Werner

BASIC

für Mikrorechner

```
140 REM ---------------------- EIN
150 :
160 PRINT
170 INPUT "Ordnung des Polynoms'
180 DIM A(N)
190 :
200 PRINT "Eingabe der Koeffizie
210 FOR I=N TO 0 STEP -1
220   PRINT I ;
230   INPUT A(I)
240 NEXT I
250 :
260 PRINT "Wertebereich des Argu
270 INPUT "untere Grenze" ; U
```

VEB VERLAG TECHNIK BERLIN

BASIC für Mikrorechner

Programmentwicklung
Sprachelemente
Anwendungen

Dieter Werner

2., durchgesehene Auflage

VEB VERLAG TECHNIK BERLIN

Werner, Dieter :
BASIC für Mikrorechner : Programmentwicklung,
Sprachelemente, Anwendungen / Dieter Werner. –
2., durchges. Aufl. – Berlin : Verlag Technik,
1987. 240 S. : 70 Bilder, 22 Taf.
ISBN 3-341-00437-8

ISBN 3-341-00437-8

2., durchgesehene Auflage

Lizenz 201 · 370/234/87
Printed in the German Democratic Republic
Lichtsatz: (52) Nationales Druckhaus Berlin, Betrieb der VOB National
Druck und buchbinderische Verarbeitung: (140) Druckerei Neues Deutschland, Berlin
Lektor: *Karl Belter*
Schutzumschlag: *Kurt Beckert*
LSV 3054 · VT 3/5842-2
Bestellnummer: 553 842 9
02500

Vorwort

In vielen Bereichen begegnen wir heute elektronischen Taschenkalkulatoren, Mikroprozessoren, Mikrorechnern und Personalcomputern. Elektronische Rechenautomaten führen jedoch nur das aus, was ihnen in Form von Programmen eingegeben wurde. Manche Computer haben zwar bereits feste Programme; aber auch diese mußten ja einmal entworfen und implementiert werden. Der weiten Verbreitung von Mikrorechnern entsprechend müssen demnach sehr viele Programme geschrieben werden, und zwar auf den unterschiedlichsten Gebieten, und oft von Menschen, die das Programmieren nur als notwendiges Übel betrachten. Gefragt sind also vornehmlich solche Hilfsmittel, die wenig Aufwand erfordern.

In diesem Zusammenhang erfährt die Programmiersprache BASIC einen schnellen Aufschwung. Als Lehrmittel entworfen, ist diese Sprache einfach zu verstehen und schnell zu erlernen. Sicher hängt es von der Vorbildung ab, ob dafür bereits einige Tage ausreichen; länger als eine Woche benötigt man jedoch dafür kaum.

Die bei BASIC mögliche interpretative Arbeitsweise gestattet eine schnelle Programmentwicklung. Diese Sprache ist deshalb vor allem für Erprobungen und Experimente geeignet, die der Vorbereitung einer endgültigen Lösung dienen. Maschinenorientierte Ergänzungen erlauben sogar die Anwendung bei kleinen Steuerungs- und Regelungseinrichtungen. Der Einsatz des Bildschirms als Ausgabegerät ermöglicht – zusammen mit entsprechenden Sprachanweisungen – auch die Implementierung von Spielen.

Ein Lehrbuch über BASIC sollte einen breiten Kreis von Interessenten ansprechen. Das sind u. a. Studenten technischer und ökonomischer Fachrichtungen, Naturwissenschaftler, Ingenieure und Betriebswirtschaftler, außerdem Lehrer für Mathematik, Physik und Polytechnik an Oberschulen sowie mathematisch interessierte Oberschüler, die in Informatikzirkeln mitarbeiten. Schließlich sind aber auch die Besitzer von Heimcomputern zu beachten.

Diesem Personenkreis entsprechend wird in dem vorliegenden Buch besonderer Wert auf eine anschauliche Darstellungsform gelegt. Dabei wird versucht, zunächst recht elementar vorzugehen, wenig vorauszusetzen und viel zu erläutern. Gegen Ende des Buches werden dagegen höhere Forderungen an den Leser gestellt. In verschiedenen Fällen wird Anfängern geraten, kompliziertere Abschnitte erst beim nochmaligen Lesen des Buches durchzuarbeiten.

Am Anfang werden Aufbau und Wirkungsweise eines Computers erläutert. Dabei wird kein spezieller Rechner vorausgesetzt. Wenn es die Beispiele erfordern, wird auf den Mikroprozessor U880 (Z80) zurückgegriffen. Es schließen sich Ausführungen über Methoden des Entwurfs von Programmen sowie über Mittel zu ihrer Implementierung an. Der komplizierte Prozeß des Umsetzens von Aufgabenstellungen in Programme kann dabei allerdings nur umrissen werden. Diejenigen Leser, die hier bereits über ausreichende Kenntnisse verfügen, können diese einleitenden Abschnitte überschlagen.

Es folgt eine allgemeine Charakterisierung der Implementierung von Programmen mit Hilfe der Sprache BASIC. Schrittweise werden die verschiedenen erforderlichen Komponenten eingeführt: Ein- und Ausgaben, numerisches Rechnen, Aufbau von Steuerstrukturen, Definition von Unterprogrammen und Funktionen, Verarbeitung von Datenfeldern, Einsatz peripherer Zusatzspeicher und Nutzung hardwareorientierter Ergänzungen. Nicht behandelt werden Probleme der (echten) Bildschirmgrafik. Ursache für diese Entscheidung ist die Tatsache, daß Grafikanweisungen – soweit sie überhaupt verfügbar sind – allzu stark vom konkreten Computer abhängen. Statt dessen wird die Gestaltung von alphanumerischen Ausgaben ausführlich besprochen.

In jedem einzelnen Abschnitt wird so vorgegangen, daß zunächst die Problemstellung diskutiert wird. Dann werden die zu ihrer Lösung geeigneten Anweisungen und Funktionen von BASIC vorgestellt. Zur Erläuterung dienen Programmbeispiele, die in der Sprachvariante BASIC-80 kodiert und mit einem MBASIC-Interpreter übersetzt wurden. Schließlich wird auch noch auf weitere Varianten hingewiesen – bei den vielen Dialekten der Sprache eine unangenehme Notwendigkeit.

In einem abschließenden Abschnitt sind einige umfangreichere Beispiele zusammengestellt.

In allen Fällen wurde bei den Programmen das Schwergewicht auf die Verständlichkeit gelegt, nicht auf die Effizienz. Daher wurde eine übersichtliche Gliederung mit vielen Erläuterungen gewählt. Dem Leser sollen nur die Möglichkeiten demonstriert werden. Er kann diese Beispiele nach seinen Vorstellungen modifizieren oder auch kombinieren. Gerade bei BASIC ist das leicht möglich! Der Speicherbedarf der Programmbeispiele liegt in den meisten Fällen unter 1000 Byte und beträgt höchstens etwa 5000 Byte. Er wird zu einem beträchtlichen Teil durch die Kommentare verursacht. Außerdem hängt er bei Textverarbeitungsprogrammen von der konkreten Länge der verwendeten Zeichenketten ab, bei Dateiverarbeitungsprogrammen vom Umfang der Dateien. Daher werden in diesem Buch bewußt keine detaillierten Angaben über den benötigten Speicherplatz gebracht.

Mein ganz besonderer Dank gilt meiner Frau *Sigrid*, die an der Bearbeitung des Manuskripts einen wesentlichen Anteil hat. Danken möchte ich auch dem VEB Verlag Technik, vor allem dem zuständigen Lektor, Herrn *Karl Belter*, für die wertvollen Hinweise und die verständnisvolle Realisierung meiner Wünsche.

Dieter Werner

Inhaltsverzeichnis

1. Computer

Einen Computer kann man nur dann voll ausnutzen und zweckentsprechend einsetzen, wenn man in ausreichendem Maße mit seinem Aufbau und seiner Funktionsweise vertraut ist. Daher soll zunächst eine Übersichtsdarstellung der gerätetechnischen Komponenten (der sog. Hardware) eines elektronischen Rechenautomaten vermittelt werden.

- Als *Hardware* bezeichnet man in der Rechentechnik die physikalisch-technischen Bestandteile von Rechenautomaten einschließlich der erforderlichen peripheren Geräte.

Demjenigen Leser, dem die folgende Zusammenfassung zu kurz erscheint, wird als Ergänzung eines der einschlägigen Bücher empfohlen, die sich in stärkerem Maße elektronischen Problemen widmen [1.1] bis [1.5].

Wenn auch die Hardware die unumgängliche Basis jeder Verarbeitung von Informationen bildet, so ist doch die Bedeutung der im Rechenautomaten verfügbaren Programme noch höher einzuschätzen.

- Die *Software* umfaßt alle programmtechnischen Komponenten eines Computersystems, das sind die Programme selbst sowie deren Dokumentationen.

Man unterscheidet die universell verwendbare, automatenspezifische *Systemsoftware* (z. B. das Betriebssystem oder den BASIC-Interpreter) und die nutzerorientierte *Anwendungssoftware*. Erst die Software eröffnet Möglichkeiten für eine unerschöpfliche Vielfalt von Anwendungen. Dabei können manche Programme (meist der Systemsoftware) vom Herstellerbetrieb des Rechenautomaten unveränderlich in Festwertspeicherschaltkreisen eingeschrieben worden sein. In diesen Fällen verwendet man den Begriff *Firmware*.

1.1. Aufbau und Arbeitsweise eines Rechenautomaten

Der heutige Rechenautomat ist das Produkt einer jahrhundertelangen Entwicklung. Er vereint die Bestrebungen zur Mechanisierung des Rechnens mit den Bemühungen zur Automatisierung von häufig zu wiederholenden Prozessen. Diese Linien verbinden sich im heutigen Computer, einem zyklisch arbeitenden, programmgesteuerten, speicherorientierten, digitalen Rechenautomaten.

1.1.1. Historische Entwicklung der Rechentechnik

Die Bemühungen der Menschen, das Rechnen wenigstens zu mechanisieren, gehen weit in die Geschichte zurück. Das älteste der uns bekannten Hilfsmittel ist der Abakus. Er wurde schon um 1100 v. u. Z. in China verwendet und ist auch heute noch im Einsatz, mindestens als anschauliche Lernhilfe. Aber erst im 17. Jahrhundert entstanden die ersten Rechenmaschinen.

Im Jahre 1623 entwickelte *Wilhelm Schickardt*, Professor an der Universität Tübingen und Freund *Keplers*, eine mechanisch arbeitende Rechenmaschine. Auch der nächste, unabhängige Erfinder einer Additionsmaschine wurde durch seine Umgebung angeregt: *Blaise Pascal* war der Sohn eines Steuereinnehmers. Die erste der von ihm gebauten Maschinen stellte er 1643 fertig, eines der späteren Exemplare kann im Mathematisch-Physikalischen Salon des Zwingers in Dresden betrachtet werden. Die erste Rechenmaschine, die auch multiplizieren und dividieren konnte, wurde 1673 von dem Mathematiker und Philosophen *Gottfried Wilhelm Leibniz* der englischen Royal Society vorgestellt.

Nachdem die wesentlichsten Wirkprinzipien geklärt waren, wurden die weiteren Fortschritte durch die Verbesserung der feinmechanischen Präzisionsarbeit sowie durch die Entwicklung der Produktivkräfte in den frühkapitalistischen Manufakturen bestimmt. Am Ende dieses Weges stehen die noch bis in die fünfziger Jahre dieses Jahrhunderts bekannten mechanischen Vierspeziesrechenmaschinen mit Antrieb durch Elektromotoren.

Die andere Wurzel unserer heutigen Rechenautomaten ist die Automatisierungstechnik. Ein wesentlicher Schritt war die Einführung der Programmsteuerung durch den Seidenweber *Joseph-Marie Jacquard* aus Lyon um 1804. Er steuerte seine Webstühle durch eine Kette von Lochkarten, in denen die gewünschten Muster kodiert waren.

Durch diese Erfindung angeregt, entwarf der an Rechenmaschinen interessierte Engländer *Charles Babbage* um 1840 den ersten programmgesteuerten Universalrechenautomaten, dessen Struktur auch heute noch aktuell ist. Die rein mechanische Realisierung dieser genialen Konzeption war aber damals unmöglich, und so gerieten seine Überlegungen wieder in Vergessenheit.

Erst hundert Jahre später war die Zeit für diese Ideen reif. Der deutsche Ingenieur *Konrad Zuse* hatte bereits 1937 die binäre Darstellung von Informationen in Rechenmaschinen vorgeschlagen. 1941 stellte er den ersten Rechenautomaten der Welt fertig. Er wurde über einen Lochstreifen gesteuert und arbeitete mit elektromechanischen Relais. Seine Struktur entsprach den Vorschlägen von *Babbage*; *Zuse* hatte sie neu erfunden. Dieser Rechenautomat wurde kurz danach bei einem Luftangriff zerstört.

Parallel dazu beschäftigte man sich in den USA mit einem relaisgesteuerten Rechenautomaten, der ab 1944 einsetzbar war. Der erste Rechner auf der Basis von Elektronenröhren, ENIAC genannt, konnte 1945 in Betrieb genommen werden, und zwar ebenfalls in den USA. Beide Rechenautomaten zeigten aber konzeptionelle Mängel in der logischen Struktur und der Ablaufsteuerung.

1.1.2. Aufbau eines Rechenautomaten

In Unkenntnis der Arbeiten von *Babbage* und *Zuse* entwarf der aus Ungarn stammende Mathematiker *John von Neumann* nach einer Besichtigung der Anlage ENIAC 1945 erneut das Konzept des programmgesteuerten Rechenautomaten, das seitdem mit seinem Namen verbunden ist. Es bildet die logische Grundlage fast aller heute arbeitenden Computer.

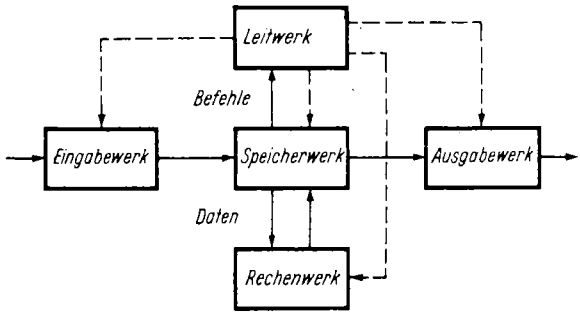


Bild 1.1. Struktur eines Rechenautomaten nach *John von Neumann*

Die Struktur **(Bild 1.1)** entspricht durchaus derjenigen, die von *Babbage* vorgeschlagen bzw. von *Zuse* angewandt wurde. Neu ist das Prinzip der *Programmsteuerung*. Während die bisherigen Rechenautomaten das Speicherwerk nur für Verarbeitungsdaten benutzten und das Programm während der Arbeit fortlaufend über Lochkarten oder Lochband eingegeben werden mußte, las es *John von Neumann* vor Beginn der Arbeit mit Hilfe des *Eingabewerks* ein und legte es zusammen mit den Daten im Speicher ab. Der Inhalt einer Speicherzelle kann also ein Befehl oder ein Datenwort sein!

Durch diesen Schritt konnte zunächst die Arbeitsgeschwindigkeit beträchtlich gesteigert werden, weil die benötigten Befehle bereits innerhalb des Rechners vorliegen. Insbesondere aber wurde es möglich, von einer sequentiellen Ausführung abzuweichen und zum Beispiel Wiederholungen (Schleifen) auszuführen.

Das *Leitwerk* interpretiert einen gegebenen Speicherinhalt als auszuführenden *Befehl* und veranlaßt die entsprechenden Operationen des Rechenautomaten.

Das *Rechenwerk* dient dazu, die in Speicherzellen abgelegten Daten miteinander zu verknüpfen. Die dabei erhaltenen Resultate werden wieder im Speicher abgelegt, wobei dieselben Zellen durchaus mehrmals benutzt werden können. Deren Inhalt wird also laufend verändert. Man spricht dann davon, daß in dieser Zelle eine *Variable* gespeichert sei.

- Unter *Daten* werden an dieser Stelle Informationen verstanden, die für eine Verarbeitung durch einen Rechenautomaten bestimmt bzw. durch eine solche Verarbeitung entstanden sind.

Am Ende der Arbeit werden die gewünschten Resultate mit Hilfe des *Ausgabewerks* aus dem Rechenautomaten herausgebracht. In der Regel handelt es sich dabei nur um Daten. Das benutzte Programm verbleibt zunächst im Rechner und wird bei der Eingabe des nächsten Programms und dessen Daten zerstört.

1.1.3. Arbeitsweise eines Rechenautomaten

Bisher wurde nur die Ausführung einer einzelnen Anweisung behandelt. Wie kommt man nun zu einer automatischen Arbeit des Computers? Der Rechner muß dazu stets wissen, welcher Befehl als nächster an der Reihe ist. Für diesen Zweck ist im Leitwerk eine besondere Merkzelle vorgesehen. Solche Speicherelemente, die nicht im Speicherwerk, sondern im Leitwerk oder im Rechenwerk vorhanden sind, heißen übrigens *Register*. Dasjenige Register im Leitwerk, das die Speicheradresse des nächsten auszuführenden Befehls enthält, das also auf diejenige Zelle des Speicherwerks zeigt, wo das Programm fortzusetzen ist, heißt *Befehlszähler* **(Bild 1.2)**.

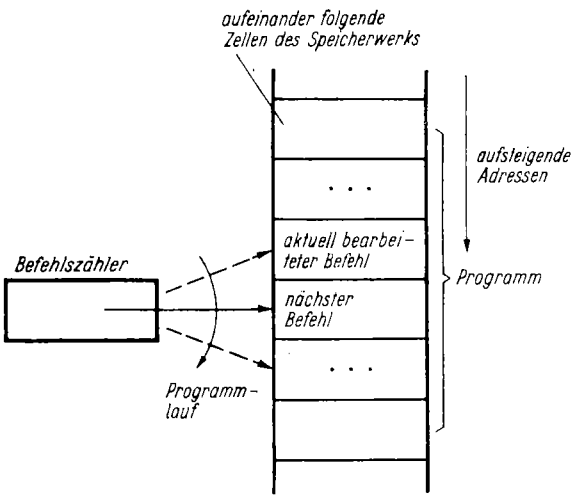


Bild 1.2. Funktionsweise des Befehlszählers

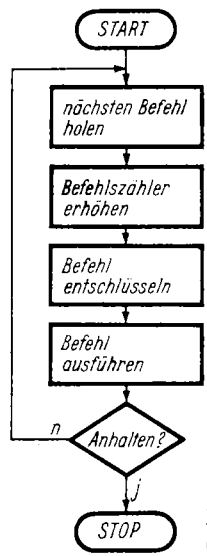


Bild 1.3. Befehlsschleife eines Rechenautomaten

Der Computer ist ein zyklisch arbeitender Automat; er arbeitet periodisch die sog. *Befehlsschleife* ab **(Bild 1.3)**. Dabei liest das Leitwerk zunächst den Inhalt derjenigen Speicherzelle, auf die der Befehlszähler hinweist, und merkt sich den betreffenden Befehl. Dann folgt der wesentliche, entscheidende Schritt: Der Inhalt des Befehlszählers wird um den Wert eins erhöht, er zeigt

also jetzt auf die Speicherzelle mit der nächsthöheren Adresse. Die sequentielle Eingabe des Programms durch Lochkarten (bei *Babbage*) oder Lochband (bei *Zuse*) wurde also bei *John von Neumann* durch das *sequentielle Lesen im Speicher* ersetzt. Anschließend entschlüsselt das Leitwerk den übernommenen Befehl und veranlaßt die Ausführung der entsprechenden Operation durch die betreffenden Teilwerke des Computers. Dabei sind weitere Veränderungen des Befehlszählers durchaus erforderlich und möglich, zum Beispiel bei Befehlen, die sich über mehrere Speicherzellen erstrecken, oder beim programmierten Verlassen der sequentiellen Abarbeitung durch sog. *Sprungbefehle*.

