $\mathfrak{M}$ die Standard-Darstellung des Wortes P in die Standard-Darstellung des Wortes R über, so schreiben wir $\mathfrak{M}(P) = R$.

Übungsaufgaben. Man konstruiere für die in den Abschnitten 2.3.1 bis 2.3.5 behandelten Aufgaben Funktionsschemata, die mit der Standard-Darstellung der Anfangs- und Resultatdaten arbeiten.

---

[1]) Dieses Teilalphabet soll selbstverständlich nicht das Leerzeichen des äußeren Alphabets enthalten. Unter einem Wort über dem äußeren Alphabet S einer Turing-Maschine soll im folgenden stets ein Wort aus vom Leerzeichen von S verschiedenen Buchstaben verstanden werden. [*Anm. d. Übers.*]

Man konstruiere Funktionsschemata, die in der unären Darstellung die folgenden Operationen im Bereich der natürlichen Zahlen realisieren:

$$x \dot{-} y = \begin{cases} x - y, & \text{falls } x \geqq y, \\ 0, & \text{falls } x < y, \end{cases}$$

$\left[\frac{x}{y}\right]$ (Quotient bei der Division von x durch y), *Rest*(x, y) (Rest bei der Division von x durch y), wobei der Quotient u bzw. der Rest v die durch die folgenden Bedingungen charakterisierten natürlichen Zahlen sind:

$$x = uy + v \quad \text{mit} \quad 0 \leqq v < y.$$

Die Zahl Null werde dabei in der unären Darstellung durch das leere Band dargestellt.

2.4. Programmierende Algorithmen

Im vorangehenden Abschnitt haben wir für einige verhältnismäßig einfache Algorithmen gezeigt, wie man sie durch Turing-Maschinen realisieren kann, indem wir für sie entsprechende Turing-Programme (Funktionsschemata) direkt aufgestellt haben. Im allgemeinen ist das Aufstellen eines Algorithmus durchaus keine einfache Angelegenheit; es erfordert meistens eine tiefgehende Analyse des Inhalts der durch den Algorithmus zu lösenden Aufgabenklasse und große Erfindungsgabe. Wollen wir einen ursprünglich in anderer Form gegebenen Algorithmus durch eine Turing-Maschine darstellen, so werden wir mit zusätzlichen Schwierigkeiten konfrontiert. Grund hierfür ist, daß die Turing-Maschine nur über außerordentlich einfache Elementaroperationen verfügt und nur einen sehr beschränkten Zugriff zu ihrem äußeren Speicher hat. In jedem Takt kann lediglich der Inhalt einer Zelle geändert werden, und es ist lediglich der Übergang zu einer Nachbarzelle möglich. Im Vergleich zum Programmieren realer elektronischer Rechenautomaten (vgl. Abschnitt 1.6) sind diese Schwierigkeiten recht erheblich. Durch die nachfolgende Bemerkung, die übrigens nicht nur auf die Turing-Programmierung, sondern auch auf die Programmierung realer Rechenautomaten anwendbar ist, lassen sich diese Schwierigkeiten häufig beträchtlich vermindern: Bei der Aufstellung neuer Programme (speziell neuer Turing-Programme) kann man versuchen, schon früher erarbeitete Programme als Teile (sogenannte *Unterprogramme*) zu verwenden. Das ist vor allem dann möglich, wenn der zu programmierende Algorithmus eine gewisse Verknüpfung von früher untersuchten und bereits programmierten Algorithmen ist.

2.4.1. Standard-Programme und formale Operationen für Programme

Wie kann man nun die soeben formulierten allgemeinen Überlegungen möglichst systematisch verwirklichen?

Erstens wird man sich einen Vorrat von fertigen Programmen (eine sogenannte *Programm-Bibliothek*) anlegen, um sie in passenden Situationen wiederholt als selbständige Programme oder als Unterprogramme von komplizierten Programmen einsetzen zu können.

Zweitens wird man versuchen, allgemeine Methoden zur Konstruktion komplizierter Programme aus einfachen Unterprogrammen auszuarbeiten.

Natürlich wird man in eine Programm-Bibliothek vor allem (Turing-)Programme für solche Algorithmen aufnehmen, die man besonders häufig benutzen kann. Einige von ihnen sollen hier aufgezählt werden:

1. *Der identische Algorithmus* (Bezeichnung: Id). Das Resultat der Anwendung dieses Algorithmus auf ein beliebiges Wort P aus einem gegebenen Alphabet ist das Wort P selbst, $\mathrm{Id}(P) = P$.

2. *Kopierende Algorithmen.* Der verdoppelnde oder einfach kopierende Algorithmus (Bezeichnung: Kop 1) fertigt eine Kopie des Wortes P an; genauer gesagt, es sei $\mathrm{Kop}\,1\,(P) = P \# P$, wobei „#" ein Symbol (*Trennzeichen*) ist, das nicht zu dem Alphabet gehört, dessen Wörter P verdoppelt (kopiert) werden sollen. Analog wird der verdreifachende oder zweimal kopierende Algorithmus Kop 2 durch

$$\mathrm{Kop}\,2\,(P) = P \# P \# P$$

gegeben. Entsprechend sei $\mathrm{Kop}\,3\,(P) = P \# P \# P \# P$ usw. Wir werden auch kopierende Algorithmen betrachten, die nicht ein ganzes Wort, sondern nur einen bestimmten Teil von diesem kopieren. So werden wir häufig den Kopieralgorithmus Kop_* brauchen, der aus Wörtern der Form $P * R$, bei denen in P und R das Trennzeichen „$*$" nicht vorkommt, den hinter dem Stern stehenden Teil R kopiert, d. h., für den

$$\mathrm{Kop}_*\,(P * R) = P * R \# R$$

gilt. Wenn aus dem Zusammenhang klar ist, welchen kopierenden Algorithmus wir meinen, so werden wir ihn einfach mit Kop bezeichnen.

3. *Ersetzungsalgorithmen.* Der Ersetzungsalgorithmus Ers_α^a (a und α sind dabei Buchstaben aus dem betrachteten Alphabet) ersetzt den Buchstaben a an allen Stellen seines Vorkommens in einem Wort P durch den Buchstaben α, d. h., es ist beispielsweise

$$\mathrm{Ers}_\alpha^a\,(babaaa) = b\alpha b\alpha\alpha\alpha.$$

4. *Abspaltungsalgorithmen.* Sie spalten aus einem gegebenen Wort P einen bestimmten Teil ab, der z. B. durch spezielle Trennzeichen gekennzeichnet ist. Wir werden vor allem den Algorithmus $\mathrm{Res}_{\#}$ benötigen, der von einem Wort P den rechts vom letzten Auftreten des Trennzeichens „#" stehenden Rest abspaltet. Es gilt also z. B.

$$\mathrm{Res}_{\#}\,(abb \# aa \# bab) = bab.$$

5. *Der Algorithmus* S. Er berechnet in unärer Darstellung die Funktion $s(x) = x + 1$, d. h., er führt das Wort aus x Strichen in das Wort aus $x + 1$ Strichen über.

Die Aufstellung von Turing-Programmen für die genannten Algorithmen, auch von solchen, die mit der Standard-Darstellung der Wörter arbeiten (vgl. Abschnitt 2.3.6), bereitet keine Schwierigkeiten und sei dem Leser als Übungsaufgabe überlassen. Sollte sich übrigens in einigen Fällen herausstellen, daß ein benötigtes Hilfsprogramm recht kompliziert ist, so braucht man es doch nur einmal aufzustellen, um es dann beliebig oft verwenden zu können.

Es ist klar, daß man sich keinen unbegrenzten Vorrat von Hilfsprogrammen anlegen kann. Glücklicherweise ist das auch nicht nötig. Wir werden nämlich sehen, daß es Algorithmen gibt, wir wollen sie *Algorithmen zur Programmherstellung* oder *programmierende Algorithmen* nennen, mit deren Hilfe man aus schon vorliegenden Turing-Programmen neue Turing-Programme gewinnen kann und die es gestatten, aus einer relativ kleinen Bibliothek von Hilfsprogrammen alle von uns benötigten Turing-Programme zu erzeugen. Zu den tatsächlich benötigten Hilfsprogrammen gehören neben den oben aufgezählten noch einige in den folgenden Abschnitten und Paragraphen zu behandelnde Programme. Wenn auch die programmierenden Algorithmen die bei der Turing-Programmierung auftretenden Schwierigkeiten nicht voll beheben, so stellen sie doch eine wesentliche Erleichterung dar.

Bevor wir die wichtigsten programmierenden Algorithmen näher beschreiben, erwähnen wir zwei oft verwendete formale Operationen für Turing-Programme:

1. *Die simultane Umbenennung der inneren Zustände* mit Ausnahme des Stopzustandes „!".

2. *Die fiktive Erweiterung des äußeren Alphabets*; hier werden zum ursprünglichen Alphabet $S = \{s_1, \ldots, s_k\}$ beliebige neue Symbole $s_{k+1}, \ldots, s_{k+n}$ hinzugefügt und die Spalten in den im Funktionsschema entstehenden n neuen Zeilen sämtlich mit dem Stopzustand „!" ausgefüllt.

Es ist unmittelbar klar, daß folgendes gilt:

Entsteht das Schema $\mathfrak{A}'$ aus dem Schema $\mathfrak{A}$ durch Anwendung der Operation 1 *oder bzw. und der Operation* 2, *so ist für beliebige Wörter P aus dem ursprünglichen Alphabet S stets $\mathfrak{A}(P) = \mathfrak{A}'(P)$, wobei der Prozeß der Umformung von P in das zugehörige Resultat R* (d. h. die Folge der dabei durchlaufenen Zwischenresultate) *bei $\mathfrak{A}$ und $\mathfrak{A}'$ derselbe ist.*

In diesem Sinne kann man die Operationen 1 und 2 als formale Umformungen von Funktionsschemata auffassen, die den durch das Funktionsschema beschriebenen algorithmischen Prozeß nicht ändern. Funktionsschemata mit dieser Eigenschaft nennt man *äquivalent*. Daher können wir bei den folgenden programmierenden Algorithmen im Bedarfsfall annehmen, daß die als Eingangsdaten auftretenden Turing-Programme (Funktionsschemata) dasselbe äußere Alphabet und — vom Stopzustand „!" abgesehen — keine gemeinsamen inneren Zustände haben.

Wir kommen nun zur Betrachtung der vier wichtigsten Arten der Verknüpfung von Algorithmen (speziell von Turing-Programmen), der *Komposition*, der *Parallelanwendung*, der *Verzweigung* und der *Iteration*.

2.4.2. Die Komposition von Turing-Programmen

Sehr häufig ist es nötig, zwei Algorithmen (speziell Turing-Programme) $\mathfrak{A}$ und $\mathfrak{B}$ in der Weise miteinander zu verknüpfen, daß auf ein gegebenes Anfangswort P zuerst der Algorithmus $\mathfrak{A}$ und auf das dabei erhaltene Wort $\mathfrak{A}(P)$ anschließend der Algorithmus $\mathfrak{B}$ angewandt wird, wodurch sich das Wort $\mathfrak{B}(\mathfrak{A}(P))$ ergibt. Offenbar wird hierdurch ein bestimmter neuer Algorithmus $\mathfrak{C}$ beschrieben, den man die *Komposition der Algorithmen* $\mathfrak{A}$ *und* $\mathfrak{B}$ nennt und den wir mit $\mathfrak{A} \circ \mathfrak{B}$ bezeichnen wollen.[1])

Es ergibt sich naturgemäß die Frage, ob man ein Funktionsschema für die Komposition $\mathfrak{A} \circ \mathfrak{B}$ aufstellen kann, wenn man über ein solches für die Algorithmen $\mathfrak{A}$ und $\mathfrak{B}$ verfügt, und wenn das der Fall ist, wie man dabei vorgehen muß. Eine positive Antwort hierauf gibt der sehr einfach zu beweisende

Satz über die Programmierung der Komposition. *Zu beliebig vorgegebenen Turing-Programmen* $\mathfrak{A}$ *und* $\mathfrak{B}$ *kann man ein Turing-Programm* $\mathfrak{C}$ *effektiv konstruieren, so daß für alle Wörter* P *über dem äußeren Alphabet des Programms* $\mathfrak{A}$ *folgendes gilt:*

$$\mathfrak{C}(P) = \mathfrak{B}(\mathfrak{A}(P)).$$

Ein die Komposition programmierender Algorithmus, der also zu gegebenen Turing-Programmen $\mathfrak{A}$, $\mathfrak{B}$ (effektiv) das gesuchte Programm $\mathfrak{C}$ liefert, besteht in der Ausführung folgender Anweisungen: Zunächst werden die Schemata $\mathfrak{A}$ und $\mathfrak{B}$ in der am Ende des vorangehenden Abschnitts beschriebenen Weise präpariert, d. h. (soweit nötig) auf das gleiche äußere Alphabet gebracht und mit unterschiedlichen inneren Zuständen versehen. Sodann wird im Funktionsschema $\mathfrak{A}$ der Stopzustand „!" überall, wo er auftritt, durch den Anfangszustand des Funktionsschemas $\mathfrak{B}$

[1]) In der Literatur ist hierfür auch die umgekehrte Schreibweise $\mathfrak{B} \circ \mathfrak{A}$ üblich. Wir bemerken ferner folgendes — und so soll die Schreibweise $\mathfrak{B}(\mathfrak{A}(P))$ stets verstanden werden: Wenn $\mathfrak{A}(P)$ nicht existiert (d. h. $\mathfrak{A}$ auf P nicht anwendbar ist) oder zwar $\mathfrak{A}(P)$ existiert, aber $\mathfrak{B}(\mathfrak{A}(P))$ nicht existiert (d. h. $\mathfrak{A}$ auf P anwendbar, aber $\mathfrak{B}$ nicht auf $\mathfrak{A}(P)$ anwendbar ist), so wird $(\mathfrak{A} \circ \mathfrak{B})(P)$ als nicht existierend angesehen (d. h., so ist $\mathfrak{A} \circ \mathfrak{B}$ auf P nicht anwendbar). [*Anm. d. Übers.*]

ersetzt. Und schließlich werden die so umgeformten Schemata $\mathfrak{A}$ und $\mathfrak{B}$ zu einem gemeinsamen Schema $\mathfrak{C}$ vereinigt (vgl. Abb. 25a). Als Anfangszustand von $\mathfrak{C}$ wird der Anfangszustand von $\mathfrak{A}$ genommen. Es ist klar, daß das Programm $\mathfrak{C}$, wenn man es auf ein Wort P in Standard-Darstellung anwendet, zunächst wie das Programm $\mathfrak{A}$ arbeitet, und zwar so lange, bis auf dem Band das Wort $\mathfrak{A}(P)$ in Standard-Darstellung vorliegt und die Anwendung von $\mathfrak{A}$ den Stopzustand „!" ergeben würde. In diesem Takt ist beim Schema $\mathfrak{C}$ der erste Buchstabe von $\mathfrak{A}(P)$ im Anfangszustand von $\mathfrak{B}$ eingestellt, so daß sich nun die Abarbeitung der Standard-Darstellung von $\mathfrak{A}(P)$ durch das Programm $\mathfrak{B}$ anschließt, die — wie gewünscht — mit (der Standard-Darstellung von) $\mathfrak{B}(\mathfrak{A}(P))$ endet. Falls $\mathfrak{A}(P)$ nicht existiert oder falls $\mathfrak{A}(P)$ existiert, aber $\mathfrak{B}(\mathfrak{A}(P))$ nicht existiert, so existiert offenbar auch $\mathfrak{C}(P)$ nicht.

Wir bemerken, daß die Forderung der Standard-Darstellung von Anfangs- und Resultatwort nur benötigt wird, um den richtigen Übergang von der Arbeit des Programms $\mathfrak{A}$ zur Arbeit des Programms $\mathfrak{B}$ zu sichern, daß nämlich die Arbeit von $\mathfrak{B}$ mit der Zelle beginnt, auf der die Arbeit von $\mathfrak{A}$ beendet wurde. Daher funktioniert unser programmierender Algorithmus auch in Fällen, in denen nicht die Standard-

	$q_1 \cdot \cdot \cdot q_m$	$r_1 \cdot \cdot \cdot r_n$
s_0	$\mathfrak{A}$, wo "!" durch den Anfangszustand von $\mathfrak{B}$ ersetzt ist	$\mathfrak{B}$
$\vdots$		
s_k		

a)

	q_1	q_2	q_3	q_4	p_0	p_1	p_2
0					$1p_2$	!	R
1					$2p_2$	!	R
2					$3p_2$	!	R
3					$4p_2$	!	R
4					$5p_2$	!	R
5					$6p_2$	!	$\ddot{R}$
6					$7p_2$	!	R
7					$8p_2$	!	R
8					$9p_2$	!	R
9					$0L$	!	R
Λ	Rq_4	Lq_3	Rq_1	p_2	$1p_2$	0!	Lp_1
1	αq_2	βq_1	Rq_1	Lq_1	L	ΛLq_0	R
α	L	R	$1L$	ΛR			
β	L	R	ΛL	$1R$			

b)

Abb. 25

Darstellung der Wörter vorausgesetzt wird, wenn nur der reibungslose Übergang von $\mathfrak{A}$ zu $\mathfrak{B}$ gesichert ist.

Als Beispiel geben wir ein Turing-Programm an, das zu einem beliebigen Paar a, b von in unärer Darstellung gegebenen natürlichen Zahlen den größten gemeinsamen Teiler von a und b in Dezimaldarstellung berechnet. Die Anfangskonfiguration soll dabei in der in Abschnitt 2.3.5 beschriebenen Weise gebildet werden. Das Schema in Abb. 14 (Abschnitt 2.2.2) führt diese Anfangskonfiguration in die unäre Darstellung des größten gemeinsamen Teilers von a und b über, wobei sich der Kopf in der Endkonfiguration unter der ersten Leerzelle rechts von dieser Darstellung befindet (vgl. Abschnitt 2.3.5). Andererseits führt das Schema aus Abb. 18 (Abschnitt 2.3.2), wenn man q_2 als Anfangszustand nimmt, die in einer solchen Konfiguration gegebene unäre Darstellung einer natürlichen Zahl n in die Dezimaldarstellung von n über (vgl. Abschnitt 2.3.2). Folglich ist für die genannten Funktionsschemata, die nicht mit der Standard-Darstellung arbeiten, der reibungslose Übergang gesichert. Wir können also das gesuchte Turing-Programm in der oben beschriebenen Weise aus den Schemata der Abb. 14 und 18 erhalten. Das resultierende Schema ist in Abb. 25b dargestellt, wobei wir die Zustände des Schemas aus Abb. 18 in p_0, p_1, p_2 umbenannt haben.

Natürlich kann man den bewiesenen Satz und den in seinem Beweis angegebenen programmierenden Algorithmus ohne Schwierigkeit auf die Komposition einer beliebigen endlichen Anzahl von Funktionsschemata erweitern.

2.4.3. Die Parallelanwendung von Turing-Programmen

Es sei „#" ein Trennzeichen, das nicht dem Alphabet S angehört, in dem die Anfangs- und Resultatdaten zweier gegebener Algorithmen $\mathfrak{A}$ und $\mathfrak{B}$ dargestellt werden. Es seien ferner P und Q Wörter über dem Alphabet S. Dann können wir die Wörter $P \# Q$ und $\mathfrak{A}(P) \# \mathfrak{B}(Q)$ als Darstellungen der Paare (P, Q) bzw. $(\mathfrak{A}(P), \mathfrak{B}(Q))$ ansehen. Die Vorschrift, auf die Komponenten P, Q des Wortes $P \# Q$ (bzw. des Paares (P, Q)) die Algorithmen $\mathfrak{A}, \mathfrak{B}$ anzuwenden und anschließend zum Wort $\mathfrak{A}(P) \# \mathfrak{B}(Q)$ (bzw. zum Paar $(\mathfrak{A}(P), \mathfrak{B}(Q))$) überzugehen, kann offenbar als ein neuer Algorithmus $\mathfrak{C}$ aufgefaßt werden, den wir die *Parallelanwendung der Algorithmen* $\mathfrak{A}$ und $\mathfrak{B}$ nennen und mit $\mathfrak{A} \parallel \mathfrak{B}$ bezeichnen wollen. So ordnet z. B. die Parallelanwendung der Algorithmen aus den Abschnitten 2.3.1 und 2.3.3 dem Anfangswort $389 \# \mathrm{III} * \mathrm{III}$, das Resultatwort $390 \# \mathrm{IIIIIIIII}$ zu (es wurde die Dezimalzahl 389 um 1 vergrößert und in der unären Zahldarstellung die Addition $6 + 4$ ausgeführt).

Satz über die Programmierung der Parallelanwendung. *Zu beliebigen Turing-Programmen $\mathfrak{A}$ und $\mathfrak{B}$ kann man ein Turing-Programm $\mathfrak{C}$ effektiv konstruieren, so daß für beliebige Wörter P, Q über dem äußeren Alphabet der Programme $\mathfrak{A}, \mathfrak{B}$ folgendes gilt:*

$$\mathfrak{C}(P \# Q) = \mathfrak{A}(P) \# \mathfrak{B}(Q).$$

Der Beweis besteht, analog wie bei der Komposition, in der Angabe eines die Parallelanwendung programmierenden Algorithmus. Seine Beschreibung ist jedoch recht schwierig, obwohl die Grundidee leicht zu übersehen ist. Natürlich wird man versuchen, das Programm $\mathfrak{E}$ so aufzubauen, daß es folgendes leistet: Zunächst soll es das Teilwort P von $P \# Q$ so umformen, wie es das Programm $\mathfrak{A}$ vorschreibt, dann soll es das Teilwort Q gemäß $\mathfrak{B}$ abarbeiten, und schließlich soll es die erarbeiteten Resultate $\mathfrak{A}(P)$ und $\mathfrak{B}(Q)$ mittels „#" verknüpfen. Hierbei kann nun der Fall eintreten, daß die Maschine bei der Bearbeitung des Wortes P nach dem Programm $\mathfrak{A}$ das Wort Q zerstört, weil sie die Zellen, die das Wort Q enthalten, bei der Abarbeitung des Programms $\mathfrak{A}$ benötigt. Genauso kann bei der Anwendung von $\mathfrak{B}$ auf Q das zuvor erarbeitete Wort $\mathfrak{A}(P)$ zerstört werden. Die Situation würde sich offenbar stark vereinfachen, wenn wir statt nur eines eindimensionalen Bandes als äußeren Speicher zwei Bänder hätten und dabei noch die Möglichkeit bestünde, Informationen von einem Band auf das andere zu übertragen. Dann könnten wir ohne gegenseitige Störung die Berechnung von $\mathfrak{A}(P)$ auf dem einen Band und die von $\mathfrak{B}(Q)$ auf dem anderen Band durchführen und brauchten anschließend die Resultate nur noch zusammenzufügen. Diese Bemerkung zeigt, daß für Maschinen anderen Typs (mit einem flexibleren, komfortableren Speicher) die Parallelanwendung von Programmen nahezu trivial wird. Für die einfachen Turing-Maschinen (-Programme) stehen die Dinge etwas schlechter, jedoch gelingt es auch hier, verschiedene Varianten von die Parallelanwendung programmierenden Algorithmen anzugeben. Wir verschieben das auf Abschnitt 2.6 und behandeln zunächst Beispiele und Sätze, die sich auf den Satz über die Programmierung der Parallelanwendung stützen.

2.4.4. Die Verzweigung von Turing-Programmen

In den vorangehenden Abschnitten sind wir bereits einige Male auf Vorschriften folgender Art getroffen: Auf die Anfangsdaten ist der Algorithmus $\mathfrak{A}$ oder $\mathfrak{B}$ anzuwenden, je nachdem, ob sie eine bestimmte Eigenschaft haben oder nicht (man vergleiche z. B. die bedingten Sprungbefehle bei elektronischen Rechenautomaten). Dabei wurde vorausgesetzt, daß für die betreffende Eigenschaft ein *Entscheidungsalgorithmus* Φ vorliegt, der auf alle in Frage kommenden Daten anwendbar ist und jeweils entweder die Antwort „ja" (sie haben die Eigenschaft) oder „nein" (sie haben die Eigenschaft nicht) liefert. Bei durch Turing-Programme realisierten Entscheidungsalgorithmen nimmt man meistens an, daß das Turing-Programm auf alle Wörter P über dem äußeren Alphabet S bzw. dem interessierenden Teilalphabet von S anwendbar ist und alle Wörter, die die betreffende Eigenschaft haben, z. B. in das nur aus dem Symbol „1" bestehende Wort überführt, und die Wörter, die die Eigenschaft nicht besitzen, in das Wort „0" überführt.

Die genannte Vorschrift kann man als einen neuen Algorithmus ansehen, der eine bestimmte Verknüpfung der drei Algorithmen Φ, $\mathfrak{A}$, $\mathfrak{B}$ darstellt, von denen der erste ein Entscheidungsalgorithmus ist. Wir wollen ihn die *Verzweigung von* $\mathfrak{A}$ *und* $\mathfrak{B}$

nach dem Führungsalgorithmus Φ nennen und in Anlehnung an die problemorientierte Sprache ALGOL mit „if Φ then $\mathfrak{A}$ else $\mathfrak{B}$" („wenn Φ, so $\mathfrak{A}$, sonst $\mathfrak{B}$") bezeichnen.

Wir betrachten ein in den Problemkreis des Euklidischen Algorithmus gehörendes Beispiel: Es seien $\Phi_1, \Phi_2, \mathfrak{A}_1, \mathfrak{A}_2, \mathfrak{A}_3$ Algorithmen, die auf alle „Paare" $a * b$ von natürlichen Zahlen anwendbar sind (a und b seien durch ihre unäre oder dezimale Darstellung kodiert) und folgendes leisten:

$$\Phi_1(a * b) = \begin{cases} 1, & \text{falls } a > b, \\ 0, & \text{falls } a \leqq b, \end{cases} \qquad \Phi_2(a * b) = \begin{cases} 1, & \text{falls } a = b, \\ 0, & \text{falls } a \neq b, \end{cases}$$

$$\mathfrak{A}_1(a * b) = a - b * b, \quad \mathfrak{A}_2(a * b) = b * a, \quad \mathfrak{A}_3(a * b) = a.$$

Dann führt der Algorithmus „if Φ_1 then $\mathfrak{A}_1$ else $\mathfrak{A}_2$" das Paar $a * b$ in $a - b * b$ oder $b * a$ über, je nachdem, ob $a > b$ oder $a \leqq b$ gilt. Die zweifache Verzweigung „if Φ_2 then (if Φ_1 then $\mathfrak{A}_1$ else $\mathfrak{A}_2$) else $\mathfrak{A}_3$" führt $a * b$ in $a - b * b$ oder $b * a$ oder a über, je nachdem, ob $a > b$ oder $a < b$ oder $a = b$ gilt. Offenbar ist das gerade ein Zyklus des Euklidischen Algorithmus.

Wir beweisen den folgenden

Satz über die Programmierung der Verzweigung. *Zu beliebigen Turing-Programmen* $\mathfrak{A}$, $\mathfrak{B}$ *und jedem Turing-Entscheidungsprogramm* Φ *mit dem gemeinsamen äußeren Alphabet* S *kann man ein Turing-Programm* $\mathfrak{C}$ *effektiv konstruieren, so daß für alle Wörter* P *über* S *folgendes gilt:*

$$\mathfrak{C}(P) = \begin{cases} \mathfrak{A}(P), & \textit{falls } \Phi(P) = 1, \\ \mathfrak{B}(P), & \textit{falls } \Phi(P) = 0. \end{cases}$$

Wir beschreiben einen programmierenden Algorithmus, der $\mathfrak{C}$ aus den Programmen Φ, $\mathfrak{A}$, $\mathfrak{B}$ erzeugt. Dabei verwenden wir bereits programmierende Algorithmen für die Komposition und die Parallelanwendung, Standard-Programme für den identischen Algorithmus Id und den Kopieralgorithmus Kop1 sowie ein Gabelungsprogramm Gab mit den inneren Zuständen p, p_0, p_1 und den äußeren Symbolen 0, 1, #, das insbesondere die folgenden Befehle umfasse:

$$1p \to \Lambda R p_1, \quad \# p_1 \to \Lambda R, \quad 0p \to \Lambda R p_0, \quad \# p_0 \to \Lambda R$$

(die übrigen Befehle werden nicht benötigt, da sie im Gesamtprogramm nicht zur Anwendung kommen).

Wir konstruieren zunächst ein Programm $\tilde{\Phi}$, das ein beliebiges Wort P in das Wort $1 \# P$ oder $0 \# P$ überführt, je nachdem, ob $\Phi(P) = 1$ oder $\Phi(P) = 0$ gilt. Man sieht sofort, daß das Programm Kop1 $\circ$ ($\Phi \parallel$ Id) das Verlangte leistet.

Der weitere Aufbau des Programms $\mathfrak{C}$ verläuft folgendermaßen (vgl. Abb. 26a): Nachdem die Programme $\tilde{\Phi}$, Gab, $\mathfrak{A}$ und $\mathfrak{B}$ in der am Ende von Abschnitt 2.4.1 beschriebenen Art präpariert sind, d. h. auf dasselbe äußere Alphabet gebracht und mit

paarweise verschiedenen Zuständen versehen sind, werden sie zu einem großen Programm zusammengeschlossen, wobei der Stopzustand von $\tilde{\Phi}$ durch den Zustand p von Gab und die Zustände p_1, p_2 von Gab durch die Anfangszustände q_1, r_1 von $\mathfrak{A}$ bzw. $\mathfrak{B}$ ersetzt werden. Als Anfangszustand von $\mathfrak{C}$ wird der Anfangszustand t_1 von $\tilde{\Phi}$ genommen.

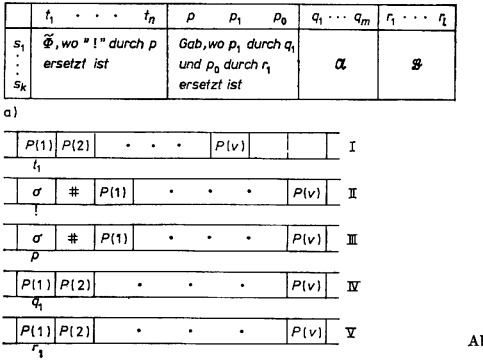


Abb. 26

Wir verfolgen die Arbeit des so konstruierten Programms $\mathfrak{C}$ (vgl. Abb. 26b): Ausgegangen wird von der Standard-Darstellung eines Wortes P (Konfiguration I). Zunächst arbeitet das Unterprogramm $\tilde{\Phi}$; während aber Φ die Konfiguration I in die Konfiguration II überführt, führt das in $\mathfrak{C}$ eingebaute, abgewandelte $\tilde{\Phi}$ die Konfiguration I in die Konfiguration III über (hierbei ist σ eines der beiden Symbole 0 oder 1, und zwar gleich $\Phi(P)$). In diesem Moment kommt das Unterprogramm Gab von $\mathfrak{C}$ zur Anwendung, das die Konfiguration III nach zwei Takten entweder in die Konfiguration IV oder in die Konfiguration V überführt, je nachdem, ob $\sigma = 1$ oder $\sigma = 0$ gilt. Damit ist die gewünschte Verzweigung erreicht, denn anschließend beginnt dasjenige der Programme $\mathfrak{A}$, $\mathfrak{B}$ zu arbeiten, das seine Arbeit aufnehmen soll.

2.4.5. Die Iteration eines Turing-Programms

Wir ändern jetzt die in Abschnitt 2.4.4 beschriebene Konstruktion des Programms „if Φ then $\mathfrak{A}$ else $\mathfrak{B}$" (vgl. Abb. 26a) in folgenden Punkten ab: (i) Für $\mathfrak{B}$ nehmen wir ein Standard-Programm für den identischen Algorithmus, (ii) im Unterprogramm $\mathfrak{A}$ ersetzen wir den Stopzustand „!" durch den Anfangszustand t_1 von $\tilde{\Phi}$. Das ent-

stehende Programm werde mit $\mathfrak{D}$ bezeichnet, und es werde wie oben t_1 als Anfangszustand des Gesamtprogramms genommen. Die Anwendung von $\mathfrak{D}$ auf ein beliebiges Wort P erfolgt in folgenden Etappen: (I) Zunächst wird geprüft, ob P die Eigenschaft Φ besitzt oder nicht. (II) Besitzt P die Eigenschaft Φ (d. h., ist $\Phi(P) = 1$), so wird auf P das Programm $\mathfrak{A}$ angewandt und das Wort P in das Wort $\mathfrak{A}(P)$ übergeführt. Statt aber nun die Arbeit zu beenden, wie das bei dem in Abschnitt 2.4.4 konstruierten Programm $\mathfrak{C}$ der Fall war, schreibt das Programm $\mathfrak{D}$ jetzt vor: (III) Es ist zu prüfen, ob $\mathfrak{A}(P)$ die Eigenschaft Φ besitzt oder nicht (da der Stopzustand von $\mathfrak{A}$ durch den Anfangszustand von Φ ersetzt wurde). (IV) Ist das der Fall, so wird $\mathfrak{A}(\mathfrak{A}(P))$ gebildet. (V) Danach wird geprüft, ob $\mathfrak{A}(\mathfrak{A}(P))$ die Eigenschaft Φ besitzt oder nicht; usw. Wir haben es hier also mit der häufig anzutreffenden Situation zu tun, daß ein Algorithmus $\mathfrak{A}$ und ein Entscheidungsalgorithmus Φ vorgegeben sind und mit ihrer Hilfe folgender Prozeß aufgebaut wird: Für ein beliebiges Wort P wird zunächst geprüft, ob P die Eigenschaft Φ besitzt oder nicht. Wenn das der Fall ist, wird $\mathfrak{A}(P)$ gebildet und der erneuten Prüfung durch Φ unterworfen. Fällt auch diese Prüfung positiv aus, so wird $\mathfrak{A}(\mathfrak{A}(P))$ gebildet usw. Das Verfahren wird so lange fortgesetzt, bis erstmalig ein Wort $\mathfrak{A}\big(\mathfrak{A}\ldots\mathfrak{A}(P)\ldots\big)$ erhalten wird, bei dem die Prüfung durch Φ negativ ausfällt, und dieses Wort wird zum Resultat des Prozesses erklärt. Das kann schon das Anfangswort P sein, wenn nämlich die Prüfung von P auf die Eigenschaft Φ negativ ausfällt. Andererseits kann durchaus auch der Fall eintreten, daß der Prozeß nicht abbricht, d. h. eine unendliche Folge von Wörtern P, $\mathfrak{A}(P)$, $\mathfrak{A}(\mathfrak{A}(P))$, ... erhalten wird, auf die sämtlich die Eigenschaft Φ zutrifft. Das beschriebene Verfahren stellt offenbar einen Algorithmus dar, den wir *die Iteration des Algorithmus* $\mathfrak{A}$ *mit dem Führungsalgorithmus* Φ nennen und in Anlehnung an die Programmiersprache ALGOL mit „while Φ do $\mathfrak{A}$" („solange Φ wende $\mathfrak{A}$ an") bezeichnen wollen.

Die zu Anfang dieses Abschnitts angestellte Betrachtung zeigt, daß eine einfache Modifizierung des die Verzweigung von Turing-Programmen programmierenden Algorithmus einen die Iteration von Turing-Programmen programmierenden Algorithmus liefert. Insbesondere haben wir damit folgenden Satz bewiesen:

Satz über die Programmierung der Iteration. *Zu jedem Turing-Programm* $\mathfrak{A}$ *und jedem Turing-Entscheidungsprogramm* Φ *läßt sich ein Turing-Programm* $\mathfrak{D}$ *für den Algorithmus „while* Φ *do* $\mathfrak{A}$" *effektiv konstruieren.*

2.4.6. Eine Programmiersprache

Wir haben in den vorangehenden Abschnitten vier Arten der Zusammensetzung von Algorithmen (die Komposition, die Parallelanwendung, die Verzweigung und die Iteration) kennengelernt und für sie formale Bezeichnungen eingeführt: „$\mathfrak{A} \circ \mathfrak{B}$", „$\mathfrak{A} \parallel \mathfrak{B}$", „if Φ then $\mathfrak{A}$ else $\mathfrak{B}$", „while Φ do $\mathfrak{A}$". Es liegt auf der Hand, daß man diese Operationen beliebig superponieren kann und diese Superpositionen durch bestimmte

Terme, wie

$$\text{„while } \Phi_2 \text{ do (if } \Phi_1 \text{ then } \mathfrak{A}_1 \text{ else } \mathfrak{A}_2)) \circ \mathfrak{A}_3\text{“}, \tag{1}$$

beschreiben kann. Wir wollen die syntaktische Struktur dieser Termsprache hier nicht genauer analysieren und uns darauf beschränken, ihren Gebrauch an Beispielen zu demonstrieren.

So sieht man zunächst leicht, daß der Term (1), wenn man die in ihm auftretenden Algorithmen Φ_1, Φ_2, $\mathfrak{A}_1$, $\mathfrak{A}_2$, $\mathfrak{A}_3$ mit der in Abschnitt 2.4.4 erklärten Bedeutung versieht, in Kurzform die übliche verbale Beschreibung des Euklidischen Algorithmus aus Abschnitt 1.1.1 wiedergibt. Andererseits können wir aufgrund von (1), wenn wir schon über Turing-Programme für die Algorithmen Φ_1, Φ_2, $\mathfrak{A}_1$, $\mathfrak{A}_2$, $\mathfrak{A}_1$ verfügen (und diese können ohne Mühe aufgestellt werden), durch rein formale Anwendung der programmierenden Algorithmen für die Komposition, Verzweigung und Iteration ein Turing-Programm für den Euklidischen Algorithmus erzeugen. Dieses wird natürlich wesentlich umfangreicher als das Funktionsschema der Abb. 14. Jedoch erfordert das hier geschilderte Verfahren keine individuellen Kunstgriffe.

Die vier programmierenden Algorithmen für die Grundverknüpfungen unserer „Programmiersprache“ können leicht zu einem Algorithmus zusammengeschlossen werden, nachdem man zu jedem Term der Programmiersprache ein ihm entsprechendes Turing-Programm effektiv aufstellen kann, vorausgesetzt natürlich, daß man über Turing-Programme für die in den Term eingehenden elementaren Algorithmen verfügt. Der Term beschreibt gerade in effektiver Weise die Reihenfolge, in der man die Grundverknüpfungen anwenden muß, um aus den Turing-Programmen für die Teilalgorithmen ein Turing-Programm für den Gesamtalgorithmus zu erhalten.

2.5. Rekursive Funktionen und Turing-berechenbare Funktionen

2.5.1. Die Berechnung von Funktionen und algorithmische Probleme

In diesem Paragraphen wollen wir ein- und mehrstellige arithmetische Funktionen behandeln, d. h. Funktionen von einer oder mehreren Veränderlichen, wobei vorausgesetzt wird, daß sowohl die Argumente als auch die Werte der betrachteten Funktionen natürliche Zahlen sind. Eine arithmetische Funktion f nennen wir *Turing-berechenbar*, wenn es eine Turing-Maschine gibt, die sie berechnet. Natürlich muß eine exakte Definition der Turing-berechenbaren Funktionen noch eine Anweisung darüber enthalten, in welcher Weise die Argumentwerte und der gesuchte Funktionswert auf dem Band zu kodieren sind. Man könnte insbesondere dafür die unäre Darstellung der natürlichen Zahlen (die Zahl x wird durch x Striche repräsentiert) oder irgendein anderes Positionssystem (z. B. das Dual- oder das Dezimalsystem) verwenden. Es ist jedoch klar, daß die Klasse der Turing-berechenbaren Funktionen nicht davon abhängt, welche der genannten Zahldarstellungen tatsäch-

lich verwendet wird, ob also z. B. die unäre oder die dezimale Darstellung benutzt wird. Wir können nämlich Turing-Programme angeben, die die unäre Darstellung in die dezimale Darstellung und die dezimale Darstellung in die unäre Darstellung überführen (vgl. Abschnitt 2.3.2). Daher können wir, sobald ein Turing-Programm $\mathfrak{A}$ zur Berechnung einer Funktion f bei unärer Zahldarstellung aufgestellt ist, dieses in ein Programm $\mathfrak{B}$ zur Berechnung derselben Funktion unter Verwendung der Dezimaldarstellung (effektiv) umwandeln. Dafür genügt es, $\mathfrak{B}$ als Komposition dreier Programme darzustellen: eines Programms für die Übersetzung der Argumente vom Dezimalsystem in ihre unäre Darstellung, des Programms $\mathfrak{A}$ und eines Programms für die Übersetzung des Funktionswerts von der unären Darstellung in das Dezimalsystem. (Hier drängt sich ein Vergleich mit den elektronischen Rechenautomaten auf, die intern im Dualsystem arbeiten, in denen es aber eine Einrichtung oder ein Programm für die Übersetzung der Anfangswerte aus dem Dezimal- in das Dualsystem gibt und auch eine Einrichtung oder ein Programm, das für die Rückübersetzung des Resultats aus dem Dualsystem in das Dezimalsystem sorgt.) Was schließlich die Zahlenpaare, -tripel usw. betrifft, so werden diese auf dem Band in der Form $x * y$, $x * y * z$ usw. dargestellt. Dabei sind x, y, z die Darstellungen der entsprechenden Zahlen in einem bestimmten System, und „$*$" ist ein von den als Symbole zur Zahldarstellung benutzten Zeichen verschiedenes Zeichen (Trennzeichen).

Es ist nützlich, folgendes zu bemerken: Obwohl wir absichtlich unsere besondere Aufmerksamkeit der Berechnung von arithmetischen Funktionen widmen, bedeutet das keine Beschränkung der Natur der betrachteten algorithmischen Probleme, die auf Turing-Maschinen lösbar sind. Das liegt daran, daß bei einem festen Alphabet A aus r Buchstaben jedes Wort R in diesem Alphabet als Niederschrift einer gewissen natürlichen Zahl im r-adischen System angesehen werden kann. Das erlaubt es, die Anfangsdaten, die durch das Wort R gegeben sind, als eine zu Beginn gegebene natürliche Zahl zu interpretieren.[1]) Da man genau dieselbe Bemerkung auch für die Wörter machen kann, die die Resultate darstellen, ist klar, daß man jeden Algorithmus zur Lösung einer nichtnumerischen Aufgabe als Algorithmus zur Berechnung einer gewissen arithmetischen Funktion interpretieren kann. Eine ähnliche Art von arithmetischer Interpretation (oder kurz Arithmetisierung) wird oft in der mathematischen Logik und in der Algorithmentheorie zur Herleitung wichtiger Sätze mit großem Erfolg verwendet; wir werden ihr im vorliegenden Abschnitt noch begegnen.

Schon im vorangehenden Abschnitt wurden Turing-Programme zur Berechnung der Summe $x + y$, des Produktes $x \cdot y$, usw. aufgestellt, oder es wurde erklärt,

[1]) Bei realen Rechenautomaten spricht man z. B. davon, daß „im Speicher des Automaten 1000 24-stellige Dualzahlen gespeichert werden können", usw. Es ist klar, daß unter Zahlen hier eigentlich Dualwörter der entsprechenden Länge zu verstehen sind. Für den Automaten ist es dabei ganz gleichgültig, ob diese Wörter als Dualdarstellung von Zahlen oder als Niederschrift einer nichtnumerischen Information interpretiert werden.

wie man sie aufstellen kann. Ohne Mühe können wir diese Liste beliebig erweitern und Programme für viele andere, häufig anzutreffende arithmetische Funktionen aufstellen (z. B. für die einstelligen Funktionen $x^2, x^3, \ldots, 2^x, x!, \ldots$, für die zweistellige Funktion x^y, für die dreistellige Funktion x^{y^z}, usw.). Für eine genauere Klärung der Verhältnisse, welche Funktionen Turing-berechenbar sind (und wie die entsprechenden Programme aufgebaut werden), ist es jedoch zweckmäßiger, in folgender Weise vorzugehen: Wir betrachten zunächst eine Reihe von natürlichen und sehr bekannten Operatoren, d. h. Methoden zur Erzeugung von neuen Funktionen aus gegebenen Funktionen, und versuchen zu ergründen, ob es möglich ist, Programme für die neuen Funktionen aufzustellen, sobald schon Programme für die Ausgangsfunktionen vorliegen.

Unter Benutzung der im vorigen Abschnitt beschriebenen Programmiersprache und der programmierenden Algorithmen stellen wir verhältnismäßig leicht fest, daß es in einer Reihe wichtiger Fälle tatsächlich gelingt, solche Programme aufzustellen. Dabei wird sich zeigen, daß die Klasse der Turing-berechenbaren Funktionen recht umfangreich ist und alle Funktionen umfaßt, die durch eine sogenannte induktive Definition (beruhend auf dem Übergang von n zu $n + 1$) beschrieben werden können. Eine solche Definition bezeichnet man auch als *rekurrente* oder *rekursive Definition*, was wörtlich soviel wie „zurücklaufende Definition" bedeutet (vom lateinischen Wort *recurro* = ich laufe zurück, bzw. *recurso* = ich kehre zurück). Weil der Aufbau der natürlichen Zahlen selbst rekursiv ist, d. h. rückläufigen Charakter trägt (um die Zahl 4 zu definieren, muß man erst die Zahl 3 definieren; hierfür ist es aber nötig, zunächst die Zahl 2 zu definieren usw. bis zur 1 bzw. 0), ist auch die Definition von arithmetischen Funktionen mit Hilfe eines „Zurückgehens" vom Unbekannten zum Bekannten ganz natürlich. Viele der in der Arithmetik studierten Funktionen werden rekursiv definiert oder können zumindest so definiert werden: die Summe, die Potenz, das Produkt, die Anzahl aller Kombinationen von k Elementen aus n Elementen, die Anzahl aller Teiler einer Zahl n, usw. Es gibt verschiedene mögliche Varianten rekursiver Definitionen, die jedoch alle in natürlicher Weise in der Definition der *rekursiven Funktion* (genauer gesagt, der *allgemein-* oder *partiell-rekursiven Funktion*, je nachdem, ob die Funktion als überall definiert vorausgesetzt wird oder nicht) zusammengefaßt werden. Wir werden zeigen, daß die Klasse der Turing-berechenbaren Funktionen mit der Klasse der rekursiven Funktionen übereinstimmt.

Im folgenden bezeichnet $\mathfrak{A}_f$ stets ein Turing-Programm, das die Funktion f berechnet.

Unser erstes Ziel ist es, drei Typen von Operatoren zu beschreiben und zu zeigen, daß die Klasse der berechenbaren Funktionen in bezug auf diese Operatoren abgeschlossen ist. Anders ausgedrückt: Wenn wir über Programme $\mathfrak{A}_{f_1}, \mathfrak{A}_{f_2}, \ldots$ für Funktionen $f_1, f_2, \ldots$ verfügen und die Funktion f durch Anwendung irgendwelcher der angegebenen Operatoren aus den Funktionen $f_1, f_2, \ldots$ erhalten werden kann, so kann auch ein Programm $\mathfrak{A}_f$ aufgestellt werden.

2.5.2.* Die elementaren Operatoren

Zu den elementaren Operatoren gehören insbesondere die Operatoren der Superposition und der Einführung fiktiver Argumente. Obwohl wir hier diese Operatoren nur für Funktionen betrachten, deren Argumente und Werte natürliche Zahlen sind, haben sie auch für Funktionen mit beliebigen Argument- und Wertebereichen einen Sinn.