Hat der Rechenautomat einen Befehl befolgt, kehrt er an den Anfang der Befehlsschleife zurück und beschäftigt sich mit der Erfüllung des nächsten. Einmal gestartet, läuft er also *ununterbrochen!* Alle weiteren Möglichkeiten – zum Beispiel auch das Anhalten – liegen allein in dem ihm übergebenen Programm:

- Ein *Programm* ist eine strukturierte Menge von *Anweisungen* (Befehlen). Es beschreibt eine Arbeitsvorschrift (einen *Algorithmus*) in einer dem Computer unmittelbar verständlichen oder mittelbar verständlich zu machenden Form.

1.1.4. Informationsdarstellung in einem Rechenautomaten

Außerhalb des Computers sind die Informationen meist in einer für den Menschen lesbaren Form vorhanden: als Ziffern(-zeichen), als Buchstaben(-zeichen) und als Sonderzeichen, wie unter anderen Komma und Punkt. Werden diese *Zeichen* aneinandergereiht, so entstehen Zeichenketten, auch *Texte* genannt. Sie sind für den Menschen verständlich, für den Rechenautomaten aber nicht. Sie müssen, spätestens durch das Eingabewerk, in eine maschinenlesbare Form gebracht werden. Bei den mechanischen Dezimalrechenmaschinen geschah das durch eine geeignete Einstellung von Ziffernrädchen. Elektronische Computer arbeiten aber mit einer anderen Art der Informationsverschlüsselung.

Es wurde bereits erwähnt, daß durch *Zuse* eine *duale Darstellung* der Informationen – manchmal auch *binär* genannt – eingeführt wurde. Das bedeutet, daß jedes Informationselement – allgemein als Bit bezeichnet – nur zwei Zustände annehmen kann.

- Die Informationseinheit bei der dualen Darstellung ist das *Bit* mit dem Wertevorrat 0 und 1.

Es ist daher erforderlich, daß jedes Textzeichen in eine solche binäre Form transformiert wird, bevor es im Rechner gespeichert werden kann. Diese Umkodierung erfolgt beispielsweise durch die Eingabetastatur des Computers. Dabei reicht natürlich ein Bit nicht aus, um alle üblichen Textzeichen kodieren zu können. Häufig faßt man daher 8 Bit zu einer größeren Einheit zusammen und bezeichnet diese als Byte.

- Ein *Byte* ist ein Informationselement mit einem Umfang von 8 Bit.

Für die Darstellung von Textzeichen als Bitmuster von der Größe eines Bytes wird heute überwiegend eine international anerkannte Vorschrift verwendet, die vor allem unter dem Namen *ASCII-Kode* bekannt ist (Tafel A.1).

Nach der Eingabe eines Zeichens in den Rechenautomaten wird das entsprechende Bitmuster im Speicherwerk abgelegt, dessen Zellen häufig gerade die Größe eines Bytes haben. Damit ist aber die Information noch nicht in die endgültige Form gebracht. Weitere Umwandlungen schließen sich an, die vom Charakter, vom Inhalt der Informationen abhängen. Sie werden durch Software oder Firmware realisiert und bringen die Information in eine für die weitere Arbeit geeignete, *interne Darstellung*.

Wie bereits gesagt, können die in einer Speicherzelle abgelegten Bitmuster nun ganz verschieden gedeutet werden:

- Das Leitwerk interpretiert einen gegebenen Speicherinhalt als *Befehl* und veranlaßt die entsprechenden Aktionen des Automaten. So würden z. B. die Mikroprozessoren 8080, Z80 und U880 nach der Dekodierung des im **Bild 1.4** gezeigten Bitmusters den Inhalt einer (durch weitere Bedingungen ausgewählten) Speicherzelle um den Wert eins erniedrigen (Befehl:

DEC (HL)). Derart benutzte Bitmuster werden als (interner) *Maschinenkode* bezeichnet. Die Menge aller Befehle, die das Leitwerk eines Computers entschlüsseln kann, heißt *Befehlsliste.*

- Die für *Daten* benutzten Bitmuster sehen aber nicht anders aus als ein Maschinenkode. Ihre unterschiedliche Bedeutung erhalten sie zunächst einmal dadurch, daß sie in das Rechenwerk transportiert und von diesem interpretiert werden. Dabei kann das Rechenwerk aber noch ganz verschieden vorgehen, je nachdem, welche spezielle Operation vom Leitwerk angewiesen wurde. Wird beispielsweise das im Bild 1.4 angegebene Bitmuster im Rahmen einer Textverarbeitung als Zeichen betrachtet, so stellt es die Ziffer "5" dar. Ist dagegen eine numerische Operation, zum Beispiel eine Addition ganzer Zahlen, durchzuführen, so repräsentiert das diskutierte Bitmuster von Bild 1.4 die binär kodierte Dezimalzahl 53.

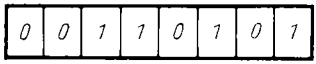


Bild 1.4. Beispiel eines Bitmusters von der Größe eines Bytes

Am Ende werden die berechneten Resultate aus der internen Darstellung wieder in den Textkode transformiert und dem Ausgabewerk übergeben, das es dann in einen für den Menschen lesbaren Klartext verwandelt. Auch die grafische Ausgabe auf einem Bildschirm ist möglich.

Glücklicherweise muß der Nutzer einer höheren Programmiersprache, wie z. B. BASIC, die verschiedenen Darstellungsformen nicht im Detail kennen. Er muß auch die entsprechenden Kodeumwandlungen nicht selbst durchführen. Bei Kenntnis der Zusammenhänge kann er aber mehr aus seinem Computer herausholen. Daher wurden hier einige Bemerkungen zu dieser Thematik gebracht. Weitere Ausführungen folgen im Abschnitt 10.1.

1.2. Mikrorechner

Der erste elektronische Rechenautomat, die Anlage ENIAC, füllte einen Saal, hatte eine Masse von 30 Tonnen und verbrauchte 200 Kilowatt. 40 Jahre später weist ein im Vergleich dazu weit leistungsfähigerer Computer, der die Sprache BASIC unmittelbar versteht, Abmessungen von 100 mm × 190 mm × 25 mm auf, wiegt 340 Gramm und benötigt zur Stromversorgung nur 4 kleine 1,5-V-Batterien (Gnomzellen). Zwischen diesen beiden Eckpunkten liegt die rasante Entwicklung der Halbleiterelektronik mit der Massenproduktion von Mikroprozessoren.

1.2.1. Entwicklung der Mikroelektronik

Etwa zur gleichen Zeit, als man die ersten elektronischen Rechenmaschinen fertigstellte, wurde ein Schaltelement erfunden, das die Rechentechnik revolutionieren sollte: der *Transistor.* Sein Einsatz führte in den fünfziger Jahren einerseits zu einer Verringerung der Abmessungen und des Leistungsbedarfs, andererseits aber auch zu einer Erhöhung der Leistungsfähigkeit.

Der nächste Sprung wurde etwa um 1960 herum vorbereitet, als man die Möglichkeit der Realisierung mehrerer Transistoren auf einem einzigen Halbleiterträger entdeckte. Die Anzahl von solchen aktiven Schaltelementen, die auf einem Siliziumchip von einigen zehn Quadratmillimetern untergebracht werden konnten, wurde jeweils im Zeitraum von etwa vier Jahren um den Faktor zehn gesteigert. Damit entstand aber auch das Problem, welche Funktionen man in einem Schaltkreis vereinigen sollte. Schließlich mußte eine hinreichend hohe Anzahl davon verkauft werden, sollten sich die Entwicklungskosten amortisieren! Schaltkreise für Taschenkalkulatoren z. B. sicherten für längere Zeit einen guten Absatz, aber der ständig steigende Integrationsgrad gestattete weit mehr. Man konnte ferner eine Reihe gleichartiger Elemente auf dem Chip unterbringen. Die begrenzte Anzahl von möglichen Anschlüssen setzte jedoch hier bald eine Grenze. Eine Ausnahme bilden lediglich die Speicherschaltkreise, die auch bei großem Umfang nur wenige Anschlüsse erfordern. Hier ist die Entwicklung inzwischen bis zu einer Kapazität von einigen Millionen Bits je Schaltkreis angelangt.

1.2.2. Mikroprozessoren

Eine andere Richtung von Überlegungen zielte darauf ab, das Stückzahlproblem durch die Entwicklung solcher Schaltkreise zu lösen, die man durch eine Programmierung an unterschiedliche Aufgaben anpassen kann. 1971 schlug die Geburtsstunde des *Mikroprozessors*. Rund 1000 Transistoren konnte man damals auf einem Chip unterbringen und realisierte damit Leitwerk und Rechenwerk eines sog. *Mikrocomputers*. Diese Komponenten wurden durch einige besondere Speicherzellen (Register) für das vorübergehende Ablegen von Operanden und Zwischenresultaten sowie von Adreßhinweisen auf das Speicherwerk ergänzt. Moderne Mikroprozessoren weisen weitaus mehr als zehnmal so viele Transistoren auf und haben die Leistungsparameter ehemals bekannter Großrechner.

1.2.3. Halbleiterspeicher

Das Speicherwerk eines Computers wird meist *Arbeitsspeicher* oder *Hauptspeicher* genannt. Das dient zur Unterscheidung von den externen Zusatzspeichern (von Geräten des Eingabe/Ausgabe-Werks, die noch zu behandeln sind [Abschn. 1.5]). Der Hauptspeicher von Mikrorechnern wird mit Hilfe hochintegrierter Halbleiterschaltkreise aufgebaut, die jeweils eine Kapazität von mehreren tausend Bytes haben. Dabei unterscheidet man zwei Arten:

- *Festwertspeicherschaltkreise*, meist kurz ROM *(Read Only Memory)* genannt, enthalten die vom Hersteller mitgelieferte Firmware, z. B. das Betriebssystem und den BASIC-Interpreter. Es gibt aber auch Schaltkreise, die der Anwender selbst programmieren und wieder löschen kann. Sie heißen EPROM *(Erasable Programmable Read Only Memory)*.
- Für die Erprobung von Programmen, für auswechselbare Software, für das Zwischenspeichern von variablen Werten wird eine andere Art von Speicherschaltkreisen benötigt, nämlich *Schreib-Lese-Speicher*, meist als RAM *(Random Access Memory)* bezeichnet. Diese haben aber die nachteilige Eigenschaft, daß die eingespeicherten Informationen beim Abschalten der Stromversorgung verlorengehen.

Auf dieser Basis lassen sich Speicherwerke verschiedener Größe aufbauen. Minimal sind einige zehntausend Bytes bereits bei Heimcomputern üblich; größere Personalcomputer weisen einen Umfang bis zu einigen Millionen Bytes auf.

1.2.4. Eingabe/Ausgabe-Schaltkreise

Für die flexible Gestaltung des Eingabe/Ausgabe-Werks von Mikrorechnern wurden ebenfalls hochintegrierte Halbleiterschaltkreise eingeführt. Sie lassen sich mit Hilfe von Steuerbefehlen an unterschiedliche Einsatzfälle anpassen und gestatten den Anschluß der verschiedenartigsten Eingabe/Ausgabe-Geräte. Sie dienen gewissermaßen als Tore des Mikrorechners in die Außenwelt. Sie haben, übrigens in ähnlicher Weise wie die Speicherzellen, Adressen (Nummern), über die sie vom Mikroprozessor her erreicht werden. Während man aber einen in einer Speicherzelle abgelegten Wert später unverändert wieder einlesen kann, wird ein zu einem *Ausgabetor* übertragenes Bitmuster dem angeschlossenen Ausgabegerät – z. B. einem Drucker – angeboten. Entsprechend bemüht sich ein vom Mikroprozessor angesprochenes *Eingabetor*, von dem zugeordneten Eingabegerät – beispielsweise der Tastatur – ein Bitmuster zu erhalten.

In beiden Fällen ist es möglich, alle Bits eines Bitmusters gleichzeitig, das heißt parallel, zu übertragen. Ein solcher Schaltkreis wird auch PIO *(Parallel Input/Output)* genannt. Ist dagegen nur eine einfache Zweidrahtleitung zwischen Mikrorechner und peripherem Gerät vorhanden – z. B. bei einer Fernschreibmaschine –, so müssen die Bits nacheinander übertragen werden. Ein entsprechender Schaltkreis heißt SIO *(Serial Input/Output)*.

Es gibt noch weitere hochintegrierte Halbleiterschaltkreise, die für die Eingabe von Signalen aus der Umgebung eingesetzt werden. Hierzu gehören solche, die man für das Zählen von Impulsen verwendet. Werden sie mit einem periodischen Taktsignal versorgt, können sie als Uhr dienen. Man nennt sie auch kurz CTC *(Counter/Timer Circuit)*.

1.3. Eingabe/Ausgabe-Geräte

Soweit Mikrorechner nicht fest als steuernder Kern in (elektronischen) Geräten und Anlagen eingebettet sind, werden sie meist im Dialogbetrieb genutzt. Das ursprüngliche Eingabe/Ausgabe-Gerät für diesen Zweck war die Fernschreibmaschine. Heute werden i. allg. Tastatur und Bildschirmgerät verwendet. Dadurch wurde ein anderes Gerät für die Ausgabe von Texten auf Papier *(Hardcopy)* erforderlich. Hierfür hat sich der serielle Drucker eingeführt.

1.3.1. Tastatur

Tastaturen von modernen Mikrorechnern enthalten zunächst die von Schreibmaschinen her bekannten Zeichen: Groß- und Kleinbuchstaben, Ziffern und Sonderzeichen. Hinzu kommen einige in der Informationsverarbeitung gebräuchliche Sonderzeichen, z. B. runde und eckige Klammern. Bei jedem Tastendruck erzeugt die Tastatur ein Bitmuster entsprechend einem *Textkode* (ASCII-Kode, siehe Tafel A.1) und wendet sich an den zugeordneten Eingabeschaltkreis zur Übernahme in den Eingabepuffer im Rechner.

Weiterhin gibt es eine Reihe spezieller *Funktionstasten*. Ein Beispiel hierfür ist die Taste „Ende der Eingabezeile". Durch das entsprechende Bitmuster teilt der Bediener dem Mikrorechner mit, daß eine Eingabezeile richtig erstellt wurde und aus dem Eingabepuffer zur Bearbeitung entnommen werden kann.

Allerdings steht meist aus räumlichen Gründen nicht für jedes dieser *Steuerzeichen* eine besondere Taste zur Verfügung. Dann muß man gleichzeitig zwei Tasten drücken: eine spezielle Umschalttaste CTRL *(Control)* und eine geeignete der Buchstabentasten. Durch die Wirkung der CTRL-Taste werden dabei die linken 3 Bit abgeschnitten und durch Nullen ersetzt. Die Steuerzeichen des ASCII-Kodes sind mit Erläuterungen in Tafel A.2 zusammengestellt.

1.3.2. Bildschirmgerät

Als bequemes, leises und papiersparendes Ausgabegerät für Mikrorechner wird allgemein der Bildschirm *(Display)* verwendet, sei es ein spezielles Datensichtgerät oder – wie beim Heimcomputer – das Fernsehgerät. Auf dem Bildschirm werden meist 16 ... 24 Zeilen mit je 64 ... 80 Zeichen untergebracht. Da das Schirmbild ständig wiederholt geschrieben werden muß, stehen die auszugebenden Informationen in einem sog. *Bildwiederholspeicher* bereit. Für jedes zu schreibende Zeichen ist dort ein Bitmuster entsprechend dem ASCII-Textkode vorhanden. Bei Farbdisplays sind zusätzliche Angaben für die Farbe des Zeichens (z. B. 4 Bit) und die Farbe des Hintergrunds (z. B. 3 Bit) erforderlich. Ein *Zeichengenerator* wertet die Bitmuster im Bildwiederholspeicher aus und steuert die zeilenweise Bewegung des Elektronenstrahls in entsprechender Weise:

- Ist nur die Wiedergabe von Buchstaben, Ziffern und Sonderzeichen nach einem 7-Bit-Textkode möglich, heißt das Bildschirmgerät *alphanumerisch*.
- Läßt der Zeichengenerator die Nutzung von allen 8 Bit eines Bytes zu, so lassen sich 128 weitere, nicht aus der lateinischen Schrift entnommene Zeichensymbole kodieren. Sie unterstützen die grafische Gestaltung von Schirmbildern. Man spricht in diesem Fall auch von pseudografischen Displays.
- *Rasterdisplays* arbeiten ebenfalls mit einem – wie beim Fernsehgerät – zeilenweise geführten Elektronenstrahl. Sie ordnen aber jedem einzelnen Bildpunkt auf einer solchen Zeile jeweils ein Bit im Bildwiederholspeicher zu, beim Farbrasterdisplay entsprechend mehrere. Dadurch entstehen größere Möglichkeiten zur grafischen Ausgabe. Oft kann man zusätzlich noch einen Zeichengenerator nutzen.
- *Grafische* Bildschirmgeräte erlauben es, den Elektronenstrahl entsprechend den Konturen der zu zeichnenden Strichdarstellungen zu führen. Es handelt sich bei diesen sog. *Vektordisplays* aber um recht kostspielige Geräte.

Der Umfang des Bildwiederholspeichers hängt vom Darstellungsprinzip ab. Ein alphanumerisches Display mit 16 × 24 Zeichen benötigt 1024 Byte, mit 24 × 80 Zeichen 1920 Byte. Für ein Rasterdisplay mit 256 × 320 Punkten sind dagegen bereits 10240 Byte erforderlich. Bei Farbdisplays ist der Speicherbedarf zwei- bis viermal so groß. Dieser Bildwiederholspeicher ist oft ein Teil des Hauptspeichers des Computers und dadurch vom Mikroprozessor unmittelbar zu erreichen.

Bei alphanumerischen Bildschirmen wird häufig eine sog. *Schreibmarke (Cursor)* benutzt, ein dicker Unterstreichungsstrich, der manchmal noch blinkt. Er wird (durch einen entsprechenden Ausgabebefehl) immer auf diejenige Position des Bildschirms gebracht, auf die die nächste Ausgabe eines Zeichens erfolgen soll.

Für das Beschreiben des Bildschirms sind drei Formen gebräuchlich:

- In der ältesten, noch stark an die Nutzung der Fernschreibmaschine erinnernden Form füllt man den Bildschirm zunächst Zeile für Zeile. Ist das Ende der letzten Zeile erreicht, rücken alle bisher geschriebenen – durch das Ausgabeprogramm gesteuert – um eine Zeile nach oben. Die oberste Zeile geht verloren, und die Schreibmarke steht auf dem Anfang der untersten. Man spricht dann davon, daß der Bildschirm *rollt*. Bei dieser Form schreibt der Bediener also stets auf die unterste Zeile und sieht den letzten Teil der Unterhaltung mit dem Rechner noch auf dem Bildschirm.
  Bei der Ausgabe größerer Texte durch den Rechner flimmert es aber dabei vor den Augen, lesen kann man nichts. Häufig sind die Ausgabeprogramme im Mikrorechner daher so aufgebaut, daß der Bediener am Ende jeder Zeile gefragt wird, ob gewartet werden soll. Für das Anhalten und das Fortsetzen der Ausgabe werden Steuerzeichen eingesetzt.
- Eine andere Form der Bildschirmnutzung, die bei der Ausgabe größerer Texte angewendet wird, ist das sog. *Blättern*. Hier wird genau ein Bildschirminhalt ausgeschrieben und dann gewartet. Nach der Eingabe irgendeines Zeichens wird die nächste Seite dargestellt. Um den Anschluß an die vorhergehende zu gewährleisten, beginnt die neue Seite oft mit der letzten Zeile der alten.
- Für die grafische Gestaltung von Schirmbildern ist eine dritte Form der Ausgabe von Interesse, bei der man das Zeichen auf eine direkt angegebene Position des Bildschirms plazieren kann. Hierauf wird im Abschnitt 10.3.2 näher eingegangen.

1.3.3. Drucker

Das Bildschirmgerät vermittelt nur ein flüchtiges Abbild des Dialogs zwischen Bediener und Computer. Benötigt man bleibende Unterlagen der Arbeit, Resultate, Listen, Protokolle, so muß zusätzlich ein Drucker vorhanden sein. Dem billigen Mikrorechner angemessen ist der einfache *Seriendrucker*. Er gibt die auf einer Zeile stehenden Zeichen nacheinander aus, und zwar teilweise sowohl vorwärts als auch rückwärts druckend. Es werden verschiedene Prinzipien angewandt, wobei in allen Fällen die Druckvorrichtung vor dem Papier bewegt wird:

- Der Schreibmaschine verwandt sind die *Typenrad-* und die *Kugelkopfdrucker*, die jeweils vollständige Zeichen auf das Papier bringen.
- *Mosaik-* bzw. *Matrixdrucker* haben dünne Stifte, die während der Bewegung des Druckkopfes angesteuert werden. Dadurch wird das Zeichen aus einem Mosaik feiner Punkte zusammengesetzt, z. B. aus 7 × 5 Punkten.
- Während die bisher genannten Drucker mit Hilfe eines Farbbandes auf normales Papier schreiben, benötigt der *Thermodrucker* ein temperaturempfindliches Papier. Er hat ein Mosaik kleinster elektrischer Widerstände. Werden sie erhitzt, so verfärbt sich das Papier, und es entsteht – wie bereits beschrieben – ein gerastertes Zeichen.
- *Pseudografische* Mosaikdrucker gestatten es, zusätzlich zu Buchstaben, Ziffern und Sonderzeichen noch weitere Symbole auszugeben. Mit ihnen lassen sich deshalb einfache grafische Darstellungen gestalten.

1.4. Externe Zusatzspeicher

Nach den bisherigen Darlegungen bestünde das Eingabewerk eines für wissenschaftlich-technische oder ökonomische Rechnungen benutzten Mikrocomputers nur aus einer Tastatur; das Ausgabewerk umfaßte außer dem Bildschirmgerät bestenfalls noch einen Drucker. Das reicht für die praktische Arbeit aber nicht aus. Beispielsweise ginge das benutzte, spezifische Anwendungsprogramm beim Abschalten der Netzspannung wieder verloren und müßte später wieder neu eingegeben werden. So entsteht der Wunsch, Programme aus dem Hauptspeicher des Mikrorechners heraus in einen Speicher zu bringen, der die Informationen nicht verliert, sie dort zeitweilig aufzuheben und bei Bedarf wieder in den Computer einzulesen. Dazu dienen Eingabe/Ausgabe-Geräte, die mit maschinenlesbaren, häufig sogar auswechselbaren und transportablen *Speichermedien* arbeiten.

Klassisches externes Speichermedium für Mikrorechner ist das *Lochband.* Heute ist es weitgehend durch Magnetbandkassette und Diskette abgelöst worden. **Tafel 1.1** gibt eine Übersicht über charakteristische Parameter. Die verschiedenen Speichermedien unterscheiden sich im wesentlichen in der *Zugriffszeit,* das ist der Zeitaufwand für das Auffinden und Lesen bzw. das Schreiben einer gewünschten Information. Im Abschnitt 9 wird die Nutzung von externen Zusatzspeichern für das Ablegen von Programmen (in *Bibliotheken*) und von Daten (in *Dateien*) ausführlicher behandelt.

Tafel 1.1. Externe Zusatzspeicher für Mikrorechner

Speichermedium	Kapazität in KByte	Zugriffszeit
Lochband	... 100	... 10 min
Magnetbandkassette	250	... 5 min
Diskette	100 ... 700	0,2 ... 0,5 s

1.4.1. Magnetbandkassetten

Die bekannten Magnetbandkompaktkassetten haben ähnliche technische Parameter wie das Lochband. Sie sind zwar wesentlich teurer, lassen sich aber löschen und wiederverwenden. Ihr Hauptvorteil besteht darin, daß ihre Nutzung vom Mikrorechner her weitgehend gesteuert werden kann, z. B. beim Suchen bestimmter Informationen. Sie lassen sich daher bequemer handhaben.

Als Eingabe/Ausgabe-Geräte dienen *Digitalkassettenlaufwerke,* komplizierte und teure elektromechanische Einrichtungen. Sie zeichnen die Informationen in geeigneter Form binär verschlüsselt auf dem Magnetband auf, und zwar in Form von *Datenblöcken.* Zusätzliche Prüfinformationen gestatten es, Informationsverluste aufzudecken.

Insbesondere Heimcomputer sehen auch den Einsatz von *Kassettenrecordern* vor. Diese haben allerdings häufig den Nachteil, daß sie nicht vom Mikrorechner her gesteuert werden können. Der Bediener muß also zunächst durch Tastendruck den Recorder starten; dann erst kann er dem Computer den Befehl zum Schreiben oder Lesen geben. Am Ende der Speicheroperation muß er den Recorder von Hand stoppen und die Kassette zurückspulen. Die Aufzeichnung kann, wie bereits erwähnt, unmittelbar binär erfolgen. Es ist aber auch – dem normalen Einsatz des Recorders entsprechend – weit verbreitet, den Bitwerten 0 und 1 zwei unterschiedliche Tonfrequenzen zuzuordnen und diese aufzunehmen.

1.4.2. Disketten

Lochband und Magnetbandkassette sind sog. sequentielle Speichermedien. Auf ihnen stehen die Informationen hintereinander aufgezeichnet. Sucht man eine bestimmte von ihnen, muß im

Regelfall das Medium von Anfang an durchgemustert werden, bis man das Gewünschte gefunden hat. Disketten gehören zu den *Direktzugriffsspeichern.* Bei ihnen läßt sich der Beginn einer gesuchten Information aus einem ebenfalls abgespeicherten Verzeichnis entnehmen. Der Zugriff erfolgt dann unmittelbar zu dieser Position. Dadurch kann das Lesen wesentlich rascher erfolgen.

Disketten sind flexible Polyesterscheiben, die mit einer magnetischen Schicht überzogen wurden. Ihr Durchmesser beträgt 130 mm *(Minidisketten)* oder 200 mm *(Standarddisketten).* Solche Magnetplatten rotieren mit 5 oder 6 Umdrehungen je Sekunde in einer Schutzhülle, die Aussparungen für den Antrieb und den Schreib-Lese-Kopf hat. Die erforderlichen Laufwerke sind relativ einfach aufgebaut. Die Informationen werden in 36 bis 80 konzentrischen *Spuren* unmittelbar binär verschlüsselt aufgezeichnet. Jede Spur enthält eine Anzahl von gleich großen Datenblöcken *(Sektoren),* z. B. 26 × 128 Byte. Sie werden durch Adreßangaben (Spurnummer, Sektornummer) und Prüfinformationen ergänzt. Zum Schreiben bzw. Lesen wird ein Magnetkopf zunächst über der gewünschten Spur positioniert und dann direkt auf die Oberfläche aufgesetzt. Disketten sind sowohl wegen ihres einfachen Aufbaus und ihrer Robustheit als auch wegen der unkomplizierten Laufwerke und des möglichen Direktzugriffs das empfehlenswerteste Speichermedium für Mikrorechner.

1.5. Betriebssystem

Eine entscheidende Rolle beim Einsatz eines Rechenautomaten spielt das Betriebssystem.

- Als *Betriebssystem* bezeichnet man ein vom Hersteller des Computers mitgeliefertes Programmsystem, das sich zwischen den Nutzer mit seiner Anwendungssoftware und die Hardware des Rechners schiebt. Es steuert die Abläufe bei der Bearbeitung von Rechenaufträgen und bietet dem Nutzer eine Reihe von Diensten an.

Ein Betriebssystem besteht zumindest in seinem Kern aus Firmware und macht es überhaupt erst möglich, daß man auf dem Computer eigene Programme abarbeiten kann. Es gibt für dieselben Computer meist eine Reihe verschiedener Betriebssysteme, die der Anpassung der im wesentlichen starren Hardware an die unterschiedlichen Anforderungen in Wissenschaft, Technik und Volkswirtschaft dienen.

1.5.1. Betriebsarten von Computern

Elektronische Rechenautomaten werden heute in verschiedener Art und Weise eingesetzt:

- Bei den ersten Computern war nur der sog. *Einzelprogrammbetrieb* möglich. Vom Bediener wurde ein Programm zunächst entwickelt, dann geprüft und schließlich gestartet. Solange er arbeitete, konnte kein weiterer Nutzer an den Rechner. Diese Betriebsart ist einfachen Mikrorechnern angemessen und daher für das genannte Aufgabenspektrum üblich. Weit verbreitet ist hierfür das Betriebssystem CP/M (Control Program for Microprocessors).
- Größere Rechner werden aber durch einen einzelnen Nutzer nicht ausgelastet. Auf modernen Minicomputern könnten fünf bis zehn Personen gleichzeitig ihre Programme entwickeln und abarbeiten, auf Großrechnern noch weit mehr. Man nutzt solche leistungsfähigen Computer daher im *Teilnehmerbetrieb.* Jeder Nutzer besitzt sein *Terminal,* bestehend aus Tastatur und Bildschirmgerät, manchmal mit Drucker. Das Betriebssystem nutzt die Denkpausen der verschiedenen Bediener und verzahnt deren Arbeit so miteinander, daß keiner durch den anderen merkbar behindert wird. Für Mikrorechner ist hier unter anderem das Betriebssystem MP/M (Multi-Programming Control Program for Microprocessors) zu nennen.
- Um eine hinreichend hohe Auslastung von Großrechnern zu erreichen, übergibt man ihnen zusätzlich zum Teilnehmerbetrieb Nutzrechnungen mit bereits geprüften Programmsystemen als sog. Jobs, die das Betriebssystem selbständig in den Arbeitsablauf des Computers einschiebt. Man spricht hier vom *Stapelbetrieb,* der aber für Mikrorechner keine Rolle spielt. Al-

lerdings könnten Mikrocomputer in lokalen Rechnernetzen größere Rechenaufgaben, die sie nicht selbst ausführen können, als Job an einen Großrechner delegieren.

- Sehr häufig werden Mikrorechner für die Lösung von Steuerungs- und Regelungsaufgaben eingesetzt, wobei sie oft direkt in die betreffende Anlage integriert werden. Hier liegen dann die Anforderungen des sog. *Echtzeitbetriebs* vor. Der Computer muß die entscheidenden Abläufe in dem zu steuernden System – z. B. einer Lichtsignalanlage an einer Straßenkreuzung – laufend überwachen, daraus hinreichend schnell Schlußfolgerungen ziehen und rechtzeitig auf das genannte System zurückwirken. Für solche Zwecke verwendet man Echtzeitbetriebssysteme, z. B. RMX-80 (Real-time Multi-task Executive).

1.5.2. Aufgaben von Betriebssystemen

Aus den verschiedenen Betriebsarten der Computer ergeben sich natürlich unterschiedliche Aufgaben für die jeweiligen Betriebssysteme. Dabei wird das folgende Spektrum überstrichen:

- *Auftragsverwaltung:* Annahme von Rechenaufträgen (z. B. vom Bediener) und deren Verwaltung nach ihrer Dringlichkeit;
- *Ablaufsteuerung:* Steuerung der Bearbeitung von Rechenaufträgen durch miteinander kooperierende Rechenprozesse;
- *Betriebsmittelverwaltung:* Bereitstellung von Elementen des Rechnersystems für die Erfüllung von Rechenaufträgen, beispielsweise Zuteilung von benötigten Speicherbereichen;
- *Eingabe/Ausgabe-Steuerung:* Abwicklung der Eingabe und Ausgabe von Informationen über die entsprechenden Geräte;
- *Datenverwaltung* (auf externen Zusatzspeichern): Organisation von *Bibliotheken* für Programme und von *Dateien* für Verarbeitungsdaten.

1.5.3. Aufbau eines Betriebssystems

Betriebssysteme sind glücklicherweise so aufgebaut, daß der Bediener von der Vielfalt der Aufgaben und der Kompliziertheit ihrer Lösung nur wenig erfährt. Das Betriebssystem bietet dem Nutzer Dienste an, die ihn bei seiner Arbeit unterstützen; die Realisierung ist Sache des Betriebssystems. Der Bediener sieht gewissermaßen nur die äußere Schale, die oberste Schicht des

Tafel 1.2. Charakteristische Bedienerkommandos

Arbeit mit der Dateiverwaltung bzw. Programmbibliothek

- Abspeichern einer Datei bzw. eines Programms
- Ausschreiben eines Verzeichnisses
- Umbenennen einer Datei bzw. eines Programms
- Ausschreiben einer (Text-)Datei bzw. eines (Maschinenkode-)Programms
- Löschen einer Datei bzw. eines Programms

Arbeit mit den Eingabe/Ausgabe-Geräten

- Transport von Dateien von Eingabe- nach Ausgabegeräten
  (z. B. vom Lochbandleser auf eine Diskette)
- Zuweisung von physischen zu logischen Geräten
- Ausgabe des Zustands von Eingabe/Ausgabe-Geräten
  (z. B. der gültigen Gerätezuweisungen oder des Füllstands einer Diskette)

Unterstützung der Programmentwicklung

- Bereitstellung von Entwicklungssoftware
  (z. B. Texteditor, Assembler, Interpreter und Compiler für höhere Programmiersprachen, Programmverbinder, Testmonitor)
- Zusammenfassung von Bedienereingaben, die häufig in gleicher Folge benötigt werden, zu Kommandoketten

Betriebssystems. Er arbeitet mit der *Bedienerverständigungsroutine,* die seine *Kommandos* entgegennimmt. Dabei bedeutet ein Kommando im Prinzip den Start des zugeordneten Programms, das meist erst noch vom externen Zusatzspeicher geladen werden muß. Charakteristische Bedienerkommandos, die von (Betriebs-)Systemprogrammen ausgeführt werden, sind in **Tafel 1.2** zusammengestellt.

Es ist aber auch möglich, solche Systemprogramme aufzurufen, die bei der Entwicklung von Anwendungsprogrammen mithelfen (z. B. den BASIC-Interpreter). Und man kann auch selbst hergestellte Programme durch Angabe ihres Namens starten.

Die Bedienerverständigungsroutine braucht zur Erledigung ihrer Aufgaben Hilfe beim Zugriff auf den externen Zusatzspeicher, also auf Magnetbandkassette oder Diskette. Hierfür gibt es *Dateiverwaltungsprogramme.* Sie gestatten das Speichern von Programmen und Dateien unter Verwendung von Namen, die der Nutzer selbst wählen kann. Einige Aspekte werden im Abschnitt 9 näher diskutiert.

Die Dateiverwaltung braucht aber ihrerseits ebenfalls Unterstützung, und zwar bei der Nutzung der externen Zusatzspeicher. Hierfür sind Eingabe/Ausgabe-Programme erforderlich, die unmittelbar mit den Speichergeräten zusammenarbeiten und *Gerätebedienungsroutinen* genannt werden. Solche Programme müssen auch für die übrigen peripheren Geräte vorhanden sein, beispielsweise für Tastatur und Bildschirmgerät. Sie werden von der Bedienerverständigungsroutine zur praktischen Realisierung des Dialogbetriebs gebraucht.

Mit diesen Bemerkungen soll die kurze Behandlung von Aufbau und Funktionsweise eines Computers abgeschlossen werden.

2. Programmentwicklung

Die Entwicklung von Programmen ist eine schöpferische Arbeit.
Sie erfordert eine hohe Qualifikation, ein systematisches, wissenschaftliches fundiertes Vorgehen. Das erzeugte Produkt muß strengen Qualitätskriterien genügen. Im folgenden sollen einige Methoden und Mittel diskutiert werden, mit deren Hilfe man solche Programme herstellen kann.

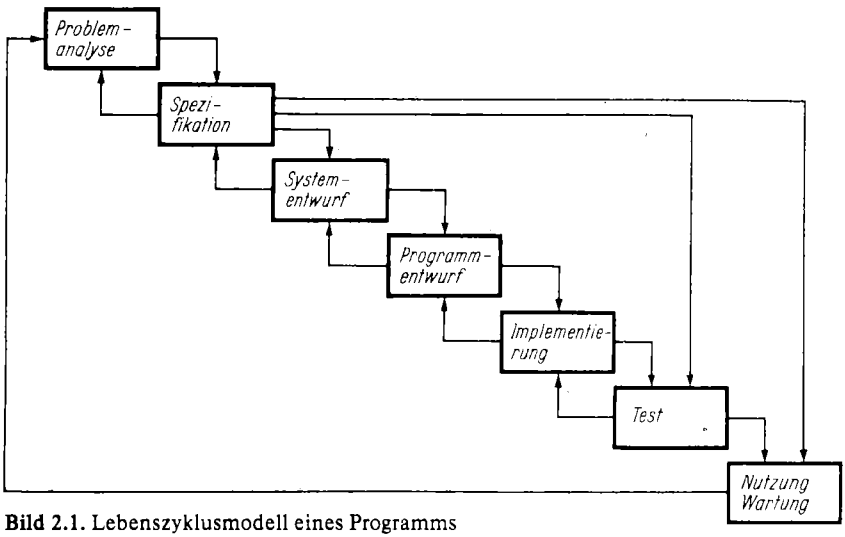


Bild 2.1. Lebenszyklusmodell eines Programms

Es gibt dabei verschiedene Entwicklungsabschnitte, die auch als *Lebenszyklus* eines Programms bezeichnet werden. Entsprechend den vielen bisher bekannten Prinzipien, Konzepten, Methoden und Mitteln der *Softwaretechnologie* gibt es auch mehrere Modelle für diesen Lebenszyklus (**Bild 2.1**). Einige Etappen auf dem Entwicklungsweg eines Programms werden in den folgenden Abschnitten kurz erläutert:

Analysieren: Untersuchen des zu lösenden Problems
Spezifizieren: Definieren der Aufgabenstellung
Entwerfen: Systementwurf, Programmentwurf
Implementieren: Kodieren, Übersetzen
Testen: Prüfen einzelner Moduln, Integration der Moduln zum Programmsystem
Dokumentieren: Beschreiben des Softwareprodukts
Warten/Pflegen: Beseitigen von Fehlern, laufendes Aktualisieren.