Durch die *Einführung von fiktiven Argumenten* wird eine gegebene Funktion f in eine Funktion h größerer Stellenzahl übergeführt. Daß die neu hinzugenommenen Argumente *fiktiv* (unwesentlich) sind bedeutet dabei, daß sich bei alleiniger Änderung dieser Argumente der Funktionswert nicht ändert (vorausgesetzt, daß die Werte der „wesentlichen" Argumente festgehalten werden). So kann man z. B. aus einer einstelligen Funktion f eine zweistellige Funktion h dadurch erhalten, daß man $h(x, y) = f(x)$ setzt. Natürlich kann man gleichzeitig auch mehrere fiktive Argumente einführen, indem man z. B.

$$h(x, y, u, v) = f(y, v)$$

setzt. In allen diesen Fällen ist die Konstruktion von $\mathfrak{A}_h$ aus $\mathfrak{A}_f$ sehr einfach. Ist z. B.

$$h(x, y) = f(x),$$

so löscht $\mathfrak{A}_h$ in dem gegebenen Paar $x * y$ zunächst das Teilwort $* y$, geht dann auf das erste Zeichen von x und arbeitet anschließend weiter wie $\mathfrak{A}_f$.

Häufig werden Funktionen, die sich nur durch fiktive Argumente voneinander unterscheiden, miteinander identifiziert. In anderen Fällen ist ihre Unterscheidung jedoch nicht nur nützlich, sondern sogar notwendig. Man kann insbesondere, wie bald ersichtlich sein wird, durch die Einführung von fiktiven Argumenten eine größere Geschlossenheit in den Programm-Prozeduren erreichen.

Eine andere Möglichkeit, aus gegebenen Funktionen neue Funktionen (sogenannte *zusammengesetzte Funktionen*) zu erzeugen, besteht darin, daß man für gewisse Argumente einer Funktion Werte anderer Funktionen einsetzt. Das ist gerade die Wirkungsweise eines sogenannten *Superpositionsoperators*.

Wir betrachten als Beispiel die einstellige Funktion φ, die durch folgende Superposition aus den einstelligen Funktionen f und g definiert wird:

$$\varphi(x) = g(f(x)).$$

Dann beschreibt der Term „$\mathfrak{A}_f \circ \mathfrak{A}_g$" der in Abschnitt 2.4.6 beschriebenen Programmiersprache offenbar einen Algorithmus zur Berechnung der Funktion φ, und aus ihm kann mit Hilfe des programmierenden Algorithmus für die Komposition ein Turing-Programm $\mathfrak{A}_\varphi$ gewonnen werden. Analog, wenn auch etwas komplizierter,

erhält man ein Turing-Programm im Fall der Superposition mehrstelliger Funktionen. Es sei z. B. die zweistellige Funktion $\varphi(x_1, x_2)$ als die Superposition

$$g\big(f_1(x_1, x_2), f_2(x_1, x_2), f_3(x_1, x_2)\big)$$

definiert. Dann kann $\mathfrak{A}_\varphi$ nach dem Term

„Kop2 $\circ$ ($\mathfrak{A}_{f_1} \parallel \mathfrak{A}_{f_2} \parallel \mathfrak{A}_{f_3}$) $\circ$ $\mathrm{Ers}_*{}^{\#} \circ \mathfrak{A}_g$“

effektiv konstruiert werden. In der Tat: Ausgehend vom Wort $x_1 * x_2$ erzeugt der Algorithmus Kop2 durch zweimaliges Kopieren das Wort $x_1 * x_2 \# x_1 * x_2 \# x_1 * x_2$, welches durch die Parallelanwendung $\mathfrak{A}_{f_1} \parallel \mathfrak{A}_{f_2} \parallel \mathfrak{A}_{f_3}$ in das Wort

$$f_1(x_1, x_2) \# f_2(x_1, x_2) \# f_3(x_1, x_2)$$

übergeführt wird. Für die anschließende Anwendung des Algorithmus $\mathfrak{A}_g$ ist es vorher noch nötig, das Symbol „#“ in das Trennzeichen „*“ überzuführen, und das leistet gerade der Algorithmus $\mathrm{Ers}_*{}^{\#}$.

Natürlich ist es bei einer Superposition keineswegs nötig, daß die Funktionen $f_1, f_2, \ldots$ (die inneren Funktionen der Superposition) von denselben Argumenten abhängen oder auch nur dieselbe Stellenzahl besitzen. Es sei z. B.

$$\varphi(x, y, u, z) = g\big(f_1(x, y), f_2(x), f_3(x, u, z)\big).$$

Dann könnten wir ein Programm $\mathfrak{A}_\varphi$ nach einer Methode aufbauen, die der oben angeführten ähnlich ist. Jedoch ist es hier bequemer, anders vorzugehen: Zunächst stellen wir Programme $\mathfrak{A}_{h_1}$, $\mathfrak{A}_{h_2}$, $\mathfrak{A}_{h_3}$ für die Funktionen

$$h_1(x, y, u, z) = f_1(x, y), \quad h_2(x, y, u, z) = f_2(x), \quad h_3(x, y, u, z) = f_3(x, u, z)$$

auf, die wir durch Einführung von fiktiven Argumenten erhalten, und danach konstruieren wir das Programm $\mathfrak{A}_\varphi$ als „Standard“-Superposition

$$\varphi(x, y, u, z) = g\big(h_1(x, y, u, z), h_2(x, y, u, z), h_3(x, y, u, z)\big).$$

Bemerkung. Bei der Notierung zweistelliger Funktionen und ihrer Superpositionen wird häufig eine sogenannte *Infix-Darstellung* verwendet, bei der das Funktionssymbol nicht vor die Argumente gestellt wird (das ist die sogenannte *Präfix-Darstellung*), sondern zwischen diese. Die Infix-Darstellung ist z. B. bei den arithmetischen Funktionen $x + y$, $x \cdot y$, usw. üblich. Ihre Präfixdarstellungen würden etwa so aussehen: $+(x, y)$, $\cdot(x, y)$, usw. Entsprechend sind gewohnte Terme der Form $(x + y) \cdot (u + v)$ oder $(x - uy) : (uz + v)$ nichts anderes als Infix-Darstellungen der Superpositionen $\cdot\big(+(x, y), +(u, v)\big)$ und $:\big(-(x, \cdot(u, y)), +(\cdot(u, z), v)\big)$.

2.5.3.* Die primitive Rekursion

Wir kommen nun zur Betrachtung von Operatoren, bei denen die Definition durch Rekursion besonders klar zum Ausdruck kommt.

Zunächst betrachten wir eine vergleichsweise einfache Form der primitiven Rekursion, die sogenannte *Iteration*. Vorgegeben seien eine einstellige Funktion f und eine feste natürliche Zahl c. Dann kann man die Zahlen $c, f(c), f(f(c)), f(f(f(c))), \ldots$ als Werte $\varphi(0), \varphi(1), \varphi(2), \varphi(3), \ldots$ einer gewissen neuen Funktion ansehen. Ihre allgemeine Definition kann mit Hilfe der folgenden beiden Gleichungen formuliert werden:

$$\varphi(0) = c \text{ (Induktionsanfang)}; \quad \varphi(x+1) = f(\varphi(x)) \text{ (Induktionsschritt)}.$$

Die erste Gleichung gibt den Anfangswert der Funktion φ an. Die zweite ist eine „zurückgehende" oder, wie man auch sagt, rekurrente Beziehung, sie erlaubt es, den gesuchten, unbekannten Wert $\varphi(x+1)$ aus dem zuvor berechneten Wert $\varphi(x)$ zu ermitteln. Wenn z. B. als Funktion $f(x)$ die lineare Funktion $2x$ und als Konstante c die Zahl 1 genommen wird, so erhalten wir die Gleichungen

$$\varphi(0) = 1; \quad \varphi(x+1) = 2\varphi(x),$$

die offenbar die Exponentialfunktion 2^x definieren. Diese Methode zur Definition einer Funktion φ heißt *die Iteration der Funktion f mit dem Anfangswert c.*

Bei der Aufstellung des Programms $\mathfrak{A}_\varphi$ unter Verwendung von $\mathfrak{A}_f$ gehen wir von der Tatsache aus, daß sich die Berechnung des Wertes $\varphi(x)$ darauf reduziert, den Algorithmus $\mathfrak{A}_f$ wiederholt anzuwenden und diese Wiederholungen so lange zu zählen, bis die Anzahl x erreicht ist. Zur Realisierung dessen betrachten wir die folgenden Algorithmen:

1. den Algorithmen $\mathfrak{D}$, der das Zahlentripel $x \# m \# z$ in das Tripel $x \# m+1 \# f(z)$ überführt;

2. den Algorithmus $\mathfrak{L}$, der die Zahl x in das Tripel $x \# 0 \# c$ umwandelt;

3. den Algorithmus Φ, der entscheidet, ob im Tripel $x \# m \# z$ die zweite Komponente m kleiner als die erste Komponente x ist.

Dann gibt der Term „$\mathfrak{L} \circ$ (while Φ do $\mathfrak{D}$)" einen Algorithmus an, der, von x ausgehend, der Reihe nach $x \# 0 \# c$, $x \# 1 \# f(c)$, $x \# 2 \# f(f(c))$, usw. erzeugt, bis schließlich das Tripel $x \# x \# \varphi(x)$ erhalten wird. Folglich liefert der Term „$\mathfrak{L} \circ$ (while Φ do $\mathfrak{D}$) $\circ$ Res", wobei Res ein Standard-Programm ist, das die dritte Komponente eines Tripels abspaltet (vgl. Abschnitt 2.4.1), den gewünschten Algorithmus, der die Funktion φ berechnet. Zur Aufstellung des Programms $\mathfrak{A}_\varphi$ ist es also nur noch nötig, Programme für die Algorithmen $\mathfrak{L}$, Φ, $\mathfrak{D}$ zu schreiben, was aber keine Mühe bereitet. So wird z. B. der Algorithmus $\mathfrak{D}$ durch den Term „Id $\|$ S $\|$ $\mathfrak{A}_f$" beschrieben, wobei S ein Programm ist, das die Funktion $s(x) = x+1$ berechnet.

Damit ist gezeigt, daß die Klasse der berechenbaren Funktionen in bezug auf Iterationen abgeschlossen ist. Wir bemerken, daß, obwohl die Iteration äußerlich gewisse Ähnlichkeiten mit der mehrfachen Superposition aufweist, hier eine wesentlich andere Situation vorliegt. Insbesondere ist die Iteration besser zur Erzeugung von schnell wachsenden Funktionen geeignet als die mehrfache Anwendung der Superposition. So kann man z. B. aus der Funktion $f(x) = 2^x$ durch Superposition der Reihe nach die Funktionen $f(f(x)) = 2^{2^x}$, $f(f(f(x))) = 2^{2^{2^x}}$, usw. erhalten. Demgegenüber liefert die einmalige Anwendung der Iteration

$$\varphi(0) = 1, \quad \varphi(x+1) = f(\varphi(x)) \quad \text{(d. h. } \varphi(x+1) = 2^{\varphi(x)}\text{)}$$

bereits die Funktion

$$\varphi(x) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \Big\} x \text{ Etagen,}$$

die erstaunlich schnell wächst und schließlich jede der Funktionen 2^x, 2^{2^x}, $2^{2^{2^x}}$, usw. übertrifft.

Die Iteration ist ein Spezialfall eines allgemeineren Schemas der Definition einer Funktion φ durch Rekursionsgleichungen, das als *Schema der primitiven Rekursion* bezeichnet wird. Eine einstellige Funktion φ wird durch primitive Rekursion, ausgehend von einer vorgegebenen zweistelligen Funktion f und einer Konstanten c, mit Hilfe der folgenden Gleichungen definiert:

$$\varphi(0) = c, \quad \varphi(x+1) = f(x, \varphi(x)).$$

Beispiel. Für $c = 1$ und $f(x, y) = (x+1) \cdot y$ erhalten wir die folgende primitive Rekursion

$$\varphi(0) = 1, \ \varphi(x+1) = (x+1) \cdot \varphi(x).$$

Hieraus ist ersichtlich, daß $\varphi(1) = 1 \cdot 1 = 1$, $\varphi(2) = 2 \cdot 1 = 2$ usw. gilt. Allgemein gilt:

$$\varphi(x) = x \cdot (x-1) \cdot (x-2) \cdots 2 \cdot 1 = x!.$$

Natürlich kann eines der Argumente von f fiktiv sein. Ist beispielsweise $f(x, y) = q(y)$, so nimmt das Schema der primitiven Rekursion die Form

$$\varphi(0) = c, \quad \varphi(x+1) = q(\varphi(x))$$

an und definiert φ als Iteration der Funktion q. Ist dagegen $f(x, y) = r(x)$, so nimmt das Schema die Form

$$\varphi(0) = c, \quad \varphi(x+1) = r(x) \tag{1}$$

an. In diesem Fall der primitiven Rekursion fehlt der eigentliche Prozeß des Zurückgehens von $\varphi(x+1)$ auf $\varphi(x)$.

Bei der rekursiven Definition einer zweistelligen Funktion φ geht man von einer einstelligen Funktion g und einer dreistelligen Funktion f aus; sie erfolgt nach dem Rekursionsschema

$$\varphi(0, n) = g(n), \quad \varphi(x + 1, n) = f\big(x, \varphi(x, n), n\big).$$

Allgemein wird eine k-stellige Funktion φ durch Rekursionsgleichungen der Form

$$\begin{aligned} &\varphi(0, n_1, n_2, \ldots, n_{k-1}) = g(n_1, n_2, \ldots, n_{k-1}), \\ &\varphi(x + 1, n_1, n_2, \ldots, n_{k-1}) = f\big(x, \varphi(x, n_1, \ldots, n_{k-1}), n_1, \ldots, n_{k-1}\big) \end{aligned} \tag{2}$$

definiert; dabei sind g und f vorgegebene Funktionen von $k - 1$ bzw. $k + 1$ Veränderlichen. Im Schema (2) wird die Rekursion über das erste Argument von φ geführt, die übrigen Argumente dienen als Parameter.

Beispiel. Es sei $g(n) = n$, $f(x, y, n) = x \cdot y \cdot n$. Hier liefert das Schema

$$\varphi(0, n) = n, \quad \varphi(x + 1, n) = (x + 1) \cdot \varphi(x, n) \cdot n$$

der Reihe nach die Werte

$$\varphi(1, n) = 1 \cdot n \cdot n = n^2, \ \varphi(2, n) = 2 \cdot n^2 \cdot n = 2n^3,$$

$$\varphi(3, n) = 3 \cdot 2n^3 \cdot n = 6n^4$$

und allgemein $\varphi(x, n) = x!n^{x+1}$.

Auf nur geringfügig kompliziertere Weise kann man mit den Konstruktionen und Überlegungen, die wir oben bei der Betrachtung der Iteration angewandt haben, einen Term der Programmiersprache für einen die Funktion φ berechnenden Algorithmus erhalten und nach diesem ein Turing-Programm $\mathfrak{A}_\varphi$ aufstellen, das von Programmen $\mathfrak{A}_f$, $\mathfrak{A}_g$ und einigen einfachen Standard-Programmen ausgeht.

Für eine einstellige Funktion φ hat der Term dieselbe Form wie bei der Iteration, d. h. „$\mathfrak{L} \circ (\text{while } \Phi \text{ do } \mathfrak{D}) \circ \text{Res}$“, mit dem Unterschied, daß jetzt $\mathfrak{D}$ ein Algorithmus ist, der das Tripel $x \# m \# z$ in das Tripel $x \# m + 1 \# f(m, z)$ überführt. Der Term für $\mathfrak{D}$ wird etwas komplizierter als früher, nämlich „$\text{Kop} \circ (\text{Id} \parallel \mathfrak{S} \parallel \mathfrak{A}_f)$“, dabei ist Kop ein Kopieralgorithmus, der das Tripel $x \# m \# z$ in das Tripel $x \# m \# m * z$ überführt.

Analog kann man das Rekursionsschema (2) für eine zweistellige oder mehrstellige Funktion φ betrachten und in der Programmiersprache einen Term für das Programm $\mathfrak{A}_\varphi$ aufstellen.

Wenn einige Argumente der Funktionen f und g fiktiv sind, kann man das Schema (2) manchmal in „nichtkanonischer“ Form angeben, z. B.

$$\varphi(0, n) = g(n), \quad \varphi(x + 1, n) = q\big(\varphi(x, n)\big) \tag{3}$$

oder

$$\varphi(0, n_1, n_2) = r(n_2), \quad \varphi(x + 1, n_1, n_2) = q\big(x, \varphi(x, n_1, n_2)\big). \tag{4}$$

Wir geben einige Beispiele für Fälle an, in denen gewisse Argumente der Ausgangsfunktionen g, f fiktiv sind:

Es sei $g(n) = n$, $q(y) = 1 + y$. Hier liefert das Schema (3)

$$\varphi(0, n) = n, \quad \varphi(x + 1, n) = 1 + \varphi(x, n)$$

der Reihe nach die Werte

$$\varphi(1, n) = 1 + n, \quad \varphi(2, n) = 1 + (1 + n) = 2 + n,$$
$$\varphi(3, n) = 1 + (2 + n) = 3 + n,$$

und allgemein $\varphi(x, n) = x + n$, d. h., das betrachtete Schema definiert also, ausgehend von den Funktionen n und $1 + y$, die Addition.

Mit $g(n) = 0$ und $f(x, y, n) = y + n$ erhalten wir sodann das folgende Schema

$$\varphi(0, n) = 0, \quad \varphi(x + 1, n) = \varphi(x, n) + n,$$

durch das die Multiplikation $x \cdot n$ definiert wird. Es wird nämlich

$$\varphi(1, n) = 0 + n = n, \quad \varphi(2, n) = n + n = 2n, \quad \varphi(3, n) = 2n + n, \text{ usw.}$$

Setzen wir $g(n) = 1$ und $f(x, y, n) = y \cdot n$, so erhalten wir das Schema

$$\varphi(0, n) = 1, \quad \varphi(x + 1, n) = \varphi(x, n) \cdot n,$$

durch das die Potenzierung n^x definiert wird. In der Tat ist

$$\varphi(1, n) = 1 \cdot n = n^1, \quad \varphi(2, n) = n \cdot n = n^2, \quad \varphi(3, n) = n^2 \cdot n = n^3, \text{ usw.}$$

Beliebige, „nichtkanonische" Schemata dieser Art kann man in „kanonische" Schemata der Form (2) umwandeln, indem man fiktive Argumente einführt. So nimmt nach Anwendung der Operatoren $g(n_1, n_2) = r(n_2)$, $f(x, y, n_1, n_2) = q(x, y)$ das Schema (4) die Form (2) an. Da die Klasse der Turing-berechenbaren Funktionen in bezug auf die Operatoren der Einführung von fiktiven Argumenten abgeschlossen ist, ist damit auch die Abgeschlossenheit dieser Klasse in bezug auf primitive Rekursionen in „nichtkanonischer" Form nachgewiesen (d. h., wenn auf den rechten Seiten der Gleichungen (2) einige Argumente fehlen).

Ein noch allgemeinerer Fall als (2) besteht darin, daß mehrere Funktionen gleichzeitig (simultan) in die induktive Definition eingehen. Für zwei einstellige Funktionen φ_1 und φ_2 sieht das *Schema der simultanen primitiven Rekursion* folgendermaßen aus:

$$\varphi_1(0) = c_1, \quad \varphi_1(x + 1) = f_1\big(x, \varphi_1(x), \varphi_2(x)\big),$$
$$\varphi_2(0) = c_2, \quad \varphi_2(x + 1) = f_2\big(x, \varphi_1(x), \varphi_2(x)\big).$$

Nach diesem Schema kann aus dem gegebenen Wertepaar $\varphi_1(0)$, $\varphi_2(0)$ zunächst das Wertepaar $\varphi_1(1)$, $\varphi_2(1)$, aus diesem sodann das Wertepaar $\varphi_1(2)$, $\varphi_2(2)$, usw.

berechnet werden. Für zweistellige Funktionen hat das Schema der simultanen Rekursion die Form

$$\varphi_1(0, n) = g_1(n), \quad \varphi_1(x+1, n) = f_1(x, \varphi_1(x, n), \varphi_2(x, n), n),$$
$$\varphi_2(0, n) = g_2(n), \quad \varphi_2(x+1, n) = f_2(x, \varphi_1(x, n), \varphi_2(x, n), n),$$

und analog für eine beliebige Anzahl von Funktionen mit beliebig vielen Argumenten.

Als Beispiel betrachten wir die simultane Definition der Funktionen $\left[\frac{x}{n}\right]$ (der ganzzahlige Teil bei der Division von x durch n) und $Rest(x, n)$ (der Rest bei der Division von x durch n). Für $n = 0$ sind diese Funktionen zunächst nicht definiert; wir setzen sie fort, indem wir festlegen, daß sie für $n = 0$ den Wert 0 annehmen sollen. Wir nehmen an, daß die Werte $\left[\frac{x}{n}\right]$ und $Rest(x, n)$ schon bekannt sind. Dann gilt beim Übergang zu $x+1$ folgendes:

1. Ist $Rest(x, n) = n - 1$, so wird der ganzzahlige Anteil um 1 erhöht, und der Rest wird 0.

2. Ist das nicht der Fall, d. h., ist $Rest(x, n) < n - 1$, so bleibt der ganzzahlige Teil ungeändert, und der Rest wird um 1 erhöht.

Dieser verbalen Beschreibung kann man die Form einer simultanen primitiven Rekursion geben, was nun ausgeführt werden soll. Zur Vorbereitung führen wir zwei Bezeichnungen ein. Es sei

$$Gl(\xi, \eta) = \begin{cases} 1, & \text{falls } \xi = \eta, \\ 0 & \text{sonst,} \end{cases} \qquad Kl(\xi, \eta) = \begin{cases} 1, & \text{falls } \xi < \eta, \\ 0 & \text{sonst.} \end{cases}$$

Das sind sehr einfache Funktionen, die ganz offensichtlich Turing-berechenbar sind, wir werden sie noch mehrfach verwenden. Das gesuchte Schema für das Paar $\left[\frac{x}{n}\right]$, $Rest(x, n)$ hat dann die Form:

$$\left[\frac{0}{n}\right] = 0, \quad \left[\frac{x+1}{n}\right] = \left[\frac{x}{n}\right] + Gl\big(Rest(x, n) + 1, n\big),$$

$$Rest(0, n) = 0, \quad Rest(x+1, n) = \big(Rest(x, n) + 1\big) \cdot Kl\big(Rest(x, n) + 1, n\big).$$

Die simultane primitive Rekursion für m Funktionen von k Argumenten kann als eine einfache primitive Rekursion für eine Funktion von k Argumenten aufgefaßt werden, deren Werte m-dimensionale Vektoren mit natürlichen Koordinaten sind. Daher unterscheidet sich das Aufstellen eines entsprechenden Turing-Programms (ausgehend von Programmen $\mathfrak{A}_{g_1}, \mathfrak{A}_{g_2}, \ldots, \mathfrak{A}_{f_1}, \mathfrak{A}_{f_2}, \ldots$) im Prinzip nicht von dem im Fall der gewöhnlichen primitiven Rekursion.

2.5.4.* Der μ-Operator

Die primitive Rekursion unterscheidet sich von der Superposition und der Einführung fiktiver Argumente dadurch, daß sie wesentlich eine Besonderheit der natürlichen Zahlen ausnutzt, die Möglichkeit des induktiven Überganges von x zu $x + 1$. Eine andere Besonderheit besteht darin, daß die Turing-Programmierung der primitiven Rekursion auf einer wiederholten (man kann auch sagen zyklischen) Anwendung des Ausgangsprogramms beruht. Hierbei ist die Anzahl der Wiederholungen vorher bekannt (und vorgegeben). Man muß nämlich für die Berechnung von $\varphi(x)$ oder $\varphi(x, n)$ den Ausgangsalgorithmus gerade x mal wiederholen. Jetzt betrachten wir eine Situation, in der die Berechnung einer neu zu definierenden Funktion zwar auch auf der wiederholten Anwendung eines gewissen Ausgangsalgorithmus beruht, jedoch ohne vorherige Information über die Anzahl der Wiederholungen. Es sei $f(x, y)$ ein Prädikat, d. h. eine Funktion, die nur die beiden Werte 0, 1 annimmt. Wir definieren eine Funktion $\varphi(x)$ durch die folgende Bedingung: Für ein beliebiges festes x_0, für das wenigstens ein Wert existiert, so daß $f(x_0, y) = 1$ gilt (d. h. das Prädikat wahr ist), sei $\varphi(x_0)$ die kleinste der Zahlen y, für die $f(x_0, y) = 1$ gilt. Wenn dagegen ein solches y nicht existiert, so sei $\varphi(x_0)$ nicht definiert. Hierin besteht gerade die Anwendung des sogenannten *μ-Operators* auf die Funktion $f(x, y)$; symbolisch wird das folgendermaßen ausgedrückt:

$$\varphi(x) = \mu y[f(x, y) = 1]$$

(gelesen: $\varphi(x)$ ist das kleinste y mit $f(x, y) = 1$).[1])

Wir nehmen nun an, daß wir über ein Programm $\overline{\mathfrak{A}}_f$ verfügen, das auf alle „Paare" $x \# y$ anwendbar ist und für das folgendes gilt:

$$\overline{\mathfrak{A}}_f(x \# y) = \begin{cases} 0, & \text{falls } f(x, y) = 1, \\ 1, & \text{falls } f(x, y) = 0 \end{cases}$$

(ein solches Programm läßt sich mittels $\mathfrak{A}_f$ leicht konstruieren). Zur Bestimmung des Wertes $\varphi(x_0)$ bietet sich dann der folgende Algorithmus an: Wir prüfen nacheinander die Bedingung $\overline{\mathfrak{A}}_f$ für die Paare $x_0 \# 0$, $x_0 \# 1$, $x_0 \# 2$, ..., und zwar so lange, bis zum ersten Mal ein y mit $f(x, y) = 1$ auftritt, falls ein solches y existiert; anderenfalls wird der Prozeß unbegrenzt fortgesetzt. Der entsprechende Term unserer Programmiersprache lautet: „$\mathfrak{L}$ o (while $\overline{\mathfrak{A}}_f$ do Id ‖ S)", dabei führe $\mathfrak{L}$ das Wort x in das Wort $x \# 0$ über. Hiernach kann man das gesuchte Programm $\mathfrak{A}_\varphi$ aufstellen, sobald das Programm $\mathfrak{A}_f$ gegeben ist. Analoge Verhältnisse liegen vor, wenn der

[1]) Während die zuvor betrachteten Operatoren arithmetische Funktionen, die überall definiert sind (man nennt sie auch *volle Funktionen*), stets in überall definierte Funktionen überführen, wird bei Anwendung des μ-Operators der Bereich der vollen Funktionen im allgemeinen verlassen. Man gelangt zu Funktionen, die nicht notwendig mehr überall definiert sind und die man *partielle Funktionen* nennt. [*Anm. d. Übers.*]

μ-Operator auf ein Prädikat $f(x_1, \ldots, x_k, y)$ einer beliebigen Stellenzahl $k+1$ angewendet wird, wodurch die Funktion

$$\varphi(x_1, \ldots, x_k) = \mu y[f(x_1, \ldots, x_k, y) = 1]$$

entsteht.

Wir nehmen nun an, daß zwei Turing-berechenbare Funktionen $g_1(x_1, \ldots, x_k, y)$ und $g_2(x_1, \ldots, x_k, y)$ vorgegeben sind. Unter Benutzung des Prädikats Gl kann man dann die Funktion

$$\varphi(x_1, \ldots, x_k) = \mu y\big[Gl\big(g_1(x_1, \ldots, x_k, y), g_2(x_1, \ldots, x_k, y)\big) = 1\big]$$

bilden, wofür wir auch kurz

$$\varphi(x_1, \ldots, x_k) = \mu y[g_1(x_1, \ldots, x_k, y) = g_2(x_1, \ldots, x_k, y)]$$

schreiben. Da die in den eckigen Klammern stehende Superposition ein Prädikat darstellt, für das mittels $\mathfrak{A}_{Gl}$, $\mathfrak{A}_{g_1}$, $\mathfrak{A}_{g_2}$ ein Turing-Programm aufgestellt werden kann, verfügen wir auch über $\mathfrak{A}_\varphi$. Analog liegen die Verhältnisse, wenn wir statt $g_1 = g_2$ die Relation $g_1 < g_2$ oder eine ähnliche betrachten. Der Unterschied besteht nur darin, daß statt des Prädikats $Gl(\xi, \eta)$ ein anderes zweistelliges Prädikat genommen werden muß (beispielsweise $Kl(\xi, \eta)$).

Eine typische Situation für die Anwendung des μ-Operators ist die Konstruktion von Umkehrfunktionen: Wir nehmen dazu an, daß zur Funktion $y = f(x)$ die Umkehrfunktion existiert, d. h. für jede natürliche Zahl y genau ein x mit $f(x) = y$ existiert; in diesem Fall kann die Umkehrfunktion $x = \varphi(y)$ offenbar mit Hilfe des μ-Operators als

$$\varphi(y) = \mu x[f(x) = y]$$

definiert werden. Viele Funktionen der Analysis, wie die Exponentialfunktion c^x und die Potenzfunktion x^c, haben, wenn man sie als Funktionen einer reellen Veränderlichen betrachtet und die Parameter (in unserem Fall den Parameter c) in geeigneter Weise wählt, die genannte Eigenschaft. Dabei sind jedoch die Umkehrfunktionen $\log_c y$ und $\sqrt[c]{y}$ (selbst für natürliches c), wenn man sie auf die natürlichen Zahlen einschränkt, keine arithmetischen Funktionen. In solchen Fällen ist es günstig, arithmetische Varianten der Umkehrfunktionen zu betrachten, nämlich als Funktionswert den ganzzahligen Anteil des tatsächlichen Wertes der Funktion, d. h. $[\varphi(y)]$, zu nehmen. Es ist klar, daß man für die Funktionen $y = c^x$ und $y = x^c$ die arithmetischen Umkehrfunktionen unter Benutzung des μ-Operators in der Form

$$[\log_r(y)] = \mu x[r^{x+1} > y], \quad \left[\sqrt[r]{y}\right] = \mu x[(x+1)^r > y]$$

schreiben kann.

2.5.5.* Die induktive Erzeugung von arithmetischen Funktionen und ihre Turing-Programmierung

In den vorangehenden Abschnitten wurden einige Operatoren ausführlich untersucht, mit deren Hilfe man aus vorgegebenen Funktionen (oder, wie man auch sagt, *Ausgangsfunktionen*) neue Funktionen erzeugen kann. Wir nehmen nun an, daß ein bestimmtes System Ω von Ausgangsfunktionen fixiert ist. Dann ist es sinnvoll, die Klasse Ω' aller derjenigen Funktionen φ zu betrachten, die man durch Anwendung der betrachteten Operatoren (in einer beliebigen Anzahl und Reihenfolge) aus den Funktionen aus Ω erhalten kann. Berücksichtigt man das Wesen dieser Operatoren, so ist klar, daß eine solche Erzeugung einer Funktion φ formal als System von Gleichungen geschrieben werden kann, in das die faktisch benötigten Schemata der Superposition, der Einführung fiktiver Argumente, der primitiven Rekursion und der Anwendung des μ-Operators eingehen.[1]) Wir wollen dieses Gleichungssystem eine *induktive Erzeugung* der Funktion φ nennen. Die genauere Analyse der in den vorangehenden Abschnitten diskutierten Beispiele zeigt, daß für alle dort betrachteten Funktionen eine induktive Erzeugung aus nur drei Ausgangsfunktionen möglich ist, nämlich

1. aus der *Nullfunktion* $O(x) = 0$,
2. aus der *Nachfolgerfunktion* $s(x) = x + 1$,
3. aus der *Vorgängerfunktion* $\bar{s}(x) = \begin{cases} x - 1 & \text{für } x > 0, \\ 0 & \text{für } x = 0. \end{cases}$

[1]) Hinsichtlich der Anwendung des μ-Operators ist dabei folgendes zu beachten: Man kann einmal annehmen, daß die Bildung von $\mu y[f(x_1, \ldots, x_k, y) = 1]$ nur dann erlaubt ist, wenn f eine volle 0-1-Funktion ist, bei der zu beliebigen $x_1, \ldots, x_k$ ein y mit $f(x_1, \ldots, x_k, y) = 1$ existiert. In diesem Fall ist auch $\varphi(x_1, \ldots, x_k) = \mu y[f(x_1, \ldots, x_k, y) = 1]$ eine volle Funktion. Daraus folgt, daß bei dieser eingeschränkten Anwendung des μ-Operators für eine Klasse Ω von vollen Funktionen die zugehörige Klasse Ω' ebenfalls nur aus vollen Funktionen besteht. Man nennt in diesem Fall die Funktionen aus Ω' die in bezug auf die Ausgangsfunktionen aus Ω *allgemein-rekursiven Funktionen*. Oder man läßt bei der Bildung von $\mu y[f(x_1, \ldots, x_k, y) = 1]$ beliebige (volle und partielle) 0-1-Funktionen zu, die nicht notwendig die angegebene Existenzbedingung erfüllen. Dann enthält die Klasse Ω' auch echt partielle Funktionen (selbst wenn Ω nur aus vollen Funktionen besteht). In diesem Fall nennt man die Funktionen aus Ω' die in bezug auf die Funktionen aus Ω *partiell-rekursiven Funktionen*. Die Anwendung des μ-Operators auf eine eventuell partielle 0-1-Funktion f ist dabei wie folgt definiert:

$$\mu y[f(x_1, \ldots, x_k, y) = 1] = y_0 \text{ genau dann, wenn}$$

$$f(x_1, \ldots, x_k, 0) = \cdots = f(x_1, \ldots, x_k, y_0 - 1) = 0,\ f(x_1, \ldots, x_k, y_0) = 1,$$

wobei diese Funktionswerte sämtlich existieren sollen, während die Funktion $\mu y[f(x_1, \ldots, x_k, y) = 1]$ im Fall der Nichtexistenz eines solchen y_0 für $x_1, \ldots, x_k$ als nicht definiert angesehen wird. Es ist also $\varphi(x_1, \ldots, x_k)$ z. B. dann nicht definiert, wenn $f(x_1, \ldots, x_k, 0) = 0$, $f(x_1, \ldots, x_k, 1)$ nicht definiert ist und $f(x_1, \ldots, x_k, 2) = 1$ ist. Die hier angegebene Variante des μ-Operators wird gerade durch die in Abschnitt 2.5.4 beschriebene Konstruktion von $\mathfrak{A}_\varphi$ geliefert, wenn durch das Programm $\mathfrak{A}_f$ eine beliebige, eventuell partielle 0-1-Funktion realisiert wird. [*Anm. d. Übers.*]

Diese Funktionen sind außerordentlich einfach und offensichtlich Turing-berechenbar. Wir bemerken, daß die Berechenbarkeit von $s(x)$ und $\bar{s}(x)$ dem Wesen nach die Fähigkeit ausdrückt, im Bereich der natürlichen Zahlen vor- und rückwärts zählen zu können.

Wir betrachten als Beispiel die Funktion $\varphi(y) = [\log_2 y]$. Sie kann als $\mu x[2^{x+1} > y]$ definiert werden, wobei die Exponentialfunktion $\psi(x) = 2^x$ und die sogenannte Diagonalfunktion $J(y) = y$ herangezogen wurden. Offensichtlich ist $J(y) = \bar{s}\big(s(y)\big)$ und $2^x = \lambda(x, 2)$, wobei $\lambda(x, n)$ die zweistellige Exponentialfunktion n^x ist. Wie wir schon gesehen haben, gilt

$$\lambda(0, n) = 1, \quad \lambda(x + 1, n) = \pi\big(\lambda(x, n), n\big),$$

wobei unter $\pi(x, n)$ die Produktfunktion $x \cdot n$ zu verstehen ist. Das Produkt wird seinerseits mit Hilfe der Summenfunktion σ, $\sigma(x, n) = x + n$, durch

$$0 \cdot n = 0, \quad (x + 1) \cdot n = \sigma(x \cdot n, n)$$

definiert, und die Summe wird schließlich mittels der Funktionen $J(n)$ und s durch

$$0 + n = n, \quad (x + 1) + n = s(x + n)$$

definiert. Fassen wir alle diese Gleichungen zusammen, so erhalten wir die folgende induktive Erzeugung (genauer gesagt, als eine mögliche induktive Erzeugung) der Funktion $[\log_2 y]$:

$$J(x) = \bar{s}\big(s(x)\big),$$

$$\begin{cases} \sigma(0, n) = J(n), \\ \sigma(x + 1, n) = s\big(\sigma(x, n)\big), \end{cases}$$

$$\begin{cases} \pi(0, n) = 0, \\ \pi(x + 1, n) = \sigma\big(\pi(x, n), n\big), \end{cases}$$

$$\begin{cases} \lambda(0, n) = 1, \\ \lambda(x + 1, n) = \pi\big(\lambda(x, n), n\big), \end{cases}$$

$$\psi(x) = \lambda(x, 2),$$

$$\varphi(y) = \mu x[\psi(x + 1) > J(y)].$$

Eine (volle) Funktion, für die es eine induktive Erzeugung aus den Ausgangsfunktionen $O(x)$, $s(x)$, $\bar{s}(x)$ gibt, heißt eine *allgemein-rekursive* oder kurz eine *rekursive Funktion*. Eine induktive Erzeugung heißt *primitiv-rekursiv*, wenn in ihr der μ-Operator nicht auftritt, d. h. nur die Operatoren der Superposition, der Einführung fiktiver Argumente und der primitiven Rekursion zugelassen werden. Eine Funktion heißt primitiv-rekursiv, wenn sie eine primitiv-rekursive induktive Erzeugung besitzt. Die von uns angegebene Erzeugung der Funktion $[\log_2]$ zeigt, daß diese Funktion

rekursiv ist. Da in ihr der μ-Operator verwendet wird, können wir nicht behaupten, daß $[\log_2]$ primitiv-rekursiv ist. Natürlich können wir auch nicht behaupten, daß die Funktion $[\log_2]$ mit Sicherheit nicht primitiv-rekursiv ist. Denn es ist ja nicht ausgeschlossen, daß eine andere induktive Erzeugung für $[\log_2]$ existiert, die primitiv-rekursiv ist, was übrigens in der Tat der Fall ist. Die „Anfangsstücke" der von uns betrachteten Erzeugung sind induktive und sogar primitiv-rekursive induktive Erzeugungen der „Hilfs-"Funktionen, die auf dem Wege der Definition von $[\log_2]$ entstehen, nämlich von Summe, Produkt usw.

Aus der Bedeutung der Operatoren der Superposition und der primitiven Rekursion sowie des μ-Operators wird klar, daß die induktive Erzeugung einer bestimmten Funktion als eine spezielle Art von Programm zur Berechnung der Werte der Funktion φ angesehen werden kann, das man etwa ein rekursives Programm für φ nennen könnte. So muß man z. B. zur Berechnung des Wertes $[\log_2 7]$ zunächst die Werte $2^0, 2^1, 2^2, \ldots$ berechnen, was uns zur Funktion 2^x führt. Die Berechnung der Werte dieser Funktion reduziert sich auf die Berechnung der Produktwerte usw., bis zu den Ausgangsfunktionen.

Indem wir die Resultate der vorangehenden Abschnitte zusammenfassen, gelangen wir zu dem folgenden

Satz. *Jede rekursive Funktion ist Turing-berechenbar, es gibt einen Algorithmus, der zu einem beliebigen rekursiven Programm einer Funktion φ ein Turing-Programm aufstellt, das die Funktion φ berechnet.*

Auf den ersten Blick erscheint die Auswahl der Funktionen $O(x)$, $s(x)$, $\bar{s}(x)$ als Ausgangsfunktionen willkürlich, und in einem gewissen Sinne ist sie es auch. Es könnten natürlich andere Ausgangsfunktionen betrachtet werden. Es ist jedoch sehr bemerkenswert und wichtig, daß schon mit diesen einfachen Ausgangsfunktionen alle diejenigen Funktionen induktiv erzeugt werden können, die in der Arithmetik eine Rolle spielen. Das bedeutet, daß eine Erweiterung der Menge der Ausgangsfunktionen durch Hinzunahme anderer natürlich scheinender arithmetischer Funktionen zu keiner Vergrößerung der Klasse der rekursiven Funktionen führt. Wir beschränken uns hier auf einige weitere Beispiele dafür, wie umfassend die Klasse der rekursiven Funktionen und folglich auch die Klasse der Turing-berechenbaren Funktionen ist.

Finite Funktionen. Unter einer *finiten Funktion* verstehen wir eine Funktion, die im folgenden Sinne fast konstant ist: Es existiert eine Konstante c, so daß f nur für endlich viele Argumente einen von c verschiedenen Wert annimmt. Es sei beispielsweise $f(0) = 2$, $f(1) = 5$ und $f(x) = 7$ für $x \geqq 2$. Wir erhalten f mit Hilfe der olgenden primitiven Rekursionen (des speziellen Typs (1) von S. 106):

$$f_1(0) = 5, \quad f_1(x+1) = 7, \quad f(0) = 2, \quad f(x+1) = f_1(x).$$

Analog zeigt man, daß jede finite Funktion primitiv-rekursiv ist. Spezielle finite

Funktionen sind

$$sg(x) = \begin{cases} 0 & \text{für } x = 0, \\ 1 & \text{für } x \neq 0, \end{cases} \qquad \overline{sg}(x) = \begin{cases} 0 & \text{für } x \neq 0, \\ 1 & \text{für } x = 0, \end{cases}$$

die man als *Signum* und *Antisignum* bezeichnet.

Einige zahlentheoretische Funktionen und Relationen. Wir haben bereits die primitive Rekursivität der Addition, der Multiplikation, der allgemeinen Potenz und anderer Funktionen nachgewiesen. Durch alleinige Anwendung der Superposition kann man aus diesen alle möglichen Polynome mit natürlichen Koeffizienten erhalten, aber auch Funktionen, die durch kompliziertere Terme, wie z. B.

$$\left((5x^2 + y)^{x^3+17} + 2xyz\right) \cdot 2^{3x^{7y+4}},$$

definiert werden.

Einige Schwierigkeiten gibt es bei den Umkehroperationen, der Subtraktion, der Division usw. Da sie im Bereich der natürlichen Zahlen nicht immer ausführbar sind, trifft man für sie gewöhnlich zusätzliche Vereinbarungen, betrachtet sogenannte arithmetische Varianten dieser (und anderer) Funktionen. Anstelle der Differenz $n - x$ betrachtet man z. B. die sogenannte *arithmetische Differenz*

$$n \dot{-} x = \begin{cases} n - x & \text{für } n \geqq x, \\ 0 & \text{für } n < x, \end{cases}$$

die durch die primitive Rekursion

$$n \dot{-} 0 = n, \quad n \dot{-} (x + 1) = \bar{s}(n \dot{-} x)$$

definiert wird. Hieraus ist ersichtlich, daß die Superposition $sg(n \dot{-} x)$ das Prädikat $Kl(x, n)$ beschreibt und die Superposition $\overline{sg}(n \dot{-} x) \cdot \overline{sg}(x \dot{-} n)$ das Prädikat $Gl(n, x)$. Folglich sind die Funktionen $\left[\frac{x}{n}\right]$ und $Rest(x, n)$, bei deren Definition durch eine simultane primitive Rekursion (vgl. Abschnitt 2.5.3) wir die Prädikate Kl und Gl benutzt haben, ebenfalls primitiv-rekursiv.

Die Teilbarkeitsrelation und die Eigenschaft, Primzahl zu sein, sind in dem Sinne primitiv rekursiv, daß ihre charakteristischen Funktionen

$$Div(x, n) = \begin{cases} 1, & \text{falls } x \text{ durch } n \text{ teilbar ist,} \\ 0 & \text{sonst,} \end{cases}$$

$$Pr(x) = \begin{cases} 1, & \text{falls } x \text{ Primzahl ist,} \\ 0 & \text{sonst} \end{cases}$$

primitiv rekursiv sind.

Es gilt nämlich $Div(x, n) = Gl(Rest(x, n), 0)$. In Übereinstimmung mit der früher getroffenen Definition der Funktion $Rest(x, n)$ sei dabei $Div(x, 0) = 1$ gesetzt. Bei der Funktion $Pr(x)$ kann man verschiedene Wege gehen. Wir wählen den folgenden: Offenbar liefert die Summe

$$\nu(n, x) = Div(x, 0) + Div(x, 1) + \cdots + Div(x, n)$$

gerade die Anzahl aller Teiler von x, die nicht größer als n sind. Diese Summe kann rekursiv beschrieben werden durch

$$\nu(0, x) = 1, \quad \nu(u + 1, x) = \nu(u, x) + Div(x, u + 1).$$

Mit Hilfe der Funktion ν erhält man die Funktion $\pi(x)$, die für gegebenes x die Anzahl aller Teiler von x angibt als $\pi(x) = \nu(x, x)$. Schließlich ist offenbar x genau dann eine Primzahl, wenn x genau drei Teiler hat (nämlich 0, 1 und x selbst). Folglich gilt

$$Pr(x) = Gl(\pi(x), 3).$$

Die Funktion $\wp$ zähle alle Primzahlen der Größe nach auf, d. h., es gelte $\wp(0) = 2$, $\wp(1) = 3$, $\wp(2) = 5, \ldots$. Offenbar ist das Produkt $Pr(y) \cdot Kl(\wp(x), y)$ dann und nur dann gleich 1, wenn y eine Primzahl ist, die $\wp(x)$ übertrifft. Folglich ist die Funktion $\wp(x)$ rekursiv, da man sie folgendermaßen definieren kann:

$$\wp(0) = 2, \quad \wp(x + 1) = \mu y[(Pr(y) \cdot Kl(\wp(x), y)) = 1].$$

2.5.6.* Die rekursive Programmierung der Turing-berechenbaren Funktionen

Sind auf Turing-Maschinen vielleicht auch Funktionen berechenbar, die nicht rekursiv sind? Eine negative Antwort auf diese Frage liefert der folgende Satz, dessen Beweis der vorliegende Abschnitt gewidmet ist.

Satz. *Die Turing-Maschine $\mathfrak{M}$ berechne die m-stellige arithmetische Funktion $f(x_1, \ldots, x_m)$. Dann kann man aus dem Funktionsschema der Maschine $\mathfrak{M}$ (einem gegebenen Turing-Programm für f) eine induktive Erzeugung von f effektiv herstellen.*

Dadurch wird auf einer höheren Stufe die bedeutsame Tatsache belegt, daß zwei Funktionenklassen, die ursprünglich auf ganz verschiedene Weise beschrieben wurden, nämlich die Klasse der Turing-berechenbaren Funktionen und die Klasse der rekursiven Funktionen, übereinstimmen. Der Beweis des Satzes ist auch für sich sehr lehrreich, weil er auf einer sehr interessanten Methode der arithmetischen Interpretation (*Arithmetisierung*) beruht, die schon in Abschnitt 2.5.1 kurz erwähnt wurde.

In unserem Fall beginnt die Arithmetisierung damit, daß jeder endlichen Konfiguration K der Maschine $\mathfrak{M}$ ein Quadrupel $(s(K), q(K), m(K), n(K))$ zugeordnet wird, dessen Komponenten wir die *Koordinaten* der Konfiguration K nennen wollen.

Es ist gut bekannt, wie nützlich die Anwendung der Koordinatenmethode z. B. in der Geometrie ist; sie gestattet es (im Rahmen der analytischen Geometrie), zur Lösung geometrischer Aufgaben gut ausgearbeitete algebraische und analytische Methoden heranzuziehen. Dabei hängen in einer konkreten Situation der Erfolg und die Einfachheit der Lösung wesentlich davon ab, wie gut das Koordinatensystem ausgewählt wurde. In unserem Fall sind (im Unterschied zur analytischen Geometrie) die Koordinaten keine beliebigen reellen, sondern natürliche Zahlen, entsprechend treten an die Stelle von algebraischen und analytischen Methoden hier zahlentheoretische (arithmetische) Konstruktionen und Methoden.