Die in diesem Buch benutzten Programmbeispiele sind allerdings relativ klein, so daß man die Vorteile eines disziplinierten Herangehens an die Programmentwicklung nicht so klar erkennt wie bei den in der Praxis üblichen komplexen Programmsystemen. Interessenten an einer detaillierteren Darstellung werden auf die Spezialliteratur zur Softwaretechnologie verwiesen [2.1] bis [2.5].

2.1. Analysieren und Spezifizieren

Am Beginn der Programmentwicklung muß eine gründliche Analyse des zu lösenden Problems stehen, die zu einer exakten *Aufgabenstellung* führt. Dabei sind folgende Aspekte zu klären:

- Was ist zu tun? Welche Aufgabe ist zu lösen, welche *Funktion* ist zu realisieren? Welche Ausgangszustände (Eingabeparameter) sollen welche Resultate (Ausgabeparameter) zur Folge haben? Es handelt sich dabei also um *interne* Eigenschaften des Programms.
- Welche Formate (d. h., welche äußere Gestalt) und welche Werte können die Eingabeparameter annehmen? Wie, wann und wo werden sie erfaßt, und über welche Eingabegeräte gelangen sie in den Computer? Entsprechende Fragen sind auch für die Ausgabeparameter zu beantworten. Welche Dateien auf welchen externen Zusatzspeichern werden wozu benutzt, und wie sind sie strukturiert? Hier handelt es sich also um *externe* Eigenschaften, um die *Schnittstellen* des Programms zu seiner Umgebung.

Weiterhin sind die *Qualitätskriterien* festzulegen, die Leistungsparameter, denen das Produkt genügen soll:

- Ein Programm muß natürlich in jedem Fall *korrekt* sein, das heißt, es muß die vorgegebene Spezifikation erfüllen.
- Ein Programm sollte *effizient* sein. Dabei handelt es sich um einen geeigneten Kompromiß zwischen den einander entgegenwirkenden Forderungen nach geringem Speicherplatzbedarf bzw. nach kurzen Abarbeitungszeiten (Abschn. 3.7).
- Ein Programm muß *robust* sein, also unempfindlich gegen eine fehlerhafte Benutzung. Auf unzulässige Formate oder Werte der Eingabeparameter hat es angemessen zu reagieren und darf nicht zum Abbruch der Arbeit führen.
- Ein Programm sollte *übersichtlich* aufgebaut sein. Es ist dann leicht zu verstehen und läßt sich einfach den veränderlichen Einsatzbedingungen anpassen (Abschn. 2.2).
- Dialogorientierte Programme sollten *nutzerfreundlich* sein, das heißt einfach zu bedienen. Sie sollten z. B. versierte Nutzer nicht durch lange Ausschriften auf dem Bildschirm und durch das Anfordern umfangreicher Eingaben über die Tastatur aufhalten, aber natürlich unbeholfene Anfänger unterstützen (Abschn. 2.6.3).

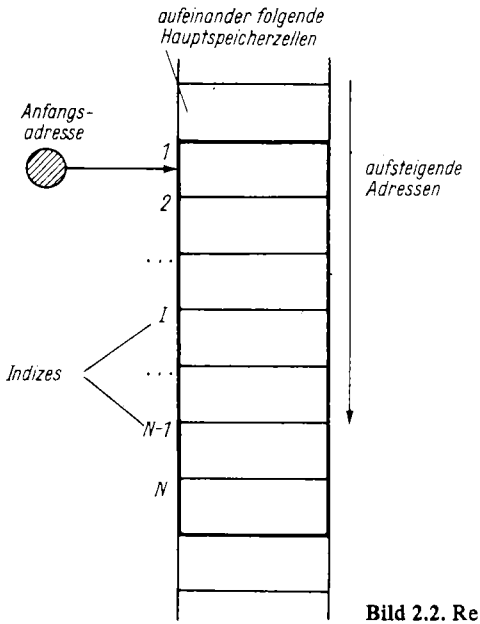


Bild 2.2. Reihung von Variablen im Hauptspeicher

Alle diese Punkte sind im Rahmen der Problemanalyse zu durchdenken und abschließend in der Programmspezifikation niederzuschreiben:

- Die umfassende und exakte Definition der Aufgabenstellung für die Entwicklung eines Programmsystems wird *Spezifikation* genannt.

Dabei werden weitreichende Festlegungen über Leistungsfähigkeit und Einsatzmöglichkeit getroffen. Es ist naheliegend, daß ein so entscheidender Schritt meist nicht die Arbeit eines einzelnen ist. So würden beispielsweise bei einem Mikrorechnereinsatz in der Automatisierungstechnik die Spezialisten des Entwicklungskollektivs gemeinsam beraten.

Als kleines Beispiel soll in den folgenden Abschnitten ein Sortierproblem behandelt werden. Gegeben ist ein sog. *Datenfeld (array).* Seine Komponenten sind Variablen gleichen Typs, die in aufeinanderfolgenden Zellen des Hauptspeichers stehen **(Bild 2.2)**, weswegen diese Datenstruktur auch als *Reihung* bezeichnet wird. Ein Datenfeld wird durch zwei Parameter charakterisiert, nämlich durch

- die *Anfangsadresse* im Hauptspeicher, das ist die Adresse der ersten Komponente, und
- die *Anzahl* seiner Komponenten.

Die einzelnen Komponenten eines Datenfelds können durch eine laufende Nummer, den *Index*, unterschieden werden. Danach heißen diese Variablen auch *indizierte Variablen.*

Die Aufgabe besteht nun darin, die Werte dieser Variablen so umzuordnen, daß sie – in Richtung wachsender Indizes gesehen – eine monoton steigende Folge bilden.

2.2. Entwerfen

Das Entwerfen eines Programms ist ein Konstruktionsprozeß, bei dem man verschiedene Etappen unterscheiden kann:

- Es ist zunächst erforderlich, ausgehend von der Spezifikation, eine *Struktur* der zu lösenden Aufgabe zu finden, das sind die auszuführenden Funktionen (Teilschritte) und ihre gegenseitigen Verknüpfungen.
- Dieser problemorientierte Systementwurf ist anschließend für eine programmtechnische Lösung zu konkretisieren. Dabei entsteht ein System miteinander verbundener *(Programm-)Moduln.*

2.2.1. Systementwurf

Die in diesem Buch behandelten Probleme haben nur einen relativ geringen Umfang, man kann sie noch vollständig überblicken. Sind aber größere Aufgabenkomplexe zu lösen, so muß man berücksichtigen, daß der Mensch nur eine begrenzte Anzahl von Objekten mit ihren Eigenschaften und wechselseitigen Beziehungen gleichzeitig erfassen kann. Um zu vermeiden, daß man wichtige Gesichtspunkte übersieht und folglich fehlerhaft programmiert, muß man die Struktur des zu lösenden Problems und damit auch die entsprechende Programmstruktur *schrittweise* herausarbeiten. Man zerlegt dazu die Gesamtaufgabe in beherrschbare Teile und betrachtet jeweils immer nur so viele Objekte, daß man sie noch gut überblicken kann. Arbeitet man mit grafischen Entwurfshilfsmitteln, so muß man beispielsweise das untersuchte (Teil-)Problem auf einem Blatt aufzeichnen können, so daß man alles gleichzeitig erfassen kann.

Ausgangspunkt des Entwurfs ist die Spezifikation. Die auszuführende Funktion wird analysiert und in kleinere Teilfunktionen zerlegt. Als Beispiel kann das bereits erwähnte Sortieren eines Datenfelds dienen. Elementare Funktionen sind hier das Vergleichen und das Vertauschen der Inhalte (Werte) zweier Speicherzellen **(Bild 2.3)**. Das sind aber bereits Funktionen, die mit Hilfe einer Programmiersprache, wie beispielsweise BASIC, ausgedrückt werden können. Die Analyse könnte dann bereits abgeschlossen werden.

Ist die geforderte Gesamtfunktion jedoch komplexer, so führt eine Zerlegung *(Dekomposition)*

nicht bereits nach dem ersten Schritt zu einfachen Teilfunktionen. Die Analyse muß wiederholt werden, möglicherweise mehrere Male. Die Betrachtung wird also, vom Gesamtanliegen kommend, schrittweise verfeinert. Man spricht von einer *Top-down-Analyse*. Dabei entsteht eine grafische Darstellung entsprechend **Bild 2.4**: Jede der in dieser Hierarchie obenstehenden Funktionen enthält die darunterliegenden und wird mit deren Hilfe realisiert.

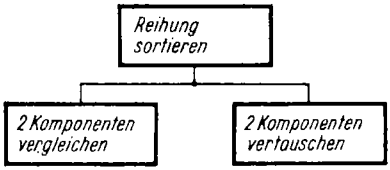


Bild 2.3. Zerlegung einer Funktion

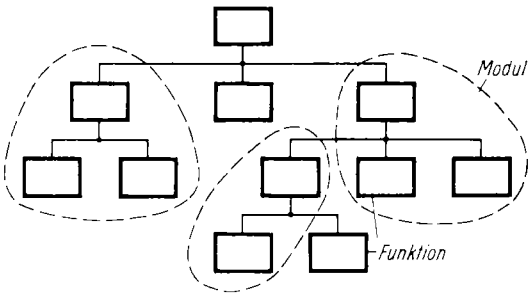


Bild 2.4. Hierarchische Baumstruktur der Funktionen bei einer Problemlösung

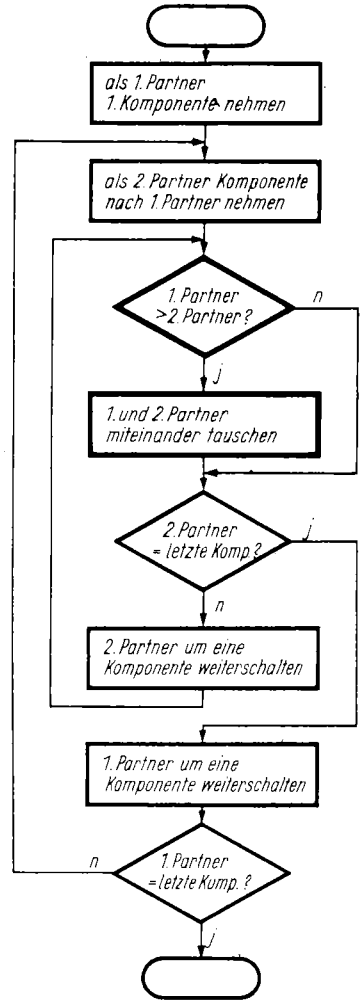


Bild 2.5. Algorithmus eines Sortierverfahrens

Ein solches Bild vermittelt einen Überblick darüber, was im einzelnen getan werden muß. Es sagt aber noch nichts darüber aus, wie, in welcher Art und Weise elementare Funktionen zur Ausführung einer komplexen Funktion eingesetzt werden. Das erfolgt durch den sog. Algorithmus.

- Ein *Algorithmus* beschreibt eindeutig und vollständig, wie ein bestimmtes Resultat bei gegebenen Voraussetzungen durch eine endliche Anzahl von Verarbeitungsschritten erzielt werden kann.

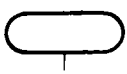
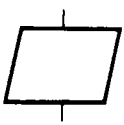
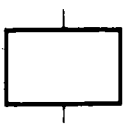
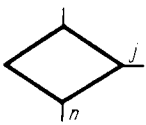
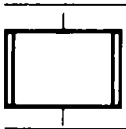
Diese Schritte bestehen aus einzelnen ausführbaren, elementaren Operationen (Funktionsaufrufen). So wird es das Sortieren eines Datenfelds erfordern, daß in einer geeigneten Weise mehrere Paare von Komponenten miteinander verglichen und erforderlichenfalls umsortiert werden.

Dafür gibt es aber verschiedene Möglichkeiten, verschiedene Vorgehensweisen, die entsprechend unterschiedliche Algorithmen bedingen. So könnte z. B. zunächst die erste Feldkomponente nacheinander mit allen weiteren verglichen werden. Findet man dabei, daß der Wert der ersten Komponente größer als der einer anderen ist, so muß vertauscht werden. Am Ende dieser Verarbeitungskette weiß man, daß die erste Komponente nunmehr die kleinste ist. Jetzt vergleicht man in entsprechender Weise die zweite Komponente mit den folgenden, aber nicht mehr mit der ersten. Auch hier wird erforderlichenfalls vertauscht. Ist dieser zweite Durchlauf beendet, so liegt die zweite Komponente als die zweitkleinste fest. Die weiteren Schritte sind nunmehr sicher klar. Der Algorithmus endet mit dem Vergleich und einer eventuellen Vertauschung der beiden letzten Komponenten.

Im vorstehenden Absatz wurde der Algorithmus mit Worten dargelegt. Diese Vorgehensweise ist aufwendig, manchmal auch nicht ganz eindeutig. Sie ist aber auch schwer verständlich, weil der Kern hinter zuviel Worten verborgen bleibt. Daher sind *formalisierte* Beschreibungsmittel für Algorithmen üblich. Besonders beliebt sind dabei grafische Darstellungen, weil sie ein anschauliches Bild des Ablaufs einer Berechnung vermitteln und daher auch die Suche nach logischen Fehlern im Algorithmus erleichtern. Sehr verbreitet ist der *Programmablaufplan*, auch Flußdiagramm oder Blockdiagramm genannt. Er bietet für bestimmte, elementare Teilschritte standardisierte grafische Symbole, die durch verbale Zusätze ergänzt werden können. Als Beispiel wird im **Bild 2.5** der diskutierte Algorithmus dargestellt. Man kann daraus bereits die Bedeutung der Sinnbilder erkennen. An der Darstellung sieht man übrigens anschaulich die Eigenschaft des Computers, entsprechend dem als Programm vorgegebenen Algorithmus Entscheidungen zu treffen und sich danach die erforderlichen Verarbeitungsschritte selbständig auszusuchen.

Einen Überblick über die wichtigsten für Programmablaufpläne festgelegten Sinnbilder gibt die **Tafel 2.1**. Sie lassen sich einfach mit Hilfe käuflicher Schablonen zeichnen. In einem Pro-

Tafel 2.1. Sinnbilder für Programmablaufpläne

	Grenzstelle (Anfang, Zwischenhalt oder Ende eines Programms)
	Eingabe oder Ausgabe
	Verarbeitungsschritt (allgemeine Darstellung einer einzelnen Operation oder einer Gruppe von Operationen)
	Alternative (Verzweigung aufgrund einer Entscheidung)
	Unterprogramm (an anderer Stelle festgelegte, strukturierte Menge von Operationen)
	Konnektor (Übergang von/zu einer anderen Stelle des Programmsystems)

grammablaufplan werden sie mit Hilfe von gerichteten Linien verbunden, die die Reihenfolge der Teilschritte angeben. Der allgemeine (sequentielle) Ablauf der Verarbeitung soll von oben nach unten erfolgen. Abweichungen (Sprünge) werden meist so dargestellt, daß Vorwärtssprünge rechts, Rücksprünge links vom Hauptablauf liegen. Die ergänzenden Erläuterungen erfolgen durch Text innerhalb der Sinnbilder; es sind aber auch (rechts seitlich) weitere Kommentare möglich. Dabei darf der Umfang eines Programmablaufplans (PAP) nicht zu groß sein; er sollte nicht über mehrere Seiten gehen. Bei größeren Problemen ist es günstiger, zunächst einen Grob-PAP mit hohem Abstraktionsgrad zu zeichnen und dann die komplexen Verarbeitungen, Entscheidungen usw. durch weitere, detailliertere PAP zu untersetzen.

Bei der bisherigen, funktionsorientierten Analyse tauchten aber bereits Probleme neuer Art auf, die mit den zu hantierenden Daten im Zusammenhang stehen. Offensichtlich ist es nicht möglich, nur mit den in der Spezifikation genannten Eingabe- und Ausgabeparametern auszukommen. Zum Beispiel ist es für die Abarbeitung des obengenannten Algorithmus erforderlich, sich die Indizes des aktuellen 1. bzw. 2. Partners des gerade untersuchten Paares zu merken. Man benötigt also zusätzliche Daten, die nur während der Verarbeitung existieren. Diese Probleme sollen im folgenden Abschnitt mit besprochen werden.

2.2.2. Programmentwurf

Es ist nun erforderlich, den allgemeinen, problemorientierten Entwurf für einen gegebenen Computer und eine bestimmte Programmiersprache zu konkretisieren. Ausgangspunkt dafür sind zweckmäßigerweise Überlegungen über die benutzten Daten:

- Externe *(globale)* Daten werden durch die Spezifikation festgelegt (Abschn. 2.1).
- Interne *(lokale)* Daten werden nur vorübergehend während der Abarbeitung des Programms benötigt. Hier sind vom Programmierer geeignete Darstellungsformen der Daten, die sog. *Datenformate*, selbst zu wählen.

Programmiersprachen bieten in der Regel Standardformate in Form von *Datentypen* an. Solche Typen sind unter anderem: ganze Zahlen, gebrochene (*reelle*) Zahlen und Ketten von hintereinander geschriebenen Textzeichen (d. h. Zeichen, Wörter, Sätze usw.). Man kann Variablen dieser Typen auch zu *Datenstrukturen* zusammenfügen. So bestehen Felder beispielsweise aus Variablen gleichen Typs.

In dem Beispiel des diskutierten Sortierverfahrens fallen zunächst, wie bereits erwähnt, zwei interne Variablen ins Auge, nämlich die Indizes des aktuellen 1. bzw. 2. Partners des gerade behandelten Paares. Sie sollen und können hier, wie von der Mathematik her gewöhnt, mit allgemeinen Zahlsymbolen bezeichnet werden, zum Beispiel mit I und J. Es handelt sich dabei um ganze Zahlen im Intervall 1 bis N (Anzahl der Elemente).

Als nächstes ist der bisherige, problemorientierte Entwurf daraufhin zu untersuchen, wie die einzelnen Funktionen – die ja einen verschiedenen Abstraktionsgrad aufweisen! – zweckmäßig zu sog. *Funktionsmoduln* zusammenzufassen sind:

- *Moduln* sind relativ selbständige Teilprogramme, die zunächst getrennt voneinander detailliert entworfen, realisiert und geprüft und dann erst miteinander zum geforderten Programm (-system) verbunden werden.

Diese Unterteilung in überschaubare Moduln erleichtert die Programmentwicklung, insbesondere die Prüfung.