Es sei nun K' die Konfiguration, die bei Arbeit der Maschine $\mathfrak{M}$ unmittelbar auf die Konfiguration K folgt; wir bezeichnen mit s, q, m, n die Koordinaten von K und mit s', q', m', n' die von K'. Unser Ziel ist es, die Koordinaten so zu wählen, daß die Funktionen

$$s' = s'(s, q, m, n), \qquad q' = q'(s, q, m, n),$$

$$m' = m'(s, q, m, n), \quad n' = n'(s, q, m, n)$$

primitiv-rekursiv werden; Koordinaten, die diese Eigenschaft besitzen, wollen wir *korrekt* nennen. Wenn uns das gelingt, wird der Erfolg des ganzen Unternehmens, nämlich der Beweis des Satzes, in vollem Umfang gesichert sein.

Wir bemerken zuerst, daß jede endliche Turing-Konfiguration K eindeutig durch das Quadrupel (s_i, q_j, K_l, K_r) bestimmt ist, dessen erste Komponente s_i das eingestellte Symbol ist, dessen zweite Komponente q_j der innere Zustand von K ist und wo K_l, K_r der linke bzw. rechte Teil der Konfiguration sind, d. h. die (möglicherweise leeren) Wörter, die sich auf dem Band links bzw. rechts von der eingestellten Zelle befinden. Beschreiben wir das noch etwas genauer:[1]) Wenn alle Zellen links von der eingestellten Zelle das leere Symbol Λ enthalten, so sei $K_l = \Lambda$; anderenfalls ist K_l das Wort, das sich vom äußersten linken nichtleeren Symbol von K bis zur linken Nachbarzelle der eingestellten Zelle erstreckt. Für die Konfiguration K aus Abb. 27 gilt z. B.

$$s_i = |, \quad q_j = q_3, \quad K_l = |\,{*}{*}, \quad K_r = {*}\,\Lambda|.$$

Wir zeigen jetzt, wie die vier Komponenten durch natürliche Zahlen kodiert werden:

$$\big(\mathrm{Kod}(s_i), \mathrm{Kod}(q_j), \mathrm{Kod}(K_l), \mathrm{Kod}(K_r)\big).$$

Das wird zugleich das Koordinaten-Quadrupel der Konfiguration K. Wir nehmen an, daß die Buchstaben des äußeren Alphabets und die Zustandssymbole durch natürliche Zahlen numeriert sind:

$$S = \{s_0, s_1, \ldots, s_{r-1}\}, \quad Q = \{q_0, q_1, q_2, \ldots, q_{k-1}\}.$$

[1]) Der Einfachheit halber wird im folgenden sowohl das Leersymbol als auch das leere Wort mit Λ bezeichnet. [*Anm. d. Übers.*]

Wir legen fest, daß $\mathrm{Kod}(s_i) = i$, $\mathrm{Kod}(q_j) = j$ ist. Der Stopzustand werde durch die Zahl k kodiert, d. h. $\mathrm{Kod}(!) = k$. Für die weiteren Betrachtungen ist es ferner bequem, s_0 als das Leerzeichen Λ anzunehmen; folglich ist im weiteren immer $\mathrm{Kod}(\Lambda) = 0$. Die Indizes der Symbole $s_0, \ldots, s_{r-1}$ kann man als Ziffern des r-adischen Positionssystems auffassen. Daher kann man das Wort K_l als r-adische Darstellung einer bestimmten natürlichen Zahl ansehen, und diese wird gerade als $\mathrm{Kod}(K_l)$

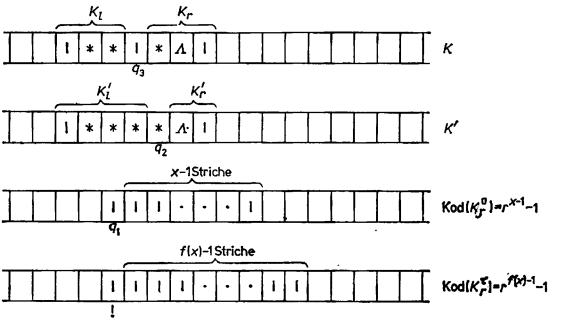


Abb. 27

genommen. So gilt z. B. für die Konfiguration K aus Abb. 27, wenn man $\mathrm{Kod}(\Lambda) = 0$, $\mathrm{Kod}(*) = 1$ und $\mathrm{Kod}(|) = 2$ setzt:

$$\mathrm{Kod}(K_l) = 2 \cdot 3^2 + 1 \cdot 3^1 + 1 \cdot 3^0 = 22 = [211]_3$$

(im triadischen System).[1]) Bei K_r ist es für uns bequemer, bei der Definition von $\mathrm{Kod}(K_r)$ das Wort K_r „spiegelbildlich" zu lesen, d. h. den weiter rechts stehenden Symbolen die höheren Positionswerte zuzuordnen. Für die Konfiguration K aus Abb. 27 wird

$$\mathrm{Kod}(K_r) = 1 \cdot 3^0 + 0 \cdot 3^1 + 2 \cdot 3^2 = 19 = [201]_3.$$

Folglich haben wir für die Konfiguration K

$$s = 2, \quad q = 3, \quad m = 22, \quad n = 19.$$

In Abb. 27 ist ferner die Konfiguration K' dargestellt, die sich aus K durch Anwendung des Befehls $|q_3 \to * Rq_2$ ergibt. Für diese ist $s' = 1$, $q' = 2$, $m' = 68$, $n' = 6$.

[1]) Der Index 3 bei $[211]_3$ weist auf das triadische Zahlensystem hin. Ohne eine solche besondere Verabredung oder Bezeichnung gelten alle Zahlen als im Dezimalsystem geschrieben.

Jetzt zeigen wir, daß s', q', m', n' als Funktionen von s, q, m, n primitiv-rekursiv sind. Wir haben bereits die äußeren Symbole und die Zustände durch Zahlen kodiert. Nun kodieren wir auch noch die Bewegungen:

$$\mathrm{Kod}(R) = 1, \quad \mathrm{Kod}(L) = 2, \quad \mathrm{Kod}(M) = 0.$$

Damit wird das Funktionsschema einer Maschine $\mathfrak{M}$ durch ein Tripel von Funktionen $S(s, q)$, $P(s, q)$, $Q(s, q)$ gegeben, die aus den Kodezahlen s, q des Eingangspaares die Kodezahl des neuen Bandsymbols, der Bewegung und des Folgezustandes erzeugen. Diese Funktionen sind zunächst nur für $s < r$ und $q < k$ definiert. Wir machen sie zu überall definierten Funktionen, indem wir ihnen für alle übrigen Argumentwerte ein und denselben konstanten Wert zuordnen. Damit werden S, P, Q zu finiten Funktionen und sind folglich (vgl. Abschnitt 2.5.5) primitiv-rekursiv. Wir wenden uns nun wieder den uns interessierenden Funktionen $s'(s, q, m, n)$, $q'(s, q, m, n)$, $m'(s, q, m, n)$ und $n'(s, q, m, n)$ zu.

Die primitive Rekursivität von s' und q' ist offensichtlich, denn es gilt

$$s'(s, q, m, n) = S(s, q) \quad \text{und} \quad q'(s, q, m, n) = Q(s, q).$$

Was nun die Funktionen m' und n' betrifft, so müssen wir in Abhängigkeit von der beim Übergang von K zu K' auftretenden Bewegung drei Fälle unterscheiden:

1. Es erfolgt keine Bewegung, d. h. $P(s, q) = 0$. Dann ist $m' = m$ und $n' = n$.

2. Es erfolgt eine Bewegung nach rechts, d. h. $P(s, q) = 1$. Dann entsteht K_l' aus K_l durch rechtsseitiges Anfügen einer Zelle, deren Inhalt das Bandsymbol mit dem Index $s' = S(s, q)$ ist. Also wird

$$m' = \mathrm{Kod}(K_l') = m \cdot r + S(s, q).$$

Der rechte Teil K_r' entsteht dagegen aus K_r, indem in K_r die äußerste linke Zelle weggelassen wird, in der das Bandsymbol steht, dessen Index der Koeffizient von r^0 in der r-adischen Darstellung von n ist. Folglich ist $n' = \left[\frac{n}{r}\right]$.

3. Es erfolgt eine Bewegung nach links, d. h. $P(s, q) = 2$. Hier wird in Analogie zum vorangehenden Fall

$$m' = \left[\frac{m}{r}\right], \quad n' = n \cdot r + S(s, q).$$

Diese verbalen Formulierungen können folgendermaßen in Formeln zusammen-

gefaßt werden:

$$m'(s,q,m,n) = Gl\big(P(s,q),0\big)\cdot m + Gl\big(P(s,q),1\big)\cdot\big(m\cdot r + S(s,q)\big) + Gl\big(P(s,q),2\big)\cdot\left[\frac{m}{r}\right];$$

$$n'(s,q,m,n) = Gl\big(P(s,q),0\big)\cdot n + Gl\big(P(s,q),1\big)\cdot\left[\frac{n}{r}\right] + Gl\big(P(s,q),2\big)\cdot\big(n\cdot r + S(s,q)\big).$$

Damit sind die Funktionen $m'(s,q,m,n)$ und $n'(s,q,m,n)$ als Superposition der primitiv-rekursiven Funktionen $Gl, P, S, [\,], +$ und $\cdot$ dargestellt. Folglich sind sie primitiv-rekursiv. Damit ist die Korrektheit der betrachteten Koordinaten bewiesen.

Als direkte Folgerung aus dieser Tatsache kann man die primitive Rekursivität auch der anderen Funktionen, die die Arbeit einer Turing-Maschine beschreiben, herleiten. Wir gehen von einer bestimmten Anfangskonfiguration K der Maschine $\mathfrak{M}$ aus, die die Koordinaten s, q, m, n besitzen möge. Dann ergibt sich nach t Takten eine Konfiguration, die wir mit K^t bezeichnen wollen (wobei $K^0 = K$ gesetzt sei). Die Koordinaten der Konfiguration K^t hängen natürlich von den Koordinaten der Anfangskonfiguration K und von t ab, d. h., sie sind Funktionen der fünf Veränderlichen t, s, q, m, n; wir bezeichnen sie mit S, Q, M, N. Es ist leicht zu sehen, daß dieses Quadrupel von Funktionen durch simultane primitive Rekursion in den Funktionen s', q', m', n' definiert werden kann. Folglich sind auch die Funktionen S, Q, M, N primitiv-rekursiv. In der Tat gilt

$$S(0,s,q,m,n) = s; \qquad Q(0,s,q,m,n) = q;$$

$$M(0,s,q,m,n) = m; \qquad N(0,s,q,m,n) = n;$$

$$S(t+1,s,q,m,n) = s'\big(S(t,s,q,m,n),Q(t,s,q,m,n),\ M(t,s,q,m,n),N(t,s,q,m,n)\big);$$

$$Q(t+1,s,q,m,n) = q'\big(S(t,s,q,m,n),Q(t,s,q,m,n),\ M(t,s,q,m,n),N(t,s,q,m,n)\big);$$

$$M(t+1,s,q,m,n) = m'\big(S(t,s,q,m,n),Q(t,s,q,m,n),\ M(t,s,q,m,n),N(t,s,q,m,n)\big);$$

$$N(t+1,s,q,m,n) = n'\big(S(t,s,q,m,n),Q(t,s,q,m,n),\ M(t,s,q,m,n),N(t,s,q,m,n)\big).$$

Dieses Schema gibt lediglich die offensichtliche Tatsache wieder, daß das Resultat nach Anwendung von $t+1$ Schritten auf die Anfangskonfiguration K identisch ist mit dem Resultat, das sich nach Anwendung eines Schrittes auf die nach t Schritten vorliegende Konfiguration K^t ergibt.

Jetzt haben wir alles dargelegt, was wir brauchen, um den Beweis des Satzes abzuschließen.

Zur Vereinfachung betrachten wir den Fall, daß die Maschine $\mathfrak{M}$ eine einstellige

Funktion $f(x)$ berechnet; der Fall einer mehrstelligen Funktion unterscheidet sich von diesem nur in unwesentlichen Details. Wir wollen ferner annehmen, daß die Argumente und Werte von f in unärer Darstellung gegeben sind; wie in Abschnitt 2.5.1 bemerkt, ist auch diese Einschränkung unwesentlich. Weiterhin setzen wir voraus, daß das Symbol „I" (Strich) durch die höchste Ziffer des bei der Arithmetisierung verwendeten Zahlensystems kodiert wird, d. h., es sei $\mathrm{Kod}(\mathrm{I}) = r - 1$. In der Anfangskonfiguration K, die der Standard-Darstellung des Arguments x entspricht, ist der linke Teil K_l leer, der rechte Teil K_r besteht aus $x \dot{-} 1$ Strichen, und ein weiterer Strich (oder das Leerzeichen, falls $x = 0$ ist) wird im Anfangszustand q_1 vom Kopf betrachtet (vgl. Abb. 27). Folglich gilt

$$\mathrm{Kod}(K_r) = (r-1)\, r^{x-2} + (r-1)\, r^{x-3} + \cdots + (r-1)\, r^0 = r^{x\dot{-}1} - 1,$$

und die Konfiguration K hat das Koordinatenquadrupel $\big(sg(x)\,(r-1),\, 1,\, 0,\, r^{x\dot{-}1} - 1\big)$. Dadurch werden bei Variation der Anfangskonfiguration nur über die unäre Niederschrift von Argumenten x die Koordinatenfunktionen S, Q, M, N zu Funktionen der beiden Veränderlichen t, x. Wir führen für diese die Bezeichnungen $\tilde{S}, \tilde{Q}, \tilde{M}$ und $\tilde{N}$ ein, d. h.

$$\tilde{S}(t, x) = S\big(t,\, sg(x)\,(r-1),\, 1,\, 0,\, r^{x\dot{-}1} - 1\big),\quad \tilde{Q}(t, x) = Q\big(t,\, sg(x)\,(r-1),\, 1,\, 0,\, r^{x\dot{-}1} - 1\big),$$

usw. Offenbar sind auch $\tilde{S}, \tilde{Q}, \tilde{M}$ und $\tilde{N}$ primitiv-rekursive Funktionen. Da der Stopzustand durch die Konstante k kodiert wird, kann die Funktion $\tau(x)$, die die Zahl der von der Maschine $\mathfrak{M}$ bei der Berechnung von $f(x)$ auszuführenden Takte angibt, durch Anwendung des μ-Operators auf $\tilde{Q}$ erhalten werden:

$$\tau(x) = \mu t[\tilde{Q}(t, x) = k].$$

Folglich ist auch sie rekursiv (allerdings im allgemeinen nicht mehr primitiv-rekursiv). Daher liefert die rekursive Funktion

$$\tilde{N}\big(\tau(x), x\big) = N\big(\tau(x),\, sg(x)\,(r-1),\, 1,\, 0,\, r^{x\dot{-}1} - 1\big)$$

die vierte Koordinate der Endkonfiguration (vgl. Abb. 27), die gleich $r^{f(x)\dot{-}1} - 1$ ist. Hiermit erhalten wir schließlich für die Funktion $f(x)$ die folgende Darstellung:

$$f(x) = \Big(1 + \big[\log_r \tilde{N}\big(\tau(x), x\big)\big]\Big) \cdot sg\big(\tilde{S}\big(\tau(x), x\big)\big).$$

Damit ist die gesuchte rekursive Beschreibung der Funktion $f(x)$ gefunden.

2.6. Varianten des äußeren Speichers

Der richtige Ablauf eines beliebigen Rechenprozesses wird dadurch gewährleistet, daß in jedem Stadium der Rechnung im Speicher (der Maschine oder des Rechners) mindestens die Daten aufbewahrt werden, die in späteren Stadien der Rechnung

benötigt werden. Diese Daten dürfen nicht vorher verstümmelt oder gar ganz ausgelöscht werden, und sie müssen darüber hinaus sogar hinreichend günstig im Speicher untergebracht werden, damit sie im gegebenen Moment gefunden und verwendet werden können. Bei realen Rechenautomaten wird das durch das Abspeichern von Zwischenergebnissen in speziellen Einrichtungen (z. B. Maschinenregister oder Speicherzellen mit bekannten Adressen) erreicht, wo sie bis zu dem Moment aufbewahrt werden, in dem sie für die weitere Arbeit benötigt werden. Erheblich komplizierter liegen die Dinge bei den Turing-Maschinen, denn bei ihnen steht nur ein Band zur Verfügung, auf das sowohl die Anfangs- und Enddaten als auch alle Zwischenergebnisse zu schreiben sind. Wir sind mit diesen Schwierigkeiten bereits bei der Parallelanwendung von Algorithmen konfrontiert worden. In diesem Abschnitt wollen wir einige allgemeine Überlegungen und Methoden darlegen, die es erlauben, die mit der Eigenart des äußeren Speichers der Turing-Maschinen verbundenen Schwierigkeiten zu überwinden. Als Folgerung erhalten wir einen Beweis des Satzes über die Programmierung der Parallelanwendung, den wir bereits in Abschnitt 2.4.3. formuliert und seitdem mehrfach benutzt haben. Die hier dargelegten Begriffe und Methoden beanspruchen jedoch durchaus selbständiges Interesse. Sie werden uns auch im folgenden noch nützlich sein.

2.6.1.* Halbbänder und die Parallelanwendung von Turing-Maschinen

Zum besseren Verständnis dafür, wie eine Turing-Maschine ihren äußeren Speicher erfolgreich ausnutzen kann, wollen wir zunächst zwei Varianten der Turing-Maschine betrachten, die sich vom bisher betrachteten Grundmodell nur durch einige Besonderheiten des äußeren Speichers, d. h. des Bandes, unterscheiden. Als erstes betrachten wir eine Maschine $\mathfrak{M}$ mit einem rechtsseitigen Halbband (vgl. Abb. 28a), d. h. einem Band, das nur nach einer Seite, und zwar nach rechts, unbeschränkt ist. Man kann annehmen, daß die Zellen des Bandes von links nach rechts mit Hilfe der Zahlen 0, 1, 2, 3, ... numeriert sind. In der Null-Zelle sei ein spezielles Zeichen „#" untergebracht. Die Form der Befehle und ihre Ausführung ist dieselbe wie beim Grundmodell, wobei aber zusätzlich die folgenden Besonderheiten auftreten:

(i) In keinem Zustand wird das Symbol „#" geschrieben;

(ii) Wenn die Maschine in irgendeinem Zustand das Symbol „#" liest, so sieht der entsprechende Befehl eine Bewegung nach rechts ohne Veränderung des Symbols „#" vor. Anders ausgedrückt: Wenn der Kopf während eines Berechnungsprozesses die Null-Zelle erreicht (was er durch Lesen des Symbols „#" erfährt), so verläßt er sie sofort wieder nach rechts.

Bei einer Maschine $\mathfrak{M}$ mit rechtsseitigem Halbband bedeute die Schreibweise $\mathfrak{M}(P) = R$, daß $\mathfrak{M}$ die Anfangskonfiguration, bei der das Anfangswort $P = P(1) \cdots P(v)$ in die Zellen mit den Nummern $1, \ldots, v$ eingetragen ist und sich der Kopf im Anfangszustand unter der Zelle mit der Nummer 1 befindet (vgl. Abb. 28a), in eine Endkonfiguration überführt, bei der das Resultatwort $R = R(1) \cdots R(s)$ in gewissen

Zellen $p+1, \ldots, p+s$ ($p \geqq 0$) steht und sich der Kopf im Stopzustand „!" unter der Zelle $p+1$ befindet. Wir wollen das die Standard-Darstellung von Anfangs- und Endinformation in einer Maschine mit rechtsseitigem Halbband nennen.

Analog wird eine Maschine $\mathfrak{M}$ mit linksseitigem Halbband definiert. Hier ist jedoch die Schreibweise $\mathfrak{M}(P) = R$ folgendermaßen zu verstehen (und das ist der wesentliche Unterschied zwischen den beiden Arten von Halbbandmaschinen): Die Anfangskonfiguration, bei der das Anfangswort $P = P(1) \cdots P(v)$ in den Zellen mit den Nummern $v, \ldots, 1$ eingetragen ist und sich der Kopf im Anfangszustand unter der Zelle mit der Nummer v befindet (vgl. Abb. 28g), wird durch $\mathfrak{M}$ in eine End-

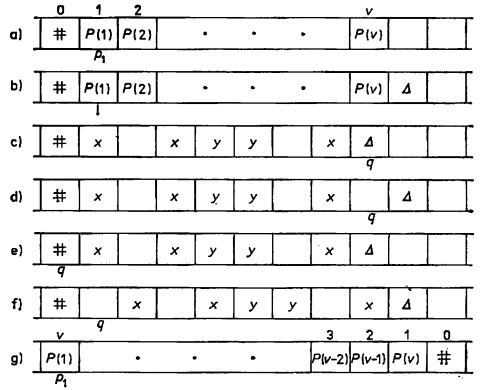


Abb. 28

konfiguration übergeführt, bei der das Resultatwort $R = R(1) \cdots R(s)$ in gewissen Zellen $p+s, \ldots, p+1$ ($p \geqq 0$) steht und der Kopf sich im Stopzustand „!" unter der am weitesten links stehenden Zelle $p+s$ befindet. Wir wollen das die Standard-Darstellung von Anfangs- und Endinformation in einer Maschine mit linksseitigem Halbband nennen.

Es ist unmittelbar klar, daß man das Funktionsschema jeder beliebigen Halbband-Maschine $\mathfrak{M}$ in ein Funktionsschema einer gewöhnlichen Turing-Maschine $\mathfrak{N}$ überführen kann, die genau die Arbeit von $\mathfrak{M}$ imitiert. Daraus folgt insbesondere, daß alle Probleme, die auf Halbband-Maschinen gelöst werden können, auch (und sogar „fast" in derselben Weise) auf gewöhnlichen Maschinen zu lösen sind. Das sieht im einzelnen so aus: Es sei $\mathfrak{M}$ gegeben; dann ist $\mathfrak{N}$ die Komposition dreier Schemata $\mathfrak{N} = \mathfrak{A} \circ \mathfrak{M} \circ \mathfrak{L}$, wobei $\mathfrak{A}$ das Wort P in $\# P$ überführt, den Kopf auf den ersten Buchstaben von P einstellt und $\mathfrak{L}$ zum Schluß das Symbol „#" links vom Resultat

R sucht, es löscht und den Kopf wieder zum ersten Buchstaben von R zurückbringt (falls R nicht leer ist).

Es zeigt sich nun (und das ist weit weniger trivial), daß auch der umgekehrte Übergang von den Maschinen mit beidseitig unbegrenztem Band zu Maschinen mit einem Halbband die Berechnungsmöglichkeiten nicht einschränkt, d. h., daß die folgenden Sätze gelten:

Satz 1. *Es existiert ein programmierender Algorithmus, der ein beliebiges Turing-Programm $\mathfrak{N}$ in ein Programm $(\# \mathfrak{N})$ für eine Maschine mit rechtsseitigem Halbband überführt, so daß folgendes gilt:* a) $\mathfrak{N}(P) = R$ *genau dann, wenn* $(\# \mathfrak{N})(P) = R$; b) *In der Maschine* $\# \mathfrak{N}$ *steht am Ende das Resultat* R *(wie zu Anfang* P*) am linken Rand des Halbbandes.*

Satz 2. *Es existiert ein programmierender Algorithmus, der ein beliebiges Turing-Programm $\mathfrak{N}$ in ein Programm $(\mathfrak{N} \#)$ für eine Maschine mit linksseitigem Halbband überführt, so daß folgendes gilt:* a) $\mathfrak{N}(P) = R$ *genau dann, wenn* $(\mathfrak{N} \#)(P) = R$; b) *In der Maschine* $\mathfrak{N} \#$ *steht am Ende das Resultat* R *(wie zu Anfang* P*) am rechten Rand des Halbbandes.*

Bevor wir die entsprechenden Beweise ausführen, zeigen wir, daß aus diesen Sätzen sehr einfach der Satz von der Programmierung der Parallelanwendung $\mathfrak{A}_1 \parallel \mathfrak{A}_2$ zweier Algorithmen $\mathfrak{A}_1, \mathfrak{A}_2$ folgt: Zunächst bilden wir die Programme $\mathfrak{A}_1 \#$ und $\# \mathfrak{A}_2$. Sodann erzeugen wir $\mathfrak{A}_1 \parallel \mathfrak{A}_2$ als Komposition der folgenden vier Programme:

1. $(\mathfrak{A}_1 \#)$ (es führt $P_1 \# P_2$ in $\mathfrak{A}_1(P_1) \# P_2$ über);

2. $\mathfrak{L}_1$, das den Kopf vom ersten Buchstaben des Wortes $\mathfrak{A}_1(P_1)$ zum ersten Buchstaben des Wortes P_2 transportiert;

3. $(\# \mathfrak{A}_2)$ (es setzt die Abarbeitung bis zum Erhalt des Wortes $\mathfrak{A}_1(P_1) \# \mathfrak{A}_2(P_2)$ fort, wobei am Ende der Kopf unter dem ersten Buchstaben des Wortes $\mathfrak{A}_2(P_2)$ steht);

4. $\mathfrak{L}_2$, das den Kopf vom ersten Buchstaben des Wortes $\mathfrak{A}_2(P_2)$ zum ersten Buchstaben des Wortes $\mathfrak{A}_1(P_1)$ transportiert.

Die Konstruktion der Standard-Programme $\mathfrak{L}_1$, $\mathfrak{L}_2$ ist trivial und sei dem Leser überlassen.

Übungsaufgabe. Man schreibe einen programmierenden Algorithmus für die Parallelanwendung von drei oder mehr Programmen.

2.6.2.* Beweis der Sätze über Halbband-Maschinen

Wir kommen nun zum Beweis der Sätze 1 und 2. Als Beispiel betrachten wir den Fall eines rechtsseitigen Halbbandes. Das Programm $(\# \mathfrak{N})$ wird die Komposition $\mathfrak{A} \circ \mathfrak{N} \circ \mathfrak{L}$ von drei Programmen, von denen $\mathfrak{N}$ von grundlegender Bedeutung ist, während $\mathfrak{A}$ und $\mathfrak{L}$ zwei Standard-Programme sind, die nicht vom Ausgangsprogramm $\mathfrak{N}$ abhängen und lediglich die Rolle von Hilfsprogrammen spielen. Um größeren

Aufwand zu vermeiden, wollen wir annehmen, daß im Programm $\mathfrak{N}$ außer dem Leerzeichen Λ nur zwei „eigentliche" Symbole x, y vorhanden sind. Für den allgemeinen Fall ist die Prozedur im Prinzip dieselbe, wird nur aufwendiger:

1. Das Programm $\mathfrak{A}$ (vgl. Abb. 29). Es schreibt ein neues Symbol „Δ" in die erste leere Zelle rechts vom Anfangswort P. Die entsprechenden Konfigurationen — zu Beginn und am Ende der Arbeit von $\mathfrak{A}$ — sind in Abb. 28a, b dargestellt.

	r_1	r_2
x	R	L
y	R	L
Λ	$\Delta L r_2$	
Δ		
#		$R!$

Abb. 29

2. Das Programm $\tilde{\mathfrak{N}}$ (vgl. Abb. 30). Zu Beginn der Anwendung des Programms $\tilde{\mathfrak{N}}$ begrenzen also die Symbole „#" und „Δ" die Zone des Bandes, die das Anfangswort P enthält. Im weiteren Arbeitsprozeß bleibt das Symbol „#" (die feste Marke) in seiner Zelle erhalten, während das Symbol „Δ" (die bewegliche Marke) in jeder Situation, in der sich der Platz innerhalb der durch die Marken „#" und „Δ" begrenzten Zone[1]) als für die Berechnung nicht ausreichend erweist, nach rechts ver-

	...	q	...	$\tilde{q}$	$q_{\#}$	q_x	q_y	q_Λ	q_Δ	q'
x					$\Lambda R q_x$	$x R q_x$	$y R q_x$	$\Lambda R q_x$		L
y		$\mathfrak{A}$			$\Lambda R q_y$	$x R q_y$	$y R q_y$	$\Lambda R q_y$		L
Λ				$\Delta L q$	$\Lambda R q_{\#}$	$x R q_\Lambda$	$y R q_\Lambda$	$\Lambda R q_\Lambda$	$\Delta L q'$	L
Δ		$\Lambda R \tilde{q}$				$x R q_\Delta$	$y R q_\Delta$	$\Lambda R q_\Delta$		
#		$R q_{\#}$								$R q$

Abb. 30

schoben wird. Nach Beendigung der Arbeit des Programms $\tilde{\mathfrak{N}}$ steht in der Zone, die zu diesem Zeitpunkt vorliegt, das Endresultat R. Das Programm $\overline{\mathfrak{N}}$ kann als Erweiterung des Programms $\mathfrak{N}$ in der Weise erhalten werden, wie es in Abb. 30 dargestellt ist. Das Alphabet $\{x, y, \Lambda\}$ des Programms $\mathfrak{N}$ wird um die Symbole „#" und „Δ" erweitert. Außerdem werden für jeden Zustand q des Programms $\mathfrak{N}$ noch sieben mit q verbundene Zustände hinzugefügt, die mit $\tilde{q}, q_{\#}, q_x, q_y, q_\Lambda, q_\Delta$ und q' bezeichnet werden. Folglich beträgt die Anzahl der Zustände das Achtfache der Anzahl der Zustände von $\mathfrak{N}$. Wir verfolgen nun die Arbeit des Programms $\overline{\mathfrak{N}}$ und vergleichen sie mit der des gegebenen Programms $\mathfrak{N}$. Solange sich der Kopf innerhalb

[1]) Zur Abkürzung werden wir sie einfach Zone nennen.

der Zone befindet, arbeitet $\mathfrak{N}$ genauso wie $\mathfrak{N}$. Sei nun der Kopf erstmalig zur Marke „Δ" gelangt, und zwar in einem bestimmten Zustand q (vgl. Abb. 28c). Dann löscht er „Δ" (Befehl $\Delta q \to \Delta R\bar{q}$) und bewegt sich nach rechts zur benachbarten leeren Zelle. Dort schreibt er „Δ" (Befehl $\Delta\bar{q} \to \Delta\, Lq$) und kehrt wieder im Zustand q dorthin zurück, wo vorher die Marke „Δ" stand, jetzt aber eine leere Zelle vorliegt (vgl. Abb. 28d). Da die Marke „#" unbeweglich ist und folglich die Zone nicht nach links erweitert werden kann, geht die Bereitstellung einer leeren Zelle am linken Ende, die für die Berechnung nach dem Programm $\mathfrak{N}$ nötig sein kann, folgendermaßen vor sich: Die gesamte Inschrift der Zone, einschließlich der Marke „Δ", wird um eine Zelle nach rechts versetzt, und auf die so gewonnene leere Zelle neben der Marke „#" wird der Kopf im Zustand q gesetzt, in dem er zu Anfang (bei der Suche nach „freien Raum") auf die Marke „#" gestoßen war (vgl. Abb. 28f). Hierdurch wird gerade das vom Programm $\mathfrak{N}$ verlangte Vorgehen erreicht. Wir verfolgen an

	p_1	p_x	p_y	p'_x	p'_y	p_2
x	Lp_x	Rp'_x	Rp'_y			L
y	Lp_y	Rp'_x	Rp'_y			L
Λ	R	L	L	xRp_1	yRp_1	L
Δ	ΛLp_2					L
#		Rp'_x	Rp'_y			$R!$

Abb. 31

Hand des Programms noch genauer, wie das vor sich geht. Auf „#" gestoßen, geht der Kopf nach rechts (Befehl $\# q \to Rq_\#$), löscht den Inhalt der Nachbarzelle und geht, sich aber dabei an deren Inhalt erinnernd, nach rechts (der Befehl des Typs $xq_\# \to \Lambda Rq_x$ führt zum Zustand q_x, der „sich an x erinnert"). Aus den Spalten des Programms, die den Zuständen $q_\#, q_x, q_y, q_\Lambda, q_\Delta$ entsprechen, ist ersichtlich, daß der Prozeß des Übertragens des Inhalts jeder Zelle in ihre rechte Nachbarzelle so lange fortgesetzt wird, bis der Kopf schließlich das Symbol „Δ" verschiebt. Dann beginnt im Zustand q' eine Bewegung nach links, die bis zum Anfang „#" führt, und schließlich kehrt der Kopf im Zustand q in die nun leere rechte Nachbarzelle von „#" zurück. Wir weisen darauf hin, daß nach Beendigung der Arbeit von $\mathfrak{N}$ die Standard-Darstellung vorliegt.

3. Das Programm $\mathfrak{L}$ (vgl. Abb. 31). Wir bemerken, daß nach Beendigung der Arbeit von $\mathfrak{N}$ das Resultat R sich irgendwo in der im Verlauf der Berechnung eventuell größer gewordenen Zone befindet. Das Programm $\mathfrak{L}$ transportiert das Wort R direkt neben die Marke „#" und löscht die rechte Marke „Δ".

Damit ist der Satz für den Fall eines rechtsseitigen Halbbandes bewiesen.

Ein programmierender Algorithmus für den Fall eines linken Halbbandes unterscheidet sich nur unbedeutend von der beschriebenen Konstruktion.

2.6.3.* Mehrstöckige und mehrdimensionale Bänder

Zum Abschluß wollen wir noch kurz auf einige andere mögliche Modifikationen des äußeren Speichers einer Turing-Maschine hinweisen.

Als erstes betrachten wir den Fall, daß jede Zelle des Bandes aus einer oberen und einer unteren Etage besteht und dementsprechend auch zwei Symbole, ein oberes und ein unteres Symbol, speichern kann. Die Befehle haben die Form

$$\begin{matrix} x \\ y \end{matrix} q \to \begin{matrix} x' \\ y' \end{matrix} Pq'$$

und werden folgendermaßen ausgeführt: Wenn die Maschine im Zustand q das obere Symbol x und das untere Symbol y liest, so schreibt sie in die obere Etage das Symbol x', in die untere Etage das Symbol y', führt die Bewegung P aus und geht in den Zustand q' über. Es ist leicht einzusehen, daß sich die neue Konzeption nur unwesentlich von der üblichen unterscheidet. Die gesamte Neuerung besteht lediglich darin, daß als äußere Symbole nun Paare (Spalten der Höhe 2) von Symbolen aus einem anderen Alphabet genommen werden.[1]) Analog ist eine Maschine mit ν-stöckigem Band im wesentlichen eine gewöhnliche Maschine, deren äußere Symbole Spalten der Höhe ν aus Buchstaben eines gewissen Alphabets sind. Diese einfache Bemerkung erlaubt uns, streng zu beweisen, daß das Heranziehen mehrstöckiger Bänder die Klasse der Turing-berechenbaren Funktionen nicht erweitert. Aus der Sicht des Programmierens kann sich jedoch das Heranziehen mehrstöckiger Bänder durchaus als nützlich erweisen.

Wir kommen nun zur Beschreibung von *Turing-Maschinen mit mehrdimensionalem äußeren Speicher*. Im Fall eines zweidimensionalen Speichers ergibt sich folgendes Bild: Die Ebene ist in quadratische Zellen unterteilt, von denen jede ein Symbol des äußeren Alphabets speichern kann. Statt der drei Bewegungsmöglichkeiten, die wir bei einem eindimensionalen Band haben, gibt es jetzt fünf: rechts, links, oben, unten, auf der Stelle bleiben. Die Befehle haben die Form $xq \to x'Pq'$ und werden wie üblich ausgeführt (vgl. Abb. 32), nur mit dem Unterschied, daß P die angegebenen fünf Bedeutungen haben kann. Man kann etwa die folgende Konzeption der Berechnung einer Funktion $R = f(H)$ verwenden: In der Anfangskonfiguration steht das Wort H in einer Zeile (horizontal), und am Ende liegt das Resultat ebenfalls in einer Zeile vor, wobei aber die Zwischeninformationen in beliebiger Weise auf die Ebene geschrieben werden dürfen. Man kann zeigen, daß auch jede auf einer Maschine mit zweidimensionalem Speicher berechenbare Funktion bereits auf einer gewöhnlichen Turing-Maschine mit eindimensionalem Band berechenbar ist. Diese Be-

[1]) Es sei bemerkt, daß man auch in anderen Fällen Symbole, die üblicherweise in realen Situationen als einheitliche Objekte aufgefaßt werden, als solche ansehen kann, die aus zwei Teilen bestehen (etwa einem oberen und einem unteren). Beispiele: die Umlaute $\ddot{a}$, $\ddot{o}$, $\ddot{u}$ oder Symbole des Typs $\bar{a}$, b', x_i usw.

hauptung bleibt auch für einen beliebigen k-dimensionalen Speicher ($k = 2, 3, \ldots$) gültig. Der Beweis hierfür beruht auf einer ähnlichen Idee, wie sie beim Beweis des Satzes vom Halbband ausgenutzt wurde. Man kann nämlich das Programm einer Maschine $\mathfrak{M}$ mit k-dimensionalem Band in das Programm einer Maschine $\mathfrak{M}'$ mit gewöhnlichem Band überführen, die in einem bestimmten Sinne die Arbeit von $\mathfrak{M}$ imitiert. Die Realisierung dieser Idee ist in diesem Fall jedoch etwas komplizierter.

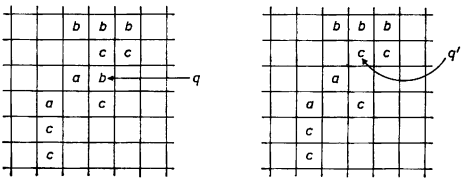


Abb. 32

Befehl : $qb \rightarrow A$ oben q'

Eine weitere wichtige Variante sind die sogenannten *Mehrband-Maschinen*. Sie verfügen über eine bestimmte Anzahl k von Bändern, die unabhängig voneinander gelesen, beschrieben und transportiert werden. Die Befehle haben die Form

$$x_1 \cdots x_k q \rightarrow x_1' \cdots x_k' P_1 \cdots P_k q' .$$

Häufig ist dabei ein Band als *Eingabeband* ausgebildet und wird eventuell ein anderes Band als *Ausgabeband* benutzt. Vom Eingabeband werden in gewissen Takten, buchstabenweise fortschreitend, die Eingabesymbole eingelesen, und auf das Ausgabeband wird, ebenfalls buchstabenweise fortschreitend, in gewissen Takten das Resultatwort gedruckt. In der Literatur wurden ferner auch Maschinen untersucht, die auf einem Band mit mehreren Köpfen operieren (vgl. Abschnitt 3.8.1), und vieles mehr. Wichtig ist, daß auch alle diese Varianten im wesentlichen dasselbe wie die einfachen Turing-Maschinen leisten.

2.7. Die Grundhypothese der Algorithmentheorie

2.7.1. Die Hypothese und ihre Bedeutung

Die Analyse der Beispiele für die Arbeit von Turing-Maschinen erweckt den Eindruck, daß der in einer solchen Maschine ablaufende Prozeß die Zeitlupenaufnahme eines von einem Menschen nach einem gewissen Algorithmus durchgeführten Rechenprozesses ist. Gleichzeitig lassen die Beispiele die Vermutung aufkommen, daß es

möglich sein müßte, jeden in irgendeiner Weise vorgegebenen Algorithmus durch das Funktionsschema einer Turing-Maschine wiederzugeben. Ist das nun aber wirklich so? Wie allgemein ist der Begriff der Turing-Maschine bzw. des Turingschen Funktionsschemas? Kann man behaupten, daß dieser Begriff so allgemein ist, daß jeder Algorithmus (im intuitiven Sinne) durch eine Turing-Maschine realisierbar ist? Diese Fragen beantwortet die Theorie der Algorithmen durch die folgende Hypothese:

Fundamentalhypothese der Algorithmentheorie. *Jeder Algorithmus kann in Form eines Turingschen Funktionsschemas vorgegeben und in einer entsprechenden Turing-Maschine realisiert werden.*

Wir wollen hier zwei Fragen näher betrachten, die sich aus der Formulierung der Hypothese ergeben:

1. Worin besteht die Bedeutung dieser Hypothese für die Algorithmentheorie?
2. Worauf gründet sich diese Hypothese?

Die obige Formulierung der Hypothese enthält eine charakteristische Besonderheit, auf die wir hier ausdrücklich aufmerksam machen wollen. In dieser Formulierung ist einerseits die Rede von „jedem Algorithmus", d. h. von dem allgemeinen Begriff des Algorithmus, der, wie schon mehrfach hervorgehoben, kein exakter mathematischer Begriff ist; andererseits ist in ihr der exakte mathematische Begriff „Turingsches Funktionsschema" enthalten. Der Sinn dieser Hypothese besteht nun gerade darin, daß sie den allgemeinen, aber verschwommenen Begriff „jeder Algorithmus" durch den spezielleren, aber exakten mathematischen Begriff „Turingsches Funktionsschema" (und seine Realisierung durch eine Turing-Maschine) präzisiert. Auf diese Weise wird die Theorie der Algorithmen auf die Untersuchung aller möglichen Turingschen Funktionsschemata (Turing-Maschinen) reduziert. Zugleich erhalten dadurch alle Fragestellungen einen vernünftigen Sinn, so z. B. die Frage, ob es für einen gegebenen Aufgabentyp einen Algorithmus gibt oder nicht. Jetzt ist diese Frage wie folgt zu verstehen: Existiert eine Turing-Maschine (ein Funktionsschema), die das Verlangte leistet, oder existiert keine solche Maschine?

Unsere Hypothese rechtfertigt die Verwendung der grundlegenden Definition der modernen Algorithmentheorie, wobei der verschwommene Begriff des Algorithmus mit dem exakten Begriff des Funktionsschemas einer Turing-Maschine identifiziert wird.

Worauf beruht nun diese fundamentale Hypothese? Zunächst wollen wir ausdrücklich darauf hinweisen, daß es sich nicht darum handeln kann, diese Hypothese so zu beweisen, wie man gewöhnlich in der Mathematik Sätze beweist. Der Wortlaut der Hypothese hat nämlich nicht den Charakter eines Satzes, weil er eine Behauptung über den allgemeinen Begriff des Algorithmus enthält, der kein exakter mathematischer Begriff ist, folglich auch kein Objekt strenger mathematischer Überlegungen sein kann.

Unsere Überzeugung von der Richtigkeit dieser Hypothese beruht hauptsächlich auf der Erfahrung. Alle bekannten Algorithmen, die im Verlauf der vieltausendjährigen Geschichte der Mathematik erdacht wurden, lassen sich mittels Turingscher Funktionsschemata beschreiben. Aber natürlich bezieht sich der Inhalt dieser Hypothese nicht nur auf die Vergangenheit und stellt nicht nur fest, daß man alle bekannten Algorithmen durch Funktionsschemata beschreiben kann, sondern er macht auch eine ganz bestimmte Aussage für die Zukunft: Wenn eine Vorschrift als Algorithmus erkannt ist, dann kann sie, unabhängig davon, in welcher Form und mit welchen Mitteln sie formuliert ist, durch das Funktionsschema einer Turing-Maschine wiedergegeben werden.

In diesem Sinne kann man die Fundamentalhypothese mit einem physikalischen Gesetz vergleichen, z. B. mit dem Gesetz von der Erhaltung der Energie, durch das ja ebenfalls Aussagen über die Zukunft gemacht werden. Die vielfältigen Erfahrungen, die in der Vergangenheit gesammelt wurden, lassen solche Prognosen als hinreichend begründet erscheinen. Wir werden hierauf im folgenden Abschnitt näher eingehen. Es sei nur noch betont, daß wir in der eigentlichen Algorithmentheorie natürlich von der Hypothese keinen Gebrauch machen, anders gesagt: Beim Beweis von Sätzen dieser Theorie wird kein Bezug auf die Hypothese genommen. Wenn also jemand die Fundamentalhypothese nicht kennt oder sie zwar kennt, aber unsere Argumente zu ihrer Stützung nicht anerkennt, so wird er in der modernen Theorie der Algorithmen auf keine formalen Schwierigkeiten stoßen. Für eine solche Person wird das, was wir als allgemeine Theorie der Algorithmen bezeichnen, nur eine Theorie der Funktionsschemata von Turing-Maschinen sein, d. h. vielleicht eine Theorie von Algorithmen einer sehr speziellen Gestalt.

Der Autor dieses Buches ist überzeugt von der Richtigkeit der Fundamentalhypothese und der sich daraus ergebenden Anerkennung der modernen Algorithmentheorie als einer Theorie, die die Natur der Dinge selbst beschreibt; er glaubt nicht, daß es sich dabei nur um eine künstlich eingeschränkte Klasse spezieller „Turing-Algorithmen" handelt.

2.7.2. Die Motivierung der Hypothese

Wir kehren nun zur Betrachtung der Fakten zurück, die unsere Überzeugung von der Gültigkeit der Hypothese bekräftigen. Zu diesen gehören in besonderem Maße die *Abgeschlossenheitssätze* und die *Gleichwertigkeitssätze*, die eine bedeutende Stellung in der Algorithmentheorie einnehmen. Auf einige von ihnen sind wir schon eingegangen, auf andere werden wir im folgenden noch stoßen.

Die allgemeine Bedeutung der Abgeschlossenheitssätze besteht darin, daß sich die Klasse der Algorithmen, die man in die Sprache der Turing-Programme „übersetzen" kann, als in bezug auf alle gebräuchlichen Verfahren zur Erzeugung neuer Algorithmen aus schon vorhandenen abgeschlossen erweist. Mit anderen Worten: Sobald für gewisse Ausgangsalgorithmen eine Beschreibung durch Turing-Programme

möglich ist, ist das auch für die daraus resultierenden komplizierteren Algorithmen der Fall. In Abschnitt 2.4 haben wir dies für so wichtige Verfahren wie die Komposition, Parallelanwendung, Verzweigung und Iteration streng bewiesen. Man kann die Liste der Verfahren wesentlich verlängern, für die man analoge Sätze streng beweisen kann, und daraus resultiert die Überzeugung, daß dieser Sachverhalt auch für alle die Verfahren zutreffen dürfte, die man gegenwärtig voraussagen kann.

Die Gleichwertigkeitssätze betreffen den Vergleich verschiedener Präzisierungen des Algorithmenbegriffs, auf die wir schon in Abschnitt 2.1 kurz hingewiesen haben. Dabei gelten zwei Algorithmenbegriffe als *gleichwertig*, wenn sie denselben Kreis von Aufgaben lösen, wenn z. B. mit ihrer Hilfe dieselben Funktionen berechnet werden können. Wir haben bereits darauf hingewiesen, daß sich viele Mathematiker bemüht haben, eine Standardform zur Beschreibung von Algorithmen zu entwickeln, die einerseits hinreichend genau und formal ist, um Gegenstand mathematischer Untersuchungen zu sein, und andererseits hinreichend allgemein, um allen bekannten und denkbaren Algorithmen eine solche Form zu verleihen. Jede solche Definition zeichnet zunächst in der verschwommenen Klasse aller Algorithmen eine gewisse scharf abgegrenzte Teilklasse von Algorithmen eines speziellen Typs aus. Es wird in diesem Buch neben der Turing-Maschine noch ein anderes abstraktes Modell eines Rechenautomaten hinreichend ausführlich untersucht werden, der sogenannte v.-Neumann-Automat (vgl. Abschnitt 3.8 bis 3.10). Schon früher (in Abschnitt 1.4.4) wurde bei der Behandlung der assoziativen Kalküle der Begriff des normalen Algorithmus erwähnt, der von A. A. Markov entwickelt wurde. Dementsprechend hat man die Klasse der Turingschen, der v.-Neumannschen und der Markovschen Algorithmen (Programme), und diese Liste könnte noch beliebig fortgesetzt werden.

Es zeigt sich jedoch — und darin besteht gerade die Aussage der Gleichwertigkeitssätze — daß alle diese und viele andere bekannte Klassen (Typen) von Algorithmen gleichwertig sind. Das bedeutet, daß für je zwei dieser Klassen, $\mathbf{K}_1$ und $\mathbf{K}_2$, folgende Behauptung gilt: Zu jedem Algorithmus $\mathfrak{A}$ aus $\mathbf{K}_1$ existiert ein gleichwertiger Algorithmus $\mathfrak{B}$ aus $\mathbf{K}_2$ (und umgekehrt). Gewöhnlich besteht der Beweis des Gleichwertigkeitssatzes in der Beschreibung einer Prozedur, die zu einem beliebig vorgegebenen Algorithmus (Programm) $\mathfrak{A}$ aus der Klasse $\mathbf{K}_1$ einen Algorithmus (Programm) $\mathfrak{B}$ der Klasse $\mathbf{K}_2$ konstruiert, der mit in $\mathbf{K}_2$ zulässigen Mitteln die Arbeit von $\mathfrak{A}$ imitiert (und umgekehrt). Unter Benutzung der Terminologie aus Abschnitt 2.4 kann man sagen, daß diese Prozedur ein programmierender Algorithmus ist, der die Programme der Klasse $\mathbf{K}_1$ in Programme der Klasse $\mathbf{K}_2$ überführt. Die Idee für eine solche *Imitation* (*Simulation*, *Modellierung*) haben wir ausführlich beim Vergleich der Turing-Maschinen mit Halbband mit den gewöhnlichen Turing-Maschinen in Abschnitt 2.6 dargestellt.

Wir haben schon in Abschnitt 2.5 bemerkt, daß unter den algorithmischen Problemen diejenigen einen besonderen Platz einnehmen, bei denen ein Algorithmus zur Berechnung der Werte einer bestimmten Funktion zu finden ist. Der Gleichwertigkeitssatz für zwei Algorithmenklassen $\mathbf{K}_1$ und $\mathbf{K}_2$ kann hier folgendermaßen formuliert werden: Die Klasse der durch Algorithmen aus $\mathbf{K}_1$ berechenbaren Funktionen

stimmt mit der Klasse aller der Funktionen überein, die durch Algorithmen aus $\mathbf{K}_2$ berechnet werden können. In diesem Sinne erweisen sich dieselben Funktionen als Turing-, Markov-, und v.-Neumann-berechenbar, und dasselbe trifft auf viele andere bekannte Präzisierungen des Begriffs „Algorithmus" zu. Man bezeichnet diese Funktionen vielfach kurz als *berechenbare Funktionen*, und in Abschnitt 2.5 wurde gezeigt, daß die Klasse der berechenbaren Funktionen mit der Klasse der rekursiven Funktionen übereinstimmt.[1]) Diese Tatsache ist natürlich kein Zufall, sie stellt vielmehr ein sehr gewichtiges Argument für die Gültigkeit der Grundhypothese dar.

[1]) Natürlich kann man auch den Begriff der rekursiven Funktion als eine Präzisierung des Algorithmenbegriffs ansehen, und sie ist in gewissem Sinne sogar die historisch älteste, die bei vielen theoretischen Untersuchungen zur Algorithmentheorie bevorzugt wird. [*Anm. d. Übers.*]

3. Algorithmische Probleme

3.1. Die universelle Turing-Maschine

Bis jetzt nahmen wir den Standpunkt ein, daß verschiedene Algorithmen durch verschiedene Turing-Maschinen realisiert werden, die sich durch ihr Funktionsschema voneinander unterscheiden. Man kann jedoch auch eine *universelle Turing-Maschine* konstruieren, die in einem gewissen Sinne jeden Algorithmus verwirklichen und damit die Arbeit jeder speziellen Turing-Maschine übernehmen kann.

3.1.1. Der Simulationsalgorithmus

Damit wir uns besser vorstellen können, wie das vor sich geht, machen wir folgendes Experiment: Auf das Band einer Maschine sei eine Anfangsinformation U geschrieben. Außerdem sei vorausgesetzt, daß ein Mensch beschreiben soll, wie diese Information von der Maschine behandelt wird und wie das Endresultat aussieht. Wenn dieser Mensch mit den Arbeitsprinzipien von Turing-Maschinen vertraut ist, braucht man ihm außer der Anfangsinformation U nur noch das Funktionsschema der Maschine mitzuteilen. Dann ist er in der Lage, indem er die Arbeit der Maschine nachahmt (simuliert), zu demselben Ergebnis zu kommen wie die Maschine. Dazu hat er der Reihe nach die Konfigurationen zu notieren, wie wir es z. B. bei der Analyse des Euklidischen Algorithmus in Abschnitt 2.3.5 getan haben. Das bedeutet aber gerade, daß dieser Mensch die Arbeit einer Turing-Maschine ausführen kann, wenn er nur ihr Funktionsschema kennt. Der Prozeß der Simulation einer Maschine gemäß ihrem Funktionsschema kann aber ganz genau beschrieben werden, und diese Beschreibung kann man einem Menschen mitteilen, auch wenn er nicht die geringste Ahnung davon hat, was eine Turing-Maschine ist. Eine solche Beschreibung soll als *Simulationsalgorithmus* bezeichnet werden. Anfangsinformation für den Simulationsalgorithmus ist außer dem Funktionsschema einer gewissen Turing-Maschine eine Anfangskonfiguration, die auf ihrem Band gespeichert ist. Ein Mensch, der über den Simulationsalgorithmus verfügt, kann dann ohne weiteres die Arbeit der entsprechenden Maschine nachahmen und am Schluß das gleiche Resultat liefern.

Den Simulationsalgorithmus kann man z. B. durch das folgende System von Anweisungen beschreiben:

Anweisung 1. Suche in der Konfiguration die (einzige) Zelle des Bandes, unter der ein Buchstabe steht.

Anweisung 2. Ermittle in der Tabelle (im Funktionsschema) die Spalte, die mit dem unter der ermittelten Zelle stehenden Buchstaben bezeichnet ist.

Anweisung 3. Gehe in der Zeile der Tabelle, die mit dem in der Zelle stehenden Buchstaben bezeichnet ist, bis zur ermittelten Spalte und notiere das dort stehende Tripel.

Anweisung 4. Ersetze den Buchstaben aus der ermittelten Zelle durch den ersten Buchstaben des notierten Tripels.

Anweisung 5. Wenn in dem notierten Tripel der zweite Buchstabe ein M ist, so ersetze den Buchstaben, der unter der ermittelten Zelle steht, durch den dritten Buchstaben des notierten Tripels.

Anweisung 6. Wenn in dem notierten Tripel der zweite Buchstabe ein L ist, so lösche den unter der ermittelten Zelle stehenden Buchstaben und setze unter die links benachbarte Zelle den dritten Buchstaben des notierten Tripels.

Anweisung 7. Wenn in dem notierten Tripel der zweite Buchstabe ein R ist, so lösche den unter der ermittelten Zelle stehenden Buchstaben und setze unter die rechts benachbarte Zelle den dritten Buchstaben des notierten Tripels.

Anweisung 8. Wenn in dem notierten Tripel der dritte Buchstabe ein „!" ist, so halte an. Der Prozeß ist beendet.

Anweisung 9. Gehe über zur Anweisung 1.

Man kann nun an die Stelle eines Menschen, der nach diesem Algorithmus arbeitet, auch eine Turing-Maschine setzen. Das ist dann die universelle Turing-Maschine, die die Arbeit jeder anderen Turing-Maschine simulieren kann. Der Simulationsalgorithmus, den wir durch ein System von neun Anweisungen festgelegt haben, kann also in passender Weise in Form eines Turingschen Funktionsschemas (universellen Schemas) wiedergegeben werden. Der vollständige und strenge Beweis dieser Tatsache, die eine weitere Bestätigung der Fundamentalhypothese der Theorie der Algorithmen ist, ist in seinen Einzelheiten zu kompliziert und zu langwierig, als daß wir ihn hier vollständig bringen könnten. Wir beschränken uns daher auf einige allgemeine Bemerkungen, die zum Verständnis des Wesens der Sache ausreichen dürften.

Wir weisen zuerst darauf hin, daß bei dem von uns beschriebenen Simulationsalgorithmus als Anfangsdaten (Anfangsinformation) das Funktionsschema der zu simulierenden Maschine und eine Anfangskonfiguration auftreten. Diese Anfangsinformation wird durch den Algorithmus zu einer Endkonfiguration verarbeitet, nämlich dem Resultat, das durch die simulierte Maschine bei Eingabe der betrach-

teten Anfangskonfiguration geliefert wird. Die universelle Maschine muß dasselbe machen. Es müssen hier jedoch zwei Umstände berücksichtigt werden:

1. Die unmittelbare Eingabe des Funktionsschemas der zu simulierenden Maschine und der entsprechenden Konfiguration auf das Band der universellen Maschine als Anfangsinformation ist unmöglich. In der universellen Maschine setzt sich nämlich, wie in jeder Turing-Maschine, eine Information aus Buchstaben ihres äußeren Alphabets zusammen, die auf ihrem Band (d. h. in einer Zeile) gespeichert sind. Wir gaben jedoch ein Funktionsschema stets durch eine „zweidimensionale" Tabelle vor, in der die Buchstaben in mehreren Zeilen und Spalten angeordnet sind. Ähnlich verhält es sich mit den Konfigurationen, bei denen die Zeichen für die Zustände unter die Buchstaben des äußeren Alphabets (unter das Band) geschrieben wurden.

2. Die universelle Maschine (wie überhaupt jede Turing-Maschine) muß ein festes endliches äußeres Alphabet besitzen. Trotzdem muß sie aber auf alle möglichen Schemata und Konfigurationen anwendbar sein, wobei Buchstaben der verschiedensten Alphabete mit einer beliebigen Anzahl von Buchstaben auftreten können.

3.1.2. Die Beschreibung der universellen Maschine

Zunächst muß in erster Linie dafür gesorgt werden, daß die Maschine überhaupt in der Lage ist, beliebige Funktionsschemata und Konfigurationen zu verarbeiten, ohne daß dabei die genannten Besonderheiten einer Turing-Maschine, nämlich die Eindimensionalität der von ihr verarbeiteten Informationen und die Endlichkeit ihres äußeren Alphabets verletzt werden. Dieser Problematik wollen wir uns jetzt zuwenden:

1. Statt als zweidimensionale Tabelle mit k Zeilen und m Spalten schreiben wir ein Funktionsschema als Wort der Länge $5km$ aus Buchstaben des inneren und äußeren Alphabets (von denen wir der Einfachheit halber voraussetzen, daß sie elementefremd sind), indem wir (in beliebiger Reihenfolge) alle möglichen Quintupel $xqx'Pq'$ für die Regeln $xq \to x'Pq'$ des Funktionsschemas hintereinander setzen. So wird beispielsweise das Schema der Abb. 12 durch folgendes Wort Ω wiedergegeben:

$$\Lambda q_1 \Lambda R q_4 \Lambda q_2 \Lambda L q_3 \Lambda q_3 \Lambda R q_1 \Lambda q_4 \Lambda M q_5 \Lambda q_5 \Lambda M q_5 | q_1 \alpha M q_2 \cdots .$$

Offenbar kann man aus dem darstellenden Wort ohne Schwierigkeiten die ursprüngliche Tabelle rekonstruieren. Analog verfährt man bei den Konfigurationen. Hier empfiehlt es sich, den Zustand statt unter den eingestellten Buchstaben unmittelbar rechts hinter ihn zu setzen. Die Konfiguration IV aus Abb. 24 wird dann durch das Wort $\mathrm{IIIII}q_4\mathrm{I}$ wiedergegeben. Offenbar kann man auch hier aus dem darstellenden Wort stets die Konfiguration rekonstruieren.

2. Zur Charakterisierung der Funktionsschemata und der Konfigurationen sind keineswegs die Zeichen für die Buchstaben des äußeren und des inneren Alphabets (der Zustände) wesentlich. Wenn beispielsweise in Abb. 12 der Buchstabe β überall

durch den Buchstaben b ersetzt wird, so ändert das gar nichts an unseren Betrachtungen. Wichtig ist nur, daß verschiedene Objekte durch verschiedene Symbole wiedergegeben werden und daß man die Buchstaben des inneren Alphabets von denen des äußeren unterscheiden kann. Insbesondere kann man natürlich auch statt L, R und M andere Zeichen für die Verschiebungen (nach links, nach rechts und keine Verschiebung) wählen, doch muß hier genau festgelegt werden, welcher Buchstabe welche Verschiebung bedeutet. Hier muß die Tatsache beachtet werden, daß jeder der drei Buchstaben eine wohlbestimmte Operation bezeichnet, die durch keine andere ersetzt werden kann.

Unter Berücksichtigung dessen können wir aber auch jeden einzelnen Buchstaben durch eine Folge z. B. von Einsen und Nullen (eine *Kodegruppe*) verschlüsseln, wobei natürlich verschiedene Buchstaben durch verschiedene Kodegruppen ersetzt werden müssen. Umgekehrt sollen selbstverständlich gleiche Buchstaben immer durch dieselbe Kodegruppe wiedergegeben werden. Damit geht jedes Wort Ω in ein Wort Ω' über, das nur aus Einsen und Nullen besteht. Damit man aber Ω aus Ω' wieder zurückgewinnen kann, muß das Kodierverfahren (die Verschlüsselung der Buchstaben durch Kodegruppen) folgende Bedingungen erfüllen:

1. Man muß jedes Wort Ω', das Kodierung eines Wort Ω ist, eindeutig in Kodegruppen zerlegen können.

2. Man muß erkennen können, welcher Kodegruppe man jedem einzelnen der Buchstaben L, R, M zuschreiben muß. Ferner müssen sich die Kodegruppen für die Buchstaben des äußeren Alphabets deutlich von den Kodegruppen für die Zustandszeichen unterscheiden.

Das folgende Kodierverfahren erfüllt offenbar diese beiden Bedingungen:

1. Als Kodegruppe nimmt man $3 + k + m$ verschiedene Wörter der Form

$$100\cdots01$$

(zwischen den Einsen stehen lauter Nullen). Dann kann ein Wort Ω', das überhaupt Kodierung eines Worts Ω ist, nur auf eine Weise in Kodegruppen zerlegt werden (genau dort, wo 11 steht, endet und beginnt jeweils eine Kodegruppe).

2. Die Buchstaben werden gemäß der Kodiertabelle auf S. 137 verschlüsselt.

In diesem Kodiersystem erhält das oben angegebene Wort die Verschlüsselung

$$10000110000011000011000110000000000011000011000000011000011011000000001\cdots.$$

Ein Wort aus Einsen und Nullen, das Verschlüsselung eines Funktionsschemas oder einer Konfiguration ist, nennen wir die *Chiffre des Funktionsschemas* bzw. die *Chiffre der Konfiguration*. Aus einer vorgegebenen Chiffre ist das Schema bzw. die Konfiguration in der ursprünglichen Form leicht reproduzierbar. Daher kann man die Vorgabe eines Schemas oder einer Konfiguration stets durch Chiffren bewerkstelligen. Anstelle der Einsen und Nullen kann man selbstverständlich auch beliebige andere Zeichen nehmen, beispielsweise a, b.

	Buchstabe	Kodegruppe		
	L	101		
	M	1001		
	R	10001		
äußeres Alphabet	s_1	100001	4 Nullen	gerade Anzahl von Nullen mindestens vier
	s_2	10000001	6 Nullen	
	.....			
	s_k	100···001	$2(k+1)$ Nullen	
inneres Alphabet (Zustandsalphabet)	q_1	1000001	5 Nullen	ungerade Anzahl von Nullen, mindestens fünf
	q_2	100000001	7 Nullen	
	.....			
	q_m	1000···001	$2(m+1)+1$ Nullen	

Jetzt ist es nicht schwer, die Formulierung der Anweisungen 1 bis 9 so abzuändern, daß man einen Simulationsalgorithmus erhält, der die Chiffre der zu simulierenden Maschine und die Chiffre einer Anfangskonfiguration in die Chiffre der Resultatkonfiguration umformt. Wir beschränken uns auf einige Beispiele:

Anweisung 1. Suche in der Konfigurations-Chiffre die (eindeutig bestimmte) Kodegruppe mit einer ungeraden Anzahl Nullen.

Anweisung 2 und 3. Suche in der Chiffre des Funktionsschemas das Paar benachbarter Kodegruppen, welches mit dem Kodegruppenpaar in der Konfigurations-Chiffre übereinstimmt, in dem die zweite Kodegruppe die ermittelte ist, und notiere das in der Chiffre des Funktionsschemas auf dieses Kodegruppenpaar folgende Kodegruppentripel.

Anweisung 5. Wenn in dem notierten Kodegruppentripel die zweite Gruppe die Gruppe 1001 ist, so ersetze in der Konfigurations-Chiffre die Kodegruppe mit der ungeraden Anzahl von Nullen durch die dritte Gruppe des notierten Tripels.

Bei der weitergehenden Untersuchung dieses Algorithmus gelangt man dahin, daß sich jede Operation mit Kodegruppen auf eine Kette von Standardoperationen zurückführen läßt, die in der Turing-Maschine ausführbar sind (Ersetzen eines Zeichens durch ein anderes, Verschiebung um einen Schritt, usw.). Dabei wird man neben den Zeichen 0 und 1, aus denen die Chiffren bestehen, noch andere Buchstaben (Hilfs- und Trennzeichen) verwenden, z. B. einen Buchstaben, der eine Chiffre von einer anderen trennt, Buchstaben, die als provisorische Kennzeichen bei der Durchmusterung der Einsen und Nullen gebraucht werden und andere.

Im Ergebnis einer solchen weiteren Aufgliederung des Simulationsalgorithmus ergibt sich schließlich die Beschreibung eines Turingschen Funktionsschemas für

eine Maschine, die im folgenden Sinne *universell* ist: Jede Aufgabe, die von einer gewissen Turing-Maschine $\mathfrak{A}$ gelöst werden kann, kann auch von der Maschine $\mathfrak{M}$ gelöst werden. Man braucht dazu nur die Chiffre des Funktionsschemas der Maschine $\mathfrak{A}$ und die Chiffre der entsprechenden Anfangskonfiguration für $\mathfrak{A}$ in geeigneter Weise auf das Band der universellen Maschine $\mathfrak{M}$ zu geben und erhält durch die die Chiffre der entsprechenden Resultatskonfiguration von $\mathfrak{A}$.

Es gilt also der folgende

Satz. *Es gibt eine universelle Turing-Maschine.*

Die Chiffre eines Funktionsschemas läßt in bezug auf die universelle Turing-Maschine folgende beiden Deutungen zu:

1. Sie beschreibt den logischen Block einer speziellen Turing-Maschine, deren Arbeitsweise auf der universellen Maschine simuliert (d. h. nachgeahmt) wird, so wie ein Rechner die Arbeit einer speziellen Turing-Maschine nachvollziehen kann; diese Auffassung haben wir bisher vertreten.

2. Sie ist ein in einer bestimmten Eingabesprache geschriebenes Programm, das in den Speicher der universellen Maschine eingegeben ist und in der Maschine einen bestimmten Algorithmus realisiert, wobei die Anfangsdaten in Form der Chiffre der Anfangskonfiguration an anderer Stelle im Speicher vorliegen.

Im Sinne der zweiten Deutung ist also die universelle Turing-Maschine mit einem modernen programmgesteuerten Rechenautomaten vergleichbar, in dessen Speicher sich ja neben den Anfangswerten für eine gestellte Aufgabe auch zugleich das zu ihrer Lösung benötigte Programm befindet.

3.2. Algorithmisch unlösbare Probleme

3.2.1. Der Satz von Church

Der Übergang vom verschwommenen intuitiven Algorithmenbegriff zum exakten Begriff der Turing-Maschine ermöglicht es nun auch, der Frage nach der algorithmischen (oder maschinellen) Unlösbarkeit einer konkreten Schar von Aufgaben einen präzisen Sinn zu geben. Die Frage lautet jetzt: Gibt es eine Turing-Maschine, die alle Aufgaben des betrachteten Typs löst, oder ist das nachweisbar nicht der Fall? Was dabei die Redeweise „die Turing-Maschine $\mathfrak{A}$ löst alle Aufgaben eines gegebenen Typs" bedeutet, wurde bereits in Abschnitt 2.2 erläutert.

Auf diese Frage gibt die Theorie der Algorithmen in einer ganzen Reihe von Fällen die Antwort: „Nein!". Insbesondere konnte im Jahre 1936 von dem amerikanischen Mathematiker A. Church, anknüpfend an grundlegende Untersuchungen des öster-

reichischen Mathematikers K. GÖDEL, das Entscheidungsproblem der mathematischen Logik (vgl. Abschnitt 2.1) negativ beantwortet werden.[1]) Er zeigte:

Satz von CHURCH. *Das Entscheidungsproblem der mathematischen Logik ist algorithmisch unlösbar.*

Dadurch war nicht nur erklärt, warum alle vorangegangenen Versuche, einen solchen Entscheidungsalgorithmus aufzustellen, erfolglos geblieben waren, sondern es war darüber hinaus bewiesen, daß derartige Versuche prinzipiell zum Scheitern verurteilt sind.

Dieser in der Theorie der Algorithmen geführte Unmöglichkeitsbeweis besitzt den gleichen Grad von Strenge wie die in anderen Gebieten der Mathematik schon früher geführten Unmöglichkeitsbeweise (wie z. B. die Unmöglichkeit der Dreiteilung eines beliebigen Winkels mit Zirkel und Lineal oder der Auflösung beliebiger algebraischer Gleichungen höheren als vierten Grades durch Radikale (vgl. Abschnitt 2.1.1) oder auch die Inkommensurabilität von Seite und Diagonale eines beliebigen Quadrats). Wir werden im folgenden die Idee für einen derartigen Unmöglichkeitsbeweis darlegen, und zwar für das sogenannte Selbstanwendbarkeitsproblem.

3.2.2. Das Anwendbarkeitsproblem, das Selbstanwendbarkeitsproblem und das Überführungsproblem

Es sei $\mathfrak{A}$ eine (durch ihr Funktionsschema gegebene) Turing-Maschine und K eine Anfangskonfiguration für $\mathfrak{A}$. Dann sind folgende beiden Fälle möglich:

1. Die Maschine $\mathfrak{A}$ ist auf die Konfiguration K anwendbar, d. h., sie gelangt von K aus in endlich vielen Takten zu einer Endkonfiguration, in der unter der eingestellten Zelle des Bandes der Stopzustand „!" notiert ist (man sagt hierfür auch, daß $\mathfrak{A}$ *für* K *konvergiert*).

2. Die Maschine $\mathfrak{A}$ ist auf die Konfiguration K nicht anwendbar, d. h., bei Anwendung von $\mathfrak{A}$ auf K tritt niemals der Stopzustand ein, der Arbeitsprozeß bricht nicht ab ($\mathfrak{A}$ *divergiert für* K).

Damit gelangen wir in natürlicher Weise zu der folgenden Problemschar:

[Spezielles Anwendbarkeits- oder Halteproblem. *Zu einem gegebenen Funktionsschema* $\mathfrak{A}$ *ist ein Algorithmus* $\mathfrak{H}_{\mathfrak{A}}$ *zu konstruieren, der auf alle Konfigurationen von* $\mathfrak{A}$ *anwendbar ist und der für eine beliebige Konfiguration* K *entscheidet, ob* $\mathfrak{A}$ *auf* K *anwendbar ist oder nicht.*

Man kann leicht Beispiele für Turing-Maschinen $\mathfrak{A}$ angeben, für die das spezielle Anwendbarkeitsproblem algorithmisch lösbar ist, d. h. für die ein Algorithmus

[1]) GÖDEL, K., Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, Monatsh. Math. Phys. **38** (1931), 349–360. CHURCH, A., A Note on the Entscheidungsproblem. J. Symb. Logic **1** (1936), 40–41. [*Anm. d. Übers.*]

(eine Turing-Maschine) $\mathfrak{H}_{\mathfrak{A}}$ angegeben werden kann, der das Verlangte leistet. Damit ergibt sich die Frage, ob es auch Turing-Maschinen $\mathfrak{A}$ gibt, für die nachweisbar kein solcher Algorithmus existiert, für die also das spezielle Halteproblem algorithmisch lösbar ist. Bevor wir auf diese Frage eingehen, betrachten wir das folgende schärfere Problem:[1])

Allgemeines Anwendbarkeits- oder Halteproblem. *Gibt es einen Algorithmus* $\mathfrak{H}$, *der auf beliebige Funktionsschemata* $\mathfrak{A}$ *und beliebige Konfigurationen* K *anwendbar ist und für jedes* $\mathfrak{A}$ *und* K *entscheidet, ob* $\mathfrak{A}$ *auf* K *anwendbar ist oder nicht?*

Hier handelt es sich also um die Frage nach einem Algorithmus, der auf alle Maschinen und auf alle Konfigurationen angewendet werden kann [während beim speziellen Halteproblem nur ein Algorithmus $\mathfrak{H}_{\mathfrak{A}}$ gesucht ist, der bei festem $\mathfrak{A}$ auf alle Konfigurationen von $\mathfrak{A}$ anwendbar ist und der auch nur für $\mathfrak{A}$ die Entscheidung über die Anwendbarkeit herbeiführt]. Zur Präzisierung des allgemeinen Halteproblems ist es erforderlich, wie schon bei der Konstruktion einer universellen Turing-Maschine in Abschnitt 3.1, die Funktionsschemata $\mathfrak{A}$ durch ihre Chiffren $\mathrm{Chif}_1(\mathfrak{A})$ und die Konfigurationen K durch ihre Chiffren $\mathrm{Chif}_2(K)$ zu ersetzen. Gefragt ist, ob es eine Turing-Maschine $\mathfrak{H}$ gibt, die z. B. auf alle Wörter der Form

$$\mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K)$$

anwendbar ist und die ein Wort der Form $\mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K)$ genau dann in „1" bzw. „0" überführt, wenn $\mathfrak{A}$ auf K anwendbar ist bzw. nicht anwendbar ist (vgl. Abschnitt 2.4.4).[2])

Dazu sei zunächst $\mathfrak{A}$ eine beliebige Turing-Maschine, deren äußeres Alphabet die Symbole 0 und 1 enthalte. Wir nennen $\mathfrak{A}$ *selbstanwendbar*, wenn $\mathfrak{A}$ auf $\mathrm{Chif}_1(\mathfrak{A})$ anwendbar ist, d. h. $\mathfrak{A}$ für $\mathrm{Chif}_1(\mathfrak{A})$ konvergiert, und *nichtselbstanwendbar*, wenn $\mathfrak{A}$ nicht auf $\mathrm{Chif}_1(\mathfrak{A})$ anwendbar ist, d. h. $\mathfrak{A}$ für $\mathrm{Chif}_1(\mathfrak{A})$ divergiert.[3]) Damit gelangen wir zu der folgenden Problemschar:

Selbstanwendbarkeitsproblem. *Gibt es einen Algorithmus* (*eine Turing-Maschine*) $\mathfrak{S}$, *der auf alle Chiffren von Funktionsschemata* $\mathfrak{A}$ *anwendbar ist und der entscheidet, ob* $\mathfrak{A}$ *selbstanwendbar ist oder nicht?*

Wir behaupten als erstes, daß folgendes gilt:

Satz. *Das Selbstanwendbarkeitsproblem ist algorithmisch unlösbar.*

---

[1]) Diese und die folgenden Ausführungen in eckigen Klammern sind Zusätze der Übersetzer.

[2]) Es wird also hier und in folgenden analogen Fällen vorausgesetzt, daß im äußeren Alphabet von $\mathfrak{H}$ die Symbole Λ, 0, 1, * (und eventuell weitere) vorkommen.

[3]) Hierbei setzen wir voraus, daß ein innerer Zustand von $\mathfrak{A}$ als Anfangszustand ausgezeichnet ist und daß Wort $\mathrm{Chif}_1(\mathfrak{A})$ in Standard-Darstellung im betrachteten Anfangszustand abgearbeitet wird. [*Anm. d. Übers.*]

Beweis (indirekt). Wir nehmen an, es gäbe eine Turing-Maschine $\mathfrak{S}$, die die Chiffre jeder selbstanwendbaren Maschine in das Symbol „1" und die Chiffre jeder nichtselbstanwendbaren Maschine in das Symbol „0" überführt. Dann kann man leicht eine Maschine $\bar{\mathfrak{S}}$ konstruieren, die die Chiffren nichtselbstanwendbarer Maschinen nach wie vor in „0" überführt, aber für die Chiffren selbstanwendbarer Maschinen divergiert. Dazu braucht man nur im Funktionsschema $\mathfrak{S}$ den Stopzustand „!" durch einen neuen Zustand q zu ersetzen und es anschließend um den Befehl $1q \to 1Mq$ zu erweitern. Offenbar wäre $\bar{\mathfrak{S}}$ auf die Chiffren nichtselbstanwendbarer Maschinen anwendbar und auf die Chiffren selbstanwendbarer Maschinen nicht anwendbar. Das ergibt jedoch einen Widerspruch: Denn wäre $\bar{\mathfrak{S}}$ auf $\mathrm{Chif}_1(\bar{\mathfrak{S}})$ anwendbar, so wäre $\bar{\mathfrak{S}}$ selbstanwendbar, und folglich dürfte $\bar{\mathfrak{S}}$ auf $\mathrm{Chif}_1(\bar{\mathfrak{S}})$ nicht anwendbar sein (da $\bar{\mathfrak{S}}$ auf die Chiffren von selbstanwendbaren Maschinen nicht anwendbar ist). Wäre jedoch $\bar{\mathfrak{S}}$ auf $\mathrm{Chif}_1(\bar{\mathfrak{S}})$ nicht anwendbar, so wäre $\bar{\mathfrak{S}}$ nichtselbstanwendbar und folglich müßte $\bar{\mathfrak{S}}$ auf $\mathrm{Chif}_1(\bar{\mathfrak{S}})$ anwendbar sein (da $\bar{\mathfrak{S}}$ auf die Chiffren aller nichtselbstanwendbaren Maschinen anwendbar ist). Damit ist der behauptete Satz bewiesen.

Mit Hilfe dieses Satzes kann man relativ einfach auch andere in der Theorie der Turing-Maschinen auftretende Probleme als algorithmisch unlösbar nachweisen. Zunächst zeigen wir, daß folgendes gilt:

Satz. *Das allgemeine Anwendbarkeitsproblem ist algorithmisch unlösbar.*

[Beweis. Zunächst konstruiert man eine Turing-Maschine $\mathfrak{C}$, die auf alle Chiffren von Turing-Maschinen anwendbar ist und für die gilt:

$$\mathfrak{C}\big(\mathrm{Chif}_1(\mathfrak{A})\big) = \mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2\big(\mathrm{Chif}_1(\mathfrak{A})\big).$$

(Bei der Bildung $\mathrm{Chif}_2(\mathrm{Chif}_1(\mathfrak{A}))$ ist $\mathrm{Chif}_1(\mathfrak{A})$ als Eingabewort für $\mathfrak{A}$ aufzufassen und in Standard-Darstellung mit einem ausgezeichneten Zustand von $\mathfrak{A}$ als Anfangszustand zu chiffrieren.) Wir nehmen nun an, es gäbe eine Turing-Maschine $\mathfrak{H}$, die das allgemeine Halteproblem löst. Dann wäre, wie man leicht nachrechnet, $\mathfrak{C} \parallel \mathfrak{H}$ eine Turing-Maschine, die das Selbstanwendbarkeitsproblem löst, aber eine solche kann es nach dem vorangehenden Satz nicht geben.

Unter Anwendung des Satzes von der universellen Turing-Maschine erhalten wir hieraus leicht, daß das spezielle Anwendbarkeitsproblem nicht immer algorithmisch lösbar ist, genauer gesagt:

Satz. *Das spezielle Anwendbarkeitsproblem für die universelle Turing-Maschine ist algorithmisch unlösbar.*

Beweis. Es sei $\mathfrak{U}$ eine universelle Turing-Maschine mit dem äußeren Alphabet S, das die Buchstaben 0, 1, * enthalte, und für die für jede Turing-Maschine $\mathfrak{A}$ und jede Konfiguration K für $\mathfrak{A}$ beispielsweise folgendes gilt:

$$\mathfrak{U}\big(\mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K)\big) = \begin{cases} \mathrm{Chif}_2\big(\mathfrak{A}(K)\big), & \text{falls } \mathfrak{A} \text{ für } K \text{ konvergiert,} \\ \text{divergent}, & \text{falls } \mathfrak{A} \text{ für } K \text{ divergiert.} \end{cases}$$

Gäbe es nun eine Turing-Maschine $\mathfrak{H}_{\mathfrak{U}}$, die das Anwendbarkeitsproblem für $\mathfrak{U}$ löst, so müßte $\mathfrak{H}_{\mathfrak{U}}$ auf alle Konfigurationen K' über S anwendbar sein, und es müßte gelten:

$$\mathfrak{H}_{\mathfrak{U}}(K') = \begin{cases} 1, & \text{falls } \mathfrak{U} \text{ für } K' \text{ konvergiert,} \\ 0, & \text{falls } \mathfrak{U} \text{ für } K' \text{ divergiert.} \end{cases}$$

Speziell wäre also $\mathfrak{H}_{\mathfrak{U}}$ auf alle Standard-Darstellungen $\mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K)$ anwendbar, und es würde gelten

$$\mathfrak{H}_{\mathfrak{U}}\big(\mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K)\big) = \begin{cases} 1, & \text{falls } \mathfrak{U} \text{ für } \mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K) \text{ konvergiert,} \\ 0, & \text{falls } \mathfrak{U} \text{ für } \mathrm{Chif}_1(\mathfrak{A}) * \mathrm{Chif}_2(K) \text{ divergiert.} \end{cases}$$

Folglich wäre $\mathfrak{H}_{\mathfrak{U}}$ eine Turing-Maschine, die das allgemeine Anwendbarkeitsproblem löst, aber eine solche kann es nach dem vorangehenden Satz nicht geben.

Die Rückführung der algorithmischen Unlösbarkeit des allgemeinen [und des speziellen] Halteproblems auf das Selbstanwendbarkeitsproblem sind bereits Beispiele für die häufig angewandte Methode der *Reduktion des Entscheidungsproblems*, die allgemein in folgendem besteht: Jedem Einzelproblem a_i aus einer Problemschar $\{a_i\}$ wird durch einen Algorithmus (d. h. in effektiver Weise) ein Problem $f(a_i)$ aus einer Problemschar $\{b_i\}$ zugeordnet, so daß aus der Lösbarkeit des Problems $f(a_i)$ stets die Lösbarkeit des Problems a_i folgt. In einem solchen Fall sagt man auch, daß die Problemschar $\{a_i\}$ durch den betreffenden Algorithmus auf die Problemschar $\{b_i\}$ *reduziert* wird. Es ist klar, daß ein Lösungsalgorithmus für die Schar $\{b_i\}$ einen Lösungsalgorithmus für die Schar $\{a_i\}$ induziert. Umgekehrt folgt aus der algorithmischen Unlösbarkeit der Schar $\{a_i\}$ die algorithmische Unlösbarkeit der Schar $\{b_i\}$. Diese Methode wird nun oft bei Prüfung auf Unentscheidbarkeit angewandt, indem man eine Problemschar $\{b_i\}$ dadurch als algorithmisch unlösbar nachweist, daß man eine bereits als algorithmisch unlösbar erkannte Problemschar $\{a_i\}$ auf sie reduziert.

Als Beispiel zeigen wir nach dieser Methode, daß das folgende Überführungsproblem algorithmisch unlösbar ist:

Allgemeines Überführungsproblem. *Gibt es einen Algorithmus, der für jede Turing-Maschine $\mathfrak{A}$ und für beliebige ihrer Konfigurationen K_1, K_2 entscheidet, ob die Maschine $\mathfrak{A}$ bei Abarbeitung der* (als Anfangskonfiguration genommenen) *Konfiguration K_1 nach einer gewissen Anzahl von Takten in die Konfiguration K_2 gelangt oder nicht* (ob mittels $\mathfrak{A}$ aus der Konfiguration K_1 die Konfiguration K_2 *erreichbar* ist)?

Wir zeigen, daß das allgemeine Halteproblem auf das allgemeine Überführungsproblem reduzierbar ist. Hierzu genügt es zu zeigen, daß man zu jeder Turing-Maschine $\mathfrak{A}$ effektiv eine Turing-Maschine $\mathfrak{A}^*$ konstruieren kann, die dasselbe äußere Alphabet wie $\mathfrak{A}$ hat und deren Zustandsmenge die Zustandsmenge von $\mathfrak{A}$ umfaßt,

so daß für jede Konfiguration K der Maschine $\mathfrak{A}$ folgendes gilt:

$$\mathfrak{A}^*(K) = \begin{cases} K^*, & \text{falls } \mathfrak{A} \text{ für } K \text{ konvergiert,} \\ \text{divergent}, & \text{falls } \mathfrak{A} \text{ für } K \text{ divergiert,} \end{cases}$$

wobei K^* eine fest vorgegebene Endkonfiguration ist (z. B. die, bei der in der eingestellten Zelle über dem Stopzustand „1" ein gegebener Buchstabe $\tau \neq \Lambda$ steht und alle anderen Zellen leer sind).[1]) Offenbar wird durch die Zuordnung

$$(\mathfrak{A}, K) \mapsto (\mathfrak{A}^*, K, K^*)$$

das als algorithmisch unlösbar erkannte allgemeine Halteproblem auf das Überführungsproblem reduziert; denn wenn mittels $\mathfrak{A}^*$ aus der Konfiguration K die Konfiguration K^* erreichbar ist, so ist (da K^* Endkonfiguration ist) $\mathfrak{A}^*(K) = K^*$, also ist $\mathfrak{A}$ für K konvergent.

[Wir überlassen es dem Leser, sich die Verhältnisse beim speziellen Überführungsproblem klar zu machen.]

3.2.3. Ein kurzer historischer Überblick

Die ersten konkreten Resultate zur algorithmischen Unlösbarkeit wurden für Probleme aus der mathematischen Logik (z. B. das Entscheidungsproblem) und der Theorie der Algorithmen (z. B. das Selbstanwendbarkeitsproblem) erzielt. Später zeigte sich, daß auch Probleme aus scheinbar viel weniger umfassenden Problemkreisen der verschiedensten Gebiete der Mathematik algorithmisch unlösbar sind.

In erster Linie muß man hier auf eine Reihe algebraischer Probleme hinweisen, die von sowjetischen Mathematikern untersucht worden sind und die zu verschiedenen Varianten des Wortproblems gehören.

Das Äquivalenzproblem für assoziative Kalküle bzw. Wortproblem für Halbgruppen (vgl. Abschnitt 1.4) wurde schon 1914 von dem norwegischen Mathematiker A. THUE formuliert. Er behandelte Algorithmen zur Lösung des Äquivalenzproblems für gewisse spezielle assoziative Kalküle. Man hat seitdem viele Versuche unternommen, einen allgemeinen Algorithmus zu konstruieren, der dieses Problem für beliebige assoziative Kalküle und für beliebige Wortpaare entscheidet (ob Äquivalenz vorliegt oder nicht). In den Jahren 1946 und 1947 konstruierten der sowjetische Mathematiker ANDREJ ANDREEVIČ MARKOV und der amerikanische Mathematiker EMIL POST unabhängig voneinander konkrete Beispiele von assoziativen Kalkülen, in denen das Äquivalenzproblem algorithmisch unlösbar ist. Also kann es auch keinen Algorithmus für beliebige assoziative Kalküle geben. Aufgrund

[1]) Die effektive Konstruktion von $\mathfrak{A}^*$ aus $\mathfrak{A}$ sei dem Leser als Übungsaufgabe überlassen.

dieses Ergebnisses stellten A. A. MARKOV und seine Schüler ganze Klassen von assoziativen Kalkülen auf, für welche die Konstruktion eines Algorithmus unmöglich ist. Im folgenden Paragraphen wird eine Variante eines Beweises für die algorithmische Unlösbarkeit des Wortproblems für assoziative Kalküle dargestellt.

Großen Eindruck machte auf die Mathematiker der Welt das Ergebnis von PETR SERGEEVIČ NOVIKOV über die algorithmische Unlösbarkeit des Identitätsproblems (Wortproblems) in der Gruppentheorie. Die Arbeit erschien im Jahre 1955.[1]) Formal ist dieses Problem ein Spezialfall des Äquivalenzproblems für assoziative Kalküle. Es werden nämlich nur assoziative Kalküle betrachtet, in denen zu jedem Buchstaben a des Alphabets eine zugelassene Substitution der Form

$$a\varphi - \Lambda$$

existiert, wobei φ ein Buchstabe desselben Alphabets ist (der mit a übereinstimmen kann). Dann ist die durch den assoziativen Kalkül erzeugte Halbgruppe eine Gruppe.

Der inhaltliche Sinn dieser zusätzlichen Forderung wird verständlich, wenn man die Wörter einer Halbgruppe (analog dem Beispiel aus Abschnitt 1.4.4) als Transformationen auffaßt, die durch Multiplikation elementarer Transformationen entstanden sind. Jedem Buchstaben eines Wortes entspricht eine elementare Transformation. Das leere Wort entspricht der identischen Transformation (die alles unverändert läßt; vgl. Abschnitt 1.4). Die zugelassene Substitution $a\varphi - \Lambda$ bedeutet dann, daß es zu der mit a bezeichneten elementaren Transformation eine elementare Transformation gibt (die mit φ bezeichnet wird), so daß ihre Hintereinanderausführung gerade die identische Transformation ergibt. Die Untersuchung solcher Transformationsgruppen ist von großem theoretischen und praktischen Interesse. Der Begriff der Gruppe ist einer der Grundbegriffe der modernen Mathematik.

Wir müssen uns noch darüber klar werden, daß das äußerst wichtige Resultat von MARKOV und POST keine Aussage über das Wortproblem in der Gruppentheorie zuläßt. Da nämlich die speziellen assoziativen Kalküle, für die A. A. MARKOV und E. POST die algorithmische Unlösbarkeit des Äquivalenzproblems bewiesen haben, keine Gruppen sind, d. h. die oben angegebene wesentliche Forderung nicht erfüllen, wird durch das Resultat von MARKOV und POST die Existenz eines Algorithmus für das Identitätsproblem der Gruppentheorie nicht ausgeschlossen. Daher hatte es zunächst durchaus noch einen Sinn, weiter nach einem solchen Algorithmus zu suchen. Alle Hoffnungen mußten jedoch aufgegeben werden, als die Arbeit von P. S. NOVIKOV bekannt wurde. In ihr wurde bewiesen, daß es einen solchen Algorithmus nicht geben kann. P. S. NOVIKOV konstruierte ein Beispiel eines assoziativen Kalküls, der die angegebene Forderung erfüllt (also ein Beispiel einer Gruppe), und bewies, daß für diesen assoziativen Kalkül das Äquivalenzproblem algorith-

[1]) Für diese Arbeit erhielt P. S. NOVIKOV 1957 den Leninpreis. (П. С. Новиков, Об алгоритмической неразрешимости проблемы тождества слов в теории групп.) Arbeiten des Steklov-Instituts für Mathematik, XLIV, Moskau 1955.

misch unlösbar ist. Also kann es erst recht keinen Algorithmus geben, der für alle endlich erzeugbaren Gruppen brauchbar ist.

Die Beispiele, die von MARKOV und NOVIKOV zum Nachweis der algorithmischen Unlösbarkeit der untersuchten Probleme konstruiert wurden, waren sehr kompliziert und wiesen Hunderte von zugelassenen Substitutionen auf. Natürlich fragte man nach möglichst einfachen Beispielen. Diese Aufgabe wurde von dem jungen Leningrader Mathematiker G. S. CEJTIN glänzend gelöst. Für die in Abschnitt 1.4.2 angegebene und CEJTIN konstruierte Halbgruppe, die nur sieben zugelassene Substitutionen enthält, ist das Äquivalenzproblem algorithmisch unlösbar.

In den letzten dreißig Jahren hat sich eine Vielzahl von mathematischen Problemen als algorithmisch unlösbar herausgestellt. Für einige von ihnen dauerte es sehr lange, bis ein Beweis gefunden wurde, obwohl sie schon lange auf der „Kandidatenliste" der unlösbaren Probleme standen. Den Rekord hatte das 10. Hilbertsche Problem inne (vgl. Abschnitt 1.1). Erst 1969 gelang es dem jungen Leningrader Mathematiker JU. V. MATIJASEVIČ, die algorithmische Unlösbarkeit dieses Problems zu beweisen. Dagegen ist für ein ähnliches Problem, das Lösbarkeitsproblem für Wortgleichungen (vgl. Abschnitt 1.4.6), die Frage nach der Existenz eines Entscheidungsalgorithmus bislang ungeklärt, obwohl auch hier stark zu vermuten ist, daß keiner existiert.

Es sei bemerkt, daß oft eine scheinbar unbedeutende und harmlose Modifikation eines Problems die Schwierigkeiten, die mit der Frage nach einem Lösungsalgorithmus verbunden sind, gründlich ändern kann, ja, daß eine geringfügige Modifizierung aus einem algorithmisch unlösbaren Problem ein lösbares machen kann und umgekehrt.[1])

Die Entdeckung algorithmisch unlösbarer Probleme führte dazu, daß ein Mathematiker, der einen bestimmten Algorithmus aufstellen möchte, damit rechnen muß, daß es den gesuchten Algorithmus gar nicht gibt. Daher muß man sich parallel zu den Bemühungen, die auf das Auffinden von Algorithmen gerichtet sind, auch mit Untersuchungen über die Nichtexistenz von Lösungsalgorithmen befassen. Unabhängig davon, welches Resultat sich schließlich einstellt, bestehen folgende Möglichkeiten: Es wird entweder ein Lösungsalgorithmus gefunden, oder es wird die algorithmische Unlösbarkeit des betrachteten Problems bewiesen.

---

[1]) Ersetzt man z. B. in der fünften zugelassenen Substitution des assoziativen Kalküls von CEJTIN (vgl. Abschnitt 1.4.2) den letzten Buchstaben *e* durch *c* (d. h., nimmt man statt der Regel *abac*→*abace* die Regel *abac*→*abacc*), so wird das Äquivalenzproblem algorithmisch lösbar. Diese Änderung hatte sich, vom Autor unbemerkt, in sein Buch „Алгоритмы и машинное решение задач" eingeschlichen [und ist dann auch in alle Auflagen von „Wieso können Automaten rechnen?" übernommen worden]. Der Leser G. N. SPIRIDOVIČ aus Leningrad schenkte dem Autor (zu Recht!) keinen Glauben, daß für den von ihm G. S. CEJTIN zugeschriebenen assoziativen Kalkül das Äquivalenzproblem algorithmisch unlösbar sei, und konnte einen Lösungsalgorithmus finden. Der Autor dankt G. N. SPIRIDOVIČ und G. S. CEJTIN dafür, daß sie ihn auf die unterlaufene Ungenauigkeit hingewiesen haben.

3.3. Die algorithmische Unlösbarkeit des Wortproblems

In diesem Abschnitt zeigen wir, daß es keinen Algorithmus geben kann, der die Äquivalenz von Wörtern in einem beliebigen assoziativen Kalkül zu erkennen gestattet. Der Beweis wird in zwei Teilen geführt. Im ersten Teil (Abschnitt 3.3.1) werden assoziative Kalküle π mit gerichteten Substitutionen der Form $P \to Q$ betrachtet (vgl. Abschnitt 1.4.1). Für diese sogenannten *einseitigen Kalküle* oder *Semi-Thue-Systeme* wird das *Übersetzungsproblem* für Wörter untersucht. Ein Wort R heißt dabei im Kalkül π *übersetzbar* in das Wort S, wenn es in π eine Ableitungskette

$$R_1 \to R_2 \to R_3 \to \cdots \to R_k$$

gibt, in der R_1 gleich R und R_k gleich S ist und wobei der Pfeil andeutet, daß das linke Wort durch einmalige Anwendung einer zugelassenen gerichteten Substitution in das rechte Wort übergeht. Wir werden sehen, daß kein Algorithmus existiert, der für einen beliebigen einseitigen Kalkül das Übersetzungsproblem für Wörter löst.