Die geeignete Auswahl der Funktionen, die jeweils in einem Modul zusammenzufassen sind, setzt gewisse Erfahrungen voraus. Man geht meist von den elementaren Funktionen (Bild 2.3) aus und benutzt folgende Kriterien:

- Ein Modul sollte *nicht zu groß* sein (ca. 100 Anweisungen der jeweiligen Programmsprache).
- Der Modul muß eine *inhaltlich abgeschlossene Gesamtfunktion* realisieren, die möglichst bei der Lösung anderer Aufgaben wieder benötigt wird.
- Der Modul muß *einfache, überschaubare Schnittstellen* zur Umgebung haben.

Durch die Bildung von Moduln entsteht aus der Systemstruktur (Bild 2.4) die *Programmstruktur* **(Bild 2.6)**. Übrigens nennt man den obersten Modul, der gewissermaßen als Chef im Programmsystem die Regie führt, oft *Hauptprogramm*. Die ihm untergeordneten Moduln heißen *Unterprogramme*. Sie werden zur Ausführung ihrer Funktion von dem jeweils übergeordneten Modul aufgerufen.

Verständlicherweise ist es beim einfachen Beispiel des obengenannten Sortierverfahrens noch nicht erforderlich, Moduln zu bilden; eher hat das ganze Sortierverfahren den Charakter eines Moduls (siehe dazu die Beispiele im Abschn. 11.4). Trotzdem sollen die Bilder 2.3 und 2.5 auf die Möglichkeit hin untersucht werden, etwas zusammenfassen zu können. Man findet, daß sich der Vergleich der Werte zweier Elemente völlig in die anderen Vergleiche (Endetests) einordnet. Dagegen hat die Funktion des Vertauschens zweier Werte einen etwas selbständigen Charakter. Damit entsteht aus dem Bild 2.3 die einfache Modulstruktur im **Bild 2.7**.

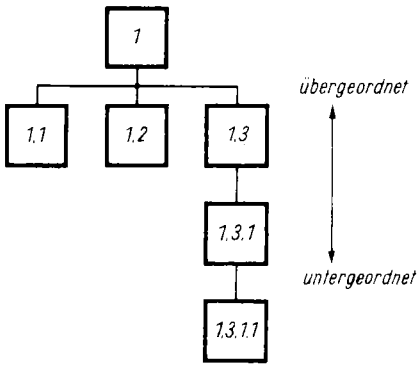


Bild 2.6. Hierarchische Baumstruktur der Moduln eines Programmsystems

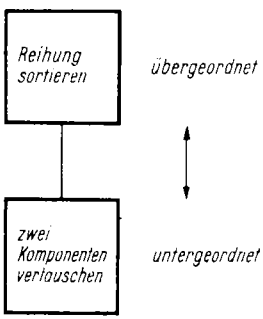


Bild 2.7. Modulhierarchie des Sortierverfahrens

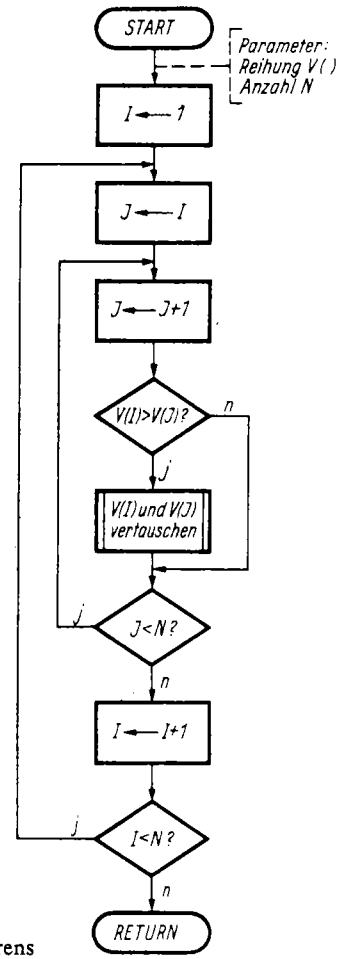
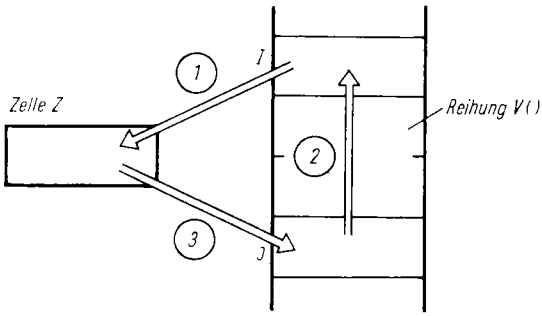


Bild 2.8. Programmablaufplan des Sortierverfahrens

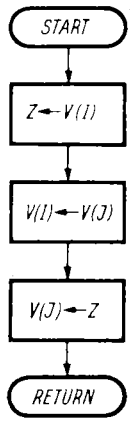
Nun ist der Algorithmus vom Bild 2.5 unter Nutzung der bisherigen Überlegungen zu konkretisieren. Die Darstellung im **Bild 2.8** wird dazu weiter formalisiert. So dient zur Kennzeichnung einer Komponente (also einer indizierten Variablen) mit dem Index I das Symbol V(I). Der Buchstabe V soll dabei auf Vektor (eindimensionales Datenfeld) hinweisen. Weiterhin steht das Symbol ← für eine Wertübertragung von rechts nach links. I ← I + 1 bedeutet also: Die Varia-

ble I erhält den (neuen) Wert I + 1, das heißt, ihr Wert wird um den Betrag eins erhöht (inkrementiert). Im Bild 2.8 steht I als Index zur Kennzeichnung des 1. Partners, J für den 2. Partner eines zu vergleichenden Paares. N ist der Index der letzten Komponente.

Jetzt müssen wir uns weiter mit demjenigen Modul beschäftigen, der die Komponenten mit den Indizes I und J vertauschen soll. Bei einer genaueren Untersuchung stellt man fest, daß an dieser Stelle eine weitere interne Variable erforderlich ist, die nur innerhalb dieses Moduls gebraucht wird. Es handelt sich um eine Speicherzelle Z, in die vorübergehend der Wert eines Elements abgelegt werden muß. Die Nutzung wird aus **Bild 2.9** ersichtlich: Da in den Programmiersprachen (meist) nur Wertzuweisungen und keine Wertvertauschungen möglich sind, muß ein Ringtausch organisiert werden. (Was würde geschehen, wenn statt dessen nacheinander die Wertzuweisungen A(J) ← A(I) und A(I) ← A(J) ausgeführt würden?)



a)



b)

Bild 2.9. Vertauschen zweier Elemente
a) Datenflußschema
b) Programmablaufplan

2.2.3. Grundstrukturen

In den vorstehenden Abschnitten wurden bereits Funktionen, die durch einzelne Anweisungen oder ganze Moduln realisiert werden, in unterschiedlichen Formen miteinander verbunden. In diesem Zusammenhang spricht man von Steuerstrukturen:

- Unter *Steuerstrukturen* versteht man die in Computern möglichen Varianten, die auszuführenden Befehle derart miteinander zu verknüpfen, daß der vom Algorithmus geforderte Ablauf der einzelnen Verarbeitungsschritte erzielt wird.

Eine anschauliche Darstellung dieser Steuerstrukturen bietet unter anderem der Programmablaufplan. Er zeigt klar diejenigen Stellen im Programm, an denen Entscheidungen gefällt werden und wo die sequentielle Arbeit des Rechenautomaten möglicherweise durch sog. Sprünge verlassen wird. Diese Sprünge sind erfahrungsgemäß eine Quelle vieler Fehler, weil sie die Übersichtlichkeit der Abläufe beeinträchtigen, insbesondere bei Sprüngen nach weit entfernt liegenden Programmteilen. Das macht sich vor allem bei späteren Änderungen störend bemerkbar. Daher wird bei der sog. *strukturierten Programmierung* vorgeschlagen, sich beim Umsetzen von Algorithmen in Programme auf wenige Grundstrukturen zu beschränken. Im Prinzip reichen drei Steuerstrukturen aus:

- Die *Folge (Sequenz)* entspricht der natürlichen Arbeitsweise eines Rechenautomaten. Hier werden zwei oder mehr Anweisungen nacheinander ausgeführt **(Bild 2.10.)**.
- Bei der *Alternative* wird eine *Bedingung* geprüft, es werden z. B. zwei Werte miteinander verglichen. Je nachdem, ob diese Bedingung erfüllt (*wahr*, TRUE) oder nicht erfüllt (*falsch*, FALSE) ist, werden unterschiedliche Anweisungen ausgeführt **(Bild 2.11)**. Danach vereinigen sich die Flußlinien wieder. Es gibt noch weitere Formen der Verzweigung; sie werden im Abschnitt 6.1 näher behandelt.

- Eine bestimmte Anweisung soll zyklisch wiederholt abgearbeitet werden, bis mit Hilfe einer Bedingung das Verlassen dieser *Schleife* möglich wird **(Bild 2.12)**. Auch hier gibt es verschiedene Varianten, die im Abschnitt 6.2 zur Diskussion stehen.

Allen genannten Grundstrukturen ist gemeinsam, daß man sie zu einer einzigen (Verbund-)Anweisung zusammenfassen kann, die – wie eine einfache Anweisung – genau *einen* Eingang und *einen* Ausgang für den Programmablauf hat. Daher lassen sich diese Grundstrukturen auch ineinanderschachteln; in einer Schleife kann z. B. eine Anweisungsfolge oder (wie im Bild 2.8) eine Alternative ausgeführt werden.

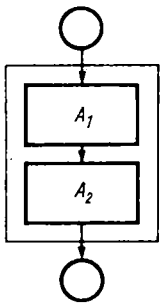


Bild 2.10. Folge (Sequenz) zweier Anweisungen A_1, A_2

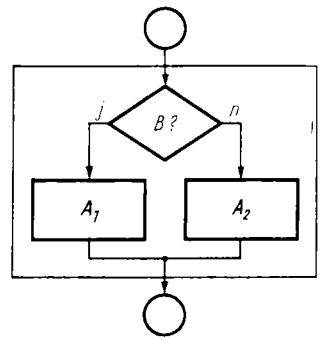


Bild 2.11. Alternative zwischen zwei Anweisungen A_1, A_2 entsprechend einer Bedingung B

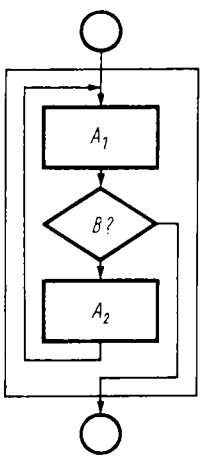


Bild 2.12. Schleife mit zwei Anweisungen A_1, A_2 und der Abbruchbedingung B

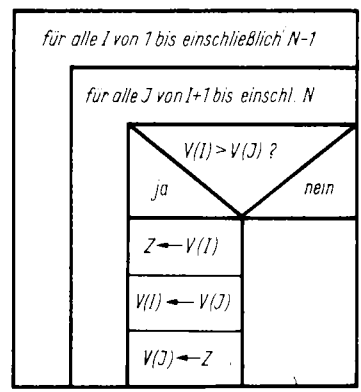


Bild 2.13. Struktogramm eines Sortierverfahrens

An den Programmierer wird appelliert, sich im Interesse der Übersichtlichkeit seiner Produkte auf diese drei Steuerstrukturen zu beschränken. Um ihn dabei zu leiten, sind auch spezielle grafische Darstellungsmittel, die *Struktogramme*, entwickelt worden, die es unmöglich machen, unzweckmäßige Steuerstrukturen zu entwerfen. Entsprechend enthalten moderne höhere Programmiersprachen keine Sprunganweisungen mehr. BASIC gehört allerdings nicht zu diesen Sprachen. Daher muß der Programmierer hier Selbstdisziplin halten.

Um dem Leser eine Vorstellung von Struktogrammen zu vermitteln, ist im **Bild 2.13** der Algorithmus des diskutierten Sortierprogramms in dieser Form dargestellt. Im folgenden werden aber nur Programmablaufpläne benutzt. Ursache dafür ist einfach der Fakt, daß diese Grafen die Struktur von Programmen, die in der Programmiersprache BASIC kodiert werden sollen, adäquater widerspiegeln.

2.3. Programmiersprachen

Im nächsten Entwicklungsschritt muß das entworfene Programm in eine für den Mikrorechner verständliche Form gebracht werden:

- Das Umsetzen eines Programmentwurfs mit Hilfe einer Programmiersprache in eine dem Computer verständliche Form nennt man *Kodieren*.

Leider sind aber bisher nur wenige Computer so konzipiert worden, daß sie die Sprache der Mathematik unmittelbar verstehen, die der Mensch zur eindeutigen Formulierung und Lösung mathematischer (Berechnungs-)Probleme geschaffen hat. Die Maschinensprache des Rechenautomaten ist meist viel elementarer. Für die Lösung dieses Problems gibt es zwei Wege:

- Der Programmierer erlernt die *Maschinensprache*. Diese Mühe ist sicher dann erforderlich, wenn Mikroprozessoren als steuernde Kerne in elektronische Geräte und Anlagen eingebettet werden; denn hier ist unter anderem wegen der anwendungsspezifischen Spezialperipherie ein hardwarenahes Programmieren erforderlich.
- Dem Computer wird die Sprache der Mathematik verständlich gemacht. Dem Programmierer steht dann eine *höhere, algorithmische Programmiersprache* zur Verfügung. Die Übersetzung in die Maschinensprache übernehmen dafür geschaffene Programmsysteme, *Compiler* bzw. *Interpreter* genannt.

Alle diese Sprachen sind künstlich geschaffene, *formale Sprachen*, die – wie die natürlichen Sprachen – über einen bestimmten Wortschatz (die *Lexik*) verfügen. Diese Sprachelemente (die *Morpheme*) werden nach bestimmten, strengen Vorschriften, den sog. *syntaktischen Regeln*, zu komplexeren Sprachausdrücken zusammengefügt, die man mit den Sätzen von natürlichen Sprachen vergleichen könnte. Sowohl die Sprachelemente als auch die Satzkonstruktionen haben genau festgelegte Bedeutungen (die *Semantik*).

2.3.1. Maschinenorientierte Sprachen

Bereits in der Einleitung wurde festgestellt:

- Die *Maschinensprache* eines Computers umfaßt alle Maschinenkodes (internen Bitmuster), die als Befehle erkannt und ausgeführt werden können.

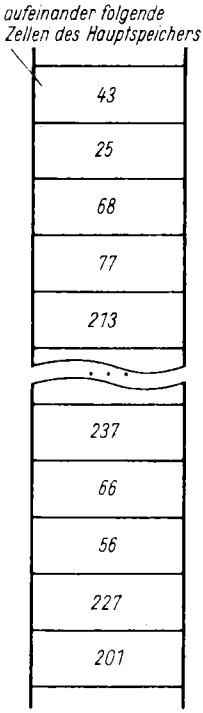
Es handelt sich also um maschinenspezifische Anweisungen, welche die auszuführende Operation und die dabei zu benutzenden Operanden enthalten. Setzt man sie zur Beschreibung eines Algorithmus ein, so erhält man ein Maschinenprogramm:

- Als *Maschinenprogramm* bezeichnet man eine strukturierte Menge von Anweisungen, die einen gegebenen Algorithmus mit Hilfe von Maschinenkodes beschreiben.

Programm 2.1. Maschinenprogramm eines Sortierverfahrens

```
 43  25  68  77 213  26  98 107  35 190  56   5
 56   3  95 126 115  84  93 183 237  66  56 238
209  18  19  98 107 237  66  56 227 201
```

Als Beispiel soll das im letzten Abschnitt diskutierte Sortierverfahren dienen. Kodiert man den in den Bildern 2.8 und 2.9 dargestellten Algorithmus für die Mikroprozessoren U880 oder Z80, so erhält man das **Programm 2.1**. Die angegebenen Dezimalzahlen stellen die Werte (Inhalte) aufeinanderfolgender Speicherzellen dar. In Anlehnung an das Bild 1.2 könnte man einen Ausschnitt dieses Maschinenprogramms auch in der im **Bild 2.14** gegebenen Form darstellen. Dabei sind die konkreten Adressen dieser Speicherzellen bei dem vorliegenden speziellen Programm uninteressant; es handelt sich hier um ein sog. *verschiebliches* Maschinenprogramm. Für den interessierten Leser seien noch folgende Randbedingungen angegeben:



Externe (globale) Daten:
Anfangsadresse im Registerpaar DE
Anzahl der Komponenten im Registerpaar HL.
Interne (lokale) Daten:
Adresse des 1. Partners im Registerpaar DE (≙I)
Adresse des 2. Partners im Registerpaar HL (≙J)
Adresse der letzten Komponente im Registerpaar BC (≙N).
(Im inneren Zyklus wird das Registerpaar DE als Zwischenspeicher benutzt.)

Bild 2.14. Maschinenprogramm des Sortierverfahrens im Hauptspeicher

Weitere Einzelheiten gingen über den Rahmen dieses Buches hinaus. Wir benötigen Programm 2.1 bzw. Bild 2.14 jedoch, um folgende Aussagen zu verstehen:

- Die Anwendung der Maschinensprache setzt *detaillierte Kenntnisse* über Aufbau und Arbeitsweise des konkreten Mikroprozessors voraus.
- Die Befehlsliste eines Mikroprozessors umfaßt mehrere hundert Maschinenkodes. Diese Zahlensprache ist daher *schwer zu erlernen*.
- Auch nach einer Lernphase wird man sicherheitshalber häufig in Tabellen nachschauen müssen. Deshalb ist die Programmierung in der Maschinensprache *uneffektiv*.
- Das entwickelte Programm ist völlig *unübersichtlich*. Eine derart schwer überschaubare Darstellungsform ist bereits während des Entwicklungsprozesses fehleranfällig. Für andere Personen als den Programmierer ist es zunächst unverständlich und läßt sich schwer verändern.
- Das Programm ist *nur* für einen *bestimmten* Computer bzw. eine Familie ähnlicher Computer geeignet. Wird es für einen Rechenautomaten mit einer anderen Maschinensprache benötigt, so muß es neu kodiert werden.

Einige der angegebenen Schwierigkeiten umgeht man bei der Anwendung der sog. Assemblersprache.

- Die *Assemblersprache* ist eine *maschinenorientierte Programmiersprache*, bei der im Unterschied zur Maschinensprache nicht interne Bitmuster, sondern symbolische Bezeichnungen für die gewünschten Operationen und die benutzten Operanden eingesetzt werden.

Jede Anweisung der Assemblersprache entspricht genau einem Befehl des Rechners. Für die Operationen sind Abkürzungen englischsprachiger Ausdrücke vorgeschrieben. Auch für die Register des Mikroprozessors sind bestimmte Benennungen erforderlich. Dagegen darf der Programmierer für die Kennzeichnung der Adressen von verwendeten Speicherzellen selbstgewählte Namen benutzen.

So wie jeder Computertyp seinen eigenen Maschinenkode hat, so hat er auch seine Assemblersprache. Sie wird in der Regel vom Gerätehersteller festgelegt.