Neben einem solchen gerichteten Kalkül π betrachten wir dann jeweils auch den Kalkül π', der aus dem Kalkül π durch Ersetzung jeder zugelassenen einseitigen Substitution der Form $P \to Q$ durch die entsprechende zweiseitige Substitution $P - Q$ hervorgeht. Ist im Kalkül π von zwei Wörtern R und S wenigstens eines in das andere übersetzbar, so sind die beiden Wörter im Kalkül π' äquivalent. Die umgekehrte Behauptung ist im allgemeinen falsch. Denn beim Nachweis der Äquivalenz sind auch Ableitungsketten zugelassen, in denen neben den in π gegebenen Substitutionen der Form $P \to Q$ die umgekehrten Substitutionen der Gestalt $Q \to P$ auftreten können. Daher läßt sich das für einseitige Kalküle bewiesene Resultat nicht automatisch auf beliebige Kalküle übertragen.

Im zweiten Teil des Beweises (Abschnitt 3.3.2) wird dann gerade diese Schwierigkeit überwunden und die algorithmische Unlösbarkeit des Äquivalenzproblems nachgewiesen.

3.3.1. Die algorithmische Unlösbarkeit des Übersetzungsproblems für Wörter

Satz 1. *Es gibt keinen Algorithmus, der in einem beliebigen (einseitigen) assoziativen Kalkül für jedes Paar von Wörtern R und S entscheidet, ob R in S übersetzbar ist oder nicht.*

Zum Beweis des Satz 1 wird das allgemeine Überführungsproblem für Turing-Maschinen (vgl. Abschnitt 3.2.2) auf das Übersetzungsproblem für einseitige Kalküle reduziert. Da das erste Problem algorithmisch unlösbar ist, muß es auch das zweite sein. Die im folgenden angeführten Begriffe und Konstruktionen sind gerade für die Reduktion des ersten Problems auf das zweite notwendig.

Wir betrachten zunächst eine gewisse Konfiguration K einer Turing-Maschine. Folgende Zellen wollen wir *aktiv* in der gegebenen Konfiguration K nennen:

a) die Zelle, unter der der Zustand notiert ist,

b) alle Zellen, in denen ein vom Leerzeichen Λ verschiedenes Zeichen gespeichert ist,

c) alle Zellen, die sowohl irgendwo links als auch irgendwo rechts von sich Zellen des Typs a) oder b) haben.

Die Gesamtheit der in K aktiven Zellen bildet einen zusammenhängenden Teil des Bandes, *den in K aktiven Teil des Bandes*. In Abb. 33 sind einige Konfigurationen abgebildet, und es ist jeweils der aktive Teil des Bandes markiert. In der Konfiguration aus Abb. 33a ist die eingestellte Zelle keine Randzelle des aktiven Teils, d. h., links und rechts von ihr gibt es noch Zellen des aktiven Teils des Bandes. Eine solche

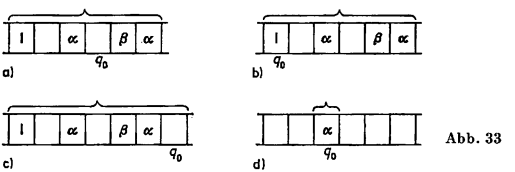


Abb. 33

Konfiguration nennen wir *innere Konfiguration*, im Unterschied zu denen in Abb. 33b, c, d, die wir als *Linkskonfiguration*, *Rechtskonfiguration* bzw. *Einerkonfiguration* bezeichnen wollen.

Es sei $\{s_1, \ldots, s_k\}$ das äußere Alphabet der Maschine $\mathfrak{M}$, wobei s_1 das Leerzeichen Λ ist, und es sei $\{q_1, \ldots, q_m\}$ das Alphabet ihrer inneren Zustände. Zur Bezeichnung der Enden des aktiven Teils des Bandes führen wir noch zusätzlich den Buchstaben h ein, der nicht in den angegebenen Alphabeten vorkommen soll. Man kann dann jede Konfiguration eindeutig in der Form hRh schreiben, wobei R ein wie in Abschnitt 3.1.2 gebildetes Wort ist. Wir nennen diese Wörter *K-Wörter* (*Konfigurationswörter*). Beispielsweise entsprechen den Konfigurationen in Abb. 33 die K-Wörter $h\,|\,\Lambda\alpha\Lambda q_0\beta\alpha h$, $h\,|\,q_0\Lambda\alpha\Lambda\beta\alpha h$, $h\,|\,\Lambda\alpha\Lambda\beta\alpha\Lambda q_0 h$ und $h\alpha q_0 h$.

Jeder Maschine $\mathfrak{M}$ ordnen wir nun folgendermaßen einen Kalkül $\pi_{\mathfrak{M}}$ zu:

1. Das Alphabet von $\pi_{\mathfrak{M}}$ besteht aus dem Buchstaben h und allen Buchstaben der Alphabete der Maschine $\mathfrak{M}$. Dann ist zwar jedes K-Wort ein Wort im Kalkül $\pi_{\mathfrak{M}}$, aber nicht umgekehrt jedes Wort des Kalküls $\pi_{\mathfrak{M}}$ ein K-Wort. Das Wort $hs_1q_1s_1q_1h$ kann z. B. kein K-Wort sein, da in ihm der Buchstabe q_1 zweimal vorkommt.

2. Die zugelassenen (gerichteten) Substitutionen des Kalküls $\pi_{\mathfrak{M}}$ werden so gewählt, daß sie genau diejenigen Umformungen von K-Wörtern ermöglichen, die

den Umformungen der Konfigurationen in der Maschine gemäß deren Befehlen entsprechen. Wir beschreiben ausführlicher, wie man das realisieren kann:

Wir betrachten zunächst einen Befehl der Form

$$sq \to s'Mq', \tag{1}$$

bei dem die eingestellte Zelle nicht verlassen wird. Man zeigt leicht, daß sich der aktive Teil des Bandes bei Anwendung von Befehlen der Form (1) nicht ändert. Wenn wir das K-Wort vor Ausführung des Befehls mit dem K-Wort vergleichen, das danach vorliegt, so sehen wir, daß dabei einfach das Buchstabenpaar sq durch das Buchstabenpaar $s'q'$ ersetzt wird. Dementsprechend ordnen wir im Kalkül $\pi_{\mathfrak{M}}$ dem Befehl (1) der Turing-Maschine die gerichtete Substitution $sq \to s'q'$ zu.

Bei einem Befehl mit einer Verschiebung kann sich dagegen, in Abhängigkeit vom Charakter der Konfiguration (innere Konfiguration, Linkskonfiguration usw.) und der Verschiebung (nach links bzw. nach rechts), der aktive Teil des Bandes ändern. Daher läßt sich ein solcher Befehl im Kalkül nicht durch eine einzige gerichtete Substitution wiedergeben (in dem Sinne, wie das beim Befehl (1) der Fall ist). Wir werden jedoch sehen, daß sich ein endliches System von Substitutionen angeben läßt, das insgesamt einem solchen Befehl gleichwertig ist.

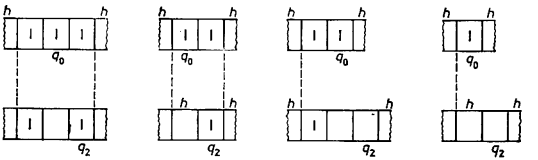


Abb. 34

Beispiel. Der Befehl $\mathrm{I}\, q_0 \to \Lambda R q_2$ aus dem Funktionsschema für die Summenbildung (vgl. Abschnitt 2.3.3, Abb. 20) bewirkt eine Umformung der Konfigurationen, wie sie in Abb. 34 beschrieben ist. Im Fall der Abb. 34a (innere Konfiguration) hat sich der aktive Teil des Bandes nicht geändert. In Abb. 34b (Linkskonfiguration) bzw. 34c (Rechtskonfiguration) wurde er verkürzt (von links) bzw. verlängert (nach rechts). In Abb. 34d (Einerkonfiguration) ist eine Verschiebung des aktiven Teils des Bandes erfolgt, ohne daß sich sein Umfang geändert hat. Die entsprechenden Transformationen der K-Wörter sind in Tabelle 3 dargestellt. Man sieht leicht, daß die Wörter der rechten Spalte nicht mit Hilfe ein und derselben gerichteten Substitution aus den entsprechenden Wörtern der linken Spalte gewonnen werden können.

Tabelle 3

Ausgangs-K-Wort	transformiertes K-Wort
$hllq_0lh$	$hl\Lambda lq_2h$
hlq_0lh	hlq_2h
$hllq_0h$	$hl\Lambda\Lambda q_2h$
hlq_0h	$h\Lambda q_2h$

Wir geben nun an, wie das einem Befehl der Form

$$sq \to s'Rq' \tag{2}$$

entsprechende System von zugelassenen gerichteten Substitutionen konstruiert werden muß. (Die Befehle der Form $sq \to s'Lq'$ werden analog behandelt.)

Wir führen dazu folgende Bezeichnungen ein: Ist die an die eingestellte Zelle nach links angrenzende Zelle aktiv, so bezeichnen wir den in ihr untergebrachten Buchstaben mit σ. Analog bezeichnet τ den in der nach rechts benachbarten Zelle stehenden Buchstaben, falls diese aktiv ist. Dabei ist der Fall nicht ausgeschlossen, daß σ oder τ oder beide das Leersymbol Λ sind.

Die dem Befehl (2) entsprechenden zugelassenen Substitutionen klassifiziert man zweckmäßigerweise nach den Konfigurationstypen:

1. Innere Konfiguration. In dem betrachteten K-Wort ist ein Teilwort der Form $\sigma sq\tau$ zu substituieren. Für jedes mögliche derartige Vorkommen (wobei also σ und τ beliebige Buchstaben des äußeren Alphabets der Maschine sein können) wird die entsprechende Substitution $\sigma sq\tau \to \sigma s'\tau q'$ benötigt.

2. Linkskonfiguration. In dem betrachteten K-Wort ist ein Teilwort der Form $hsq\tau$ zu substituieren. Die hierfür benötigten Substitutionen haben die Form

$$hsq\tau \to hs'\tau q', \quad \text{falls } s' \neq \Lambda,$$

$$hsq\tau \to h\tau q', \quad \text{falls } s' = \Lambda.$$

Die letzte Substitution drückt die Tatsache aus, daß im Fall $s' = \Lambda$ die zuvor eingestellte, s enthaltende Zelle nicht aktiv bleibt (vgl. Abb. 35, S. 150).

3. Rechtskonfiguration. In dem betrachteten K-Wort ist ein Teilwort K der Form σsqh zu substituieren. Die hierfür benötigen Substitutionen haben die Form $\sigma sqh \to \sigma s'\Lambda q'h$. Sie drücken die Tatsache aus, daß der aktive Teil des Bandes nach rechts verlängert wird (vgl. Abb. 36, S. 150).

4. Einerkonfiguration. In dem betrachteten K-Wort ist ein Teilwort der Form $hsqh$ zu substituieren. Die hierfür benötigten Substitutionen haben die Form

$$hsqh \to hs'\Lambda q'h, \quad \text{falls } s' \neq \Lambda,$$

$$hsqh \to h\Lambda q'h, \quad \text{falls } s' = \Lambda.$$

Die letzte Substitution drückt die Tatsache aus, daß im Fall $s' = \Lambda$ die einzige aktive Zelle lediglich verschoben wird (vgl. Abb. 37).

Damit ist die Liste der einem Maschinenbefehl der Form (2) entsprechenden zugelassenen gerichteten Substitutionen des Kalküls $\pi_{\mathfrak{M}}$ abgeschlossen.

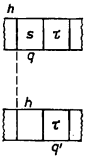


Abb. 35

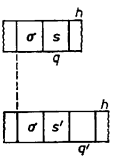


Abb. 36

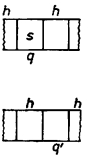


Abb. 37

Beispiel. Man konstruiere den der Turing-Maschine Σ für die Addition (vgl. Abschnitt 2.3.3, Abb. 20) entsprechenden Kalkül π_{Σ}.

Das Alphabet von π_{Σ} ist $\{|, \Lambda, *, q_0, q_1, q_2, !, h\}$. Dem Befehl $\Lambda q_2 \to |Mq_1$ entspricht die einzige Substitution $\Lambda q_2 \to |q_1$. Dem Befehl $|q_0 \to \Lambda R q_2$ entsprechen die folgenden Substitutionen

für die inneren Konfigurationen

$$\begin{cases} ||q_0| \to |\Lambda|q_2 \\ ||q_0\Lambda \to |\Lambda\Lambda q_2 \\ ||q_0 * \to |\Lambda * q_2 \\ \Lambda|q_0| \to \Lambda\Lambda|q_2 \\ \Lambda|q_0\Lambda \to \Lambda\Lambda\Lambda q_2 \\ \Lambda|q_0 * \to \Lambda\Lambda * q_2 \\ *|q_0| \to *\Lambda|q_2 \\ *|q_0\Lambda \to *\Lambda\Lambda q_2 \\ *|q_0 * \to *\Lambda * q_2 \end{cases}$$

für die Linkskonfigurationen

$$\begin{cases} h|q_0| \to h|q_2 \\ h|q_0\Lambda \to h\Lambda q_2 \\ h|q_0 * \to h * q_2 \end{cases}$$

für die Rechtskonfigurationen

$$\begin{cases} ||q_0 h \to |\Lambda\Lambda q_2 h \\ \Lambda|q_0 h \to \Lambda\Lambda\Lambda q_2 h \\ *|q_0 h \to *\Lambda\Lambda q_2 h \end{cases}$$

für die Einerkonfigurationen $\quad h|q_0 h \to h\Lambda q_2 h.$

Analog erhält man die den übrigen Befehlen entsprechenden Substitutionen.

Wir bemerken nun die folgenden Eigenschaften des zu einer Maschine $\mathfrak{M}$ konstruierten Kalküls $\pi_{\mathfrak{M}}$:

Behauptung 1. *Jedes K-Wort für die Maschine $\mathfrak{M}$ ist ein Wort in $\pi_{\mathfrak{M}}$.*

Behauptung 2. *Ist $\tilde{K}$ das K-Wort, das die Konfiguration K beschreibt, so ist auf $\tilde{K}$ in $\pi_{\mathfrak{M}}$ nicht mehr als eine zugelassene Substitution an genau einer Stelle anwendbar, und diese Substitution führt $\tilde{K}$ in das K-Wort $\tilde{K}'$ für diejenige Konfiguration K' über, in die die Maschine $\mathfrak{M}$ die Konfiguration K überführt.*

Behauptung 3. *Ist K Endkonfiguration für die Maschine $\mathfrak{M}$, so ist auf $\tilde{K}$ keine zugelassene Substitution mehr anwendbar.*

Aus diesen Behauptungen folgt unmittelbar, daß eine Konfiguration K_1 in der Maschine $\mathfrak{M}$ genau dann in eine Konfiguration K_2 überführbar ist, wenn das K-Wort $\tilde{K}_1$ im Kalkül $\pi_{\mathfrak{M}}$ in das K-Wort $\tilde{K}_2$ übersetzt werden kann. Also ist das Überführungsproblem für Turing-Maschinen auf das Übersetzungsproblem für Kalküle mit gerichteten Substitutionen reduzierbar. Damit ist Satz 1 bewiesen.[1])

Läßt man für K_2 nur Endkonfigurationen der Maschine $\mathfrak{M}$ zu, so erhält man aus der bewiesenen Reduzierbarkeit den folgenden Satz:

Satz 2. *Es gibt keinen Algorithmus, der für einen beliebigen assoziativen Kalkül mit gerichteten Substitutionen und für ein beliebiges Paar $\tilde{K}_1$, $\tilde{K}_2$ von Wörtern, dessen zweites Wort ein Endwort ist, entscheidet, ob $\tilde{K}_1$ in $\tilde{K}_2$ übersetzbar ist oder nicht.*

3.3.2. Die Unentscheidbarkeit des Äquivalenzproblems

Es seien R und S zwei K-Wörter im Kalkül $\pi_{\mathfrak{M}}$. Ist R durch gerichtete Substitutionen in S übersetzbar, so sind R und S in $\pi'_{\mathfrak{M}}$ erst recht äquivalent. Tauchen nun in $\pi'_{\mathfrak{M}}$ neue Äquivalenzen dadurch auf, daß wir die in $\pi_{\mathfrak{M}}$ zugelassenen Substitutionen als ungerichtete Substitutionen auffassen? Das folgende Lemma beantwortet diese Frage.

Lemma. *Entspricht das Wort S einer Endkonfiguration von $\mathfrak{M}$, und ist R zu S in $\pi'_{\mathfrak{M}}$ äquivalent, so ist R bereits allein durch gerichtete Substitutionen in S überführbar.*

Aus diesem Lemma folgt unmittelbar, daß auch das Äquivalenzproblem für Wörter in ungerichteten assoziativen Kalkülen (Thue-Systemen) algorithmisch unlösbar ist.

[1]) Wählt man als $\mathfrak{M}$ eine Maschine, für die das spezielle Überführungsproblem algorithmisch unlösbar ist (z. B. eine universelle Turing-Maschine), so gelangt man zu folgendem allgemeineren Satz: *Es läßt sich ein assoziativer Kalkül π mit gerichteten zugelassenen Substitutionen angeben, für den das (spezielle) Übersetzungsproblem algorithmisch unlösbar ist, d. h., für den nachweisbar kein Algorithmus existiert, um für beliebige Wortpaare P, R aus dem Alphabet von π zu entscheiden, ob im Kalkül π das Wort P in das Wort R übersetzt werden kann oder nicht.* Auf diesem Wege wurden übrigens von A. A. Markov die ersten Beispiele für assoziative Kalküle mit algorithmisch unlösbarem Äquivalenzproblem konstruiert, die auf Hunderten von zugelassenen Substitutionen beruhten. [*Anm. d. Übers.*]

Denn ein derartiger Algorithmus löste zugleich das Übersetzungsproblem für Wörter in Endwörter mit Hilfe gerichteter Substitutionen, aber nach Satz 2 kann ein solcher Algorithmus nicht existieren.

Zum Beweis des Lemmas. Es gelte $R \sim S$. Dann gibt es im Kalkül $\pi'_{\mathfrak{M}}$ eine Ableitungskette, die von R zu S führt:

$$R = R_1 - R_2 - R_3 - \cdots - R_{k-1} - R_k = S. \tag{3}$$

Es seien R_j und R_{j+1} zwei benachbarte Wörter in dieser Kette. Wird der Übergang von R_j zu R_{j+1} durch eine zugelassene gerichtete Substitution des Kalküls $\pi_{\mathfrak{M}}$ realisiert, so schreiben wir

$$R_j \to R_{j+1}.$$

Wird dagegen der Übergang von R_j zu R_{j+1} durch Ersetzen der rechten Seite einer zugelassenen Substitution des Kalküls $\pi_{\mathfrak{M}}$ durch deren linke Seite realisiert (oder geht, was dasselbe besagt, R_{j+1} durch Anwendung einer zugelassenen gerichteten Substitution in R_j über), so schreiben wir

$$R_j \leftarrow R_{j+1}.$$

Wir betrachten zunächst die folgenden möglichen Fälle für ein Worttripel in der Kette (3):

$$R_{j-1} \leftarrow R_j \to R_{j+1} \tag{4}$$

$$R_{j-1} \to R_j \leftarrow R_{j+1}. \tag{5}$$

Nach Behauptung 2 aus Abschnitt 3.3.1 müssen im Fall (4) die Wörter R_{j-1} und R_{j+1} übereinstimmen, denn auf das Wort R_j ist ja nur eine einzige Substitution aus $\pi_{\mathfrak{M}}$ anwendbar. Wenn in der Ableitungskette (3) also ein derartiges Tripel auftritt, so kann man sie dadurch verkürzen, daß man zwei Wörter entfernt (beispielsweise R_{j-1} und R_j). Im Fall (5) können dagegen die Wörter R_{j-1} und R_{j+1} durchaus verschieden sein. In der Sprechweise der Turing-Maschinen bedeutet das, daß man aus einer gegebenen Konfiguration der Maschine im allgemeinen nicht eindeutig die vorhergehende Konfiguration rekonstruieren kann, daß zwei verschiedene Konfigurationen dieselbe Nachfolgekonfiguration haben können.

Wir kehren nun zur Betrachtung der Kette (3) zurück. Da R_k Endkonfiguration ist, kann nur $R_{k-1} \to R_k$ gelten (vgl. Behauptung 3 aus Abschnitt 3.3.1). Zeigen in der Kette alle Pfeile von links nach rechts, so ist das Lemma bewiesen. Existieren jedoch Pfeile von rechts nach links, so betrachten wir den letzten derartigen Pfeil. Er möge vor dem j-ten Wort stehen. Dann kommt in der Kette aber das Tripel $R_{j-1} \leftarrow R_j \to R_{j+1}$ vor, so daß diese um zwei Wörter verkürzt werden kann, und wir erhalten so eine kürzere Ableitungskette, die von R zu S führt. Setzen wir diese Prozedur so lange fort, bis wir zu einer Kette kommen, die nicht mehr verkürzt werden kann, so sind alle Pfeile in ihr von links nach rechts gerichtet. Also ist R in der Tat mit Hilfe von gerichteten Substitutionen in S überführbar.

3.4. Die Qualität von Algorithmen und Berechnungen

Gewöhnlich gibt man sich nicht einfach mit der Feststellung zufrieden, daß für eine gegebene Klasse von Aufgaben ein sie lösender Algorithmus existiert, sondern man bemüht sich, einen möglichst zweckmäßigen Algorithmus zu finden. Die Beurteilung der Qualität von Algorithmen kann nun nach ganz unterschiedlichen Gesichtspunkten erfolgen: Eine erste Möglichkeit besteht darin, die Kompliziertheit der Beschreibung eines Algorithmus als Maß für seine Güte zu nehmen; so kann man z. B. einfach die Anzahl der Befehle (Anweisungen) abzählen, aus denen er sich zusammensetzt, oder dabei auch noch die Länge der Niederschriften der einzelnen Befehle in Rechnung stellen. Eine andere Möglichkeit ist, die Kompliziertheit der Rechenprozesse zu berücksichtigen, die bei dem gegebenen Algorithmus zu realisieren sind. So kann man beispielsweise die Elementaroperationen abzählen, die man braucht, um das Resultat zu erhalten, oder den Umfang des dabei benutzten „Speichers". Dabei ist folgendes zu beachten: Es kann passieren, daß man zur Lösung gewisser Probleme einen relativ einfach und kurz zu formulierenden Algorithmus vorlegen kann, dessen tatsächliche Anwendung aber sehr umfangreiche Berechnungen erfordert, die z. B. im Durchmustern einer Unmenge von Varianten bestehen können. Es kann also vorkommen, daß sich eine gewisse Ausführungsvorschrift selbst sehr einfach und kurz formulieren läßt, daß die Durchführung der durch sie beschriebenen Handlung jedoch sehr aufwendig ist. Das trifft z. B. auf den Suchalgorithmus für eine Gewinnstrategie in einem Spiel zu (vgl. Abschnitt 1.2). Umgekehrt gibt es hinreichend kompliziert zu beschreibende Verfahren, deren Ausführung relativ schnell zum Ziel führt.

3.4.1. Signalisierende Funktionen

Wir definieren jetzt Begriffe, die im Sinne der zweiten der genannten Möglichkeiten die Kompliziertheit der Berechnungen auf Turing-Maschinen beschreiben. Das wird es uns gestatten, die Frage, um wieviel leichter dieses als jenes Problem lösbar ist, zu präzisieren und manchmal sogar zu beantworten.

Es sei P ein Wort auf dem Band einer Turing-Maschine $\mathfrak{M}$, das eine gegebene Anfangsinformation darstellt. Mit $t_{\mathfrak{M}}(P)$ bezeichnen wir die Anzahl der Arbeitstakte, die die Maschine $\mathfrak{M}$ braucht, um zur zugehörigen Endkonfiguration zu gelangen. Wir betrachten weiterhin das kleinste Stück des Bandes, welches das Ausgangswort P und alle die Zellen enthält, die im Laufe dieses Prozesses wenigstens einmal durch den Kopf der Maschine berührt werden. Die Länge dieses Abschnitts ist gerade der Umfang des von der Maschine $\mathfrak{M}$ während der Rechnung effektiv benutzten äußeren Speicher. Diese Länge bezeichnen wir mit $s_{\mathfrak{M}}(P)$. Da man die Ausgangsgröße P variieren kann, haben wir damit für die gegebene Maschine $\mathfrak{M}$ zwei Funktionen des Arguments P definiert, die wir die *Zeitsignalisierende* bzw. die *Speichersignalisierende* der Maschine $\mathfrak{M}$ nennen wollen; sie messen den Aufwand an Zeit bzw. an Speicherraum für Berechnungen auf der Maschine $\mathfrak{M}$.

Neben diesen Funktionen benutzt man zweckmäßigerweise noch aus ihnen abgeleitete Funktionen mit natürlichen Zahlen als Argumenten, die zu einer Maschine $\mathfrak{M}$ und zu einem festen Teilalphabet A ihres äußeren Alphabets folgendermaßen definiert werden:

$T_{\mathfrak{M}}(n) = \max t_{\mathfrak{M}}(P)$ über alle Wörter P der Länge n im Alphabet A;

$S_{\mathfrak{M}}(n) = \max s_{\mathfrak{M}}(P)$ über alle Wörter P der Länge n im Alphabet A.

Von jetzt an bezeichnen wir die Länge eines Wortes P stets mit $|P|$. Die Funktionen $T_{\mathfrak{M}}(n)$ und $S_{\mathfrak{M}}(n)$ bezeichnen wir ebenfalls als Zeit- bzw. Speichersignalisierende; aus dem Kontext wird jeweils klar sein, welche der Signalisierenden gemeint sind. Selbstverständlich ist eine Berechnung auf einer betrachteten Maschine umso einfacher, je kleiner diese Funktionen sind, d. h., je langsamer sie mit wachsendem n wachsen. Meistens ist es nicht möglich (und auch nicht notwendig), genaue und übersehbare Formeln für die uns interessierenden Funktionen $T_{\mathfrak{M}}(n)$, $S_{\mathfrak{M}}(n)$ anzugeben. Nicht selten lassen sich jedoch näherungsweise Ausdrücke für sie finden, und man kann oft hinreichend gute untere oder obere Abschätzungen angeben.

3.4.2. Abschätzungen der Zeitsignalisierenden für die Beispiele aus Abschnitt 2.3

Wir betrachten als Beispiele zunächst die Turing-Maschinen, die wir in Abschnitt 2.3 zur Lösung der folgenden drei Aufgaben konstruiert haben:

I. Übergang von n zu $n + 1$ im Dezimalsystem;

II. Übersetzung der unären Darstellung einer Zahl in ihre Dezimaldarstellung;

III. Addition von in unärer Darstellung gegebenen Zahlen.

Wir bezeichnen die entsprechenden Maschinen mit $\mathfrak{M}_1$, $\mathfrak{M}_2$ und $\mathfrak{M}_3$ und betrachten ihre Zeitsignalisierenden ausführlicher. Dabei benutzen wir jene Abschätzungen, die in Abschnitt 2.3 im Zusammenhang mit der Diskussion der Aufgaben I bis III und ihrer Lösung auf den Maschinen $\mathfrak{M}_1$, $\mathfrak{M}_2$, $\mathfrak{M}_3$ angegeben wurden.

Wir erinnern daran, daß $\mathfrak{M}_1$ auf Wörter P angewandt wird, die Dezimaldarstellungen natürlicher Zahlen sind. Wie in Abschnitt 2.3.1 schon bemerkt wurde, arbeitet die Maschine $\mathfrak{M}_1$ am längsten auf den Wörtern P, die nur aus Neunen bestehen. Für diese ist

$$t_{\mathfrak{M}_1}(P) = |P| + 1.$$

Also ist $T_{\mathfrak{M}_1}(n) = n + 1$.

Die Maschine $\mathfrak{M}_2$ wird auf Wörter P angewandt, die unäre Darstellungen natürlicher Zahlen sind. Dabei bedeutet $|P| = n$, daß P aus n Strichen besteht. Als Resultat erhält man die Dezimaldarstellung R der Zahl n, wobei

$$|R| \leqq]\log_{10} n[+ 1$$

ist. Im Abschnitt 9.2 wurde bereits folgende Ungleichung erhalten:

$$n(n+3) \leqq T_{\mathfrak{M}_2}(n) \leqq n(n+3) + 2n(\lfloor \log_{10} n \rfloor + 1).$$

Die untere Schranke $n(n+3)$ und die obere Schranke $n(n+3) + 2n(\lfloor \log_{10} n \rfloor + 1)$ stimmen hier nicht überein. Für $n \to \infty$ strebt ihr absoluter Fehler, der bei Ersetzen der einen Größe durch die andere entsteht, sogar gegen ∞. Es ist jedoch von Nutzen zu bemerken, daß für $n \to \infty$ das Verhältnis dieser Größen gegen 1 strebt, d. h., der relative Fehler strebt für $n \to \infty$ gegen Null. In Übereinstimmung mit einer allgemein üblichen Terminologie nennen wir arithmetische Funktionen $f(n)$ und $g(n)$, für die

$$\lim_{n\to\infty} f(n) = \infty, \quad \lim_{n\to\infty} g(n) = \infty \quad \text{und} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

gilt, *asymptotisch gleich* und schreiben dafür auch $f(n) \sim g(n)$. Damit können wir sagen, daß die als untere und als obere Abschätzung für $T_{\mathfrak{M}_2}(n)$ erhaltenen Funktionen asymptotisch gleich sind. Damit ist natürlich auch jede von ihnen zur Signalisierenden $T_{\mathfrak{M}_2}(n)$ asymptotisch gleich. Also haben wir für $T_{\mathfrak{M}_2}(n)$ die im Sinne der asymptotischen Gleichheit geltende Abschätzung

$$T_{\mathfrak{M}_2}(n) \sim (3+n)\, n \sim n^2.$$

Die Maschine $\mathfrak{M}_3$ wird auf Wörter P der Form

$$\underbrace{|| \ldots ||}_{k\text{-mal}} * \underbrace{|| \ldots ||}_{s\text{-mal}}$$

angewandt. Dabei wird die Zeit $t_{\mathfrak{M}_3}(P) = 2k(k+s+1)$ benötigt (vgl. Abschnitt 2.3.3). Ist $|P| = k + s + 1 = n$, so wird das Maximum der Rechenzeit für $k = n - 1$ erreicht. Folglich gilt

$$T_{\mathfrak{M}_3}(n) = 2(n-1)\, n \sim 2n^2.$$

3.4.3. Die Suche nach einem besten Algorithmus

Sobald wir einen Algorithmus zur Lösung einer Aufgabenschar haben, ist es leicht, andere Algorithmen zu konstruieren, die dasselbe tun, aber bedeutend langsamer sind. Es ist natürlich wichtiger zu wissen, ob man einen besten Algorithmus konstruieren kann und wie man einen solchen findet. Wir wenden uns wieder den Beispielen I bis III zu und versuchen zu klären, ob es für sie Turing-Programme gibt, die schneller zum Resultat führen, d. h., für die die Zeitsignalisierenden langsamer wachsen. Im Fall I ist es klar, daß eine Beschleunigung unmöglich ist, denn für das Wort $99\cdots9$ (n Ziffern) braucht man, wie man es auch macht, wenigstens $n+1$ Takte zum Aufschreiben des Resultats (einer Eins und n Nullen). Im Fall II kann man jedoch ein schneller arbeitendes Programm $\mathfrak{M}_2'$ konstruieren. Wir schreiben es nicht

auf, sondern geben nur an, wie der entsprechende Rechenprozeß verläuft und worin er sich von dem Prozeß unterscheidet, der nach dem Programm $\mathfrak{M}_2$ abläuft (d. h. mit dem Schema in Abb. 18): In jedem Zyklus wird der am weitesten links stehende Strich des noch vorhandenen Blocks von Strichen gelöscht (und nicht der am weitesten rechts stehende Strich, wie beim Programm $\mathfrak{M}_2$). Der andere Unterschied besteht

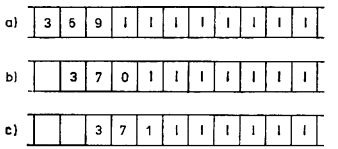


Abb. 38

darin, daß die Dezimaldarstellung der im folgenden Zyklus abzuarbeitenden Zahl um eine Zelle nach rechts verrückt und damit direkt an den verbliebenen Block von Strichen herangeführt wird. In Abb. 38 sind die Niederschriften auf dem Band nach dem 369., 370. und 371. Zyklus und ihre gegenseitige Lage dargestellt. Offenbar ist die Dauer des j-ten Zyklus nicht länger als $2\,(]\log_{10} j[+ 1) + 1$ Takte, davon wird ein Takt zum Löschen des Strichs benötigt, während für den Übergang zur Dezimaldarstellung der Zahl j und den Transport ihrer Ziffern um eine Zelle nach rechts höchstens $2\,(]\log_{10} j[+ 1)$ Takte benötigt werden. Also gilt

$$T_{\mathfrak{M}_3'}(n) \leqq 2 \sum_{j=1}^{n} (]\log_{10} j[+ 1) + n \leqq 2n(]\log_{10} n[+ 1) + n \sim 2n \cdot \log_{10} n\,.$$

Das ist aber wesentlich besser als $T_{\mathfrak{M}_2}(n)$. Es gilt sogar:

$$\lim_{n \to \infty} \left(T_{\mathfrak{M}_3'}(n) / T_{\mathfrak{M}_2}(n)\right) = 0\,,$$

d. h., $T_{\mathfrak{M}_3'}$ ist von kleinerer Ordnung als $T_{\mathfrak{M}_2}$. Der erhaltene Zeitgewinn zeigt, daß es besser ist, in jedem Zyklus die Dezimaldarstellung zu verschieben, deren Länge relativ klein ist, als den nächsten umzuwandelnden Strich vom rechten Ende zu holen, das weit entfernt liegt.

Kann man dieses Resultat noch weiter verbessern? Gibt es vielleicht ein Turing-Programm $\mathfrak{M}_2''$, so daß $T_{\mathfrak{M}_2''}$ von kleinerer Ordnung als $T_{\mathfrak{M}_3'}$ ist? Wir kommen auf diese Frage später zurück (vgl. Abschnitt 3.6.2).

Zur Lösung der Aufgabe III kann man schließlich ein Programm $\mathfrak{M}_3'$ benutzen, bei dem zunächst der Stern durch einen Strich ersetzt und anschließend der am weitesten links stehende Strich gelöscht wird. Offensichtlich ist $T_{\mathfrak{M}_3'}(n) = n + 1$ von kleinerer Ordnung als $T_{\mathfrak{M}_3}(n)$ (denn $T_{\mathfrak{M}_3}(n)$ ist asymptotisch gleich n^2). Es ist außerdem hinreichend klar, daß eine weitere Beschleunigung des Prozesses unmöglich ist.

3.4.4. Die Erkennung der Symmetrie

Wir betrachten jetzt die folgende Schar von Aufgaben gleichen Typs: Von einem beliebigen Wort $P = P(1)P(2)\cdots P(n)$ im Alphabet {0, 1} ist festzustellen, ob es symmetrisch ist oder nicht, d. h., ob es mit seinem Spiegelbild $P(n)P(n-1)\cdots P(1)$ übereinstimmt. Wir nehmen wie üblich an, daß der entsprechende *Erkennungsalgorithmus* im Fall einer positiven Antwort eine Eins und im Fall einer negativen Antwort eine Null ausgibt. Man kann leicht Turing-Programme angeben, die dieses Problem lösen; wir beschränken uns darauf, den Ablauf eines solchen Rechenprozesses zu beschreiben. Die Arbeit beginnt mit der Konfiguration aus Abb. 39. Der Kopf merkt sich das Symbol $P(1)$, löscht es, begibt sich zum rechten Ende des Wortes und vergleicht $P(1)$ mit dem dort stehenden $P(n)$. Sind diese Symbole verschieden, so ist das Wort P nicht symmetrisch, die zu Beginn der Rechnung auf dem Band

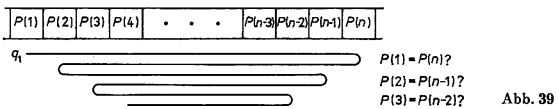


Abb. 39

stehende Niederschrift wird gelöscht, und es wird auf das Band das Resultat „0" geschrieben. Im anderen Fall kehrt der Kopf nach links zurück, sucht $P(2)$ und beginnt den zweiten Zyklus, in dem $P(2)$ und $P(n-1)$ verglichen werden, usw. Die Maschine arbeitet am längsten, wenn das Wort in der Tat symmetrisch ist. In diesem Fall werden $\frac{n}{2}$ Vergleichszyklen durchgeführt, in denen der Kopf zunächst n Zellen nach rechts, dann $n-1$ Zellen nach links, dann $n-2$ Zellen nach rechts, dann $n-3$ Zellen nach links geht usw. In Abb. 39 ist dieses Pendeln des Kopfes durch eine Schlangenlinie unter dem Band dargestellt. Insgesamt werden hierbei also $n + (n-1) + (n-2) + \cdots = \frac{n(1+n)}{2} \sim \frac{n^2}{2}$ Takte benötigt. Man kann diese Zeit etwa um die Hälfte verkürzen, wenn man den Algorithmus dadurch vervollkommnet, daß die Maschine in beiden Bewegungsrichtungen des Kopfes einen Vergleich durchführt, also sich z. B. schon nach dem erfolgten Vergleich von $P(1)$ mit $P(n)$ bei der Rückkehr nach links das Symbol $P(n-1)$ merkt und dann vorn mit $P(2)$ vergleicht. Bei diesem Verfahren erhält man eine Maschine $\mathfrak{N}_1$ mit

$$T_{\mathfrak{N}_1}(n) \sim \frac{n^2}{4}\,.$$

Es ist leicht einzusehen, daß noch weitere Verbesserungen möglich sind. Bei jeder Wanderung des Kopfes merke sich die Maschine $\mathfrak{N}_2$ ein Paar von Symbolen und ver-

gleiche es mit dem entsprechenden Paar am anderen Ende des Wortes. Zunächst wird also das Paar $P(1)P(2)$ mit dem Paar $P(n-1)P(n)$ verglichen, dann das Paar $P(3)P(4)$ mit dem Paar $P(n-3)P(n-2)$, usw. Das erfordert natürlich, im Vergleich zur Maschine $\mathfrak{M}_1$, eine größere Anzahl von inneren Zuständen, die Arbeitszeit wird jedoch wiederum, halbiert, d. h.

$$T_{\mathfrak{M}_2}(n) \sim \frac{n^2}{8}.$$

Allgemein gilt: Zu jeder Konstanten l kann man eine Turing-Maschine $\mathfrak{M}_l$ konstruieren, die auf jedem Weg je zwei l-Tupel von Symbolen vergleicht, und für deren Zeitsignalisierende gilt asymptotisch

$$T_{\mathfrak{M}_l}(n) \sim \frac{n^2}{4l}.$$

Obwohl wir auf diese Weise fortlaufend eine Beschleunigung des Rechenprozesses erreichen, ist für keine der bisher von uns betrachteten Maschinen zur Erkennung der Symmetrie die Zeitsignalisierende $T_{\mathfrak{M}}$ der Ordnung nach kleiner als n^2. Es gilt für sie, genauer gesagt, die folgende

Behauptung. Es existiert eine von $\mathfrak{M}$ abhängige, aber von n unabhängige Konstante $C_{\mathfrak{M}} > 0$, so daß

$$T_{\mathfrak{M}}(n) \geqq C_{\mathfrak{M}} \cdot n^2 \quad (n = 1, 2, 3, \ldots) \tag{1}$$

gilt.

Ist nun überhaupt eine Erkennung der Symmetrie in einer Zeit möglich, deren Ordnung kleiner als n^2 ist, d. h., existiert eine Turing-Maschine $\mathfrak{M}_0$, die die Symmetrie erkennt und für die

$$\lim_{n\to\infty} \left(T_{\mathfrak{M}_0}(n)/n^2\right) = 0$$

gilt (z. B. eine Maschine $\mathfrak{M}_0$, für die $T_{\mathfrak{M}_0}(n) \leqq n\sqrt{n}$ oder $T_{\mathfrak{M}_0}(n) \leqq \dfrac{n^2}{\log n}$ ist)?

Im Zusammenhang damit erinnern wir an die Tatsache, daß wir für den Übergang von der unären Darstellung zur Dezimaldarstellung eine Turing-Maschine $\mathfrak{M}_2'$ mit einer Signalisierenden

$$T_{\mathfrak{M}_2'}(n) \leqq 2n \cdot \log n$$

angeben konnten, die größenordnungsmäßig besser als die ursprüngliche Maschine $\mathfrak{M}_2$ mit $T_{\mathfrak{M}_2}(n) \leqq n^2$ war. Hier entsteht demgegenüber jedoch sofort der Eindruck, daß sich jede Erkennungsprozedur für die Symmetrie in irgendeiner Weise auf Vergleichszyklen des oben betrachteten Typs gründen muß, und es ist nicht zu sehen, auf wessen Kosten eine wesentliche Beschleunigung erreicht werden könnte. Diese Vermutung wird durch den folgenden Satz von Ja. M. Barzdiń bestätigt.

Satz. *Jede Turing-Maschine* $\mathfrak{M}$, *die die Symmetrie erkennt, erfüllt eine Bedingung* (1).

Wir beweisen diesen Satz in Abschnitt 3.6.1. Dazu müssen wir jedoch vorbereitend einige spezifische Besonderheiten der Turing-Berechnung etwas ausführlicher untersuchen, die auch selbständiges Interesse beanspruchen. Ihnen ist Abschnitt 3.5 gewidmet.

3.5. Die Spuren von Turing-Berechnungen

3.5.1.* Spuren

Wir betrachten den mit $\mathfrak{M}[P]$ bezeichneten Rechenprozeß, den die Turing-Maschine $\mathfrak{M}$ ausführt, wenn sie in ihrer Anfangskonfiguration das Wort

$$P = P(1)P(2)\cdots P(n)$$

erhalten hat. Die Grenzen zwischen zwei benachbarten Zellen numerieren wir so durch, wie das in Abb. 40a angegeben ist. Einer fixierten Grenze j kann man dann auf folgende Weise eine Folge von inneren Zuständen der Maschine zuordnen: Wenn der Kopf im Rechenprozeß $\mathfrak{M}[P]$ die Grenze j niemals überschreitet, so sei diese Folge leer (enthalte also keine Elemente). Der Kopf möge nun die Grenze j genau s mal überschreiten, und zwar das erste Mal im Zustand $q(1)$, das zweite Mal im Zustand $q(2)$, usw. Dann nennen wir die Folge $q(1)q(2)\cdots q(s)$, die ein Wort im Alphabet Q der inneren Zustände ist, die *Spur des Prozesses* $\mathfrak{M}[P]$ *im Punkt* j und schreiben dafür $\mathrm{Spur}_{\mathfrak{M}}(P, j)$. Diese Bezeichnung enthält einen Hinweis auf die betrachtete Maschine $\mathfrak{M}$ und das Ausgangswort P. Wenn aus dem Kontext ersichtlich ist, welche Maschine oder welches Wort gemeint ist, verwenden wir die kürzeren Bezeichnungen $\mathrm{Spur}(P, j)$ oder $\mathrm{Spur}(j)$. War die erste Bewegungsrichtung des Kopfes von links nach rechts, so wird die zweite von rechts nach links sein, und die weiteren Bewegungsrichtungen werden sich ebenfalls abwechseln. Offensichtlich kann bei $j > 0$ die erste Bewegungsrichtung nur die von links nach rechts und bei $j \leqq 0$ nur die von rechts nach links sein. Ist der Prozeß $\mathfrak{M}[P]$ unendlich, so kann es passieren, daß die Spur an einigen Grenzen ebenfalls unendlich wird. Da wir jedoch im weiteren nur endliche Prozesse betrachten werden, die mit der Ausgabe eines bestimmten Wortes R enden (das vom Anfangswort P abhängt), werden auch die Spuren endlich sein. Es bezeichne $|\mathrm{Spur}(j)|$ die Länge der Spur im Punkt j, d. h. die Anzahl der Übergänge des Kopfes über die Grenze j. Offensichtlich ist jeder Übergang des Kopfes über eine Grenze mit der Durchführung eines Arbeitstaktes der Maschine verbunden. Daher gilt die Ungleichung

$$t_{\mathfrak{M}}(P) \geqq \sum |\mathrm{Spur}(P, j)|, \tag{1}$$

wobei sich die Summation über eine beliebige Menge von Grenzen erstreckt. Die echte Ungleichheit kann einmal deshalb gelten, weil nur ein Teil der vom Kopf überschrittenen Grenzen betrachtet wurde, und zum anderen deshalb, weil sich in einigen Takten der Kopf nicht von der eingestellten Zelle wegbewegt und folglich auch keinerlei Grenzüberschreitungen verursacht hat. Wenn man nun aus der Analyse des Prozesses $\mathfrak{M}[P]$ entnehmen kann, daß es in einigen Punkten hinreichend lange Spuren gibt, so kann die Ungleichung (1) zur Aufstellung unterer Schranken für die Signalisierende $t_{\mathfrak{M}}(P)$ herangezogen werden. Eben mit einer solchen Methode beabsichtigen wir im folgenden untere Abschätzungen für die Kompliziertheit von Berechnungen in einigen für uns interessanten Fällen zu erhalten.

3.5.2.* Das Experiment mit dem aufgeschnittenen Band

Die Bedeutung des Spurbegriffs besteht darin, daß die Spur die Art und Weise widerspiegelt, in der die Maschine mittels ihrer inneren Zustände Information von einer Seite der Grenze auf die andere überträgt. Das wird durch das folgende Gedankenexperiment veranschaulicht. Wir nehmen an, daß wir die Spur für den Prozeß $\mathfrak{M}[P]$ im Punkt $j > 0$ kennen. Wir nehmen ferner an, daß das im Punkt j beginnende rechte Halbband entfernt wurde (zusammen mit dem sich auf ihm eventuell befindlichen Stück des Wortes P). Das Experiment besteht dann in folgendem: Wir setzen die Maschine auf das unveränderte linke Halbband so an, daß der Kopf das Symbol $P(1)$ im Startzustand liest. Dann arbeitet die Maschine auf dem Halbband zunächst genau wie vorher auf dem ganzen Band, und zwar so lange, bis der Kopf das erste Mal das Halbband im Zustand $q(1)$ verlassen will. Statt dessen versetzen wir den Kopf in der Randzelle des linken Halbbandes in den Zustand $q(2)$ (der uns bekannt ist, da wir die Spur kennen) und geben so der Maschine die Möglichkeit, so lange weiterzuarbeiten, bis sie das linke Halbband im Zustand $q(3)$ erneut verlassen will. Dann geben wir dem Kopf in der Randzelle den Zustand $q(4)$ und verhindern so ein Verlassen des Bandes, usw. Ist $s = |\mathrm{Spur}(j)|$ gerade, so verläßt der Kopf das linke Halbband nicht mehr, nachdem er in den Zustand $q(s)$ gebracht wurde, und bleibt schließlich irgendwo auf dem linken Halbband im Stopzustand stehen. Damit betrachten wir das Experiment als abgeschlossen. Ist s dagegen ungerade, so wird der Kopf das letzte Mal in den Zustand $q(s-1)$ versetzt und verläßt dann in einem gewissen Moment das linke Halbband im Zustand $q(s)$; damit wird das Experiment in diesem Fall beendet. Offenbar verhält sich die Maschine auf dem linken Halbband unter den Bedingungen des Experiments ganz genauso, wie sie sich dort im gewöhnlichen Prozeß $\mathfrak{M}[P]$ ohne entferntes rechtes Halbband verhält. Das bedeutet aber gerade, daß die im Experiment als bekannt angenommene Spur im Punkt j alle Informationen aus dem entfernten rechten Halbband enthält, die für die Arbeit auf dem linken Halbband benötigt werden. Genauso ist in ihr die gesamte Information über das linke Halbband enthalten, die für die Arbeit auf dem rechten Halbband benötigt wird.

3.5.3.* Ersetzungen

Wir nehmen nun an, daß im Prozeß $\mathfrak{M}[H]$, den die Maschine mit dem Anfangswort H ausführt, in einem gewissen Punkt i die gleiche Spur entsteht, wie beim Anfangswort P im Punkt j (vgl. Abb. 40a, b):

$$\mathrm{Spur}(P, j) = \mathrm{Spur}(H, i).$$

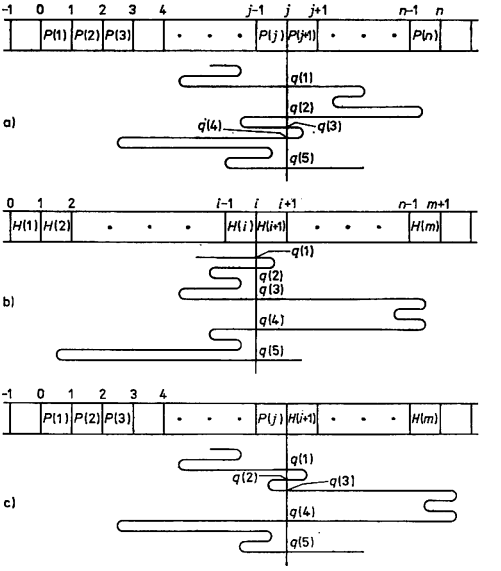


Abb. 40

Es bezeichne P_0^j den Anfangsabschnitt des Wortes P der Länge j und P_i^n seinen Endabschnitt der Länge $n - j$. Eine analoge Bedeutung haben die Bezeichnungen H_0^i und H_i^m für das Wort $H = H(1)H(2)\cdots H(m)$.

Wir betrachten nun die Anfangskonfiguration mit dem Wort $P_0^j H_i^m$ (vgl. Abb. 40c) und den entsprechenden Prozeß $\mathfrak{M}[P_0^j H_i^m]$. Eine Analyse des oben beschriebenen

Experiments zeigt, daß hier folgendes vor sich geht: Der Prozeß $\mathfrak{M}[P_0{}^jH_i{}^m]$ verläuft links vom Punkt j genauso wie der Prozeß $\mathfrak{M}[P]$ links vom Punkt j, und er verläuft rechts vom Punkt j wie der Prozeß $\mathfrak{M}[H]$ rechts vom Punkt i. Die Spur bleibt im Punkt j die gleiche wie vorher:

$$\mathrm{Spur}(P, j) = \mathrm{Spur}(H, i) = \mathrm{Spur}(P_0{}^jH_i{}^m, j).$$

Die Schlangenlinien in Abb. 40a, b, c illustrieren etwa den Weg des Kopfes in den Prozessen $\mathfrak{M}[P]$, $\mathfrak{M}[H]$ und $\mathfrak{M}[P_0{}^jH_i{}^m]$.

Wir nehmen nun an, daß die Maschine $\mathfrak{M}$ eine gewisse Eigenschaft von Wörtern entscheidet. Ist dann die Länge von $\mathrm{Spur}(P, j)$ gerade, so bleibt der Kopf am Ende des Prozesses $\mathfrak{M}[P_0{}^jH_i{}^m]$ links vom Punkt j stehen, und zwar auf einer Zelle mit dem Inhalt „1", falls P die von $\mathfrak{M}$ entschiedene Eigenschaft besitzt, bzw. mit dem Inhalt „0", falls P diese Eigenschaft nicht besitzt. Ist dagegen die Länge von $\mathrm{Spur}(P, j)$ ungerade, so bleibt die Maschine rechts vom Punkt j stehen, und das Resultat wird wie für das Wort H angegeben. Uns interessiert nun speziell der Fall, daß die Wörter P und H bezüglich der gegebenen Maschine $\mathfrak{M}$ äquivalent sind, d. h., daß sie entweder beide die durch die Maschine $\mathfrak{M}$ entschiedene Eigenschaft besitzen oder sie beide nicht besitzen. Dann kann man das Gesagte folgendermaßen zusammenfassen:

Ersetzungslemma. *Die Wörter $P = P_0{}^jP_j{}^n$ und $H = H_0{}^iH_i{}^m$ seien in bezug auf die Entscheidungsmaschine $\mathfrak{M}$ äquivalent, und ihre Spuren an den Grenzen zwischen $P_0{}^j$ und $P_j{}^n$ bzw. zwischen $H_0{}^i$ und $H_i{}^m$ seien gleich. Dann sind die vier Wörter $P_0{}^jP_j{}^n = P$, $H_0{}^iP_j{}^n$, $P_0{}^jH_i{}^m$ und $H_0{}^iH_i{}^n = H$ paarweise äquivalent, d. h., die Wörter $P_0{}^j$ und $P_j{}^n$ können ohne Einfluß auf das Ergebnis durch $H_0{}^i$ bzw. $H_i{}^m$ ersetzt werden.*

Wir geben noch eine direkte Folgerung aus dem Lemma an:

Behauptung A. *Es seien $P = P_0{}^jP_j{}^n$ und $H = H_0{}^jH_j{}^n$ zwei symmetrische Wörter der Länge n, und es sei $j \leqq \frac{n}{2}$. Falls dann irgendeine die Symmetrie erkennende Turing-Maschine $\mathfrak{M}$ im Punkt j* (d. h. an der Grenze zwischen $P_0{}^j$ und $P_j{}^n$ und an der Grenze zwischen $H_0{}^j$ und $H_j{}^n$) *die gleichen Spuren erzeugt, so gilt $P_0{}^j = H_0{}^j$.*

Denn nach dem Lemma muß dann das Wort $P_0{}^jH_j{}^n$ ebenfalls symmetrisch sein. Wenn wir jedoch in einem symmetrischen Wort $H = H_0{}^jH_j{}^n$ das Anfangsstück $H_0{}^j$ einer Länge kleiner oder gleich der Hälfte des ganzen Wortes H durch ein Stück derselben Länge, aber verschieden von $H_0{}^j$, ersetzen, so erhalten wir ein Wort, das sicher nicht symmetrisch ist.

3.6. Untere Kompliziertheitsabschätzungen

3.6.1.* Die Kompliziertheit der Symmetrieerkennung

Jetzt sind wir in der Lage, den Satz von BARZDIŃ zu beweisen (vgl. Abschnitt 3.4.4). Seine Richtigkeit folgt unmittelbar aus dem Lemma 1, das weiter unten bewiesen wird. Um unnötige Schwierigkeiten zu vermeiden, nehmen wir im weiteren an, daß die benutzten natürlichen Zahlen Vielfache von 4 sind, daß also $\frac{n}{2}$ und $\frac{n}{4}$ natürliche Zahlen sind (im allgemeinen Fall müßte man mit Rundungen arbeiten, alle Abschätzungen behalten jedoch Gültigkeit).

Lemma 1. *Es sei $\mathfrak{M}$ eine beliebige, die Symmetrie von Wörtern erkennende Turing-Maschine. Dann kann man eine positive Konstante $D_{\mathfrak{M}}$ (die nur von der Maschine $\mathfrak{M}$ abhängt) angeben, so daß folgendes gilt: Für alle hinreichend großen n (d. h. für alle n von einer gewissen Zahl n_0 ab) gibt es symmetrische Wörter P der Länge n, so daß*

$$|\mathrm{Spur}_{\mathfrak{M}}(P, j)| \geqq D_{\mathfrak{M}} \cdot n \quad \text{für} \quad j = \frac{n}{4} + 1, \frac{n}{4} + 2, \ldots, \frac{n}{2} \tag{1}$$

gilt.

Die von uns interessierende Abschätzung erhält man aus diesem Lemma trivial. Denn für jedes Wort P, das die Bedingung (1) erfüllt, gilt aufgrund der Ungleichung (1) aus Abschnitt 3.5.1 die folgende Ungleichung

$$t_{\mathfrak{M}}(P) \geqq \sum_{j=\frac{n}{4}+1}^{\frac{n}{2}} |\mathrm{Spur}_{\mathfrak{M}}(P, j)| \geqq \frac{n}{4} \cdot D_{\mathfrak{M}} \cdot n = \frac{D_{\mathfrak{M}}}{4} \cdot n^2.$$

Satz 1 gilt also, wenn als Konstante $C_{\mathfrak{M}}$ die Zahl $\frac{D_{\mathfrak{M}}}{4}$ genommen wird.

Wir beweisen, daß das Lemma 1 für $D_{\mathfrak{M}} = \frac{1}{8 \cdot \log_2 k}$ gilt, wobei k die Anzahl der inneren Zustände der Maschine $\mathfrak{M}$ ist. Für dieses $D_{\mathfrak{M}}$ sichern wir nicht nur, daß es unter den symmetrischen Wörtern der Länge n ein Wort P gibt, das die Bedingung (1) erfüllt, sondern daß es sogar „erdrückend viele" solcher Wörter gibt. Genauer gesagt: Ist $N(n)$ die Anzahl der symmetrischen Wörter P der Länge n, für die die Bedingung (1) falsch ist, und $S(n)$ die Anzahl aller symmetrischen Wörter der Länge n, so gilt

$$\lim_{n \to \infty} \frac{N(n)}{S(n)} = 0, \tag{2}$$

d. h., die Bedingung (1) ist nur für einen geringen Bruchteil der Menge aller Wörter

der Länge n nicht erfüllt. Um die Behauptung (2) zu zeigen, genügt es die, folgende Behauptung zu beweisen:

Lemma 2. *Es bezeichne* $N(j, n)$ *für* $j = \frac{n}{4} + 1, \frac{n}{4} + 2, \ldots, \frac{n}{2} - 1, \frac{n}{2}$ *die Anzahl der symmetrischen Wörter* P *der Länge* n, *für die die Ungleichung*

$$|\mathrm{Spur}_{\mathfrak{M}}(P, j)| < \frac{n}{8 \cdot \log_2 k}$$

gilt. Dann ist

$$N(j, n) \leqq 2^{\frac{3n}{8}}. \tag{3}$$

Denn $N(n)$ überschreitet offenbar nicht den Wert $\sum_{j=\frac{n}{4}+1}^{\frac{n}{2}} N(j, n)$, es ist also nach Lemma 2

$$N(n) \leqq \frac{n}{4} \cdot 2^{\frac{3n}{8}}.$$

Andererseits ist die Anzahl $S(n)$ aller symmetrischen Wörter der Länge n gleich $2^{\frac{n}{2}}$ (da wir n als gerade vorausgesetzt haben; allgemein gilt: $S(n) = 2^{\lceil \frac{n}{2} \rceil}$), nämlich gleich der Anzahl aller 0-1-Wörter der Länge $\frac{n}{2}$ (die linke Hälfte eines symmetrischen Wortes der Länge n kann jede der $2^{\frac{n}{2}}$ möglichen Formen haben, und die rechte Hälfte ist dann als Spiegelbild der linken Hälfte eindeutig bestimmt). Also gilt für $n \to \infty$:

$$\frac{N(n)}{S(n)} \leqq \frac{2^{\frac{3n}{8}} \cdot n}{4 \cdot 2^{\frac{n}{2}}} \to 0.$$

Wir beweisen nun die Ungleichung (3) aus Lemma 2: Dazu nehmen wir an, wir hätten eine Liste aller paarweise verschiedenen Spuren, die kürzer als $\frac{n}{8 \cdot \log_2 k}$ sind: $\sigma_1, \sigma_2, \ldots, \sigma_\alpha, \ldots, \sigma_\pi$; wir nehmen also an, daß es π solche Spuren für die Maschine $\mathfrak{M}$ gibt. Wir bezeichnen weiterhin mit $L(\alpha)$ die Anzahl aller symmetrischen Wörter der Länge n, die im Punkt j die Spur σ_α haben. Dann gilt:

$$N(j, n) = L(1) + L(2) + \cdots + L(\alpha) + \cdots + L(\pi). \tag{4}$$

Wir geben zunächst eine Abschätzung für π an. Da jede Spur ein Wort im k-buchstabigen Alphabet $\{q_1, q_2, \ldots, q_k\}$ ist, ist π nicht größer als die Anzahl der verschiedenen

Wörter einer Länge kleiner als $\frac{n}{8 \cdot \log_2 k}$ in einem k-buchstabigen Alphabet, d. h., es gilt

$$\begin{aligned} \pi &\leqq k + k^2 + \cdots + k^{\frac{n}{8 \cdot \log_2 k} - 1} \\ &= \frac{k^{\frac{n}{8 \cdot \log_2 k}} - 1}{k - 1} + 1 \leqq k^{\frac{n}{8 \cdot \log_2 k}} = 2^{\frac{n}{8}}. \end{aligned} \tag{5}$$

Nun schätzen wir den Summanden $L(\alpha)$ ($\alpha = 1, 2, \ldots, \pi$) ab. Er ist gleich der Anzahl aller der symmetrischen Wörter P der Länge n, die im Punkt j die Spur σ_α haben. Aufgrund der Behauptung A aus Abschnitt 3.5.3 (genau an dieser Stelle unseres Beweises benutzen wir die in Abschnitt 3.5 hergeleiteten Tatsachen über Spuren) haben alle diese Wörter P denselben Anfangsabschnitt der Länge j. Wir bezeichnen ihn mit R. Die Anzahl aller symmetrischen Wörter der Länge n mit dem Anfangsabschnitt R ist andererseits gleich $2^{\frac{n}{2}-j}$; denn da für die erste Hälfte des symmetrischen Wortes P der Anfangsabschnitt R der Länge j bereits festliegt, bleibt nur noch die Möglichkeit, alle $2^{\frac{n}{2}-j}$ Varianten für den Abschnitt $P(j+1)P(j+2)\cdots P\left(\frac{n}{2}\right)$ der Länge $\frac{n}{2} - j$ zu durchlaufen. Also gilt

$$L(\alpha) \leqq 2^{\frac{n}{2}-j} \quad (\alpha = 1, 2, \ldots, \pi). \tag{6}$$

Unter Berücksichtigung von (5) und (6) erhalten wir aus der Ungleichung (4)

$$N(j, n) \leqq \pi \cdot 2^{\frac{n}{2}-j} \leqq 2^{\frac{n}{8}} \cdot 2^{\frac{n}{2}-\frac{n}{4}} = 2^{\frac{3n}{8}}.$$

Damit ist die im Lemma 2 behauptete Ungleichung (3) und also der Satz aus Abschnitt 3.4.4 bewiesen. Das heißt, daß es in der Klasse der Turing-Maschinen keine (der Ordnung nach) bessere Erkennungsmöglichkeit für die Symmetrie von Wörtern als den sukzessiven Vergleich der von der Mitte der Wörter gleich entfernten Abschnitte einer festen Länge gibt.

3.6.2.* Die Kompliziertheit der Übersetzung von der unären in die dezimale Darstellung

Im vorangehenden Beweis wurde wesentlich von den spezifischen Eigenschaften der Arbeit von Turing-Maschinen Gebrauch gemacht, die ihren Ausdruck im Ersetzungslemma und dem vorher durchgeführten Gedankenexperiment mit dem aufgeschnittenen Band fanden. Diese Eigenschaften können auch zur Aufstellung unterer Schranken in anderen Fällen herangezogen werden. Wir wenden uns noch-

mals der Aufgabe II aus Abschnitt 2.3 zu, für die im Abschnitt 3.4.3 ein verbesserter Algorithmus $\mathfrak{M}_2'$ skizziert wurde, der zum gewünschten Resultat mit einer oberen Abschätzung der Rechenzeit von $2n \cdot \log_{10} n$ führte.

Übungsaufgabe. Man zeige, daß es zu jeder Turing-Maschine $\mathfrak{M}$, die für eine beliebige Zahl n aus ihrer unären Darstellung ihre Dezimaldarstellung herstellt, eine Konstante $C_{\mathfrak{M}} > 0$ gibt, so daß

$$T_{\mathfrak{M}}(n) \geqq C_{\mathfrak{M}} \cdot n \cdot \log_{10} n$$

gilt. Das Turing-Programm $\mathfrak{M}_2'$ ist also hinsichtlich der Ordnung (d. h. bis auf einen konstanten Faktor) das beste.

Hinweis. Die Anfangskonfiguration sei entsprechend Abb. 41 gewählt. Man zeige, daß eine mit dieser Anfangskonfiguration arbeitende Maschine $\mathfrak{M}$ in den Punkten $1, 2, \ldots, n$ paarweise verschiedene Spuren erzeugt. Man zeige weiter, daß die Summe der Länge von n paarweise verschiedenen Spuren bereits die für $T_{\mathfrak{M}}$ aufzustellende Ungleichung erfüllt.

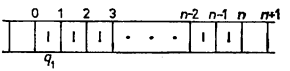


Abb. 41

3.7. Die Existenz beliebig komplizierter Probleme

Für die algorithmischen Probleme, die wir bisher genauer betrachteten, gelang uns die Angabe von Turing-Programmen, die die Lösung in einer Anzahl von Schritten lieferten, die im Vergleich zur Länge der Anfangsinformation nicht übermäßig groß war, die Zeitsignalisierenden wurden durch eine quadratische Funktion majorisiert:

$$T_{\mathfrak{M}}(n) \leqq C_{\mathfrak{M}} \cdot n^2.$$

In einigen Fällen konnten wir dabei sogar zeigen, daß die von uns vorgelegten Programme von allen überhaupt möglichen am besten sind, d. h., daß eine wesentliche Beschleunigung der Rechenzeit (der Ordnung nach) auf Turing-Maschinen nicht erreichbar ist. Nun dürfte jedoch intuitiv klar sein, daß es Probleme geben wird, die bezüglich der Rechenzeit auf Turing-Maschinen, ja sogar allgemein auf beliebigen Maschinen, eine so schnelle Lösung nicht zulassen. Wir könnten sogar Probleme angeben, für die alle bisher bekannten Algorithmen zu ihrer Lösung sehr komplizierte Berechnungen erfordern und für die es voraussichtlich auch keine hinsichtlich ihrer Rechenzeit wesentlich besseren Algorithmen gibt. In diesem Paragraphen wollen wir folgende prinzipielle Frage erörtern: Gibt es unter allen algorithmisch entscheidbaren Problemen beliebig komplizierte?

3.7.1. Präzisierung der Fragestellung

Wir versuchen zunächst, diese etwas verschwommene Fragestellung zu präzisieren. Im Zusammenhang damit erklären wir auch, in welchen Termini wir eine Antwort darauf erhalten möchten.

Wir nehmen an, das auf einer Turing-Maschine $\mathfrak{M}$ zu lösende Problem bestehe in der Berechnung der Werte einer gewissen arithmetischen Funktion $\varphi(n)$ für $n = 0, 1, 2, 3, \ldots$. Wir wissen schon (vgl. Abschnitt 2.5.3), daß eine solche Funktion sehr schnell wachsen kann. Man denke etwa an die Funktion φ, die für das Argument n den Wert $2^{2^{\cdot^{\cdot^{2}}}}$ (mit n Etagen) annimmt, obwohl natürlich auch ein so beeindruckendes Wachsen noch übertroffen werden kann. Wenn die Argumente und Werte der Funktion in unärer Darstellung aufgeschrieben werden, so ist für eine beliebige, φ berechnende Maschine $\mathfrak{M}$ der Wert $T_{\mathfrak{M}}(n)$ sicher nicht kleiner als $\varphi(n)$. Denn die Maschine braucht ja allein schon so viele Takte, um die $\varphi(n)$ Striche auszudrucken, die den Funktionswert markieren. In diesem Sinne kommen wir bereits bei der Berechnung schnell wachsender Funktionen zu beliebig komplizierten Problemen. Diese Sachlage bleibt auch dann bestehen, wenn man die Zahlen in einem beliebigen Positionssystem, etwa im Dezimalsystem, aufschreibt. Wenn nämlich $\varphi(n)$ sehr schnell wächst, so wächst auch $]\log_{10}\varphi(n)[$ (die Anzahl der Dezimalziffern in der Darstellung von $\varphi(n)$) sehr schnell. Beispiele dieser Art können uns natürlich nicht ganz befriedigen. Obwohl in den geschilderten Situationen von der Maschine viel Zeit zum Ausdrucken des Resultats benötigt wird, braucht der Prozeß der Berechnung selbst ja ganz und gar nicht kompliziert zu sein. Beispiele, in denen das Resultat nicht so umfangreich, seine Wortlänge also kurz ist, würden uns mehr befriedigen, denn dort wäre fast die gesamte Zeit zur eigentlichen Berechnung des Resultats nötig. Wir beschränken uns daher im folgenden auf die Betrachtung von Problemen, bei denen es sich um die Erkennung (Entscheidung) einer bestimmten Eigenschaft der Anfangsinformation handelt. Dann besteht das Resultat nämlich in der Ausgabe der einbuchstabigen Wörter „1" (ja) oder „0" (nein). In der Terminologie der Logik handelt es sich hierbei um die Berechnung von *Prädikaten*. Unter diesen Problemen gibt es, wie wir schon wissen, auch algorithmisch unentscheidbare (z. B. das Problem der Selbstanwendbarkeit von Turing-Maschinen, das Äquivalenzproblem für die Wörter eines assoziativen Kalküls und andere). Ist ein Erkennungsproblem jedoch algorithmisch entscheidbar, so wird die für die Entscheidung einer individuellen Aufgabe des Problems bei der Anwendung des konkreten Algorithmus (z. B. des Turing-Programms) notwendige Rechenzeit fast ausschließlich zum Auffinden der Antwort (und nicht für ihre Niederschrift!) verwendet.

Folgende Frage ist nun im Zusammenhang mit den durchgeführten Überlegungen von Interesse: Existiert eine berechenbare Funktion $f(P)$, die für alle Wörter P im Alphabet $\{0, 1\}$ definiert ist und natürliche Zahlen als Werte hat, für die folgendes gilt: Ist $\mathfrak{M}$ eine beliebige Turing-Maschine, die im Dualalphabet eine gewisse Eigen-

schaft Γ der Wörter P erkennt, so gibt es stets eine Turing-Maschine $\mathfrak{N}$, die ebenfalls die Eigenschaft Γ erkennt und für die $t_{\mathfrak{N}}(P) \leqq f(P)$ für alle Wörter P gilt.

Die Existenz einer solchen berechenbaren Funktion, wäre sie auch noch so schnell wachsend, könnten wir als Begründung dafür ansehen, daß algorithmisch entscheidbare Probleme nicht beliebig kompliziert sein können; denn dann könnte man immer mit Algorithmen auskommen, die mit einer vorher begrenzten Anzahl von Schritten bei der Abarbeitung eines Wortes der Länge n das Resultat finden. Außerdem würden wir über eine berechenbare Funktion f verfügen, die uns eine annehmbare obere Schranke liefert.

3.7.2. Die Erkennung der f-Selbstanwendbarkeit

Es zeigt sich nun, daß die oben gestellte Frage negativ zu beantworten ist, daß es also eine solche berechenbare Funktion nicht gibt. Es gilt, genauer gesagt, der folgende Satz.

Satz von Cejtin. *Es sei $f(P)$ eine beliebige berechenbare Funktion. Dann gibt es eine Eigenschaft Γ für Dualwörter, für die folgendes gilt:*

a) *Γ ist effektiv erkennbar (d. h., es gibt eine Turing-Maschine, die für jedes Wort P entscheidet, ob es die Eigenschaft Γ hat oder nicht),*

b) *für jede die Eigenschaft Γ erkennende Maschine $\mathfrak{N}$ ist für unendlich viele Wörter P die Ungleichung $t_{\mathfrak{N}}(P) > f(P)$ erfüllt.*

Den Beweis für dieses Theorem erhält man durch eine geeignete (und lehrreiche!) Modifizierung des Theorems aus Abschnitt 3.2.2 über die algorithmische Unlösbarkeit des Problems der Selbstanwendbarkeit. Es sei dazu $f(P)$ eine beliebige berechenbare Funktion. Wird auf das Band einer beliebigen Maschine $\mathfrak{M}$ ihre eigene Ciffre $\mathrm{Chif}_1(\mathfrak{M})$ geschrieben, so sind folgende drei Fälle möglich: 1. In den ersten $f\big(\mathrm{Chif}_1(\mathfrak{M})\big)$ Takten bleibt $\mathfrak{M}$ nicht stehen; 2. die Maschine stoppt spätestens nach $f\big(\mathrm{Chif}_1(\mathfrak{M})\big)$ Takten und gibt als Resultat das Symbol „1" aus; 3. die Maschine stoppt spätestens nach $f\big(\mathrm{Chif}_1(\mathfrak{M})\big)$ Takten und gibt ein von „1" verschiedenes Resultat aus. Wenn der Fall 3 eintritt, nennen wir die Maschine $\mathfrak{M}$ *f-selbstanwendbar*, während sie in den Fällen 1 und 2 *nicht-f-selbstanwendbar* heißt. Damit entsteht auf natürliche Weise das Erkennungsproblem für die f-Selbstanwendbarkeit: Für ein beliebiges Dualwort P ist zu entscheiden, ob es Chiffre einer f-selbstanwendbaren Maschine ist (kürzer: ob P Selbstanwendbarkeitswort ist) oder nicht. Trotz seiner äußeren Ähnlichkeit mit dem Selbstanwendbarkeitsproblem ist das Entscheidungsproblem für die f-Selbstanwendbarkeit algorithmisch lösbar. Wir geben einen Entscheidungs-algorithmus an:

1. Für ein gegebenes Wort P wird zunächst geprüft, ob es Chiffre einer Turing-Maschine ist oder nicht. Das ist nach den Ausführungen aus Abschnitt 1.4.2 effektiv möglich. Ist P keine Chiffre, so ist P erst recht kein Selbstanwendbarkeitswort, und die (negative) Antwort steht fest.

2. Ist jedoch P Chiffre einer Maschine $\mathfrak{M}$, so berechnen wir zunächst die Zahl $f(P)$. Das kann effektiv erfolgen, denn nach Annahme ist die Funktion $f(P)$ berechenbar; daher gibt es einen Algorithmus (eine Turing-Maschine), der es gestattet, zu jedem Wort P den Wert $f(P)$ zu finden.

3. Sodann wird aus der Chiffre P das Funktionsschema $\mathfrak{M}$ aufgestellt.

4. Es wird die Arbeit der Maschine $\mathfrak{M}$, ausgehend von Wort P als Anfangskonfiguration höchstens $f(P)$ Takte lang verfolgt. Damit können wir effektiv feststellen, welcher der obigen drei Fälle eintritt, und erfahren, ob P ein Selbstanwendbarkeitswort ist oder nicht.

Für diesen Algorithmus ist es wesentlich, daß wir über eine berechenbare obere Grenze (nämlich $f(P)$) für die Anzahl der zu verfolgenden Arbeitstakte der Maschine verfügen. Beim allgemeinen Problem der Selbstanwendbarkeit war, im Unterschied dazu, eine unbegrenzte Anzahl von Arbeitstakten zu verfolgen.

Es sei nun $\mathfrak{K}$ eine beliebige Turing-Maschine, die die f-Selbstanwendbarkeit erkennt (wir haben uns gerade davon überzeugt, daß solche Maschinen existieren); dann gilt für eine beliebige Chiffre $\mathrm{Chif}_1(\mathfrak{M}')$: (i) $\mathfrak{K}$ führt $\mathrm{Chif}_1(\mathfrak{M}')$ in „1" über, wenn $\mathfrak{M}'$ f-selbstanwendbar ist; (ii) $\mathfrak{K}$ führt $\mathrm{Chif}_1(\mathfrak{M}')$ in „0" über, wenn $\mathfrak{M}'$ nicht f-selbstanwendbar ist. In Analogie zum Beweis aus Abschnitt 3.2 überprüfen wir, wie die Maschine $\mathfrak{K}$ ihre eigene Chiffre verarbeitet:

1. Ist $\mathfrak{K}$ f-selbstanwendbar, so liefert $\mathfrak{K}$ nach Definition der f-Selbstanwendbarkeit (Variante 3) ein von „1" verschiedenes Resultat. Nach (ii) heißt das aber, daß $\mathfrak{K}$ das Symbol „0" ausgibt, folglich kann $\mathfrak{K}$ nicht f-selbstanwendbar sein. Das ist aber ein Widerspruch, d. h., dieser Fall kann nicht eintreten.

2. Es ist also $\mathfrak{K}$ nicht f-selbstanwendbar, dann ist nach Definition die Variante 1 oder die Variante 2 möglich. Die Variante 2 führt, genau wie die Variante 3, zu einem Widerspruch. Also kann nur die Variante 1 vorliegen, daß die Maschine $\mathfrak{K}$ zwar ihre Nicht-f-Selbstanwendbarkeit (denn $\mathfrak{K}$ kann ja nach Voraussetzung dieses Problem für jede Maschine entscheiden!) erkennt, aber dabei $t_{\mathfrak{K}}(\mathrm{Chif}_1(\mathfrak{K})) > f(\mathrm{Chif}_1(\mathfrak{K}))$ gilt.

Zum Beweis des Theorems müssen wir nur noch zeigen, daß außer für $\mathrm{Chif}_1(\mathfrak{K})$ die Bedingung $t_{\mathfrak{K}}(P) > f(P)$ noch für unendlich viele weitere Wörter P gilt. Es sei dazu $\mathfrak{M}$ eine beliebige Turing-Maschine, die man aus der Maschine $\mathfrak{K}$ durch fiktive Erweiterung ihres äußeren Alphabets erhält. Wie wir schon in Abschnitt 2.4.1 feststellten, arbeiten solche Maschinen $\mathfrak{M}$ taktweise genauso wie die Maschine $\mathfrak{K}$ (wenn man sie auf das gleiche Anfangswort P in Standard-Darstellung ansetzt). Speziell gilt dabei $t_{\mathfrak{M}}(P) = t_{\mathfrak{K}}(P)$. Da es aber unendlich viele derartige fiktive Erweiterungen $\mathfrak{K}_1, \mathfrak{K}_2, \ldots$ gibt, die natürlich ebenfalls die f-Selbstanwendbarkeit entscheiden, und für jede solche Maschine $\mathfrak{K}_i$ und ihre Chiffre $\mathrm{Chif}_1(\mathfrak{K}_i)$ die Beziehung

$$t_{\mathfrak{K}_i}(\mathrm{Chif}_1(\mathfrak{K}_i)) > f(\mathrm{Chif}_1(\mathfrak{K}_i))$$

gilt, haben wir wegen

$$t_{\mathfrak{K}}(\mathrm{Chif}_1(\mathfrak{K}_i)) = t_{\mathfrak{K}_i}(\mathrm{Chif}_1(\mathfrak{K}_i)) \quad \text{für } i = 1, 2, 3, \ldots$$

die Beziehung

$$t_{\mathfrak{B}}(\mathrm{Chif}_1(\mathfrak{R}_i)) > f(\mathrm{Chif}_1(\mathfrak{R}_i)).$$

Damit ist der Satz bewiesen.

Durch eine Verfeinerung der Konstruktion läßt sich der Satz weiter verschärfen:

Satz von Rabin. *Zu jeder berechenbaren Funktion $f(P)$ läßt sich eine entscheidbare Eigenschaft Γ für Dualwörter angeben, so daß die Zeitsignalisierende $t_{\mathfrak{B}}$ für jede Γ erkennende Maschine $\mathfrak{R}$ die Bedingung $t_{\mathfrak{B}}(P) > f(P)$ für alle bis auf höchstens endlich viele Wörter P erfüllt.*

3.8. v.-Neumann-Automaten

Die Besonderheit der Arbeit einer Turing-Maschine besteht darin, daß in jedem Takt nur in einer einzigen Zelle eine Informationsumwandlung erfolgt. Alle übrigen Zellen warten in dieser Zeit auf den Maschinenkopf. Natürlich gibt es Situationen, in denen diese Wartezeit durch die Natur der zu lösenden Aufgabe gerechtfertigt ist. Das ist z. B. dann der Fall, wenn die Informationsumwandlung in einer gegebenen Zelle α vom Resultat der Informationsumwandlung in einer davon verschiedenen Zelle β abhängt. Man kann sich aber auch leicht die andere Situation vorstellen, in der eine solche Abhängigkeit nicht gegeben ist und die einzige Ursache dafür, daß die Informationsumwandlung in den Zellen α und β nicht gleichzeitig (parallel) erfolgt, in der Unfähigkeit der Turing-Maschine liegt, so etwas zu tun.

In diesem Zusammenhang liegt die Idee nahe, Maschinen mit paralleler Arbeitsweise zu betrachten, Maschinen also, in denen eine Informationsumwandlung gleichzeitig in mehreren Zellen erfolgen kann.

3.8.1. Systeme von Turing-Maschinen

Die Betrachtung eines Systems von Turing-Maschinen mit gemeinsamem äußeren Speicher (Band) ist wohl die einfachste Form, die sich hier anbietet. Wir beginnen mit einem Beispiel, das auch für spätere Betrachtungen nützlich ist: Es sei $\mathfrak{M}_0$ eine Turing-Maschine mit dem äußeren Alphabet $\{\Lambda, l, r, |\}$ und den Zuständen π, π', ϱ, ϱ', ϱ''. Im Programm der Maschine gibt es die unten angeführten beiden Gruppen von Befehlen (neben eventuell anderen, die für uns jetzt nicht interessant sind):

Gruppe I. $|\pi \to R, r\pi \to L\pi', |\pi' \to L$;

Gruppe II. $l\varrho \to R, |\varrho \to \varrho', |\varrho' \to \varrho'', |\varrho'' \to R\varrho$.

Diese Befehle ändern nicht die Bandinschrift, sondern sie steuern nur die Bewegung des Kopfes. Wenn mit der Konfiguration aus Abb. 42a begonnen wird, so bewegt sich der Kopf nach den Befehlen der Gruppe I nach rechts und bleibt im Zustand π.

Wenn er zum Symbol r kommt, wird er „reflektiert" und geht nach Übergang in den Zustand π' nach links. Wenn mit der Konfiguration aus Abb. 42b begonnen wird, bewegt sich der Kopf nach den Befehlen der Gruppe II langsam nach rechts, wobei er jeden Strich drei Takte lang „genau betrachtet".

Wir nehmen nun an, daß gleichzeitig die Köpfe von zwei Exemplaren der Maschine $\mathfrak{M}_0$ auf ein Band gesetzt werden, das, wie auch oben, die Niederschrift des Wortes $l\,|\,|\cdots|\,|r$ enthält, wobei von der in Abb. 42c dargestellten Konfiguration ausgegangen werde. Nach den Befehlen der Gruppen I und II beginnt jeder der Köpfe, sich nach rechts zu bewegen. Man zeigt ohne Mühe, daß folgendes gilt: Ist die Wortlänge n des Wortes $l\,|\,|\cdots|\,|r$ eine gerade Zahl, so gelangt der „schnelle" Kopf (der mit dem

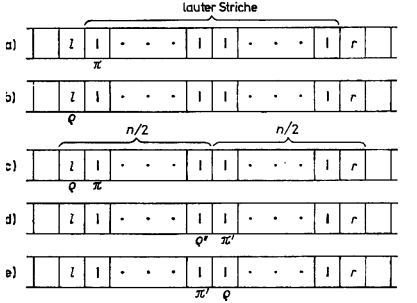


Abb. 42

Anfangszustand π versehen ist) nach $3\left(\frac{n}{2}-1\right)$ Takten zunächst zum Buchstaben r und von dort, nachdem er die Richtung gewechselt hat und in den Zustand π' übergegangen ist, im Zustand π' zur Zelle, die den $\left(\frac{n}{2}+1\right)$-ten Buchstaben des betrachteten Wortes $l\,|\,|\cdots|\,|r$ enthält. In derselben Zeit gelangt der „langsame" Kopf (der mit dem Anfangszustand ϱ versehen ist) im Zustand ϱ'' zur Zelle, die den $\left(\frac{n}{2}\right)$-ten Buchstaben des betrachteten Wortes enthält. Am Ende des $3\left(\frac{n}{2}-1\right)$-ten Taktes wird also die Konfiguration aus Abb. 42d erreicht, bei der die Köpfe gerade in der Mitte des Wortes nebeneinanderstehen. Im nächsten Takt „überspringen" sich die Köpfe, und es entsteht die Konfiguration aus Abb. 42e. Anschließend läuft der langsame Kopf weiter nach rechts und der schnelle weiter nach links. Wir bemerken sofort, daß man nicht immer mit einer so glücklichen Koexistenz von zwei oder mehreren Turing-Maschinen rechnen kann. Es kann durchaus passieren, daß beide

Köpfe „versuchen", in die gleiche Zelle zu gelangen (das geschieht in unserem Beispiel, wenn man ein Wort $l l \cdots l r$ ungerader Länge betrachtet). Dann sind bei den ursprünglichen Turing-Maschinen nicht vorgesehene zusätzliche Befehle notwendig die eine eindeutige Arbeit des Systems garantieren. So kann ein zusätzlicher Befehl beispielsweise Prioritäten festlegen: Wird eine Zelle mit dem Inhalt x gleichzeitig von zwei Köpfen gelesen, die sich in den Zuständen q_1 bzw. q_2 befinden, so ist in die Zelle das Symbol zu schreiben, das durch den Befehl mit der linken Seite xq_1 angegeben wird. Es ist allerdings meistens bequemer, beim Zusammenwirken mehrerer Maschinen auf einem gemeinsamen äußeren Speicher von vornherein ein Zusammentreffen von zwei oder mehr Köpfen auf einer Zelle durch ein anderes Prinzip zu verhindern, das in folgendem besteht: Wenn sich zwei Köpfe soweit einander genähert haben, daß die Gefahr besteht, daß sie in einer Zelle zusammentreffen, so tritt ein Befehl in Kraft, der diese Gefahr beseitigt. Auf diese Weise gelangt man für beliebiges $\nu = 2, 3, \ldots$ zum Begriff des *Systems von ν Turing-Maschinen mit gemeinsamem äußeren Speicher*. Ein solches System läßt sich natürlich auch als Maschine neuen Typs auffassen (*Mehrkopfmaschine*). Dabei nimmt man meistens an, daß man ν Exemplare von ein und derselben Turing-Maschine $\mathfrak{M}$ vorliegen hat. Wir können jedoch sagen, ohne dabei endgültigen Definitionen vorzugreifen, daß ein System von ν Turing-Maschinen für kein ν die Parallelisierung des Berechnungsprozesses voll verwirklichen kann, denn auch hier befinden sich in jedem Takt die meisten Zellen (nämlich alle, bis auf ν Zellen) nach wie vor in einer erzwungenen Passivität.

3.8.2. v.-Neumann-Automaten

Wir beschreiben nun eine Maschine ganz anderen Typs, den sogenannten *v.-Neumannschen Zellularautomaten*, bei dem das Parallelisierungsprinzip voll durchgesetzt ist. Obwohl auch im Zellularautomaten die Anzahl der gleichzeitig bearbeiteten Zellen zu jedem Zeitpunkt endlich ist, kann sie im Laufe der Zeit unbeschränkt wachsen. In dieser Hinsicht übertrifft der Zellularautomat jedes beliebige System aus einer festen endlichen Anzahl von Turing-Maschinen. Ganz grob kann man sich unter einem Zellularautomaten ein System vorstellen, das in der Lage ist, immer wieder neue Exemplare einer Turing-Maschine heranzuziehen und die Bearbeitung eines für alle gemeinsamen äußeren Speichers durch sie zu organisieren.

Wir kommen nun zu den exakten Definitionen: Ein *v.-Neumann-Element* oder kurz *Element* ist ein Baustein, der sich in jedem Takt $t = 1, 2, 3, \ldots$ in einem von endlich vielen Zuständen $r_1, r_2, \ldots, r_k$ befindet.[1]) Diese Zustände bilden das *Alphabet R des Elements*. Ein solches Element hat zwei Eingangskanäle, nämlich einen linken und einen rechten, über die in jedem Takt t ebenfalls je ein Zustand aus R ankommt (vgl. Abb. 43). Der Zustand eines Elements im Takt $t + 1$ hängt aus-

[1]) In der Literatur werden die Elemente auch als *Zellen* bezeichnet, woraus sich die Bezeichnung *Zellularautomat* ableitet. [*Anm. d. Übers.*]

schließlich davon ab, in welchem Zustand es sich im vorhergehenden Takt t befunden hat und welche Zustände im Takt t über die Eingangskanäle angekommen sind. Jedes Element realisiert also eine Funktion $\Psi(p, r, q)$ von drei Veränderlichen. Argumente und Werte dieser Funktion sind die Elemente des Alphabets R. Die Gleichung

$$\Psi(r_i, r_j, r_m) = r_n$$

bedeutet dabei folgendes: Befindet sich das Element im Zustand r_j und erhält es über den linken Kanal den Zustand r_i und über den rechten Kanal den Zustand r_m, so geht es im folgenden Takt in den Zustand r_n über. Diese Gleichung schreiben wir auch als *v.-Neumann-Befehl* in der Form $r_i r_j r_m \to r_n$.

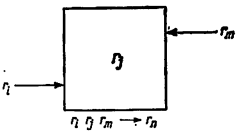


Abb. 43

Die Arbeitsweise eines Elements wird also vollständig durch die Funktion Ψ beschrieben, d. h. durch ein bestimmtes System von k^3 v.-Neumann-Befehlen, das man auch als sein *v.-Neumann-Programm* bezeichnet. Einen Zustand r nennen wir einen *passiven Zustand* oder *Ruhezustand*, wenn für ihn die Bedingung $\Psi(r, r, r) = r$ (d. h. $rrr \to r$) gilt. Ist $\Psi(r, r, r) \neq r$, so nennen wir r einen *aktiven* (oder *aktivierenden*) Zustand. Befindet sich ein Element in einem Ruhezustand, so sagen wir auch, es sei *in Ruhe* oder *passiv*, während ein Element, das sich in einem aktiven Zustand befindet, als *aktiv* oder *erregt* bezeichnet wird. Wir nehmen in Zukunft an, daß die Zustandsmenge R einen speziellen Ruhezustand $\emptyset$ enthält. Es gelte also stets

$$\emptyset\emptyset\emptyset \to \emptyset. \tag{1}$$

Ein Element, das sich im Ruhezustand $\emptyset$ befindet, kann nur dadurch erregt werden, daß es über wenigstens einen Kanal einen den Zustand $\emptyset$ erregenden Zustand erhält.

Nachdem ein gewisses Element $\mathfrak{H}$ fixiert wurde, läßt sich der v.-Neumannsche Zellularautomat zu dem gegebenen Element $\mathfrak{H}$ als zweiseitig unendliches Band beschreiben, das aus lauter Exemplaren dieses Elements gebildet ist, die in der in Abb. 44a dargestellten Weise miteinander verbunden sind.

Legen wir zum Zeitpunkt t die Zustände der Elemente in einer bestimmten Weise fest, so erhalten wir eine bestimmte *v.-Neumann-Konfiguration* $K(t)$. Mit $\alpha(t)$ bezeichnen wir für ein beliebiges Element α des Bandes seinen Zustand zum Zeitpunkt t in der Konfiguration $K(t)$. Mit $\alpha^-(t)$ bzw. $\alpha^+(t)$ bezeichnen wir die Zustände seines

linken bzw. rechten Nachbarn. Wir bemerken, daß das Element α die Zustände $\alpha^-(t)$ bzw. $\alpha^+(t)$ über den linken bzw. rechten Kanal erhält. Damit kann gemäß dem Programm der Zustand $\alpha(t+1)$ im folgenden Moment bestimmt werden. Da das für alle Elemente des Bandes zutrifft, entsteht nach einem Takt im Ergebnis einer gleichzeitigen Umwandlung der Zustände aller Elemente sofort die nächste Konfiguration $K(t+1)$. Ganz genauso gewinnt man danach die Konfigurationen $K(t+2)$, $K(t+3)$, ... Darin besteht gerade die *Arbeitsweise* eines v.-Neumannschen Zellularautomaten. Das Wesentliche ist also die Tatsache, daß überall und gleichzeitig die Informationen, die in der v.-Neumann-Konfiguration kodiert sind, umgewandelt werden.

Wir betrachten im weiteren nur *endliche* v.-Neumann-Konfigurationen, d. h. Konfigurationen, bei denen sich fast alle Elemente, d. h. alle bis auf endlich viele (deren Anzahl jedoch nicht fixiert ist), im Ruhezustand Ø befinden. Den kleinsten

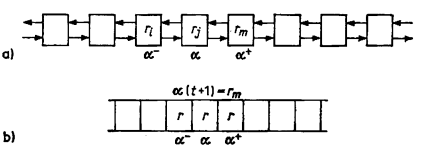


Abb. 44

Abschnitt des v.-Neumann-Bandes, außerhalb dessen sich alle Elemente im Ruhezustand Ø befinden, nennen wir die *aktive Zone* des Automaten (in der gegebenen Konfiguration). Aus der Bedingung (1) folgt offensichtlich, daß in einem zu Beginn mit einer endlichen Konfiguration versehenen v.-Neumann-Automaten auch im weiteren Verlauf der Arbeit stets nur endliche Konfigurationen auftreten können, denn die aktive Zone vergrößert sich in jedem Takt um höchstens zwei Elemente, nämlich um höchstens eins an jedem Ende. Daher ist der von uns formal als unendliches Objekt definierte v.-Neumann-Automat zu jedem Zeitpunkt im wesentlichen ein endliches Objekt, das jedoch in der Lage ist, beliebig zu wachsen. Wir haben hier also eine volle Analogie zum Turing-Band, das ja formal ebenfalls unendlich, jedoch zu jedem Zeitpunkt im wesentlichen endlich ist. Das Turing-Band wächst immer dann, wenn der Vorrat an Speicherplätzen nicht mehr ausreicht, um den Rechenprozeß fortzusetzen. Der Unterschied besteht vor allem darin, daß die Zellen des Turing-Bandes nur die Funktion der Informationsspeicherung haben, die Elemente (oder Zellen) eines v.-Neumann-Automaten dagegen die Informationsspeicherung mit einer Informationsverarbeitung verbinden. Unter Berücksichtigung des Zusammenwirkens der Elemente des v.-Neumann-Bandes, das in Abb. 44a schematisch dargestellt ist, werden wir im weiteren der Einfachheit halber auch den v.-Neumann-Automaten in Form eines gewöhnlichen Bandes (wie bei den Turing-Maschinen) darstellen (vgl. Abb. 44b).

3.8.3. Ein Beispiel: Die v. Neumannsche Uhr

Das Element $\mathfrak{H}$ habe vier Zustände $\emptyset$, 0, ε, 1. Bei der Niederschrift seines Programms in Abb. 45 sind folgende vereinfachenden Vereinbarungen getroffen, von denen wir von nun an stets Gebrauch machen werden: Erstens werden alle passiven Befehle, d. h. solche, die keine Veränderung von $\alpha(t)$ hervorrufen (d. h. mit $\alpha(t+1) = \alpha(t)$), weggelassen. Zweitens verwenden wir eine zusammenfassende Niederschrift für gleichartige Befehle; die erste Zeile der Abb. 45 steht z. B. für 16 einzelne Befehle,

	$\alpha^-(t)$	$\alpha(t)$	$\alpha^+(t)$	$\alpha(t+1)$
1	beliebig	1	beliebig	0
2	$\mathfrak{H}$	0	beliebig	1
3	1	0	beliebig	ε
4	1	ε	beliebig	1

Abb. 45

die man erhält, wenn man für $\alpha^-(t)$ und $\alpha^+(t)$ unabhängig voneinander die Werte $\emptyset$, 0, ε und 1 einsetzt. Entsprechend diesem Programm erhält man, ausgehend von der Konfiguration $K(t)$ aus Abb. 46, die weiteren Konfigurationen $K(t+1)$, $K(t+2)$, ... In der Abbildung haben wir jeweils nur die aktiven Zonen dargestellt. Man sieht übrigens leicht, daß durch unser Programm die zu Beginn gegebene aktive Zone nicht vergrößert wird. Es ist nützlich zu bemerken, daß sich beim Übergang

$\mathfrak{H}$	1	0	1	ε	1	ε	1	0	$\mathfrak{H}$	$K(t)$
$\mathfrak{H}$	0	ε	0	1	0	1	0	ε	$\mathfrak{H}$	$K(t+1)$
$\mathfrak{H}$	1	ε	0	0	ε	0	ε	ε	$\mathfrak{H}$	$K(t+2)$

Abb. 46

von $K(t)$ zu $K(t+1)$ gleichzeitig in allen acht Elementen der aktiven Zone die Zustände ändern, beim Übergang zu $K(t+2)$ jedoch nur in fünf Elementen.