Als Beispiel für ein in der Assemblersprache kodiertes Programm wurde im **Programm 2.2** wieder das bekannte Sortierverfahren dargestellt. Man sieht ein, daß sich durch die gewählten Abkürzungen für die Befehle ein besseres Verständnis ergibt (z. B. LD = load, ADD = add, CP = compare). Auch ist die Struktur durch die Angabe der Ziele von Sprüngen (SORT1, SORT2, SORT3) besser zu erkennen. Trotzdem gelten die obengenannten Aussagen über die Nachteile der Maschinensprache im Prinzip weiterhin, wenn auch in eingeschränktem Maß.

Programm 2.2. Assemblerprogramm eines Sortierverfahrens

```
SORT:   LBC     HL
        ADD     HL,DE
        LD      B,H
        LD      C,L       ; BC <= ENDADRESSE
SORT1:  PUSH    DE        ; STAPEL <= ADR. 1.PARTNER
        LD      A,(DE)    ; A <= 1.PARTNER
SORT2:  LD      H,D
        LD      L,E
        INC     HL        ; HL <= ADR. NEU 2.PARTNER
        CP      (HL)
        JR      C,SORT3   ; 1.PARTNER < 2.PARTNER
        JR      C,SORT3   ; 1.PARTNER = 2.PARTNER
        LD      E,A
        LD      A,(HL)    ; WERTE TAUSCHEN
        LD      (HL),E
SORT3:  LD      D,H
        LD      E,L       ; DE <= ADR. 2.PARTNER
        OR      A         ; C-FLAG LOESCHEN
        SBC     HL,BC
        JR      C,SORT2   ; ADR. 2.PARTNER < ENDADR.
        POP     DE        ; DE <= ADR. 1.PARTNER
        LD      (DE),A    ; ABSPEICHERN 1.PARTNER
        INC     DE        ; DE <= ADR. NEU 1.PARTNER
        LD      H,D
        LD      L,E
        SBC     HL,BC
        JR      C,SORT1   ; ADR. NEU 1.P. < ENDADR.
        RET
        END
```

Ein in der Assemblersprache kodiertes Programm besteht aus Textzeichen. Es ist daher für den Menschen lesbar, nicht aber für den Computer. Man muß den Programmtext erst in den Maschinenkode umwandeln. Wird beispielsweise das Assemblerprogramm 2.2 übersetzt, so entsteht das Maschinenprogramm 2.1.

2.3.2. Höhere Programmiersprachen

Nach der Schilderung der Schwierigkeiten bei der Programmierung eines Rechenautomaten in einer maschinenorientierten Sprache wünscht man sich sicher, bequemere und effektivere Mittel zur Verfügung zu haben. Auch sollten die kodierten Programme rechnerunabhängig sein, um auf allen Computern eingesetzt werden zu können. Ein mögliches Ziel könnte es sein, für die Beschreibung von Algorithmen eine Sprache zu benutzen, die an die Darstellungsform der Mathematik angelehnt ist. Dabei müßte es möglich sein, die Beschreibung so umfassend und eindeutig zu formulieren, daß sie ohne Rückfragen beim Programmierer verstanden werden kann.

- Als *höhere Programmiersprachen* bezeichnet man solche Programmiersprachen, die einen hohen Abstraktionsgrad aufweisen. Sie lehnen sich häufig in ihrem Aufbau und in der Bedeutung ihrer Elemente an konkrete Einsatzgebiete an.

Es existiert eine große Anzahl solcher Sprachen. Als Beispiele sollen genannt werden:

ALGOL	(ALGOrithmic Language)
PASCAL	(nach dem französischen Pionier der Rechentechnik)
FORTRAN	(FORmula TRANslation)
PL/1	(Programming Language 1)
BASIC	(Beginner's All-purpose Symbolic Instruction Code)
COBOL	(COmmon Business-Oriented Language)
CB	(Commercial BASIC).

Fachsprachen sind problemorientierte Spezialsprachen für spezielle Wissensgebiete, Industriezweige, Anwendungsfälle, Geräte oder Anlagen. Sie berücksichtigen die Problematik, daß in unseren Tagen – insbesondere durch das stürmische Eindringen der Mikroelektronik in alle Zweige von Wissenschaft, Technik und Wirtschaft – viele Menschen mit Problemen der Informatik konfrontiert werden. Für sie ist der Rechner ein notwendiges Hilfsmittel, aber nicht Arbeitsgegenstand. Darauf müssen die Fachsprachen Rücksicht nehmen. Als Beispiele könnte man nennen:

methodenorientierte Fachsprachen (Entscheidungstabellen, Petrinetze)
anwendungsorientierte Fachsprachen (Steuerung von Werkzeugmaschinen und Industrierobotern, grafische Datenverarbeitung).

Programm 2.3. BASIC-Programm eines Sortierverfahrens

```
100 REM ++++++++++++++++++ SORTIEREN ++++++++++++++++++++
110 LET I = 1
120 LET J = I
130 LET J = J+1
140 IF V(I)>V(J) THEN GOSUB 200
150 IF J<N THEN GOTO 130
160 LET I = I+1
170 IF I<N THEN GOTO 120
180 RETURN
190 REM ------------------ VERTAUSCHEN -------------------
200 LET Z = V(I)
210 LET V(I) = V(J)
220 LET V(J) = Z
230 RETURN
240 REM ==========================================
```

Um zu demonstrieren, wie ein in einer höheren Programmiersprache kodiertes Programm aussieht, wurde wieder das Sortierverfahren herangezogen und in BASIC kodiert. Das Resultat ist im **Programm 2.3** dargestellt. Man sieht die außerordentlich weitgehende Ähnlichkeit mit den Programmablaufplänen in den Bildern 2.8 und 2.9. Die offensichtlichen Vorteile höherer Programmiersprachen (HPS) lassen sich anschaulich erkennen, ohne daß man natürlich das einzelne Beispiel schon als Beweis werten kann:

- Der Umfang der Sprachelemente einer HPS ist wesentlich kleiner als der einer Maschinensprache. Der Programmierer braucht die konkrete Rechnerarchitektur nicht zu kennen. Daher ist eine HPS meist *leicht zu erlernen*. Das gilt insbesondere für die Sprache BASIC.
- Weil die Anweisungen einer HPS komplexer als die Befehle eines Rechenautomaten sind, wird der Aufwand für die Kodierung gesenkt. Der Programmierer kommt mit seiner Arbeit *schneller* voran.
- Die in einer HPS abgefaßten Programmtexte sind *gut lesbar*. Dabei helfen leichtverständliche Bezeichnungen für die auszuführenden Anweisungen (zum Beispiel LET, IF-THEN, GOTO). Daher enthalten solche Programme erfahrungsgemäß *weniger Fehler*. Diese lassen sich auch leichter als in Maschinenprogrammen auffinden, weil die Übersetzerprogramme umfangreiche Analysen durchführen und Fehler melden.

- Da die in einer HPS kodierten Programme gut zu verstehen sind, lassen sie sich auch *leicht und sicher ändern*. Dadurch sind die Wartungskosten geringer als bei der Anwendung maschinenorientierter Sprachen.
- Ein Programm, das in einer HPS geschrieben wurde, läuft auf *allen* Rechnern, die ein Übersetzerprogramm für diese HPS haben. Sorgen bereiten lediglich solche Übersetzer, die mit Dialekten der betreffenden HPS arbeiten. Leider ist das bei BASIC sehr verbreitet.

2.4. Implementieren

■ Zum *Implementieren* eines Programms zählt man diejenigen Arbeitsschritte, die erforderlich sind, um vom Programmentwurf bis zu einem Maschinenprogramm zu kommen.

Hierzu können folgende Etappen gerechnet werden:

Kodieren des Programmentwurfs in der gewählten Programmiersprache
Erfassen des Programmtextes und eventuell erforderlicher Korrekturen
Übersetzen des Programms.

Eine besondere Rolle spielt das *Prüfen* des Programms, ob es die vorgegebenen Spezifikationen erfüllt. Diesem Problem wird ein eigener Abschnitt gewidmet sein.

Einige Bemerkungen sind noch zum Übersetzen zu machen. Das Übersetzerprogramm arbeitet dabei im Prinzip ähnlich wie ein Mensch, der die Darstellung eines Sachverhalts von einer Sprache in eine andere übersetzt. Dabei gibt es bekanntlich zwei Vorgehensweisen:

- Der fremdsprachige Text wird während des Lesens übersetzt. Benötigt man ihn nochmals, muß der Übersetzungsvorgang wiederholt werden. Das entspricht der Arbeitsweise eines *Interpreterprogramms*.
- Der Text wird schriftlich übertragen, die Übersetzung erfolgt nur einmal. So arbeitet ein *Compilerprogramm*.

Bei dieser Gegenüberstellung fragt man sich möglicherweise, ob ein Interpreter überhaupt sinnvoll ist. Darauf soll im folgenden noch eingegangen werden.

2.4.1. Editieren

Nach dem Kodieren liegt das Programm in Textform vor. Da die weiteren Entwicklungsschritte hiervon ausgehen, spricht man vom *Quelltext* eines Programms, vom Quellprogramm.

■ Das in einer Programmiersprache kodierte, in Textform vorliegende Programm wird *Quellprogramm* genannt. Besteht das Quellprogramm aus einzelnen Moduln, so nennt man sie auch *Quellmoduln*.

Das Quellprogramm steht zunächst in Gestalt von Textzeichen auf dem Papier. Es muß jetzt in eine solche Darstellung übergeführt werden, daß der Computer damit arbeiten kann. Zu diesem Zweck ist es auf einem maschinenlesbaren Datenträger abzuspeichern. Es bildet dort eine *Datei*, eine zusammengehörige Menge von Daten, die gemeinsam wiedergelesen werden können. Auch die synonyme Bezeichnung *File* ist hierfür üblich:

■ Ein Quellprogramm wird in Form einer *Quelldatei* (eines *Quellfiles*) auf einem externen Zusatzspeicher abgelegt.

Diesen Arbeitsgang bezeichnet man als *Erfassung* des Quellprogramms. Sowohl dafür als auch für meist erforderliche Korrekturen wird ein Textverarbeitungsprogramm benötigt. Bei Mikrorechnern wird es i. allg. als Editor bezeichnet.

■ Ein *Editor* ist ein Textverarbeitungsprogramm zur Erfassung und zur Korrektur von Quellprogrammen.

Editoren für Programmiersprachen sind meist *zeilenorientiert*, das heißt, es wird immer jeweils eine Textzeile bearbeitet. Die entsprechenden Funktionen können vom Programmierer im Dia-

logbetrieb durch *Kommandos* aufgerufen werden:

Erfassen (jeweils einer Zeile)
Streichen (von Zeilen)
Einfügen (zusätzlicher Zeilen)
Ändern (jeweils einer Zeile):
    Streichen (von Zeichen)
    Einfügen (zusätzlicher Zeichen)
    Austauschen (von Zeichen).

Auf weitere Einzelheiten wird im Abschnitt 3.4 im Zusammenhang mit BASIC-Programmen detailliert eingegangen.

Bessere Arbeitsmöglichkeiten bieten Editoren, die einen größeren Ausschnitt aus dem zu bearbeitenden Text auf dem Bildschirm anzeigen. Für die zeilenorientierte Sprache BASIC sind sie aber nicht unbedingt erforderlich.

2.4.2. Übersetzen

Das erfaßte Quellprogramm muß nun übersetzt werden, damit es der Computer versteht. Dazu dienen Übersetzerprogramme. Man unterscheidet verschiedene Arten.

- Ein *Assembler* ist ein Programm, das ein in der Assemblersprache geschriebenes Quellprogramm in ein Maschinenprogramm übersetzt.

Er geht dabei im Prinzip folgendermaßen vor, wobei hier Feinheiten und Varianten nicht interessieren:

- Zunächst sieht sich der Assembler das Quellprogramm von Anfang bis Ende durch, sammelt (engl.: *to assemble*) dabei die vom Nutzer eingeführten, symbolischen Namen von Speicherzellen und ordnet ihnen selbständig die entsprechenden numerischen Adressen zu.
- Anschließend arbeitet der Assembler das Programm noch einmal durch und ersetzt alle Textsymbole (für Maschinenbefehle, Register und Speicherzellen) durch Bitmuster des Maschinenkodes. Dabei wird die im ersten Durchlauf erstellte *Symboltabelle* mit benutzt.

Die Arbeit eines Assemblers ist also nicht schwer, weil Assembleranweisungen und Maschinenkode nur zwei verschiedenen Darstellungsformen desselben Sachverhalts sind. Im Gegensatz dazu ist die Aufgabe eines Interpreters bzw. eines Compilers viel komplizierter, denn er muß Programme, die in einer höheren Programmiersprache geschrieben sind, für einen gegebenen Computer verständlich machen. Diese Übersetzer kennen zwei ganz verschiedene Sprachen: die betreffende (maschinenunabhängige) höhere Programmiersprache (Quellsprache) und die jeweilige Maschinensprache (Zielsprache), und sie müssen einen echten Übersetzungsvorgang realisieren. In diesem Zusammenhang haben sie zunächst zwei Aufgabenkomplexe zu erledigen, bis sie begriffen haben, was der Programmierer in seinem Programm ausdrücken wollte:

- Der Quelltext muß überhaupt erst einmal gelesen werden; die einzelnen Sprachelemente müssen erkannt werden (*lexikale Analyse*). Auch bei der Übersetzung eines fremdsprachigen Textes würde man so vorgehen. Dabei könnte unter anderem bemerkt werden, daß der Text Elemente enthält, die in der betreffenden Sprache nicht definiert sind und also nicht übersetzt werden können.
- Weiterhin müssen die einzelnen erkannten Elemente in ihrem Zusammenhang untersucht werden. Unter Beachtung der hierfür definierten Sprachregeln ist festzustellen, welche Sprachkonstruktionen aufgebaut worden sind (*syntaktische Analyse*). Auch hierbei können Fehler festgestellt werden, so daß der Inhalt der Sprachanweisungen sich nicht eindeutig erkennen läßt.

Nach dem Abschluß dieser Arbeiten weiß der Übersetzer erst einmal darüber Bescheid, was im Programm steht. Wie oft er es zu diesem Zweck (von Anfang bis Ende) durchmustern mußte, hängt von der Kompliziertheit und von der Darstellungsform der betreffenden höheren Programmiersprache ab.

Nun geht es darum, daß der Übersetzer sein Wissen über den Inhalt des Programms in geeigneter Weise an den Computer weitergibt. Dabei unterscheiden sich Interpreter und Compiler, wie bereits in der Einleitung dieses Abschnitts erwähnt wurde:

- Ein *Interpreter* ist ein Übersetzerprogramm, das jede erkannte Sprachanweisung sofort mit Hilfe entsprechender Funktionsmoduln (Unterprogramme) vom Computer ausführen läßt.

Dieses Prinzip läßt sich offensichtlich nur dann konsequent und wirkungsvoll anwenden, wenn die höhere Programmiersprache derart einfach aufgebaut und günstig dargestellt ist, daß der Interpreter das Programm (möglichst) nicht erst bis zum Ende durchlesen muß, bevor er die ersten Anweisungen übersetzen kann. Eine solche für Interpreter gut geeignete Sprache ist BASIC. Hier weiß der Interpreter beim Lesen jeder Zeile ganz genau, welche Aktionen er dem Computer zu übertragen hat, ohne daß er die folgenden Teile des Programms bereits kennt.

Diese Vorgehensweise hat aber auch Nachteile. Hat der Interpreter eine Anweisung ausführen lassen, so vergißt er sie und geht zur nächsten über. Kommt er später wieder einmal an diese Stelle, so übersetzt er die betreffende Anweisung erneut. Da Übersetzen eine komplizierte und daher zeitaufwendige Arbeit ist, kommt ein Interpreter recht langsam voran.

Als Beispiel soll wieder einmal das bereits mehrfach diskutierte Sortierverfahren herangezogen werden. Wird ein Datenfeld mit 100 Elementen zugrunde gelegt, so benötigt man folgende Sortierzeiten (Bürocomputer A5120):

Maschinenprogramm 2.1 bzw. 2.2:	131...171 Millisekunden
BASIC-Programm 2.3 (MBASIC-Interpreter):	70...143 Sekunden.

Anders geht ein Compiler vor:

- Ein *Compiler* übersetzt ein in einer höheren Programmiersprache kodiertes Quellprogramm geschlossen in ein Zielprogramm, z. B. in ein Maschinenprogramm.

Die Information an den Computer (die Ausführung des Programms) wird also so lange aufgeschoben, bis der Quelltext vollständig in die Maschinensprache übertragen wurde. Das hat zunächst den Vorteil, daß eine Anweisung beim selben Programmlauf nicht mehrfach übersetzt werden muß. Bereits dadurch wird die gesamte Arbeitszeit verkürzt. Vor allem aber ist bei jeder weiteren Abarbeitung keine erneute Übersetzung erforderlich. Dabei ist zu bedenken, daß das Interpretieren eines Quellprogramms erheblich mehr Zeit erfordert als das Abarbeiten des entsprechenden Maschinenprogramms!

Bei dieser Sachlage wird man sich sicher fragen, ob Interpreter überhaupt eine Daseinsberechtigung haben. Es ist aber im Gegenteil so, daß sie immer weiter vordringen! Ihr entscheidender Vorteil liegt nämlich darin, daß sich das eingegebene Quellprogramm sofort mit Hilfe des Computers auf seine Richtigkeit hin überprüfen läßt. Wird ein Fehler aufgespürt, so kann man ihn korrigieren und das Programm gleich wieder interpretieren lassen. Der Programmtest geht also recht zügig voran. Im Gegensatz dazu sind beim Compilieren viele Arbeitsschritte erforderlich, bis der Programmierer ein fehlerhaftes Quellprogramm korrigiert, übersetzt und erneut als Maschinenprogramm gestartet hat. Aus den hier gebrachten Darlegungen läßt sich das gar nicht in vollem Maß erkennen. Wer bereits praktisch gearbeitet hat, wünschte sich für jede Sprache zum Prüfen der entwickelten Programme einen Interpreter. Leider läßt sich dieser Wunsch nicht realisieren. BASIC aber bietet diese Möglichkeit. Obgleich im Regelfall mit Interpretern gearbeitet wird, so gibt es aber auch BASIC-Compiler. Man kann also ein BASIC-Programm zunächst mit Hilfe des Interpreters erproben. Ist es fehlerfrei, so wird es durch den Compiler in ein Maschinenprogramm übersetzt, das wesentlich kürzere Abarbeitungszeiten erfordert. Dadurch wird die Möglichkeit geschaffen, die Programmiersprache BASIC auch in zeitkritischen Fällen einzusetzen, beispielsweise in Regelungseinrichtungen.

Die von Assemblern und Compilern erzeugten Maschinenprogramme wird man sich in der Regel für eine wiederholte Abarbeitung aufheben wollen:

- Ein Maschinenprogramm wird in Form einer *Maschinenkodedatei* (eines *Programmfiles*) auf einem externen Zusatzspeicher abgelegt.

Es wird von dort durch ein Kommando des Bedieners an das Betriebssystem geholt und anschließend gestartet.

2.5. Testen

Im vorigen Abschnitt wurde darauf hingewiesen, daß die Übersetzerprogramme bereits Fehler im Quelltext feststellen können. Es handelt sich dabei um Verstöße gegen die festgelegte Lexik (nichtdefinierte Sprachelemente benutzt) und Syntax (unzulässige Sprachkonstruktionen aufgebaut):

- *Syntaktische Fehler* sind Verstöße gegen die Regeln, die den formalen Aufbau der Sprache definieren.