Wir nehmen nun an, die Anfangskonfiguration habe eine aktive Zone der Länge ν, in die nur Nullen geschrieben sind. Wir verfolgen die Arbeit unseres Automaten. Für $\nu = 3$ sind in Abb. 47 die entstehenden Konfigurationen angegeben. Daneben stehen die Nummern der jeweils beim Übergang zur folgenden Konfiguration tatsächlich benutzten Befehle. Wie aus Abb. 47 zu sehen ist, mündet der Automat, beginnend mit $t = 11$, in einen periodischen Prozeß; es ist nämlich $K(11) = K(3)$. In der äußersten rechten Zelle der aktiven Zone wird folglich, beginnend mit $t = 10$, alle $8 = 2^3$ Takte eine Eins auftreten, und in keinem anderen Takt steht dort eine

Eins. Analog tritt auch bei jedem anderen ν alle 2^ν Takte die Eins am rechten Ende der aktiven Zone auf. In diesem Sinne gestattet es das von uns betrachtete Element, eine „Uhr" mit der Periode 2^ν auf einer aktiven Zone der Länge ν zu konstruieren.

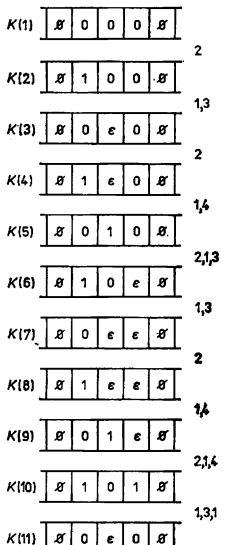


Abb. 47

3.8.4. Eine v.-Neumann-Modellierung der Turing-Maschinen

Es sei eine Turing-Maschine $\mathfrak{M}$ mit dem äußeren Alphabet $\{s_1, s_2, \ldots, s_k\}$ und den Zuständen $\{q_1, q_2, \ldots, q_m\}$ und eine beliebige Konfiguration K dieser Maschine gegeben. Jede Zelle des Bandes kann man in der Konfiguration K durch eine Spalte der Form $\begin{matrix} x \\ q \end{matrix}$ charakterisieren, wobei x das in der Zelle stehende Symbol ist und q durch folgende Bedingung definiert sei: Wird in K die Zelle α durch den Kopf gelesen, so sei q der entsprechende Zustand der Maschine, wird α jedoch nicht gelesen, so sei q das Symbol Λ. Wir nehmen dabei an, daß das Symbol Λ von $q_1, q_2, \ldots, q_m$

verschieden ist. Daß dieses Symbol mit dem Leersymbol des äußeren Alphabets der Maschine $\mathfrak{M}$ übereinstimmt, stört uns nicht. Es gibt offensichtlich genau $(m+1)\cdot k$ verschiedene solche Spalten. Man kann sie als „zweistöckige" Buchstaben eines gewissen neuen „zweistöckigen" Alphabets R auffassen. Unter diesen „zweistöckigen" Buchstaben kommt natürlich auch der Buchstabe $\begin{matrix}\Lambda\\ \Lambda\end{matrix}$ vor, der die leeren und zugleich „unbeobachteten" Zellen in der Konfiguration K charakterisiert. Das bedeutet, daß alle bis auf endlich viele Zellen des Bandes eben durch diesen „zweistöckigen" Buchstaben $\begin{matrix}\Lambda\\ \Lambda\end{matrix}$ charakterisiert werden. Wenn wir jetzt die Buchstaben aus R als Zustände eines gewissen v.-Neumann-Elements interpretieren, so wird deutlich, daß jede Turing-Konfiguration K in natürlicher Weise durch eine gewisse v.-Neumann-Konfiguration $\bar{K}$ in R kodiert wird. In Abb. 48a und 48b sind beispielsweise die die Turing-Konfigurationen aus Abb. 42a und 42b kodierenden v.-Neumann-Konfigurationen dargestellt. Außerdem gilt: Man kann zu jedem gegebenen Turing-Programm $\mathfrak{M}$ ein v.-Neumann-Programm $\bar{\mathfrak{M}}$ konstruieren, so daß der v.-Neumann-Automat mit dem Programm $\bar{\mathfrak{M}}$ immer dann die K kodierende Konfiguration $\bar{K}$ in einem Takt in die K' kodierende Konfiguration $\bar{K}'$ überführt, wenn $\mathfrak{M}$ in einem Takt K in K' überführt. In diesem Sinne *modelliert* der v.-Neumann-Automat $\bar{\mathfrak{M}}$ die (Arbeitsweise der) Turing-Maschine $\mathfrak{M}$, wobei offenbar der Buchstabe $\begin{matrix}\Lambda\\ \Lambda\end{matrix}$ die Rolle des Ruhezustands spielt.

Wir erklären die Übergangsprozedur von $\mathfrak{M}$ zu $\bar{\mathfrak{M}}$ am Beispiel des Turing-Programms $\mathfrak{M}_0$ aus Abschnitt 3.8.1. In diesem Programm kommt z. B. der Befehl $r\pi \to Ln'$ vor. Dafür steht für jedes Tripel der Form $\begin{matrix}x & y & r\\ \Lambda & \Lambda & \pi\end{matrix}$ mit beliebigen äußeren Buchstaben x, y der Maschine $\mathfrak{M}_0$ im Programm $\bar{\mathfrak{M}}_0$ der Befehl

$$\begin{matrix}x & y & r\\ \Lambda & \Lambda & \pi\end{matrix} \to \begin{matrix}y\\ \pi'\end{matrix}.$$

Dieser Befehl beschreibt genau die Tatsache, daß der Kopf von der Zelle α^+ zur Zelle α gebracht wird und der Zustand π' erscheint. Hinzu kommen für den betrachteten Turing-Befehl in $\mathfrak{M}_0$ weiterhin alle v.-Neumann-Befehle der Form

$$\begin{matrix}x & r & y\\ \Lambda & \pi & \Lambda\end{matrix} \to \begin{matrix}r\\ \Lambda\end{matrix} \quad \text{und} \quad \begin{matrix}r & x & y\\ \pi & \Lambda & \Lambda\end{matrix} \to \begin{matrix}x\\ \Lambda\end{matrix}.$$

Indem wir für jeden Befehl des Programms $\mathfrak{M}_0$ analog verfahren, erhalten wir offenbar ein Programm $\bar{\mathfrak{M}}_0$ eines v.-Neumann-Automaten, der $\mathfrak{M}_0$ modelliert. Die oberen 12 Zeilen der Abb. 49 enthalten gerade alle aktiven Befehle aus $\bar{\mathfrak{M}}_0$; die passiven Befehle wurden, wie vereinbart, weggelassen. Es ist leicht einzusehen, daß die am konkreten Beispiel illustrierte Prozedur stets zum Ziel führt, d. h., sie führt ein beliebiges gegebenes Turing-Programm $\mathfrak{M}$ in ein „modellierendes" v.-Neumann-Programm $\bar{\mathfrak{M}}$ über.

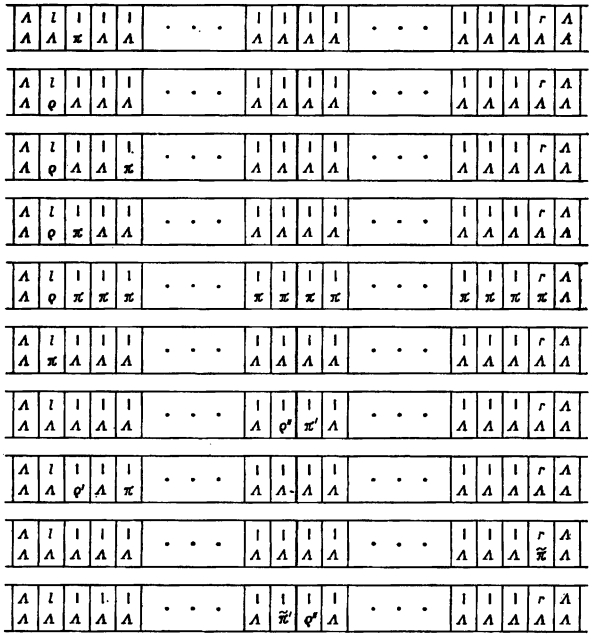


Abb. 48

Wir wenden uns nochmals dem v.-Neumann-Programm $\mathfrak{M}_0$ zu. Wir bemerken als erstes, daß es die Turing-Prozesse in der Maschine $\mathfrak{M}_0$ für die Anfangskonfigurationen aus Abb. 42a und 42b modelliert, wenn als Anfangskonfigurationen die v.-Neumann-Konfigurationen aus Abb. 48a bzw. 48b genommen werden. Das bedeutet, daß bei Verwendung der Anfangskonfiguration aus Abb. 48a in der unteren Etage des v.-Neumann-Bandes ein „schneller" Transport des Symbols π nach rechts erfolgt, während analog bei der Anfangskonfiguration aus Abb. 48b ein „langsamer" Transport des Symbols ϱ nach rechts erfolgt. Ein auf die angegebene Weise konstruiertes Programm $\mathfrak{M}$ (und speziell auch unser Programm $\mathfrak{M}_0$) bestimmt jedoch das Verhalten des Zellularautomaten auch eindeutig für solche v.-Neumann-Konfigurationen N, die keine Kodierungen von Turing-Konfigurationen sind, also z. B. auch

	$\alpha^-(t)$	$\alpha(t)$	$\alpha^+(t)$	$\alpha(t+1)$	Erläuterung
1	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}x\\ \pi\end{matrix}$	$\mathrm{I}\,\pi \to R$
2	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \Lambda\end{matrix}$	
3	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}r\\ \pi\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}r\\ \Lambda\end{matrix}$	$r\,\pi \to L\,\pi'$
4	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}r\\ \pi\end{matrix}$	$\begin{matrix}y\\ \pi'\end{matrix}$	
5	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi'\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \Lambda\end{matrix}$	$\mathrm{I}\,\pi' \to L$
6	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi'\end{matrix}$	$\begin{matrix}y\\ \pi'\end{matrix}$	
7	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}l\\ \varrho\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}l\\ \Lambda\end{matrix}$	$l\,\varrho \to R$
8	$\begin{matrix}l\\ \varrho\end{matrix}$	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}x\\ \varrho\end{matrix}$	
9	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho'\end{matrix}$	$\mathrm{I}\,\varrho \to \varrho'$
10	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho'\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho''\end{matrix}$	$\mathrm{I}\,\varrho' \to \varrho''$
11	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho''\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \Lambda\end{matrix}$	$\mathrm{I}\,\varrho'' \to R\varrho$
12	$\begin{matrix}\mathrm{I}\\ \varrho''\end{matrix}$	$\begin{matrix}x\\ \Lambda\end{matrix}$	$\begin{matrix}y\\ \Lambda\end{matrix}$	$\begin{matrix}x\\ \varrho\end{matrix}$	
13	beliebig	$\begin{matrix}l\\ \pi\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \Lambda\end{matrix}$	$\begin{matrix}l\\ \varrho\end{matrix}$	Verdopplung
14	$\begin{matrix}l\\ \pi\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \Lambda\end{matrix}$	beliebig	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	
15	$\begin{matrix}l\\ \varrho\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	beliebig	$\begin{matrix}\mathrm{I}\\ \varrho\end{matrix}$	
16	beliebig	$\begin{matrix}l\\ \varrho\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	$\begin{matrix}l\\ \Lambda\end{matrix}$	Ergänzungen zu $\mathrm{I}\,\pi \to R$, $l\,\varrho \to R$, $\mathrm{I}\,\varrho \to \varrho'$
17	$\begin{matrix}l\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \varrho'\end{matrix}$	
18	$\begin{matrix}\mathrm{I}\\ \varrho\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi\end{matrix}$	beliebig	$\begin{matrix}\mathrm{I}\\ \Lambda\end{matrix}$	

1–12 : Programm $\widetilde{\mathfrak{M}}_0$ 1–18 : Programm $\mathfrak{M}_l$

Abb. 49

in dem Fall, wenn es in N mehr als ein Element in einem Zustand der Form $\begin{matrix}x\\ q\end{matrix}$ mit $q \neq \Lambda$ gibt. Diese Zustände, die bei Turing-Interpretation bedeuten, daß die Zelle durch einen Kopf betrachtet wird, wollen wir kurz *zu betrachtende Zustände* nennen. Wenn wir beim Automaten $\mathfrak{M}_0$ beispielsweise mit der Konfiguration aus Abb. 48c mit zwei zu betrachtenden Zuständen beginnen, so findet gleichzeitig ein schneller Transport des Symbols π und ein langsamer Transport des Symbols ϱ statt. Es

wird also ein System von zwei Exemplaren der Maschine $\mathfrak{M}_0$ mit gemeinsamem Band modelliert. Eine analoge Situation liegt immer dann vor, wenn die zu betrachtenden Zustände nicht zu nahe beieinander liegen. Sonst kann ein völlig anderes Bild entstehen. Als Anfangskonfiguration werde beispielsweise die Konfiguration aus Abb. 48d mit zwei benachbarten zu betrachtenden Zuständen genommen. Da es unter den aktiven Befehlen des Programms $\mathfrak{M}_0$ (vgl. Abb. 49) keine gibt, deren linker Teil das Tripel $\begin{matrix} l & | & | \\ \varrho & \pi & \Lambda \end{matrix}$ oder das Tripel $\begin{matrix} \Lambda & l & | \\ \Lambda & \varrho & \pi \end{matrix}$ ist, finden in den ersten beiden Zellen (der aktiven Zone!) keinerlei Änderungen statt. Dafür erscheint im folgenden Takt in der dritten Zelle der Zustand $\begin{matrix} | \\ \pi \end{matrix}$, im nächsten Takt erscheint der Zustand $\begin{matrix} | \\ \pi \end{matrix}$ in der vierten Zelle usw. Schließlich erscheint im $(n-1)$-ten Takt die Konfiguration aus Abb. 48e, die sich dann nicht mehr ändert.

Wir betrachten nun das Programm $\mathfrak{N}_l$ mit 18 aktiven Befehlen, das man aus $\mathfrak{M}_0$ durch Hinzunahme der Befehle 13 bis 18 erhält (vgl. Abb. 49). Wir überzeugen uns davon, daß dafür die folgende Behauptung gilt, die wir im Abschnitt 3.9 benötigen werden.

Behauptung A. *Für beliebiges geradzahliges n führt das Programm $\mathfrak{N}_l$ in $3\left(\frac{n}{2}-1\right)+1$ Takten die Konfiguration aus Abb. 48f in die Konfiguration aus Abb. 48g über.*

In der Tat: Die Befehle 13 und 14 führen die Konfiguration aus Abb. 48f in einem Takt in die Konfiguration aus Abb. 48d über. Man kann also sagen, daß in diesem Takt eine Verdopplung des Kopfes stattfindet: Aus einer Turing-Maschine, die die erste Zelle betrachtet, werden zwei Turing-Maschinen, die die ersten beiden Zellen betrachten. Durch die Befehle 15 bis 18 wird in den beiden nächsten Takten die Konfiguration aus Abb. 48h erhalten. Sie gestatten den beiden entstandenen Turing-Maschinen den Übergang in eine „normale" Arbeitsweise und beseitigen die „Hemmung", die beim ursprünglichen Versuch auftrat, das Programm $\mathfrak{M}_0$ auf die Konfiguration aus Abb. 48d anzuwenden. Weiter wird dann genau die gemeinsame Arbeit von zwei Exemplaren der Turing-Maschine $\mathfrak{M}_0$ modelliert, die wir in Abb. 42c, d studiert haben, d. h. der gleichzeitige Transport des „schnellen Kopfes" und des „langsamen Kopfes", bei dem nach $3\left(\frac{n}{2}-1\right)$ Takten die Konfiguration aus Abb. 48g erscheint.

Übungsaufgabe. Man konstruiere ein v.-Neumann-Programm $\mathfrak{N}_r$ aus 18 Schemata von aktiven Befehlen, das in folgendem Sinne zum Programm $\mathfrak{N}_l$ dual ist: a) In $\mathfrak{N}_r$ gibt es zu jedem zu betrachtenden Zustand $\begin{matrix} x \\ q \end{matrix}$ aus $\mathfrak{N}_l$ einen „dualen" Zustand $\begin{matrix} \boldsymbol{x} \\ \bar{q} \end{matrix}$; b) angesetzt auf die Konfiguration aus Abb. 48i führt das Programm $\mathfrak{N}_r$ diese in $3\left(\frac{n}{2}-1\right)+1$ Takten in die Konfiguration aus Abb. 48j über.

Hinweis. Die Buchstaben l und r vertauschen ihre Rollen. Die Suche nach der Mitte des Wortes $l\,|\cdots|\,r$ beginnt nicht am linken, sondern am rechten Ende.

Die Bedeutung der betrachteten v.-Neumann-Programme besteht darin, daß es mit den verschiedenen Transportgeschwindigkeiten der unteren Symbole π, φ im Fall von $\mathfrak{N}_l$ oder der unteren Symbole $\bar{\pi}$, $\bar{\varphi}$ im Fall von $\mathfrak{N}_r$ gelingt, die Mitte einer aktiven Zone der Länge n in einer Zeit zu finden, die n nicht mehr als 1,5mal überschreitet. Dieser Faktor wird im Abschnitt 3.9 eine wesentliche Rolle spielen.

3.9. Eine Aufgabe über die Synchronisierung einer Schützenkette

Im vorliegenden Abschnitt behandeln wir eine Aufgabe, die zuerst von dem bekannten amerikanischen Automatentheoretiker E. F. Moore veröffentlicht wurde, und erörtern einige Lösungsmöglichkeiten.[1]) Sie wird uns die Schwierigkeiten besser verstehen lassen, die bei der Organisation eines Informationsverarbeitungsprozesses auf v.-Neumann-Automaten (und allgemein bei parallelisiertem Berechnungsprozeß) auftreten. Die Lösung der Aufgabe wird außerdem in Abschnitt 3.10 benutzt.

3.9.1. Schützenketten

Wir nehmen an, daß für in einer Reihe nebeneinander stehende Schützen die folgende gegenseitige Kommunikationsmöglichkeit gegeben ist.

1. Jeder Schütze kann sich nur mit seinem unmittelbar linken und seinem unmittelbar rechten Nachbarn verständigen. Der linke (rechte) Flügelmann kann sich dabei natürlich nur an seinen rechten (linken) Nachbarn wenden.

2. In jeder Sekunde (oder einer anderen vorher fixierten Zeiteinheit) kann je eine Mitteilung übermittelt werden. Es wird angenommen, daß alle Schützen über ein streng synchronisiertes Sekundenmaß verfügen.

3. Für jeden Schützen besteht eine Mitteilung darin, daß er seinen beiden Nachbarn je ein vereinbartes Zeichen aus einem gegebenen Zeichenvorrat übergibt und von diesen je ein vereinbartes Zeichen empfängt.

Zu einem gewissen Zeitpunkt erhält nun der linke oder rechte Flügelmann einen Umschlag mit dem Befehl „Feuer!", der an alle Schützen gerichtet ist und von ihnen gleichzeitig ausgeführt werden soll. Dann entstehen folgende Fragen:

a) Welche Instruktion muß jeder Schütze vorher erhalten, wobei diese Instruktion für alle Schützen die gleiche sein soll, damit bei der zugelassenen Kommunikationsprozedur in der Schützenkette dieser Befehl von allen gleichzeitig ausgeführt werden kann?

---

[1]) Vgl. Moore, E. F., The firing squad synchronization problem, Sequential Machines, Addison Wesley Publ. Co., Reading, Mass. — Palo Alto — London 1964, S. 213—214. Moore nennt J. Myhill als Autor dieser Aufgabe.

b) Wie groß ist die Minimalzeit (die kleinste Anzahl von Sekunden), die von der Erteilung bis zur Ausführung des Befehls verstreicht?

Hierbei ist es wichtig zu unterstreichen, daß die Anzahl der Schützen (die Länge der Kette) keinem der Schützen bekannt ist und daß diese Anzahl (wir bezeichnen sie mit n) im allgemeinen beliebig sein kann. Außerdem ist in dem Moment, in dem der Umschlag dem linken oder rechten Flügelmann überreicht wird, niemandem als dem Empfänger etwas davon bekannt.

Wir betrachten die folgende mögliche Instruktion an die Schützen, die auf den ersten Blick sehr natürlich erscheint. Wir beschreiben jenen Teil der Instruktion, der sich auf den Fall bezieht, daß der Umschlag dem linken Flügelmann überreicht wird. Er besteht aus mehreren Anweisungen, von denen die beiden ersten die Ausführung des üblichen Befehls: „Von links nach rechts durchzählen!" sichern.

Anweisung 1. Wenn du der linke Flügelmann bist und den Befehl „Schützenkette Feuer!" erhältst, merke dir die Nummer Eins und teile sie dem rechten Nachbarn mit.

Anweisung 2. Wenn du nicht der rechte Flügelmann bist und dir dein linker Nachbar die Nummer ν mitteilt, so merke dir die Nummer $\nu + 1$ und teile sie in der nächsten Sekunde deinem rechten Nachbarn mit.

Nach Abarbeiten dieses Teils der Instruktion wissen alle Schützen in der n-ten Sekunde nach Erteilung des Befehls ihre Nummer in der Reihe. Die weiteren Anweisungen regulieren den umgekehrten Informationsfluß vom rechten Flügelmann zum linken.

Anweisung 3. Wenn du der rechte Flügelmann bist und dir dein linker Nachbar die Nummer $n - 1$ mitteilt, so antworte ihm in der nächsten Sekunde mit „fertig!" und zähle von da an in Gedanken rückwärts: $n - 1$, $n - 2$, ..., und zwar in jeder Sekunde eine Zahl.

Anweisung 4. Hast du dir die Nummer ν gemerkt und gibt dir dein rechter Nachbar die Mitteilung „fertig!", so zähle von der nächsten Sekunde an rückwärts: $\nu - 1$, ...; ist $\nu > 1$ (d. h., stehst du nicht ganz links), so gib außerdem in der nächsten Sekunde die Mitteilung „fertig!" an deinen linken Nachbarn weiter.

Anweisung 5. Hast du bis Null gezählt, so gib Feuer.

Analoge Anweisungen werden für den Fall gegeben, daß der rechte Flügelmann den Befehl erhält. Damit ist die Beschreibung der Instruktion beendet.

Wenn sich die Schützen nach dieser Instruktion richten, so schießen offenbar alle Schützen gleichzeitig genau $2n$ Sekunden nach dem Zeitpunkt, zu dem der linke- oder rechte Flügelmann den Befehl „Feuer!" erhielt. Es ist noch nützlich zu bemerken, daß bei keiner Instruktion, die sich auf die zugelassene Kommunikationsprozedur beschränkt, die bis zur Mitteilung des Befehls an alle Schützen notwendige Zeit kürzer als n sein kann, denn keine Information kann vom linken Flügelmann aus den rechten vor dieser Zeit erreichen.

3.9.2. Schützen und Automaten

Wir präzisieren jetzt die Frage a), indem wir fordern, daß die Instruktion für die Schützen als v.-Neumann-Programm formalisiert ist. Mit anderen Worten, uns interessiert, ob ein gewisses fixiertes v.-Neumann-Element $\mathfrak{H}$ die Rolle eines Schützen aus unserer Aufgabe übernehmen kann. Vor der Befehlsübergabe hat man sich eine Schützenkette der Länge n als eine v.-Neumann-Konfiguration mit einer aktiven Zone der Länge n vorzustellen, in der sich alle Elemente außer den Randelementen im gleichen Zustand befinden. Diese zusätzliche Forderung drückt die Tatsache aus, daß kein Schütze, bis auf die beiden Flügelmänner, vor einem anderen irgendwie ausgezeichnet ist. Im Moment des Feuerns müssen sich überhaupt alle Elemente der aktiven Zone der Länge n im gleichen Zustand befinden, und zwar in einem Zustand (dem „Feuer"-Zustand), der in keiner der vorangegangenen Konfigurationen schon einmal aufgetreten sein darf (es ist kein ungeordnetes Schießen vorher erlaubt!).

Eine nähere Betrachtung der in Abschnitt 3.9.1 behandelten Instruktion zeigt, daß sie nicht als v.-Neumann-Programm formalisierbar ist. Das liegt daran, daß n in der Aufgabe eine beliebige natürliche Zahl sein kann. Da sich aber jeder Schütze seine Nummer merken muß, wird damit vorausgesetzt, daß er sich über einige Zeit hinweg eine beliebige natürliche Zahl merken kann. Mit anderen Worten, es ist also notwendig, daß die Anzahl der verschiedenen Zustände, in denen sich der Schütze befinden kann, unbeschränkt ist. Ein v.-Neumann-Element kann jedoch nur endlich viele Zustände annehmen. Nebenbei bemerkt, es kann sich natürlich auch kein realer Schütze in einer Sekunde (oder in einer beliebigen anderen fixierten Zeiteinheit) zu große Zahlen merken. Daher ist die Forderung, daß sich (für beliebige n) eine Schützenkette wie ein v.-Neumann-Automat verhalten soll, hinreichend natürlich. Wir kommen nun zur genauen Formulierung der Aufgabe für v.-Neumann-Automaten. Es ist für uns bequem, wenn wir annehmen, daß die Zustände der uns interessierenden v.-Neumann-Elemente „zweistöckige" Buchstaben sind, ähnlich wie wir das bei der Modellierung einer Turing-Maschine im Abschnitt 3.8 beschrieben haben. Wir nehmen also an, daß die Zustände die Form $\begin{matrix} x \\ q \end{matrix}$ haben. Dabei möge das obere Symbol x unter anderem die Werte Λ und | und das untere Symbol q unter anderem die Werte q_0, Λ, $*$ annehmen können. Wir verwenden im folgenden die Bezeichnungen und Resultate aus dem vorangehenden Paragraphen. Die Aufgabe erscheint nun in folgender Form: Es ist ein v.-Neumann-Programm zu konstruieren, das für jedes n die Anfangskonfigurationen aus Abb. 50a, b in die Konfiguration aus Abb. 50c überführt, wobei zusätzlich gefordert ist, daß bis zum Erreichen der Konfiguration aus Abb. 50c das Symbol „$*$" niemals in der unteren Etage auftritt. Wir merken an, daß sich die Abb. 50a und 50b nur dadurch unterscheiden, daß die erste und letzte Zelle der aktiven Zone ihre Rollen vertauscht haben. Inhaltlich bedeutet das Symbol $\begin{matrix} | \\ q_0 \end{matrix}$ am linken Ende der aktiven Zone den Befehlsempfang durch

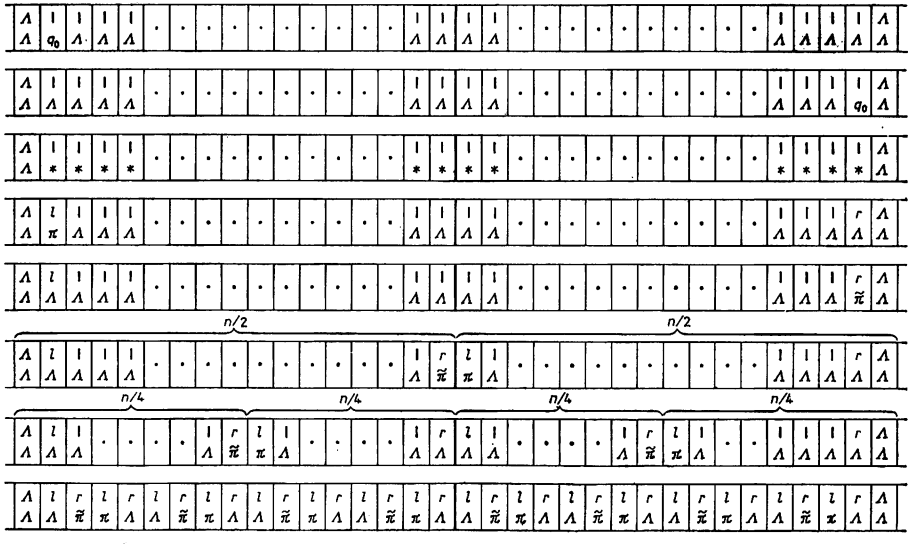


Abb. 50

den linken Flügelmann und das Symbol $\overset{|}{q_0}$ am rechten Ende den Befehlsempfang durch den rechten Flügelmann; das Symbol $\overset{|}{A}$ charakterisiert einen abwartenden Schützen und das Symbol $\overset{|}{*}$ einen feuernden Schützen.

3.9.3. Eine Lösung der Aufgabe[1])

Um die Lösung von unwesentlichen Einzelheiten zu befreien, nehmen wir an, daß es unter den äußeren Symbolen (d. h. den Symbolen der oberen Etage) spezielle Symbole l, r (inhaltliche Bedeutung: für den linken bzw. rechten Flügelmann) gibt und daß statt der Anfangskonfigurationen aus Abb. 50a, b die Konfigurationen aus Abb. 50d, e betrachtet werden. Eine solche Annahme läßt sich dadurch rechtfertigen, daß der Übergang von Abb. 50a zu Abb. 50d durch einen Abschnitt im v.-Neumann-Programm realisiert werden kann, der eine gewöhnliche Turing-Maschine modelliert, die folgendermaßen arbeitet: Der Kopf bewegt sich im Zustand q_0 nach rechts, bis er den äußersten rechten Strich gefunden hat, ersetzt diesen durch das Symbol r, kehrt dann nach links zurück, bis er den äußersten linken Strich gefunden hat, ersetzt diesen durch das Symbol l und geht in den Zustand π über. Für das weitere ist es wichtig zu bemerken, daß für die beschriebene Überführung der Konfiguration aus Abb. 50a in die Konfiguration aus Abb. 50d genau n Takte benötigt werden. Dieselbe Bemerkung trifft auf die Umarbeitung der Konfiguration aus Abb. 50b in die Konfiguration aus Abb. 50e zu.

Eine weitere, von uns zunächst vorgenommene Vereinfachung besteht darin, daß nur solche Anfangskonfigurationen aus Abb. 50d, e betrachtet werden, bei denen die Länge der aktiven Zone eine Potenz von 2 ist (d. h. die Anzahl der Schützen in der Reihe eine Zweierpotenz ist). Später wird erklärt, wie man sich von dieser Einschränkung lösen kann.

In Abb. 51 ist ein Programm $\mathfrak{P}$ aus 44 Schemata von Befehlen abgebildet, dessen erste 18 Befehle die aktiven Befehle des Programms $\mathfrak{P}_l$ und dessen nächste 18 Befehle die aktiven Befehle des Programms $\mathfrak{P}_r$ sind. Wir erinnern uns daran, daß die Programme $\mathfrak{P}_l$ und $\mathfrak{P}_r$ am Ende von Abschnitt 3.8 betrachtet wurden. Unser Ziel ist es, uns von der Gültigkeit der folgenden Behauptung zu überzeugen.

Behauptung B. *Für jede Zweierpotenz n führt das Programm $\mathfrak{P}$ die Konfigurationen aus Abb. 50d, e in die Konfiguration aus Abb. 50c über, wobei nicht mehr als $3n$ Takte benötigt werden.*

Berücksichtigen wir noch die obigen Bemerkungen über die Zeit, die für den Übergang von Abb. 50a zu Abb. 50d und von Abb. 50b zu Abb. 50e benötigt wird, so

[1]) Zunächst möge der Leser selbst über eine Lösung nachdenken. In dem in der Fußnote auf S. 181 erwähnten Artikel bittet Moore dringend diejenigen, die eine Lösung kennen, sie nicht denjenigen zu sagen, die selbst eine Lösung suchen, um ihnen nicht die Freude an diesem spannenden Problem zu nehmen.

sehen wir, daß die hier vorgelegte Lösung der Aufgabe mit einer Gesamtzeit verbunden ist, die $5n$ nicht überschreitet.

Zum Beweis der Behauptung B verfolgen wir etwa den Prozeß, bei dem die Konfiguration aus Abb. 50d nach dem Programm $\mathfrak{P}$ verarbeitet wird. In dieser Abbildung wurde $n = 2^5 = 32$ gewählt. Zuerst werden die Befehle aus $\mathfrak{P}_t$ angewendet; da die Konfiguration aus Abb. 50d mit der Konfiguration aus Abb. 48f übereinstimmt, erscheint nach $3\left(\frac{n}{2}-1\right)+1$ Takten die Konfiguration aus Abb. 48g (vgl. Behauptung A am Ende von Abschnitt 3.8). Im folgenden Takt wird mit Hilfe der Befehle 37, 38 des Programms $\mathfrak{P}$ in einem Takt die Konfiguration aus Abb. 50f erzeugt. So erfolgt also nach $3\left(\frac{n}{2}-1\right)+2 \leqq \frac{3n}{2}$ Takten eine Zerlegung der aktiven Zone der Länge n in je zwei Teilzonen der Länge $\frac{n}{2}$; die rechte Teilzone hat die gleiche Form wie die Anfangskonfiguration in Abb. 50d, sie ist nur halb

	$\alpha^-(t)$	$\alpha(t)$	$\alpha^+(t)$	$\alpha(t+1)$	Erläuterungen
1 · · · 18					$\mathfrak{N}_t$ (vgl. Abb. 49)
19 · · · 36					$\mathfrak{N}_r$ (dual zu $\mathfrak{N}_t$)
37	beliebig	$\begin{matrix}\mathrm{I}\\ \varrho''\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi'\end{matrix}$	$\begin{matrix}r\\ \tilde{\pi}\end{matrix}$	Abschluß der linken Teilung
38	$\begin{matrix}\mathrm{I}\\ \varrho''\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \pi'\end{matrix}$	beliebig	$\begin{matrix}l\\ \pi\end{matrix}$	
39	beliebig	$\begin{matrix}\mathrm{I}\\ \tilde{\pi}'\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \tilde{\varrho}''\end{matrix}$	$\begin{matrix}r\\ \tilde{\pi}\end{matrix}$	Abschluß der rechten Teilung
40	$\begin{matrix}\mathrm{I}\\ \tilde{\pi}'\end{matrix}$	$\begin{matrix}\mathrm{I}\\ \tilde{\varrho}''\end{matrix}$	beliebig	$\begin{matrix}l\\ \pi\end{matrix}$	
41	beliebig	$\begin{matrix}l\\ \pi\end{matrix}$	$\begin{matrix}r\\ \Lambda\end{matrix}$	$\begin{matrix}\mathrm{I}\\ *\end{matrix}$	"Feuer!"
42	beliebig	$\begin{matrix}l\\ \Lambda\end{matrix}$	$\begin{matrix}r\\ \tilde{\pi}\end{matrix}$	$\begin{matrix}\mathrm{I}\\ *\end{matrix}$	
43	$\begin{matrix}l\\ \pi\end{matrix}$	$\begin{matrix}r\\ \Lambda\end{matrix}$	beliebig	$\begin{matrix}\mathrm{I}\\ *\end{matrix}$	
44	$\begin{matrix}l\\ \Lambda\end{matrix}$	$\begin{matrix}r\\ \tilde{\pi}\end{matrix}$	beliebig	$\begin{matrix}\mathrm{I}\\ *\end{matrix}$	

Abb. 51

so lang; die linke Teilzone ist die halbierte Zone aus Abb. 50e. Den beschriebenen Teil des Prozesses nennen wir *linke Teilung*. Mit dieser Bezeichnung drücken wir den Sachverhalt aus, daß wir zur Zerlegung der Zone in zwei Teilzonen kamen, indem wir von der Konfiguration aus Abb. 50d mit dem betrachtenden Zustand $\overset{l}{\pi}$ am linken Rand ausgingen. Analog verifiziert man, daß in der gleichen Anzahl von Takten die *rechte Teilung* realisiert wird, d. h. die Konfiguration aus Abb. 50e mit dem betrachtenden Zustand $\overset{r}{\pi}$ am rechten Ende in die Konfiguration aus Abb. 50f übergeht. In diesem Fall werden zu Beginn die Befehle aus $\mathfrak{R}_r$ angewandt, bis die Konfiguration aus Abb. 48j entsteht, und dann sichern die Befehle 39 und 40 den Übergang in einem Takt zur Konfiguration aus Abb. 50f. Eine *Teilung* ist, mit anderen Worten, eine Zerlegung der Schützenkette in zwei kürzere Ketten. Jede hat dabei ihren linken und rechten Flügelmann.

Jetzt können wir den Beweis der Behauptung B vollenden: Es sei $n = 2^s$. Dann wird der uns interessierende Prozeß in s Zyklen zerlegt. Im ersten Zyklus, der nicht länger als $\frac{3n}{2}$ Takte dauert, entsteht die beschriebene linke Teilung. Im zweiten Zyklus entsteht gleichzeitig eine rechte Teilung in der linken Teilzone der Länge $\frac{n}{2} = 2^{s-1}$ und eine linke Teilung in der rechten Teilzone derselben Länge; dazu werden nicht mehr als $\frac{3n}{4}$ Takte benötigt. Als Ergebnis (vgl. Abb. 50g) erscheinen vier Zonen, je mit der Länge $\frac{n}{4}$, die im folgenden Zyklus gleichzeitig einer weiteren Teilung unterworfen werden, wozu nicht mehr als $\frac{3n}{8}$ Takte gebraucht werden, usw. Nach dem $(s-1)$-ten Zyklus entsteht schließlich eine Konfiguration der Form, wie sie in Abb. 50h dargestellt ist. Jetzt gelangen die Befehle 41 bis 44 des Programms $\mathfrak{R}$ zur Anwendung und erzeugen in einem Takt die „Feuer"-Konfiguration aus Abb. 50c. Die Gesamtzahl der benutzten Takte überschreitet also nicht die Zahl

$$\frac{3n}{2} \cdot \left(1 + \frac{1}{2} + \cdots + \frac{1}{2^{s-1}}\right) + 1 \leqq 3n.$$

Damit ist die Behauptung B bewiesen.

Wir erläutern noch in allgemeinen Zügen, wie das Programm geändert werden muß, damit es bei beliebigem n, das also nicht mehr notwendig eine Potenz von 2 sein muß, funktioniert. Dafür genügt es, die Prozedur der Teilung für den Fall abzuändern, daß die aktive Zone eine ungerade Länge hat. Das läßt sich etwa so erreichen: Das Element, das sich bei ungerader Länge in der Mitte der Zone befindet, gelangt in einen neuen Zustand, der in einem natürlichen Sinne dem Paar der Zustände $\overset{r}{\pi}\,\overset{l}{\pi}$ „gleichwertig" ist, und erhält die Möglichkeit, seinen Einfluß

auf beide Teilzonen als für sie gemeinsames begrenzendes Element auszudehnen. Die Realisierung dieser Idee erfordert, im Vergleich mit dem Programm $\mathfrak{R}$, eine gewisse Vergrößerung der Anzahl der Zustände, die Abschätzung für die Rechenzeit bleibt die vorige, d. h., die Taktanzahl bleibt unter $3n$.[1])

3.10. Ein Vergleich von Berechnungen auf v.-Neumann-Automaten und auf Turing-Maschinen

Der vorliegende Abschnitt ist der Erörterung der folgenden Frage gewidmet: Welche Klassen von Aufgaben kann man auf v.-Neumann-Automaten lösen, und worin besteht die Besonderheit der entsprechenden Lösungsprozesse im Vergleich zu dem, was wir bereits über Turing-Maschinen wissen?

3.10.1. Eine Kodierung

Aus den gleichen Motiven wie früher bei der Untersuchung von Turing-Maschinen betrachten wir die v.-Neumann-Automaten als Instrument zur Berechnung von Funktionen. Ohne Beschränkung der Allgemeinheit können wir uns auf die Untersuchung von Wortfunktionen konzentrieren, deren Argumente und Werte Wörter im Alphabet {0, 1} sind. Wir werden uns dabei auf die folgende Definition für die v.-Neumann-Berechenbarkeit stützen: Der v.-Neumann-Automat $\mathfrak{R}$ *berechnet die Funktion* f, wenn $\mathfrak{R}$ für jedes Wort P aus dem Definitionsbereich von f die P darstellende oder, wie man auch sagt, P kodierende Konfiguration in die das Wort $R = f(P)$ kodierende Konfiguration überführt. Wir müssen also nur noch präzisieren, welche Form der Kodierung wir für die Wörter benutzen wollen. Am natürlichsten erscheint es wohl, als Kode für das Wort $P = P(1)\cdots P(n)$ einfach das Wort selbst zu nehmen, genauer gesagt, die Konfiguration

$$\cdots \emptyset\emptyset P(1)P(2)\cdots P(n)\emptyset\emptyset\cdots,$$

deren aktiver Teil mit dem Wort P übereinstimmt. Dabei tritt aber eine Schwierigkeit auf, die wir an einem Beispiel erläutern wollen.

Die betrachtete Funktion f sei durch $f(000) = 000$ und $f(P) = 1$ für alle $P \neq 000$ definiert. Speziell ist dann $f(0000) = 1$. Diese Funktion ist außerordentlich einfach, und es wäre wohl mehr als befremdend, wenn sich herausstellen würde, daß sie nach

[1]) Man kann zeigen, daß bei keiner Lösung der Aufgabe die Zeit vom Befehlsempfang durch den Flügelmann bis zum Feuern kleiner als $2n - 2$ sein kann. Eine erste Lösung mit der minimal möglichen Zeit erhielt der Japaner E. Goto; in der Lösung von Goto hatte jedoch jeder Schütze, d. h. jedes v.-Neumann-Element, viele tausend Zustände. Der sowjetische Mathematiker V. I. Levenštejn fand eine Lösung mit der Minimalzeit $2n - 2$ und 9 Zuständen.

der benutzten Berechenbarkeitsdefinition nicht v.-Neumann-berechenbar ist. Es sei also $\mathfrak{N}$ ein diese Funktion bei der vorgeschlagenen Kodierung berechnender v.-Neumann-Automat. Unter den fünf Befehlen dieses Automaten, die als linke Seiten die Tripel $\emptyset\emptyset 0$, $\emptyset 00$, 000, $00\emptyset$ und $0\emptyset\emptyset$ haben, muß wenigstens einer aktiv sein (d. h., es muß $\alpha(t+1)$ verschieden von $\alpha(t)$ sein), denn sonst wäre auf die Anfangskonfiguration $\cdots \emptyset 0000\emptyset \cdots$ kein aktiver Befehl anwendbar, und folglich wäre sie selbst passiv, d. h., der Automat $\mathfrak{N}$ würde sie nicht verändern. Das widerspricht aber der Forderung $f(0000) = 1$. Also gibt es einen aktiven Befehl der angegebenen Form. Dann kann aber die Konfiguration $\cdots \emptyset 000\,\emptyset \cdots$ nicht passiv sein, d. h., der Automat beginnt, sie umzuformen. Weil aber $f(000) = 000$ sein soll, muß nach einer bestimmten Anzahl von Takten (etwa nach σ Takten) wieder die Konfiguration $\cdots \emptyset 000\emptyset \cdots$ vorliegen, für die dann ein neuer Transformationszyklus beginnt, und das wiederholt sich zyklisch aller σ Takte. Der Automat gelangt also, wenn er mit der Anfangskonfiguration $\cdots \emptyset 000\emptyset \cdots$ beginnt, in einen unendlichen Prozeß, und es ist (zumindest ohne zusätzliche Festlegungen) nicht klar, in welchem Stadium dieses Prozesses die Berechnung als abgeschlossen und für das Ablesen des Resultats als geeignet anzusehen ist.

Dieses Beispiel zeigt uns, daß die Art und Weise der Kodierung, die wir (natürlich mehr oder minder willkürlich) heranziehen, nicht nur Informationen über das Wort enthalten muß, sondern auch darüber, ob es als Eingabegröße, Zwischenresultat oder Endresultat fungiert. Zusätzlich wollen wir fordern, daß die das Endresultat darstellende Konfiguration passiv ist. Das entspricht der Forderung, daß der Automat nach Auffinden des Endresultats stoppt. Es gibt natürlich ganz unterschiedliche Kodierungen, die diesen Forderungen genügen. Um etwas Bestimmtes vor Augen zu haben, bleiben wir bei einer buchstabenweisen Kodierung, wie wir sie faktisch schon, wenn auch nicht so klar, bei der Modellierung einer Turing-Maschine durch einen v.-Neumann-Automaten benutzt haben:

1. Es werden v.-Neumann-Automaten mit einem zweistöckigen Zustandsalphabet betrachtet. Unter den Zuständen zeichnen wir aus: den Ruhezustand $\begin{matrix}\Lambda\\ \Lambda\end{matrix}$, die Zustände $\begin{matrix}0\\ q_0\end{matrix}$, $\begin{matrix}1\\ q_0\end{matrix}$ als Kodesymbole für den ersten Buchstaben im Anfangswort (Null oder Eins), die Zustände $\begin{matrix}0\\ !\end{matrix}$, $\begin{matrix}1\\ !\end{matrix}$ als Kodesymbole für den ersten Buchstaben im Resultatwort und die Zustände $\begin{matrix}0\\ \Lambda\end{matrix}$, $\begin{matrix}1\\ \Lambda\end{matrix}$ als Kodesymbole der Null und der Eins für alle anderen Positionen im Anfangs- und Resultatwort. Die Konfiguration in Abb. 52c stellt die Anfangskonfiguration mit dem Wort $P = P(1)\cdots P(n)$ und die Konfiguration in Abb. 52d die Resultatkonfiguration mit dem Wort $R = R(1)R(2)\cdots R(s)$ dar.

2. Es wird vorausgesetzt, daß jeder Befehl

$$\alpha^-(t)\alpha(t)\alpha^+(t) \to \alpha(t+1),$$

auf dessen linker Seite nur die Zustände $\begin{matrix}\Lambda\\ \Lambda\end{matrix}, \begin{matrix}0\\ \Lambda\end{matrix}, \begin{matrix}1\\ \Lambda\end{matrix}, \begin{matrix}0\\ !\end{matrix}, \begin{matrix}1\\ !\end{matrix}$ stehen, passiv ist, d. h., für ihn gelte $\alpha(t+1) = \alpha(t)$. Insbesondere ist jede Resultatkonfiguration passiv. Das Erscheinen des Symbols „!" in der unteren Etage einer passiven Konfiguration kann als Signal des Automaten, durch seinen Stop in der passiven Konfiguration ein erhaltenes Resultat ausgeben zu wollen, gedeutet werden.

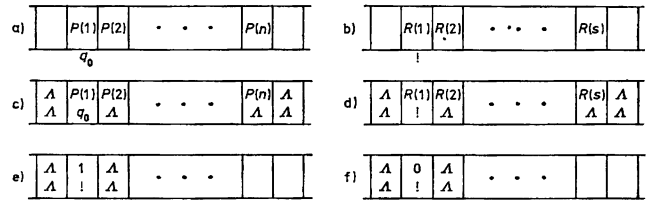


Abb. 52

Genau wie im Fall der Turing-Berechnungen kann man die Kompliziertheit des in einem v.-Neumann-Automaten $\mathfrak{N}$ ablaufenden Rechenprozesses durch signalisierende Funktionen charakterisieren, für die wir die früheren Bezeichnungen beibehalten. Die Zeitsignalisierende $t_{\mathfrak{N}}(P)$ gibt also die Anzahl der Takte an, die der Automat $\mathfrak{N}$ benötigt, um die Anfangskonfiguration, die das Wort P kodiert, in die zugehörige (passive) Endkonfiguration überzuführen.

Folgende Bemerkung zur Abschätzung der signalisierenden Funktion $t_{\mathfrak{N}}(P)$ ist für das Weitere wichtig: Wir nehmen an, daß es zur Bestimmung des Wertes R der Funktion f für ein beliebiges Argument P notwendig ist, das Wort P vollständig (oder fast vollständig) zu kennen, daß es also nicht ausreicht, nur Information über einen kleinen Teil von P, etwa über ein kurzes Anfangsstück bereitzustellen. Dann muß, da in der Anfangskonfiguration nur der erste Buchstabe aus P durch ein von $\begin{matrix}0\\ \Lambda\end{matrix}, \begin{matrix}1\\ \Lambda\end{matrix}$ verschiedenes Symbol kodiert wird, wegen der Eigenschaft 2 der betrachteten Kodierung notwendigerweise

$$t_{\mathfrak{N}}(P) \geqq |P| = n$$

sein. Denn der Automat schafft es einfach nicht, in einer geringeren Anzahl von Takten das gesamte Anfangswort zu lesen.

Die von uns verwendete Methode zur Kodierung der Anfangsinformation hat die Eigenschaft, daß sie dem v.-Neumann-Automaten am Anfang der Rechnung keinen Vorteil gibt, im Vergleich zu einer Turing-Maschine, die den ersten Buchstaben eines Wortes liest. Erst im Prozeß der eigentlichen Arbeit kann der v.-Neumann-Automat seine Vorteile ausnutzen, wobei er wie ein unbeschränkt wachsendes System von Turing-Maschinen mit einem gemeinsamen Speicher wirkt.

Wir unterstreichen nochmals, daß in allen nichttrivialen Fällen die Ungleichung

$$t_{\mathfrak{M}}(P) \geqq |P|$$

gilt, die wir an späterer Stelle ausnutzen werden.

3.10.2. Die Berechenbarkeit nach v. Neumann und nach Turing

Wir stellen nun einige grundlegende Tatsachen zusammen, die die „rechnerischen Fähigkeiten" der Turing-Maschinen und der v.-Neumann-Automaten betreffen. Die Beweise dafür werden in den folgenden Abschnitten geführt. Wir beschränken uns hier auf die Angabe der Resultate und einige Kommentare.

Zu Abschnitt 3.8 wurde bereits gezeigt, wie man zu einer Turing-Maschine $\mathfrak{M}$ einen sie modellierenden v.-Neumann-Automaten $\mathfrak{M}$ konstruieren kann. Die Maschine $\mathfrak{M}$ berechne nun die Funktion $f(P)$. Wenn dabei die Maschine $\mathfrak{M}$ für ein beliebiges Wort P aus dem Definitionsbereich der Funktion f die Konfiguration aus Abb. 52a in die Konfiguration aus Abb. 52b in $t_{\mathfrak{M}}(P)$ Takten überführt, so führt $\mathfrak{M}$ in der gleichen Anzahl von Takten die Konfiguration aus Abb. 52c in die Konfiguration aus Abb. 52d über, d. h. $t_{\mathfrak{M}}(P) = t_{\mathfrak{M}}(P)$. Also gilt der folgende Satz.

Satz 1. *Jede Turing-berechenbare Funktion f ist auch v.-Neumann-berechenbar, wobei es zu jeder Turing-Berechnung von f eine ebenso schnelle v.-Neumann-Berechnung gibt.*

Wir wissen bereits, daß sich ein v.-Neumann-Automat in einer Weise verhalten kann, die sich wesentlich von einer einfachen Nachbildung einer Turing-Maschine oder von einem festen endlichen System von Turing-Maschinen mit einem gemeinsamen Speicher unterscheidet. In Zusammenhang damit ergeben sich nun einige Fragen, die die zusätzlichen rechnerischen Möglichkeiten der v.-Neumann-Automaten gegenüber den Turing-Maschinen betreffen. Die erste und grundlegende Frage ist natürlich die folgende: Gibt es v.-Neumann-berechenbare Funktionen, die nicht Turing-berechenbar sind? Die aufgrund der Fundamentalhypothese der Algorithmentheorie zu erwartende negative Antwort auf diese Frage läßt sich nun tatsächlich beweisen:

Satz 2. *Jede v.-Neumann-berechenbare Funktion ist auch Turing-berechenbar.*

Den Beweis dieses Satzes kann man als eine weitere Stütze für die Fundamentalhypothese ansehen.

Obwohl, wie die Sätze 1 und 2 zeigen, die v.-Neumann-Automaten und die Turing-Maschinen dasselbe leisten, tun sie das jedoch auf verschiedene Weise. Daher kann es aus der Sicht der Qualität der Algorithmen sehr wohl passieren — und wir werden gerade zeigen, daß es sich in der Tat so verhält —, daß der v.-Neumann-Automat insgesamt vorteilhafter bei der Lösung gewisser Aufgabenscharen ist. Wir können vermuten, daß immer dann eine solche Situation eintritt, wenn man für die Aufgaben

des betrachteten Typs den Lösungsprozeß stark parallelisieren kann. Der folgende Satz zeigt, daß das wirklich so ist.

Satz 3. *Es existiert ein v.-Neumann-Automat $\mathfrak{N}_0$, der die Symmetrie von Wörtern im Alphabet $\{0, 1\}$ in linearer Zeit erkennt, d. h. in einer Zeit, die $C_{\mathfrak{N}_0} \cdot |P|$ nicht übertrifft, wobei $C_{\mathfrak{N}_0}$ eine von P unabhängige Konstante ist.*

Wir erinnern zum Vergleich daran (vgl. Abschnitt 3.6), daß für eine die Symmetrie von Wörtern erkennende Turing-Maschine $\mathfrak{M}$ stets eine Ungleichung der Form

$$T_{\mathfrak{M}}(n) \geqq C_{\mathfrak{M}} \cdot n^2 \tag{1}$$

gelten muß, wobei $C_{\mathfrak{M}}$ eine von n unabhängige positive Konstante ist. Damit verfügen wir über ein konkretes Beispiel für einen v.-Neumann-Automaten $\mathfrak{N}_0$, so daß für jede Turing-Maschine $\mathfrak{M}$, die dasselbe wie $\mathfrak{N}_0$ leistet (nämlich die Symmetrie von Wörtern erkennt), die Ungleichung

$$T_{\mathfrak{M}}(n) \geqq D_{\mathfrak{M}} \cdot T^2_{\mathfrak{N}_0}(n) \tag{2}$$

gilt, wobei $D_{\mathfrak{M}}$ eine von n unabhängige positive Konstante ist. Für $D_{\mathfrak{M}}$ kann man dabei $C_{\mathfrak{M}} / C^2_{\mathfrak{N}_0}$ nehmen.

Natürlich fragt man nun, ob es für gewisse Funktionen möglich ist, durch die Verwendung von v.-Neumann-Automaten bei der Berechnung eine noch bedeutendere Zeiteinsparung zu erreichen. Gibt es beispielsweise eine Funktion f und einen sie berechnend v.-Neumann-Automaten $\mathfrak{N}$, so daß für eine beliebige, f berechnende Turing-Maschine $\mathfrak{M}$ eine Ungleichung der Form

$$T_{\mathfrak{M}}(n) \geqq E_{\mathfrak{M}} \cdot T^2_{\mathfrak{N}}(n) \tag{3}$$

gilt, wobei $E_{\mathfrak{M}}$ eine von n unabhängige positive Konstante ist? Der folgende Satz klärt den hier vorliegenden Sachverhalt.

Satz 4. *Für jede berechenbare Funktion f und jeden f berechnenden v.-Neumann-Automaten $\mathfrak{N}$ gibt es eine f berechnende Turing-Maschine $\mathfrak{M}$, so daß für jedes Anfangswort P die Ungleichung*

$$t_{\mathfrak{M}}(P) \leqq 18 \cdot t^2_{\mathfrak{N}}(P) \tag{4}$$

gilt.

Der Satz 4 beantwortet also die zuletzt gestellte Frage negativ. Durch die Verwendung von v.-Neumann-Automaten läßt sich keine größere Zeiteinsparung als von der Ordnung der Quadratwurzel im Vergleich zu einer „ökonomischen" Turing-Berechnungen erzielen. Das Beispiel der Erkennung der Symmetrie zeigt dabei, daß eine solche Zeitraffung mitunter tatsächlich erreichbar ist.

Den Beweis von Satz 3 werden wir in Abschnitt 3.10.3 führen. Die Sätze 2 und 4 sind einfache Folgerungen einer allgemeinen Betrachtung, der der Abschnitt 3.10.4 gewidmet ist. Dort wird nämlich ein Algorithmus beschrieben, der zu einem beliebigen v.-Neumann-Programm $\mathfrak{N}$ ein Turing-Programm $\hat{\mathfrak{N}}$ konstruiert, das in einem be-

stimmten Sinne das v.-Neumann-Programm $\mathfrak{N}$ modelliert. Berechnet $\mathfrak{N}$ die Funktion f, so berechnet auch $\mathfrak{N}$ die Funktion f. Dabei wird jedoch jeder Arbeitstakt des Automaten $\mathfrak{N}$ durch eine große Anzahl von Takten der Maschine $\mathfrak{N}$ modelliert (darin besteht der wesentliche Unterschied zur früher betrachteten Modellierung einer Turing-Maschine durch einen v.-Neumann-Automaten). Diese Zahl bleibt auch nicht fest; sie wächst im allgemeinen mit der Nummer des modellierten Taktes. Es wird sich aber zeigen, daß die Turing-Maschine $\mathfrak{N}$ bei dem angegebenen Imitationsprozeß nicht mehr als $18t^2$ Takte zur Modellierung von t Takten des Automaten $\mathfrak{N}$ braucht.

Wir bemerken zum Abschluß, daß die Algorithmen, die $\mathfrak{M}$ in $\mathfrak{M}$ (bei einer v.-Neumann-Modellierung) bzw. $\mathfrak{N}$ in $\mathfrak{N}$ (bei einer Turing-Modellierung) überführen, in der Terminologie von Abschnitt 2.4 als programmierende Algorithmen für die v.-Neumann-Programmierung bzw. die Turing-Programmierung aufgefaßt werden können.

3.10.3. Die Symmetrieerkennung auf v.-Neumann-Automaten

Man kann sofort „im Kopf" einen v.-Neumann-Automaten konstruieren, dessen Existenz im Satz 3 behauptet wurde. Wir ziehen hier jedoch einen etwas umständlicheren, aber, wie es uns scheint, lehrreicheren Weg vor, indem wir uns auf eine Analyse und die Lösung der Aufgabe über die Synchronisation einer Schützenkette stützen. Wir benötigen ein Programm, durch das die Anfangskonfiguration aus Abb. 52c in die Konfigurationen aus Abb. 52e oder 52f übergeführt wird, je nachdem, ob das Wort $P = P(1)\cdots P(n)$ symmetrisch ist oder nicht. Dabei wird außerdem gefordert, daß diese Transformation in linearer Zeit zu erfolgen hat (d. h. in einer Zeit, die $C \cdot n$ nicht überschreitet, wobei C eine geeignete Konstante ist). Wir zerlegen diese Aufgabe in zwei Teilaufgaben:

Erste Aufgabe. Die Anfangskonfiguration sei so wie in Abb. 53a gewählt, wobei der Einfachheit halber n gerade sei ($n = 2\nu$). Hierbei seien L und R zwei spezielle untere Symbole der zweistöckigen Buchstaben. Ihre inhaltliche Bedeutung besteht in der Unterscheidung der Buchstaben der linken Hälfte des Wortes P von den Buchstaben der rechten Hälfte. Gesucht ist ein Programm $\mathfrak{N}_s$, das die Konfiguration aus Abb. 53a in linearer Zeit $\left(\text{man kann sogar fordern: „in der Zeit } \nu = \frac{n}{2}\text{"}\right)$ in die Konfiguration aus Abb. 52e oder 52f überführt, und das auf die Frage „Ist das Wort P symmetrisch?" richtig antwortet. Ein solches Programm ist in Abb. 54 dargestellt. Die ihm zugrunde liegende Idee ist außerordentlich einfach. Während der ν Takte erfolgt in jedem Takt eine gleichzeitige Verschiebung der rechten Halbkonfiguration um eine Position nach links, der linken Halbkonfiguration um eine Position nach rechts und ein Vergleich der sich an der Nahtstelle berührenden Symbole der Halbkonfigurationen.

In der Abb. 53a ist der Vergleich „$P(\nu) = P(\nu + 1)$?", in der Abb. 53b der Vergleich „$P(\nu - 1) = P(\nu + 2)$?", usw. dargestellt. Ist das Wort P symmetrisch,

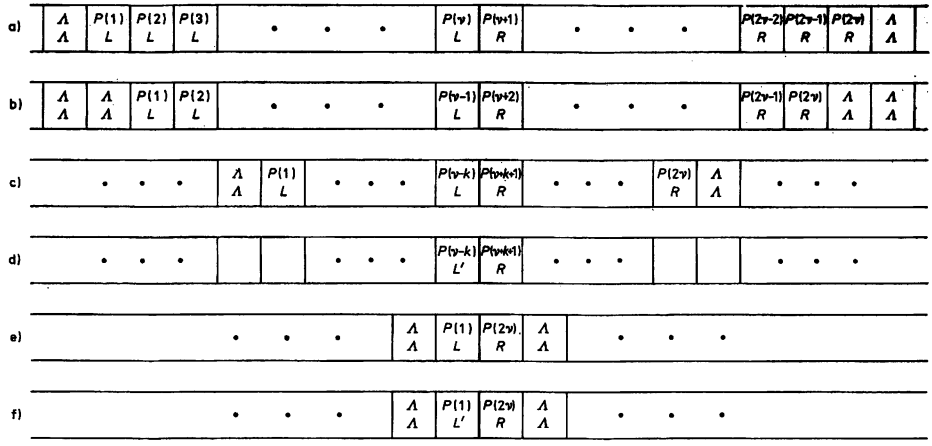

a)
Λ Λ
P(1) L
P(2) L
P(3) L
P(ν) L
P(ν+1) R
P(2ν-2) R
P(2ν-1) R
P(2ν) R
Λ Λ
b)
Λ Λ
Λ Λ
P(1) L
P(2) L
P(ν-1) L
P(ν+2) R
P(2ν-1) R
P(2ν) R
Λ Λ
Λ Λ
c)
Λ Λ
P(1) L
P(ν-k) L
P(ν+k+1) R
P(2ν) R
Λ Λ
d)
P(ν-k) L'
P(ν+k+1) R
e)
Λ Λ
P(1) L
P(2ν) R
Λ Λ
f)
Λ Λ
P(1) L'
P(2ν) R
Λ Λ

Abb. 53

so erscheint im $(\nu - 1)$-ten Takt die Konfiguration aus Abb. 53e, die im nächsten Takt in die Konfiguration aus Abb. 52e übergeht. Ist das Wort P dagegen nicht symmetrisch, und wird das zum ersten Mal beim Vergleich „$P(\nu - k - 1) = P(\nu + k)$?" deutlich, d. h., ist $P(\nu - k - 1) \neq P(\nu + k)$, so wird in diesem Takt an der Nahtstelle das untere Symbol L' eingeführt (vgl. Abb. 53d), das bei allen weiteren Verschiebungen erhalten bleibt. Als vorletzte Konfiguration erscheint die Konfiguration aus Abb. 53f, die in einem Takt in die Konfiguration aus Abb. 52f übergeht. Also genügt es zum Beweis von Satz 3, die folgende Aufgabe zu lösen.

	$\alpha^-(t)$	$\alpha(t)$	$\alpha^+(t)$	$\alpha(t+1)$	Erläuterungen
1	beliebig	$\begin{matrix} x \\ R \end{matrix}$	$\begin{matrix} y \\ q \end{matrix}$	$\begin{matrix} y \\ q \end{matrix}$	x, y, q beliebig; Verschiebung der rechten Halbkonfiguration
2	$\begin{matrix} z \\ q \end{matrix}$	$\begin{matrix} x \\ L \end{matrix}$	$\begin{matrix} y \\ L \end{matrix}$	$\begin{matrix} z \\ q \end{matrix}$	x, y, z, q beliebig; Verschiebung der linken Halbkonfiguration bis zum Entdecken einer Unsymmetrie
3	$\begin{matrix} z \\ L \end{matrix}$	$\begin{matrix} x \\ L \end{matrix}$	$\begin{matrix} x \\ R \end{matrix}$	$\begin{matrix} z \\ L \end{matrix}$	
4	$\begin{matrix} z \\ L \end{matrix}$	$\begin{matrix} x \\ L \end{matrix}$	$\begin{matrix} y \\ R \end{matrix}$	$\begin{matrix} z \\ L' \end{matrix}$	x, y, z, q beliebig, $x \neq y$; Verschiebung der linken Halbkonfiguration nach Entdecken einer Unsymmetrie
5	$\begin{matrix} z \\ L \end{matrix}$	$\begin{matrix} x \\ L' \end{matrix}$	beliebig	$\begin{matrix} z \\ L' \end{matrix}$	
6	$\begin{matrix} \Lambda \\ \Lambda \end{matrix}$	$\begin{matrix} x \\ L \end{matrix}$	$\begin{matrix} y \\ R \end{matrix}$	$\begin{matrix} 1 \\ ! \end{matrix}$	x, y, z beliebig, $x \neq y$; Ausgabe des Resultats
7	$\begin{matrix} \Lambda \\ \Lambda \end{matrix}$	$\begin{matrix} x \\ L \end{matrix}$	$\begin{matrix} y \\ R \end{matrix}$	$\begin{matrix} 0 \\ ! \end{matrix}$	
8	$\begin{matrix} \Lambda \\ \Lambda \end{matrix}$	$\begin{matrix} x \\ L' \end{matrix}$	$\begin{matrix} z \\ R \end{matrix}$	$\begin{matrix} 0 \\ ! \end{matrix}$	

Abb. 54

Zweite Aufgabe. Gesucht ist ein v.-Neumann-Programm $\mathfrak{N}_\pi$, das die Konfiguration aus Abb. 52a in die Konfiguration aus Abb. 53a unter Einhaltung folgender Bedingungen transformiert:

a) Die Transformation erfolgt in einer Zeit, die $C \cdot n$ nicht überschreitet, wobei C eine geeignete Konstante ist.

b) $\mathfrak{N}_\pi$ hat außer dem Zuständen $\begin{matrix} \Lambda \\ \Lambda \end{matrix}$ (Ruhezustand), $\begin{matrix} 0 \\ L' \end{matrix}, \begin{matrix} 1 \\ L' \end{matrix}, \begin{matrix} 0 \\ R' \end{matrix}, \begin{matrix} 1 \\ R \end{matrix}$ mit $\mathfrak{N}_s$ keine Zustände gemeinsam.

c) Die Zustände $\begin{matrix} 0 \\ L \end{matrix}, \begin{matrix} 1 \\ L' \end{matrix}, \begin{matrix} 0 \\ R' \end{matrix}, \begin{matrix} 1 \\ R \end{matrix}$ erscheinen erstmals im letzten Takt der Transformation, d. h. bei Auftreten der Konfiguration aus Abb. 53a.

Man sieht leicht, daß die Vereinigung der Listen der aktiven Befehle der Programme $\mathfrak{N}_s$ und $\mathfrak{N}_\pi$ ein Programm $\mathfrak{N}_0$ ergibt, das die Forderungen von Satz 3 erfüllt: Zunächst wird nach dem Programm $\mathfrak{N}_\pi$ der Übergang von der Konfiguration aus

Abb. 52a zur Konfiguration aus Abb. 53a durchgeführt, und anschließend erfolgt aufgrund des Programms $\mathfrak{P}_s$ der Übergang von der Konfiguration aus Abb. 53a zur Konfiguration aus Abb. 52e oder 52f. Dabei wird die geforderte Zeitbegrenzung nicht überschritten.

Die Rolle des Programms $\mathfrak{P}_n$ besteht offensichtlich darin, im gleichen Zeitpunkt unter jeden Buchstaben des Wortes P die Information darüber zu bringen, in welcher Hälfte des Wortes er sich befindet, in der linken oder in der rechten. Die Analogie zur Aufgabe über die Schützen ist offensichtlich. Die Besonderheit der neuen Situation besteht in den folgenden beiden Bedingungen:

1. Wir haben es jetzt mit „Schützen" zweier Kategorien zu tun (Nullen und Einsen), vorher hatten wir nur eine Sorte (Striche).

2. An die Stelle des zuvor für alle Schützen gemeinsamen Befehls „Feuer!" (unteres Zeichen „*") erhalten jetzt die Schützen der linken Hälfte der Kette den Befehl „Feuer nach links!" (unteres Zeichen „L") und die Schützen der rechten Hälfte der Kette den Befehl „Feuer nach rechts!" (unteres Symbol „R").

Diese beiden Besonderheiten sind jedoch unwesentlich. Die von uns dargelegte Lösung (vgl. Abschnitt 3.9) kann leicht den neuen Bedingungen angepaßt werden, wobei die Zeitabschätzung die gleiche bleibt, d. h., bis zum synchronen Feuern vergehen nicht mehr als $5n$ Takte. Die notwendigen Modifizierungen erfordern lediglich eine Vergrößerung der Anzahl der inneren Zustände des v.-Neumann-Elements (des „Schützen"). Statt der beiden oberen Symbole „l" und „r" brauchen wir jetzt vier Symbole „l^0", „l^1", „r^0", „r^1", um uns nämlich zu merken, welche Buchstaben („0" oder „1") an den Stellen standen, an denen die entsprechenden Zyklen der Teilung zu Ende gingen. Im Schlußtakt ermöglicht das die Wiederherstellung des Anfangswortes P in der oberen Etage. Die Vergrößerung der Anzahl der Zustände ist außerdem deshalb nötig, weil von der ersten Teilung an zur Unterscheidung der rechten von der linken Hälfte in der linken und rechten Halbzone der Länge $\frac{n}{2}$ jeweils verschiedene Zustände benutzt werden müssen. Das ermöglicht es, im Schlußtakt überall in der linken Halbzone das untere Symbol „L" und überall in der rechten Halbzone das untere Symbol „R" einzuführen. Wir lassen die weiteren Einzelheiten weg und betrachten den Satz 3 damit als bewiesen.

3.10.4. Die Turing-Modellierung eines v.-Neumann-Automaten

Die Idee, die dieser Modellierung zugrunde liegt, ist in Abb. 55 dargestellt. Wir betrachten die v.-Neumann-Konfiguration aus Abb. 55a, in der rechts von $x(s)$ und links von $x(1)$ nur Ruhezustände vorliegen mögen. Die aktive Zone dieser Konfiguration wird also durch das Wort $x(1)x(2)\cdots x(s)$ bedeckt, wobei im allgemeinen unter dem $x(i)$ ($i = 1, 2, \ldots, s$) auch der Ruhezustand Ø vorkommen darf. Diese Konfiguration werde in einem Takt in die Konfiguration aus Abb. 55b übergeführt, deren

aktive Zone offenbar in keinem Fall länger als $s + 2$ ist und durch das Wort $y(0)y(1)\cdots y(s)y(s+1)$ im Alphabet aller Zustände überdeckt wird. In Abb. 55c bis 55f sind die Konfigurationen einer „dreistöckigen" Turing-Maschine $\mathfrak{N}$ dargestellt, die bestrebt ist, in ihrer zweiten Etage die Arbeit des gegebenen v.-Neumann-Automaten $\mathfrak{N}$ zu imitieren. In Abb. 55c und 55f ist gezeigt, wie in $\mathfrak{N}$ die Konfigurationen aus Abb. 55a und 55b kodiert werden. Zur Imitation eines Taktes des

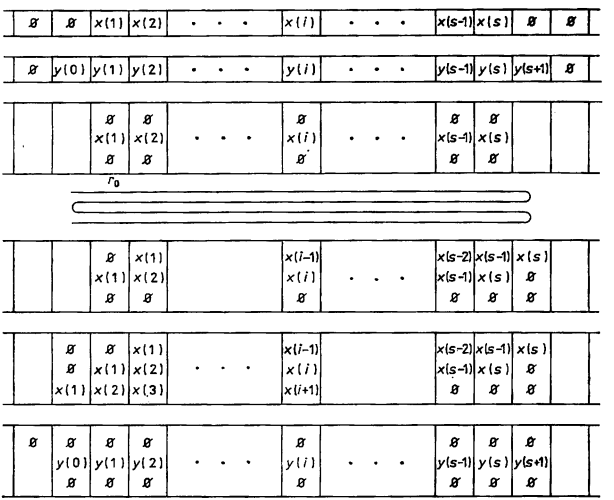


Abb. 55

v.-Neumann-Automaten läuft der Kopf von $\mathfrak{N}$ hin und her, wie das unter Abb. 55c durch Schlangenlinien angedeutet ist. Bei dem ersten Lauf von links nach rechts schreibt die Maschine in die obere Etage jeder besuchten Zelle den Inhalt der zweiten Etage ihrer linken Nachbarzelle. Im Ergebnis entsteht die Konfiguration aus Abb. 55d. Beim anschließenden Rücklauf von rechts nach links wird in die untere Etage jeder Zelle der Inhalt der zweiten Etage ihrer rechten Nachbarzelle geschrieben. Nach einem vollständigen Hin- und Rücklauf ist also in jeder Zelle der Zone der Länge $s + 2$ die Information über den Zustand des entsprechenden v.-Neumann-Elements und über die Zustände ihres rechten und linken Nachbarn enthalten. Bei dem sich nun anschließenden Lauf der Maschine $\mathfrak{N}$ von links nach rechts realisiert sie in jeder

Zelle die Ersetzung des in ihr enthaltenen Tripels von v.-Neumann-Zuständen durch den Zustand, der im entsprechenden Befehl des Programms von $\mathfrak{N}$ vorgeschrieben ist. Dieser Zustand wird in die zweite Etage geschrieben; die beiden anderen Etagen werden gleichzeitig geleert. Am Ende dieses dritten Laufes hat die Niederschrift auf dem Turing-Band bereits die in Abb. 55f angegebene Form. Abschließend kehrt der Kopf nach links zurück, ohne dabei irgendwelche Inhalte zu ändern, und er geht in den in Abb. 55c mit r_0 bezeichneten Zustand über, wenn er die Zelle mit dem Inhalt $y(0)$ erreicht hat. Damit ist die Maschine zum folgenden Zyklus bereit, in dem der nächste v.-Neumann-Befehl imitiert wird. Es ist klar, daß man eine solche Turing-Maschine $\mathfrak{N}$ effektiv angeben kann. Damit haben wir zugleich eine weitere Stütze für die Fundamentalhypothese der Algorithmentheorie.

Beim Aufstellen des Programms für $\mathfrak{N}$ müssen allerdings einige hinreichend evidente Details berücksichtigt werden. So muß die Maschine $\mathfrak{N}$ beispielsweise in jedem Zyklus die Zelle markieren, von der aus sie ihren Lauf beginnt, damit sie diese beim Rücklauf wiederfindet und anschließend ihren Weg genau eine Zelle weiter fortsetzt.

Wir schätzen zunächst die Anzahl der Takte ab, die die Maschine $\mathfrak{N}$ braucht, um t Takte des v.-Neumann-Automaten $\mathfrak{N}$ zu imitieren, wenn er mit einer Anfangskonfiguration beginnt, deren aktive Zone die Länge d hat. Offensichtlich durchläuft $\mathfrak{N}$ dabei t Zyklen, deren Gesamtlänge nicht größer ist als die folgende arithmetische Summe:

$$4(d+2)+4(d+4)+\cdots+4(d+2t)=4(d+t+1)\,t. \tag{4}$$

Wir nehmen nun an, daß der v.-Neumann-Automat $\mathfrak{N}$ den Funktionswert $f(P)$ einer berechenbaren Funktion f in $t_{\mathfrak{N}}(P)$ Takten berechnet. Es ist leicht zu übersehen, wie man die Maschine $\mathfrak{N}$ in eine Maschine $\mathfrak{N}^*$ überführen kann, die ebenfalls die Funktion f berechnet, und wie man die Rechenzeit dieser Maschine abschätzt. Die Maschine $\mathfrak{N}^*$ arbeitet zunächst wie $\mathfrak{N}$, wobei sie die $t_{\mathfrak{N}}(P)$ Takte des Automaten $\mathfrak{N}$ imitiert. Dazu braucht sie, wie aus (4) zu ersehen ist, nicht mehr als $4(|P|+t_{\mathfrak{N}}(P)+1)\,t_{\mathfrak{N}}(P)$ Takte. Anschließend muß sie die zu dieser Zeit vorliegende Bandinschrift auf die Standardform einer abschließenden Turing-Konfiguration bringen. Dafür ist es offenbar notwendig, daß der Kopf die zum Schluß erhaltene Niederschrift, deren Länge nicht größer als $|P|+2t_{\mathfrak{N}}(P)$ ist, noch einmal vorwärts und zurück „abschreitet". Damit erhalten wir für die von der Maschine $\mathfrak{N}^*$ insgesamt benötigte Zeit die Abschätzung

$$t_{\mathfrak{N}^*}(P)\leqq 4\big(|P|+t_{\mathfrak{N}}(P)+1\big)\,t_{\mathfrak{N}}(P)+2\big(|P|+2t_{\mathfrak{N}}(P)\big). \tag{5}$$

Wir erinnern daran, daß in allen nichttrivialen Situationen $t_{\mathfrak{N}}(P)\geqq|P|$ ist (vgl. die Bemerkung am Ende von Abschnitt 3.10.1). Wenn wir folglich $|P|$ und ebenso auch 1 in (5) durch $t_{\mathfrak{N}}(P)$ ersetzen, so erhalten wir: Für jede nichttriviale Situation gilt

$$t_{\mathfrak{N}^*}(P)\leqq 12t_{\mathfrak{N}}^2(P)+6t_{\mathfrak{N}}(P)\leqq 18t_{\mathfrak{N}}^2(P).$$

Das wurde aber im Satz 4 behauptet.

Schlußbemerkungen

Es folgen einige abschließende Bemerkungen zum Inhalt des Buches. Einige davon (I bis V) verweisen auf bibliographische Quellen und verfolgen den Zweck, die Auswahl weiterführender Literatur für interessierte Leser zu erleichtern. Andere wiederum (VI bis VIII) haben den Charakter von weiterführenden Bemerkungen.

I. Auf Grund der Tatsache, daß der Begriff der *berechenbaren arithmetischen Funktion* von der gewählten Formalisierung des Algorithmenbegriffs (vgl. die Abschnitte 2.5.5, 2.5.6, 2.7.2, 3.8.4, 3.10.1) unabhängig ist, spielt er in der Algorithmentheorie eine hervorragende Rolle und bildet die Grundlage für die Formulierung anderer wesentlicher Begriffe.

Wir betrachten als Beispiel den Begriff der *entscheidbaren Menge* (von natürlichen Zahlen). Er präzisiert die intuitive Vorstellung von dem, was man im Bereich der natürlichen Zahlen algorithmisch erkennen kann. Seine genaue Definition lautet: Eine Menge M von natürlichen Zahlen heißt *entscheidbar* (oder *rekursiv*), wenn ihre charakteristische Funktion f_M berechenbar ist; dabei ist definitionsgemäß

$$f_M(n) = \begin{cases} 1, & \text{falls } n \in M, \\ 0, & \text{falls } n \notin M. \end{cases}$$

Ein anderer, von uns noch nicht direkt benutzter Begriff ist der Begriff der *rekursiv aufzählbaren* Menge von natürlichen Zahlen. Eine nichtleere Menge M heißt rekursiv aufzählbar, wenn es eine berechenbare Funktion f gibt, so daß die Folge

$$f(0), f(1), f(2), \ldots, f(n), \ldots \tag{1}$$

aus genau den Elementen der Menge M besteht. Wir unterstreichen, daß in (1) die Elemente nicht notwendig in wachsender Größe vorliegen müssen; außerdem sind in (1) durchaus Wiederholungen zugelassen.

Die Art und Weise der Übertragung dieser und auch anderer Definitionen auf Mengen, die keine Zahlenmengen sind, ist ganz natürlich. So heißt beispielsweise

eine Menge von Wörtern rekursiv aufzählbar, wenn bei einer effektiven Arithmetisierung dieser Wörter (vgl. Abschnitt 1.1.1) eine rekursiv aufzählbare Menge von natürlichen Zahlen erhalten wird. Man zeigt leicht, daß jede entscheidbare Menge auch rekursiv aufzählbar ist. Die Umkehrung davon ist im allgemeinen nicht richtig. So bilden z. B. die Chiffren der selbstanwendbaren Turing-Maschinen eine rekursiv aufzählbare Menge, die nicht entscheidbar ist. Rekursiv aufzählbar sind auch die Satzmengen beliebiger, durch ein endliches Axiomensystem charakterisierter formalisierter mathematischer Theorien.

Obwohl die Klasse der rekursiv aufzählbaren Mengen umfangreicher als die Klasse der entscheidbaren Mengen ist, bilden auch sie nur einen verschwindenden Bruchteil in der Klasse aller Teilmengen der Menge der natürlichen Zahlen (die Klasse der rekursiv aufzählbaren Mengen ist abzählbar unendlich, die Klasse aller Teilmengen ist überabzählbar).

Wir bemerken weiter, daß man unter berechenbaren Funktionen in der Algorithmentheorie nicht nur überall definierte Funktionen versteht, sondern auch *partielle Funktionen*. Denn es kann sich bei der Betrachtung eines Algorithmus herausstellen, daß er partiell ist, d. h., daß er für gewisse Eingabegrößen zu keinem Resultat führt (eine Turing-Maschine kann auf gewisse Konfigurationen nicht anwendbar sein, und der μ-Operator kann einen unendlichen Prozeß erzeugen). Obwohl wir im Buch die Betrachtung partieller Funktionen nach Möglichkeit vermieden haben, kann man mühelos sehen, daß die meisten der von uns gebrachten Fakten auch für partielle Funktionen richtig bleiben. So gilt beispielsweise die folgende Behauptung (vgl. die Sätze aus Abschnitt 2.5.5 und 2.5.6): *Die Klasse der partiell-rekursiven Funktionen stimmt mit der Klasse der auf einer Turing-Maschine berechenbaren Funktionen überein.*

In Abschnitt 3.2.2 haben wir einen Spezialfall einer *Reduktionsmethode* eines algorithmischen Problems $\mathfrak{A}$, das aus einzelnen Problemen $a_1, a_2, \ldots$ besteht, auf ein algorithmisches Problem $\mathfrak{B}$ aus den Einzelproblemen $b_1, b_2, \ldots$ kurz beschrieben. In der allgemeinsten Situation ist die Reduktionsprozedur von $\mathfrak{A}$ auf $\mathfrak{B}$ so beschaffen, daß die Entscheidung für ein Einzelproblem a_i auf mehrere als entschieden angenommene Probleme b_j reduziert wird. Dabei wird im Verlauf der Realisierung dieser Prozedur erklärt, welche der „bereitgestellten" Resultate b_j angefordert werden müssen und wie sie den weiteren Gang des Prozesses beeinflussen. Solche Prozeduren nennt man *bedingte Algorithmen* oder *Reduktionsalgorithmen*, im Unterschied zu den üblichen absoluten Algorithmen. Es ist dabei zugelassen, daß die Problemscharen $\mathfrak{A}$ und $\mathfrak{B}$ beide algorithmisch unentscheidbar sind. Ist dabei $\mathfrak{A}$ auf $\mathfrak{B}$ reduzierbar, jedoch $\mathfrak{B}$ nicht auf $\mathfrak{A}$, so bedeutet das inhaltlich, daß das Problem $\mathfrak{B}$ in einem höheren Grade algorithmisch unentscheidbar ist als das algorithmisch unentscheidbare Problem $\mathfrak{A}$. In einem solchen Fall sagt man, daß das Problem $\mathfrak{B}$ einen *höheren Unentscheidbarkeitsgrad* hat als das Problem $\mathfrak{A}$.

Das Studium der berechenbaren Funktionen, der entscheidbaren und der rekursiv aufzählbaren Mengen, der bedingten Algorithmen und der ihnen entsprechenden Unentscheidbarkeitsgrade nimmt in der Algorithmentheorie einen zentralen Platz

ein. Eine systematische Behandlung dieses Materials kann man in folgenden Lehrbüchern finden:

[1] Успенский, В. А., Лекции о вычислимых функциях, Физматгиз, Москва 1960.

[2] Мальцев, А. И., Алгоритмы и рекурсивные функции, Наука, Москва 1965 (Deutsche Übersetzung: Malcew, A. I., Algorithmen und rekursive Funktionen, Akademie-Verlag, Berlin 1974.).

[3] Hermes, H., Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit, 2. Auflage, Springer-Verlag, Berlin—Heidelberg—New York 1971.

[4] Péter, R., Rekursive Funktionen, 2. Auflage, Akadémiai Kiado, Budapest/ Akademie-Verlag, Berlin 1957.

[5] Davis, M., Computability and Unsolvability, McGraw-Hill, New York 1958.

[6] Kleene, S. C., Introduction to Metamathematics, 5. edition, North Holland, Amsterdam 1967.

[7] Rogers, H., Theory of recursive functions and effective computability, McGraw-Hill, New York 1967.

[8] Shoenfield, J. R., Degrees of unsolvability, Elservier Publ. Co., Amsterdam—London—New York 1971.

[9] Schnorr, C. P., Rekursive Funktionen und ihre Komplexität, Teubner-Verlag, Stuttgart 1974.

II. Die Existenz algorithmisch unentscheidbarer Probleme (vgl. Abschnitt 3.2.2) und auch algorithmisch entscheidbarer, jedoch schwierig zu entscheidender Probleme (vgl. Abschnitt 3.7.2) wurde zunächst durch die Konstruktion künstlicher Beispiele für die Selbstanwendbarkeit und die f-Selbstanwendbarkeit belegt. Gibt es solche Erscheinungen aber auch bei natürlichen Problemen, die auch sonst von Interesse sind? Für algorithmisch unlösbare Probleme ist das in der Tat so (vgl. Abschnitt 3.2.1 und 3.2.3). Was dagegen algorithmisch entscheidbare Probleme mit einer hohen unteren Zeitabschätzung für ihre Lösung betrifft (z. B. mit einer Zeitabschätzung des Typs $T_{\mathfrak{M}}(n) > 2^n$), so wurden erst vor relativ kurzer Zeit hinreichend „natürliche" Beispiele für solche Probleme gefunden. Leider können wir sie im Rahmen dieses Buches nicht darstellen.

Das Studium der Qualität von Algorithmen und Berechnungen ist ein relativ junges Gebiet der Algorithmentheorie, und es hat sich noch keine hinreichend vollständige Darstellung in der Literatur gefunden. Die im folgenden genannten Bücher systematisieren einen Teil des vorliegenden Materials. Die Bücher [9], [10] und [12] enthalten vor allem Resultate, die sich auf Kompliziertheitsabschätzungen von Berechnungen mit signalisierenden Funktionen beziehen (vgl. Abschnitt 3.4.1). Zu den auf eine Kompliziertheitsabschätzung der Beschreibung von Algorithmen abzielenden Untersuchungen (vgl. den Beginn von Abschnitt 3.4), die auf Arbeiten von A. A. Markov und A. N. Kolmogorov zurückgehen, kann man außer auf [11] fast nur auf Zeitschriftenartikel verweisen.

[10] Трахтенброт, Б. А., Сложность алгоритмов и вычислений, НГУ, Новосибирск 1967.

[11] Агафонов, В. Н., Сложность алфоритмов и вычислений II, НГУ, Новосибирск 1975.

[12] Böhling, K. H., und B. v. Braunmühl, Komplexität bei Turingmaschinen, Bibliographisches Institut, Mannheim—Wien—Zürich 1974.

III. Die Theorie der Programmierung für reale Rechenautomaten kann man durchaus als Anwendungsgebiet der gegenwärtigen Algorithmentheorie ansehen, in der viele Ideen aus ihren abstrakteren Gebieten ihre Verkörperung und weitere Entwicklung erfahren. So ist beispielsweise eines der zentralen Probleme der Theorie der Programmierung die Schaffung von Programmiersprachen höherer Stufe (z. B. ALGOL, FORTRAN, PL/1 u. a.) und von entsprechenden Compilern, d. h. von Übersetzungsprogrammen. Diese weitgehende Verallgemeinerung der Idee einer Programmiersprache und eines programmierenden Algorithmus (vgl. Abschnitt 2.4) ist die Grundlage für die Automatisierung der Programmierung. Der Programmierer schreibt nämlich einen Algorithmus in einer für ihn bequemen Sprache auf. Anschließend übersetzt die Maschine diese Niederschrift („geführt" durch den Compiler) in die Maschinensprache, d. h., sie erarbeitet ein gewöhnliches Programm aus Maschinenbefehlen (im Sinne von Abschnitt 1.6). Hierzu gibt es inzwischen umfassende Literatur.

IV. Neben den eindimensionalen v.-Neumann-Automaten kann man analog auch zwei- und höherdimensionale v.-Neumann-Automaten betrachten. Die Theorie dieser Automaten wurde durch J. v. Neumann nicht nur zum Studium ihrer rechnerischen Fähigkeiten entwickelt, sondern vor allem auch zur Modellierung des Prozesses der Selbstreproduktion. Es kann hier auf folgende Literatur verwiesen werden:

[13] Neumann, J. v., Theory of self-reproducing Automata, Univ. of Illinois Pr., Urbana—Chicago—London 1970.

[14] Codd, E. F., Cellular Automata, Academic Press, New York—London 1968.

[15] Burks, A. W. (ed.), Essays on Cellular Automata, Univ. of Illinois Pr., Urbana—Chicago—London 1970.

V. Zusammenfassungen historischen Charakters, einen Überblick über die Erfolge sowjetischer Mathematiker auf dem Gebiet der Algorithmentheorie und in benachbarten Gebieten sowie eine umfangreiche Bibliographie findet man in

[16] История отечественной математики, т. 4, кн. 2, (гл. V. VI), Науковая думка, Киев 1970.

Eine relativ allgemeinverständliche Darlegung einiger Fragen der Algorithmen- und Automatentheorie ist zu finden in

[17] Turing, A. M., Computing Machinery and Intelligence, Mind 59 (1950), 433—460.

[18] Minsky, M., Computation: Finite and Infinite Machines, Prentice Hall, Englewood Cliffs 1967.

VI. Im Zusammenhang mit den Sätzen über die algorithmische Unlösbarkeit bestimmter Aufgabenklassen bemerken wir, daß sie keinen Anlaß dafür geben, in Agnostizismus zu verfallen. Denn jeder dieser Sätze bezieht sich auf eine ganze Klasse von Aufgaben und konstatiert lediglich die Unlösbarkeit aller Aufgaben dieser Klasse nach einer einheitlichen effektiven Methode, eben einem Algorithmus. Das bedeutet ganz und gar nicht, daß es unter den Einzelaufgaben, die in der betreffenden Klasse zusammengefaßt sind, welche geben sollte, die prinzipiell unentscheidbar (grundsätzlich nicht lösbar) sind. Beispielsweise darf man den Satz aus Abschnitt 3.2.2 nicht so verstehen, daß eine Chiffre einer Turing-Maschine existiert, von der grundsätzlich nicht festgestellt werden kann, ob sie selbstanwendbar ist oder nicht. Er besagt nur, daß der betrachtete Typ von Aufgaben zu allgemein und zu weit ist, als daß ein einheitlicher Algorithmus zur Lösung aller dieser Aufgaben existieren kann. In jedem solchen Fall besteht das Ziel mathematischer Untersuchungen darin, immer umfassendere Algorithmen zu schaffen, die zur automatischen Lösung auch immer umfangreicherer Teilklassen der betrachteten Aufgabenklasse führen.

Die Sätze über die algorithmische Unlösbarkeit zeigen weiter, daß sich die Mathematik nicht auf die Konstruktion von Algorithmen reduzieren läßt, daß der Erkenntnisprozeß in der Mathematik nicht bis ins letzte automatisiert werden kann. Schon in relativ begrenzten Gebieten der Mathematik (wie z. B. in der Theorie der Gruppen mit endlich vielen Erzeugenden) treten Problemscharen auf, die nicht durch einen Rechenautomaten (genauer gesagt: von keiner Turing-Maschine) gelöst werden können.

VII. Trotzdem muß man feststellen, daß das Anwendungsgebiet für algorithmische Prozesse sehr weit ist und sich keineswegs nur in Berechnungsprozessen erschöpft, die wir in der Mathematik antreffen. Für viele Prozesse, die üblicherweise durchaus als sehr schwierig und kompliziert gelten, kann man Algorithmen konstruieren, die ihrer Idee nach hinreichend einfach sind. Die praktischen Schwierigkeiten, die bei der Realisierung dieser Algorithmen auftreten, sind damit verbunden, daß die nach dem Algorithmus ablaufenden Prozesse sehr lang sind, d. h. eine enorme Anzahl von Operationen erfordern (obwohl die Operationen selbst einfach sind). Diese Bemerkungen beziehen sich vor allem auf Spielprozesse (etwa speziell das Schachspiel), da hier der Erfolg sehr stark vom Durchdenken einer sehr großen Anzahl von Varianten bei der Auswahl einer optimalen Variante abhängt. Die Schaffung und weitere Vervollkommnung schnell arbeitender Rechenmaschinen erweiterte die Zahl der praktisch realisierbaren Algorithmen ganz bedeutend.

Der Satz über die Existenz beliebig komplizierter Probleme zeigt jedoch, daß es bei jedem beliebigen Niveau der Technik unter den algorithmisch lösbaren Problemen stets auch solche geben wird, für die es (bei dem gegebenen Niveau) praktisch realisierbare Algorithmen nicht gibt. Deshalb hat der „reine" Nachweis der algorithmischen Lösbarkeit eines gewissen Problems, d. h. ein reiner Existenzbeweis oder auch ein Existenzbeweis durch Angabe eines das Problem lösenden Algorithmus,

aus dem keine (hinreichend gute) Kompliziertheitsabschätzung abzuleiten ist, nur einen begrenzten praktischen Wert. Wir können sagen, indem wir einen Ausspruch von P. S. Novikov heranziehen, daß ein „reiner" Existenzbeweis für die algorithmische Lösbarkeit einer gewissen Aufgabenklasse nicht mehr bedeutet, als daß es nicht möglich ist, die algorithmische Unlösbarkeit für diese Klasse nachzuweisen.

Abschließend richten wir die Aufmerksamkeit nochmals darauf, daß jede realisierbare Rechenmaschine nur als eine gewisse Annäherung an den abstrakten Begriff der Turing-Maschine oder den v.-Neumann-Automaten angesehen werden kann. In realen Maschinen ist der Speicherumfang stets begrenzt, während in einem v.-Neumann-Automaten und in einer Turing-Maschine ein unendliches Band zur Verfügung steht; aber natürlich ist eine technische Realisierung eines unendlichen Bandes unmöglich. Dagegen liegt eine bedeutende Vergrößerung des Speicherumfangs im Vergleich zum bisher erreichten Niveau durchaus im Bereich des Möglichen und ist durchaus wünschenswert. Insbesondere hinsichtlich der Erweiterung des äußeren Speichers und der Erhöhung der Rechengeschwindigkeit kann man weitere große Erfolge bei der Entwicklung der Rechenautomaten erwarten. Neben dem technischen Fortschritt hat jedoch auch die reine mathematische Forschung einen entscheidenden Beitrag bei der Klärung der Fragen zu leisten, welche Maschinentypen und welche Algorithmentypen besonders für eine sehr schnelle Lösung von Aufgaben einer bestimmten Art geeignet sind. Es erscheint vor allem wichtig, die Typen von Aufgaben auszusondern, die eine starke Parallelisierung ihres Lösungsprozesses zulassen.

Namen- und Sachverzeichnis

Berichtigungen

Abb.	*Seite*	*Spalte*	*Reihe*	*statt*	*lies*
31	126	2	2	Lp_x	ΛLp_x
31	126	2	3	Lp_y	ΛLp_y
54	195	4	7	y	x
				R	R
55e)	197	8	—	$x(s-2)$	$x(s-2)$
				$x(s-1)$	$x(s-1)$
				$\varnothing$	$x(s)$