Aber auch nach der Beseitigung syntaktischer Fehler könnte das Programm möglicherweise immer noch nicht richtig arbeiten, beispielsweise vorzeitig abbrechen, nicht zum Ende gelangen oder falsche Resultate erzeugen. Es entspräche dann nicht der vorgeschriebenen Spezifikation. Solche Fehler liegen in der Logik des Programms:

- *Logische Fehler* beruhen darauf, daß entweder falsche Algorithmen benutzt *(Verfahrensfehler)* oder richtige Algorithmen fehlerhaft kodiert wurden *(Programmfehler)*.

Um solche Fehler aufzuspüren, muß das Programm geprüft (*getestet*) werden:

- *Testen* nennt man das wiederholte Ausführen eines Programms mit der Absicht, Fehler zu finden. Das einmalige Abarbeiten des Programms mit geeigneten Testdaten zu Testzwecken heißt *Testfall*.

Findet man einen Fehler, so war der Testfall erfolgreich; andernfalls war er ein Mißerfolg. Während das Implementieren eine konstruktive Tätigkeit darstellt, könnte man das Testen als *destruktiv* bezeichnen. Ein Programmierer ist daher eigentlich nicht in der Lage, sein eigenes Programm vorurteilsfrei zu testen.

Sicher sind diese Probleme vor allem für komplexe Programmsysteme von entscheidender Bedeutung. Trotzdem sollen auch an dieser Stelle noch einige Bemerkungen zur Vorgehensweise beim Programmtest gemacht werden.

2.5.1. Vorbereitung

Das Testen beginnt genaugenommen nicht erst nach der Implementierung, sondern bereits davor. So ist es zweckmäßig, das kodierte Programm vor der maschinenlesbaren Erfassung noch einmal in Ruhe durchzusehen und Testfälle am Schreibtisch durchzuspielen *(Trockentests)*. Dabei sind kollektive Diskussionen wertvoll. Erst danach sollte das Programm implementiert werden.

Es ist weiterhin zu empfehlen, in das Programm von vornherein *Prüfpunkte* einzubauen. Es handelt sich dabei um Kontrollausgaben, die an wichtigen Stellen des Programmablaufs, z. B. an Schnittstellen zwischen Moduln, die Werte von dort interessierenden Größen ausgeben. Um sich vor einer Informationsflut zu schützen, kann man diese Ausgaben auch so programmieren, daß sie nur bei Abweichungen vom erwarteten Verhalten erfolgen. Am Ende der Testarbeiten nimmt man dann diese Ausgabeanweisungen wieder aus dem Programm heraus.

In dem schon mehrfach diskutierten Sortieralgorithmus wären beispielsweise die beiden Rücksprünge im Bild 2.8 geeignete Stellen für solche Testausgaben. In der inneren Schleife könnte man prüfen, ob tatsächlich der erste der beiden verglichenen Partner den kleineren Wert erhalten hat. Nach einem Durchlauf durch die äußere Schleife müßte der erste Partner den kleinsten Wert im Vergleich zu allen Speicherzellen mit höheren Adressen besitzen.

2.5.2. Testfälle

Ist die Implementierung abgeschlossen, so können praktische Testfälle auf dem Computer realisiert werden. Bei diesem empirischen Probieren geht man folgendermaßen vor:

- Es werden geeignete *Eingabedaten* ausgewählt.
- Die nach der Spezifikation *zu erwartenden* Resultate sind zu ermitteln und aufzuschreiben.
- Es wird ein *Testlauf* unter Verwendung der Testdaten durchgeführt.

Programm 2.4. Testrahmen für das BASIC-Programm 2.3

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"       TESTRAHMEN FUER EIN SORTIERPROGRAMM       "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++ TESTRAHMEN +++++++++++++++++++
1080 :
1090 READ N   : REM Anzahl von Elementen
1100 DIM V(N+1) : REM Datenfeld
1110 LET I = 1
1120 READ V(I)
1130 LET I = I+1
1140 IF I<=N THEN GOTO 1120
1150 GOSUB 1440 : REM Ausdrucken
1160 GOSUB 1240 : REM Sortieren
1170 GOSUB 1440 : REM Ausdrucken
1180 PRINT "Sortieren beendet."
1190 PRINT
1200 END
1210 :
1220 REM +++++++++++++++++++ SORTIEREN +++++++++++++++++++++
1230 :
1240 LET I = 1
1250 LPRINT
1260 LET J = I
1270 LET J = J+1
1280 IF V(I)>V(J) THEN GOSUB 1370 : REM Vertauschen
1290 GOSUB 1450 : REM Ausdrucken
1300 IF J<N THEN GOTO 1270
1310 LET I = I+1
1320 IF I<N THEN GOTO 1250
1330 RETURN
1340 :
1350 REM ------------------- VERTAUSCHEN --------------------
1360 :
1370 LET Z = V(I)
1380 LET V(I) = V(J)
1390 LET V(J) = Z
1400 RETURN
1410 :
1420 REM ----------- AUSDRUCKEN DES DATENFELDS ------------
1430 :
1440 LPRINT
1450 LET K = 1
1460 LPRINT V(K) ;
1470 LET K = K+1
1480 IF K<=N THEN GOTO 1460
1490 LPRINT
1500 RETURN
1510 :
1520 REM ------------------- TESTWERTE --------------------
1530 :
1540 DATA 7
1550 DATA 57,42,33,99,16,33,28
1560 :
1570 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
57  42  33  99  16  33  28

42  57  33  99  16  33  28
33  57  42  99  16  33  28
33  57  42  99  16  33  28
16  57  42  99  33  33  28
16  57  42  99  33  33  28
16  57  42  99  33  33  28

16  42  57  99  33  33  28
16  42  57  99  33  33  28
16  33  57  99  42  33  28
16  33  57  99  42  33  28
16  28  57  99  42  33  33

16  28  57  99  42  33  33
16  28  42  99  57  33  33
16  28  33  99  57  42  33
16  28  33  99  57  42  33

16  28  33  57  99  42  33
16  28  33  42  99  57  33
16  28  33  33  99  57  42

16  28  33  33  57  99  42
16  28  33  33  42  99  57

16  28  33  33  42  57  99

16  28  33  33  42  57  99
```

- Die *tatsächlichen* Resultate werden mit den erwarteten verglichen und eventuelle Unterschiede sorgfältig analysiert.

Die Testfälle kann man zunächst einmal so festlegen, daß man eine Übereinstimmung mit der Spezifikation prüft, ohne die interne Struktur des Programms zu beachten. Dabei sollten folgende Fälle untersucht werden:

- Es sind *typische* Beispiele für alle Aufgaben zu wählen, die laut Spezifikation zu lösen sind.
- Weiterhin sind Testdaten für *Sonder- und Grenzfälle* des Programms vorzusehen (beispielsweise erster und letzter zulässiger Wert, Eingabe des Werts null).
- Die Robustheit des Programms ist durch *unzulässige* Eingaben zu prüfen.
- Schließlich sind auch noch *zufällige,* vielleicht unsinnig erscheinende Testdaten zu benutzen. Sie können helfen, Lücken aufzudecken, die durch Routine oder durch eigentlich selbstverständliche, hier aber unzutreffende Annahmen des Programmierers entstanden sind.

Diese Testfälle sind zu ergänzen durch solche, die die interne Programmstruktur beachten. Man testet hier durch ein bewußtes Abarbeiten von Programmzweigen, den sog. *Pfaden.* Dabei sind folgende Mindestforderungen zu beachten:

- Jede *Anweisung* des Programms ist mindestens einmal auszuführen.
- Es ist jede mögliche *Bedingung* auszuwerten und jede mögliche *Entscheidung* zu fällen, und zwar mindestens einmal und unter Berücksichtigung eventueller Kombinationen *(Mehrfachbedingungen).*

Bei allem Aufwand für das Prüfen eines Programms muß man aber leider feststellen, daß ein vollständiges Testen praktisch unmöglich ist. Daher kann man auch keine Garantie dafür geben, daß ein Programm fehlerfrei ist. Man kann nur beweisen, daß Fehler vorhanden sind, nicht aber, daß sie alle beseitigt wurden!

Als Beispiel soll das BASIC-Programm 2.3 mit einem Testrahmen umgeben werden. Das entsprechende **Programm 2.4** für einen Testfall enthält natürlich eine Reihe von weiteren, noch unbekannten BASIC-Anweisungen. Trotzdem soll es an dieser Stelle nicht ausführlich kommentiert werden. Der Leser soll versuchen, es wie einen Text zu lesen und zunächst Vermutungen

über die Bedeutung der einzelnen Sprachelemente anstellen. Es sei nur soviel gesagt, daß im ersten Teil die vorgegebenen Testdaten in das Feld eingelesen werden. Vor Beginn des Sortierens und nach jedem Teilschritt der inneren Schleife wird das Datenfeld zur Kontrolle ausgedruckt. Nachdem ein erster Partner mit allen hinter ihm gespeicherten verglichen worden ist, wird eine Leerzeile eingeschoben. Die entstehenden Ausgaben sind nach dem Quelltext im Programm 2.4 angegeben. Die verwendeten Testwerte (DATA-Anweisungen) lassen sich leicht austauschen.

2.5.3. Modultest und Integrationstest

Bisher wurde nur über die Testmethodik gesprochen, aber nicht über den Testgegenstand. Wie bereits erwähnt, sollten größere Programme in Moduln unterteilt implementiert werden, um das Prüfen zu erleichtern. Dabei geht man zweckmäßigerweise folgendermaßen vor (Bild 2.6):

- Zuerst werden diejenigen Moduln für sich allein untersucht *(Modultest)*, die auf keine anderen Moduln zurückgreifen (Moduln 1.1, 1.2 und 1.3.1.1).
- Als nächstes bezieht man die übergeordneten Moduln in den Test mit ein, wobei jeweils alle untergeordneten Moduln bereits geprüft sein müssen (Moduln 1.3 und 1.3.1). Die Testfälle betreffen jetzt sowohl die *Funktionen* des übergeordneten Moduls als auch alle *Schnittstellen* zu den untergeordneten Moduln (Integrationstest).
- Dieses Verfahren wird *schrittweise wiederholt,* bis sämtliche Moduln miteinander verknüpft sind.

Man geht bei diesem Verfahren also von den elementaren Funktionen aus und faßt sie zu immer komplexeren zusammen; man baut das Programm gewissermaßen *von unten nach oben* auf. Daher wird diese Vorgehensweise auch als *Bottom-up-Synthese* bezeichnet.

Ein abschließender Funktionstest dient dem vollständigen Vergleich des Programmverhaltens mit der Spezifikation. Eine besondere Rolle spielen dabei die vorgegebenen Qualitätsparameter, wie z. B. übersichtlicher Aufbau, kurze Laufzeiten und nutzerfreundliche Einsatzmöglichkeiten.

2.5.4. Häufige Fehlerquellen

Im folgenden soll stichwortartig eine Reihe von typischen Fehlern zusammengestellt werden. Es ist klar, daß die Kenntnis von Fehlermöglichkeiten allein noch nicht dazu führt, Fehler zu vermeiden. Vielleicht sind die Darlegungen aber insofern von Interesse, daß bestimmte Gefahrenschwerpunkte aufgelistet werden:

- Semantische Fehler
  - falsches Vorzeichen
  - falsche Zahlen oder falsche Variablenbezeichnungen
  - falsche Operations- oder Funktionsbezeichnung
  - falsches Sprungziel
  - fehlende Anfangswertzuweisung bei Variablen
- Abbruchfehler in Schleifen
  - Schleifen mit vorgegebener Anzahl von Durchläufen werden einmal zu viel oder einmal zu wenig abgearbeitet.
  - Infolge von Rundungsfehlern bei Berechnungen kann ein vorgegebener Wert nicht exakt erreicht werden.
- Schnittstellenfehler
  - Derselbe symbolische Name wird an verschiedenen Stellen des Programms für verschiedene Variablen benutzt.
  - Zwischen Haupt- und Unterprogramm besteht ein Mißverständnis über die zu verwendenden Parameter.

- Verfahrensfehler
  - Das Bilden der Differenz zweier großer Zahlen, deren Werte benachbart sind, führt zum Verlust an Genauigkeit.
  - Wenn sich bei Berechnungen die Rundungsfehler aufschaukeln, erreicht man nicht die gewünschte Genauigkeit, sondern nur unbrauchbare Resultate.
- Fehleingaben
  - falsche Werte
  - zuviel/zuwenig Werte.

An dieser Stelle soll die Übersicht abgebrochen werden. Sie sollte auf Fehlerquellen hinweisen, kann aber nicht die eigene praktische Arbeit ersetzen. Und schließlich lernen wir aus unseren Fehlern!

2.5.5. Lokalisierung und Korrektur von Fehlern

Hat man auf einem der genannten Wege festgestellt, daß ein Fehler vorhanden ist, muß man ihn zunächst einmal finden. Im allgemeinen wird man dabei folgendermaßen vorgehen:

- Aus der fehlerhaften Reaktion des Programms versucht man auf die mögliche Ursache rückzuschließen. Dabei können die oben genannten *Testausgaben* recht nützlich sein, weil sie bereits eine gewisse Lokalisierung der Fehlerauswirkung, wenn leider auch noch nicht der Fehlerquelle, gestatten. Durch spezialisierte Testfälle, bei denen jeweils wenig veränderte Pfade durchlaufen werden, lassen sich weitere Informationen gewinnen.
- Oft kann man die Fehlerquelle nur schrittweise lokalisieren. Man muß dann in den Modul hineingehen und einzelne Pfade genauer untersuchen. Hier bewähren sich *Testhilfsmittel,* mit denen man auf dem Programmpfad liegende Anweisungen protokollieren oder die Abarbeitung an gewünschten *Haltepunkten* stoppen kann. Im Abschnitt 3.6 werden entsprechende BASIC-Anweisungen besprochen.
- Beim *Korrigieren* mit Hilfe des Editors ist größte Zurückhaltung zu üben. Oft glaubt man irrtümlicherweise, den Fehler bereits gefunden zu haben und ändert vorschnell. Erfahrungsgemäß schadet dieser blinde Eifer sehr häufig. Man sollte nur das korrigieren, was eindeutig als falsch nachgewiesen wurde! Das korrigierte Programm ist anschließend zu übersetzen und erneut zu testen, woran sich möglicherweise weitere Korrekturen anschließen.

2.6. Dokumentieren

Im Verlauf der Implementierung sind auch die erforderlichen Dokumentationen anzufertigen. Diese Arbeit ist bei Programmierern recht unbeliebt und wird deshalb häufig immer wieder aufgeschoben oder ganz ignoriert. Das ist einerseits kurzsichtig, weil man oft selbst wieder darauf angewiesen ist; andererseits behindert man damit den Austausch von Programmen, von dem man schließlich selbst profitiert.
Auch für kleinere Programme sind folgende Dokumentationen zweckmäßig:

Bedienanleitung für den Einsatz des Programms
Programmbeschreibung für die Wartung.

Auf jeden Fall empfiehlt sich mindestens eine angemessene Kommentierung des Programmtextes.

2.6.1. Kommentieren des Quellprogramms

Jedes Quellprogramm ist in bestimmtem Maß lesbar, obgleich das bei Anwendung einer höheren Programmiersprache viel einfacher möglich ist als bei der Assemblersprache. Man kann die Anschaulichkeit aber wesentlich durch eine geeignete, übersichtliche *Gliederung des Programmtextes* erhöhen. Dazu dienen Absätze (leere Zeilen), trennende Zeilen mit Sonderzeichen, wie

Strichen oder Sternen, und leicht zu erkennende Überschriften. Dabei sollten dieselben Begriffe und Abkürzungen wie in den Entwürfen (Programmablaufplänen) benutzt werden.

Kommentare sind auch bei höheren Programmiersprachen nicht überflüssig. Sie sind unter anderem an folgenden Stellen zweckmäßig:

Bedeutung von vereinbarten *Datenstrukturen*
Funktion von *Unterprogrammen*, benutzte Parameter
Zweck und Wirkungsweise komplizierter *Programmkonstruktionen.*

Als Beispiel für eine entsprechende Gestaltung eines Quellprogramms kann das Programm 2.4 dienen.

2.6.2. Programmbeschreibung

Häufig müssen vorhandene Programmsysteme modifiziert werden. So sind sie beispielsweise an bestimmte Einsatzforderungen oder -bedingungen anzupassen, für neue Aufgaben zu erweitern; vielleicht sind auch bisher verborgen gebliebene Fehler entdeckt worden und zu beseitigen. Für solche Zwecke benötigt man folgende Angaben:

- Beschreibung der *Aufgabe*, die gelöst werden sollte;
- Überlegungen, die im Rahmen der *Problemanalyse* angestellt worden sind, und die auf dieser Basis entstandene *Spezifikation* (Abschn. 2.1);
- Vorgehen beim problemorientierten *Systementwurf*, herauskristallisierte *Funktionen* und ihre Beziehungen zueinander (Abschn. 2.2.1);
- Zusammenfassung der Funktionen in zu implementierenden *Moduln*, Beziehungen zwischen den Moduln (*Schnittstellen* und *Hierarchien*), zugrunde gelegte *Datenstrukturen* und benutzte *Algorithmen* (Abschn. 2.2.2);
- Zweckmäßigerweise sind ausgewählte Entwurfsunterlagen (z. B. *Programmablaufpläne*) beizufügen.

Diese Darlegungen dürfen nicht zu umfangreich sein. Das Niveau sollte hinreichend abstrakt sein, um einen Überblick vermitteln zu können. Die Beziehung zum (kommentierten) Quellprogramm muß jedoch gewahrt bleiben.

2.6.3. Bedienanleitung

Während die Programmbeschreibung für den (System-)Programmierer bestimmt ist, der das Programm pflegen soll, wendet sich die Bedienanleitung an den Anwender, der das Programm zur Lösung seiner Aufgaben benötigt und es einsetzen möchte. Hier ist eine andere Auswahl von Angaben erforderlich:

- Zweck des Programms, *Anwendungsmöglichkeiten*, Grenzen des Einsatzbereichs
- Hinweise zur *Installation* des Programms auf einem Rechner: erforderliches Betriebssystem (z. B. CP/M), benötigter Umfang an Speicherplatz, besondere Geräte (z. B. Magnetbandkassettengeräte) usw.
- Angabe zur *Inbetriebnahme*, beispielsweise Kommandos zum Starten und zum Abbrechen des Programmlaufs, Testbeispiele
- Hinweise zur *Nutzung* des Programms, Übersicht über die zulässigen Kommandos des Bedieners und die möglichen Meldungen des Programms
- Bedeutung, Form und zulässige bzw. mögliche Werte der *Eingabe- und Ausgabedaten*, Aufzeichnungsformate auf externen *Speichermedien.*

Bei diesen Hinweisen ist auf den Anwender Rücksicht zu nehmen! Er möchte kein Zusatzstudium absolvieren, nur um das Programm einsetzen zu können. So sollten zulässige Bedienerkommandos thematisch geordnet angegeben werden, nicht aber alphabetisch! Eine solche Reihenfolge ist dagegen für die Erläuterung möglicher Meldungen des Programms zweckmäßig. Die

Materialien sollten nicht allzu umfangreich sein, sonst kann man nicht mit ihnen arbeiten. Ein nutzerfreundliches Programm benötigt nicht viel Bedienhinweise!

Manchmal werden erläuternde Texte für den Anwender direkt in das Programm aufgenommen. Man ruft sie mit Hilfe eines Kommandos (HELP) ab und läßt sie sich auf dem Bildschirm anzeigen. Diese Ausgaben können auch automatisch erfolgen, wenn das Programm bemerkt, daß der Bediener unsicher ist. Das ist bequem, erfordert aber für jede Schreibmaschinenseite etwa 2000 Byte. Diesen Luxus kann man sich daher nur dann leisten, wenn man über einen externen Direktzugriffsspeicher (z. B. eine Diskette) verfügt.

Schließlich freut sich jeder Nutzer auch, wenn der Bedienanleitung noch ein vollständiges *Beispiel* für einen typischen Einsatzfall beigegeben ist.

3. Programmieren mit BASIC

In den Abschnitten 1 und 2 wurden die Hardware von Mikrorechnern und einige bei der Entwicklung von Software anzuwendende Mittel und Methoden im Überblick dargestellt, um auch Anfängern auf dem Gebiet der Informatik eine Vorstellung von diesen Problemkreisen zu geben. Jetzt wird zur Behandlung der Programmiersprache BASIC übergegangen. Im folgenden Abschnitt werden zunächst einige allgemeine Aussagen zur Arbeit mit dieser Sprache gebracht: Wie sollte ein Programm aufgebaut sein? Wie gelangt es in den Computer hinein? Wie wird es gestartet und wieder angehalten? Auf welchem Wege erhält man ein fehlerfreies und effizientes Programm?

In diesem Abschnitt werden auch die ersten BASIC-Anweisungen eingeführt. Zu ihrer Definition wird jeweils die folgende Darstellungsweise gewählt:

- Festgelegte BASIC-Elemente (wie Schlüsselwörter und Trennzeichen) werden **halbfett** gedruckt. Sie müssen vom Programmierer in unveränderter Form verwendet werden.
- Ergänzende Parameter und problemspezifische Teile werden mit kleinen *kursiven* Buchstaben bezeichnet. Sie sind im konkreten Fall durch die jeweiligen Bezeichnungen, Werte oder dergleichen zu ersetzen.
- In eckigen Klammern angegebene BASIC-Elemente und Parameter kann man wahlweise verwenden oder weglassen.

3.1. Was ist BASIC?

3.1.1. Historische Entwicklung

Die Programmiersprache BASIC wurde in den Jahren 1962 bis 1964 von *John G. Kemeny* und *Thomas E. Kurtz* am Dartmouth College in Hanover/New Hampshire in den USA entwickelt [3.6]. Sowohl die vollständige Bezeichnung

Beginner's All-purpose Symbolic Instruction Code

als auch die Abkürzung BASIC dokumentieren den Zweck dieser Programmiersprache: Sie ist für den Ausbildungsbetrieb bestimmt, wendet sich an den Anfänger und ist einfach erlernbar. Außerdem wurde sie für ein breites Einsatzgebiet entworfen, um auch bei unterschiedlichsten Anwendungsfällen mit einer einzigen Sprache auskommen zu können.

Die folgende Entwicklung zeigte auch in diesem Fall, daß sich bewußt einfach gehaltene und dadurch leicht zu erfassende Konzepte in der Praxis durchsetzen. Die bereits erwähnte, durch die Mikroelektronik ermöglichte Ausbreitung des Mikrorechners in alle Gebiete von Wissenschaft, Technik und Wirtschaft erforderte eine auch von Nichtinformatikern einfach anzuwendende Programmiersprache. In dieser Phase bot sich BASIC an und wurde unter anderem auch von den Herstellern von Personal- und Heimcomputern adaptiert. Die Hersteller wollten aber aus Absatzgründen die Besonderheiten ihrer Produkte herausstellen und boten individuelle Modifikationen in der Sprache an.

Als Folge dieses Prozesses gibt es eine unübersehbare Fülle von Sprachdialekten. Da ein Übersetzerprogramm aber jede Abweichung von der ihm geläufigen Sprachdefinition als syntaktischen Fehler zurückweist, können BASIC-Programme nur in beschränktem Maß von dem Computer des einen Herstellers zu dem eines anderen übergeführt werden [3.2]. Man spricht

manchmal von einem babylonischen Sprachgewirr, wenn man die Situation auf dem Gebiet der Computersprachen insgesamt charakterisieren will. BASIC aber stellt ein Babylon für sich allein dar!

Um aus dieser Situation einen Ausweg zu finden, unternehmen zuständige Institutionen intensive Standardisierungsversuche. Zunächst wurde bereits 1978 ein Standard für ein sog. *Minimal-BASIC* herausgebracht [3.1]. Programme, die nur Sprachelemente von Minimal-BASIC benutzen, müßten danach auf allen Computern in der gleichen Weise abgearbeitet werden können. Das ist aber nur eine geringe Hilfe, weil der Umfang dieser Teilsprache recht gering ist.

Die nächsten Aktivitäten der genannten Institutionen richteten sich nun auf die Standardisierung eines Maximalumfangs, von dem auf konkreten Computern möglicherweise jeweils nur ein Teil *(Subset)* implementiert zu werden brauchte. Dabei ist aber eine sehr große, uneinheitlich aufgebaute Sprache entstanden, die nicht mehr die ursprüngliche Einfachheit von BASIC aufweist [3.4] [3.5] [3.7].

In einem Lehrbuch muß ein Kompromiß gewählt werden. Daher sollen hier im wesentlichen nur diejenigen Sprachelemente besprochen werden, die heutzutage in den am weitesten verbreiteten BASIC-Versionen vorgefunden werden können. Die gewählten Programmbeispiele wurden dabei mit einem MBASIC-Interpreter für den Dialekt BASIC-80 übersetzt, der unter dem Betriebssystem CP/M läuft.

3.1.2. Charakterisierung von BASIC

Es wurde bereits gesagt, daß es viele höhere Programmiersprachen gibt. Jede von ihnen hat ihre Vorteile, die zu ihrer Entwicklung und Verbreitung führten. Aber jede hat auch ihre schwachen Seiten. Wo ist hier BASIC einzuordnen?

BASIC ist eine sehr *einfache* Sprache. Sie ist daher leicht zu erlernen. Je nach der Vorbildung reichen dazu wenige Tage. Dabei hilft auch die Tatsache, daß BASIC *interpretativ* übersetzt wird: Ein BASIC-Interpreter wirkt wie eine rechnergestützte Lehr- und Lernhilfe. Diese Eigenschaften haben dazu geführt, daß BASIC vor allem dort beliebt wurde, wo kleine wissenschaftlich-technische Berechnungen und Arbeiten zur Büroautomatisierung von Nichtinformatikern ausgeführt werden müssen. Jüngste Analysen haben zu dem Ergebnis geführt, daß die weitaus meisten Programme für Mikrorechner in der Welt in BASIC kodiert wurden, mit großem Abstand gefolgt von PASCAL, Commercial Basic, COBOL und FORTRAN. Diese Position wird auch dadurch nicht beeinträchtigt, daß in BASIC meist nur kleinere Programmsysteme implementiert werden. Schließlich ist BASIC die führende Sprache für Heimcomputer geworden, und die hier geschriebenen Programme werden in keiner Statistik erfaßt!

BASIC ist eine relativ *maschinennahe* Sprache. So wie in einem Computer die Speicherzellen (und damit die darin enthaltenen Befehle) numeriert sind, so geschieht es in BASIC mit den Quellprogrammzeilen (und damit mit den darin stehenden Anweisungen). Alle benutzten Variablen sind in BASIC – wie in der Assemblersprache – von jeder Stelle her nutzbar. Natürlich sind die Anweisungen einer höheren Programmiersprache wie BASIC komplexer als einzelne Befehle des Computers, aber ein BASIC-Programm spiegelt die interne (physische) Ablaufstruktur deutlich wider. Der Sprungbefehl ist in BASIC direkt als GOTO-Anweisung enthalten. Eine strukturierte Programmierung wird wenig unterstützt. Deshalb wurden in neueren Sprachdialekten zusätzliche Sprachelemente von höheren Programmiersprachen eingeführt. Sie wirken aber wie Fremdkörper in BASIC, dürfen häufig nur unter Beachtung von Einschränkungen verwendet werden und erschweren dadurch das Erlernen und den Einsatz von BASIC. Manche anderen Erweiterungen sind dagegen recht zweckmäßig; sie werden auch im folgenden mit benutzt.

3.2. Arbeiten mit BASIC

Im folgenden soll auf die Arbeitsweise eines Programmierers eingegangen werden, der einen Computer vor sich stehen hat, auf dem ein BASIC-System implementiert ist. Die grundsätzliche

Struktur eines solchen Systems ist im **Bild 3.1.** dargestellt. Wie kommt der Programmierer als Bediener des Rechners nun mit diesem System in Kontakt? Welche Möglichkeiten hat er, die Arbeitsmodi zu beeinflussen? Was unterscheidet ein Bedienerkommando von einer Programmzeile?

Diese Fragen sind Inhalt dieses Abschnitts.

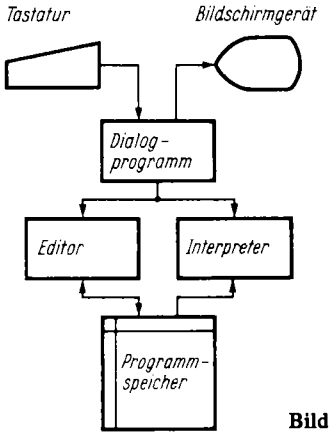


Bild 3.1. Datenfluß in einem BASIC-System

3.2.1. Dialogbetrieb

Wenn man einen Computer einschaltet, meldet sich zunächst das *Betriebssystem* über eine Ausschrift auf dem Bildschirm und fordert ein Bedienerkommando an. Wie bereits im Abschnitt 1.5.3 besprochen, kann man auf diesem Wege auch eigene Programme starten; in unserem Fall wird das BASIC-System gebraucht. Nach Eingabe des entsprechenden Namens wird dieses Programm vom externen Zusatzspeicher in den Hauptspeicher geladen und gestartet. Es meldet sich ebenfalls auf dem Bildschirm und fragt seinerseits nach den Wünschen des Bedieners. In dieser Phase könnte der Bildschirm folgendes Aussehen haben:

```
CP/M 2.2
A>MBASIC
BASIC-80 Rev.5.2
31558 Bytes free
Ok
```

Dabei wurde der vom Bediener geschriebene Teil durch eine Einrahmung hervorgehoben. Die Ausschrift OK zeigt an, daß nunmehr ein *BASIC-Kommando* erwartet wird. Andere Systeme schreiben statt dessen HALLO, READY oder DONE aus. Manchmal wird auf der neuen Zeile noch durch ein bestimmtes Zeichen (zum Beispiel: >) darauf hingewiesen, daß ein Kommando des Programmierers einzugeben ist.

Verschiedene Interpreter wünschen in dieser Phase zunächst Angaben des Bedieners über den *Umfang des Speichers*, der vom BASIC-System benutzt werden darf. Das ist in solchen Fällen von Interesse, wenn dem Programmierer ein Teil des Speichers zur unmittelbaren Nutzung reserviert bleiben soll (Abschn. 10.5.1).

Der Bediener gibt jetzt über die Tastatur seine Kommandos ein. Dabei sind *Großbuchstaben* erforderlich. Einige BASIC-Systeme gestatten kleine Buchstaben und wandeln sie selbständig in große um. Die eingetasteten Zeichen werden zunächst in einem *Eingabepuffer* zwischengespeichert und gleichzeitig auf dem Bildschirm angezeigt. Die Länge dieses Puffers hängt vom konkreten System ab und beträgt 72 bis 255 Zeichen; evtl. also mehr, als auf einer Bildschirmzeile dargestellt werden können.

Nun sollte der Programmierer seine Eingabe zunächst noch einmal prüfen, denn jetzt ist noch eine *Korrektur* möglich. Dabei läßt sich durch Drücken einer dafür bestimmten Taste (BS, CTRL-H; evtl. auch DEL; siehe Tafel A.4) das jeweils letzte Zeichen im Puffer löschen. Bei bildschirmorientierten Systemen verschwindet das Zeichen dann auch auf dem Schirm. Möchte man die gesamte Zeile löschen, muß man eine andere Taste betätigen (CAN, CTRL-X; evtl. auch CTRL-U).

Manche BASIC-Systeme sehen für die Korrektur spezielle *Tasten* vor:

- Mit ← und → kann die *Schreibmarke* innerhalb der Zeile nach links bzw. rechts *verschoben* werden.
- Mit DEL (DELete) *löscht* man das markierte *Zeichen,* mit CL LN (CLear LiNe) die ganze *Zeile.*
- Durch INS (INSert) kommt man in den Einfügungszustand. Alle jetzt eingegebenen *Zeichen* werden vor der aktuellen Position der Schreibmarke *eingeschoben.*

Ist das BASIC-Kommando nunmehr nach Ansicht des Bedieners fehlerfrei, so muß er die auf seinem Rechner festgelegte *Zeilenendetaste* betätigen. Dabei treten recht vielfältige Kennzeichnungen auf: ENTER, ET (End of Text), EL (End of Line), RET (RETurn), CR (Carriage Return) und andere.

Nach diesem Tastendruck wird die eingegebene Zeile vom Dialogprogramm an den *Interpreter* übergeben. Dieser analysiert sie und führt sie anschließend aus; evtl. veranlaßt er auch eine Fehlermeldung auf dem Bildschirm. Danach zeigt das Dialogprogramm wieder in der bereits besprochenen Form seine Bereitschaft an, weitere Kommandos entgegenzunehmen.

Hat der Programmierer seine Arbeit mit BASIC beendet und will wieder mit dem Betriebssystem in Verbindung treten, so muß er das Kommando

SYSTEM

verwenden. Der Interpreter schließt dann alle eingeleiteten Maßnahmen ab und übergibt die weitere Steuerung wieder dem *Betriebssystem.* Dessen Bedienerverständigung fragt daraufhin den Programmierer nach seinen weiteren Wünschen. Manche Interpreter verwenden an dieser Stelle die Kommandos BYE oder QUIT.

3.2.2. Anweisungen

Bei den *BASIC-Anweisungen* sind zwei Klassen zu unterscheiden:

- *Systemanweisungen* bestimmen die Arbeitsweise des BASIC-Systems und dienen zur Bearbeitung von BASIC-Programmen.

Ein Beispiel dafür ist die im vorigen Abschnitt behandelte Systemanweisung SYSTEM. Eine Zusammenstellung der wichtigsten Systemanweisungen ist in der Tafel A.5 enthalten. Einige Mikrocomputer, die vorrangig zur Nutzung von BASIC bestimmt sind, haben für die am häufigsten benötigten Systemanweisungen spezielle *Steuertasten;* manche sogar auch für die Sprachanweisungen. Dadurch verringert sich der Aufwand beim Eingeben von Kommandos und Programmen beträchtlich.

- *Sprachanweisungen* dienen der Formulierung von Verarbeitungsschritten, die mit Hilfe von BASIC-Programmen erfolgen sollen.

Wird beispielsweise im Anschluß an die im Abschnitt 3.2.1 besprochenen Startschritte das Kommando PRINT 5 + 7 erteilt, so ergeben sich auf dem Bildschirm folgende Ausschriften:

```
PRINT 5 + 7
 12
Ok
```

Der Rechner ermittelt also den Wert des Ausdrucks 5 + 7 und schreibt ihn aus. Auch längere

Aufgaben lassen sich durch direkte Bedienereingaben lösen:

```
PRINT "Seitenlaenge"
Seitenlaenge
Ok
INPUT A,B
?5,7
Ok
LET F = A * B
Ok
PRINT F
 35
Ok
```

Offensichtlich muß der Rechner die Resultate nicht unbedingt sofort ausdrucken. Er kann sich die Werte merken, die für die *Variablen* eingegeben (A und B) oder errechnet (F) wurden, wobei man diese Variablen mit *allgemeinen Zahlsymbolen* bezeichnet. *Probieren Sie auf der Basis der bisher benutzten Sprachelemente eigene Beispiele!*

Beide Klassen von Anweisungen haben einen bestimmten Aufbau:

Sie beginnen stets mit einem *Schlüsselwort* (Abschn. 3.3.2). Durch diese Struktur erkennt der Interpreter bei der Analyse des Eingabepuffers also sehr schnell, was auszuführen ist, und kann die entsprechenden Teilprogramme einsetzen, die für die weitere Bearbeitung gebraucht werden.

Nach dem Schlüsselwort werden häufig noch *Parameter* angegeben, die zur Spezifizierung der auszuführenden Operation erforderlich sind.

Man kann auch *mehrere* Sprachanweisungen *auf einer Zeile* angeben und demzufolge mit einem einzigen Kommando ausführen lassen. Als Trennzeichen bieten die meisten Interpreter den *Doppelpunkt* an. Zum Beispiel lassen sich folgende Sprachanweisungen zu einem Kommando zusammenfassen:

```
LET A = 3 : LET B = 4 : LET F = A * B : PRINT F
 12
Ok
```

Nicht alle Sprachanweisungen können in Kommandos verwendet werden. Beispiele hierfür sind die Anweisungen REM und END, die bereits im nächsten Abschnitt behandelt werden, und die DATA-Anweisung zur Definition von Eingabedaten (Abschn. 4.2.2).

3.2.3. Arbeitsmodi

Bei der bisher besprochenen direkten Betriebsweise arbeitet der Computer also wie ein Tischrechner:

- Im *Kommandomodus* führt das BASIC-System jede eingegebene Zeile sofort aus.

Der Kommandomodus wird beim Start des BASIC-Systems automatisch eingestellt und ist die normale Arbeitsweise. Wie aber kann man unter diesen Umständen ein Programm entwickeln, speichern und abarbeiten?

- Beginnt eine Eingabezeile mit einer natürlichen Zahl, der *Zeilennummer,* so wird sie als Programmzeile betrachtet und nicht dem Interpreter, sondern dem Editor übergeben, der sie im Programmspeicher ablegt. Das BASIC-System befindet sich dabei in einem *Lernmodus.*

Man kann also den Kommandomodus vorübergehend verlassen und in der angegebenen Weise, Zeile für Zeile, ein Programm aufbauen. Dieses Programm läßt sich mit Hilfe des RUN-Kommandos (Abschn. 3.5.1) starten:

- Im *Programmodus* führt das BASIC-System ein gespeichertes Programm durch *fortlaufende Interpretation* der Sprachanweisungen entsprechend ihrer Numerierung aus.

3.2.4. Aufbau eines BASIC-Programms

Aus dem bisher Gesagten geht hervor, daß man ein BASIC-Programm von vornherein mit Zeilennummern versehen entwerfen muß. Diese Zeilennummern dienen zur Festlegung der *Abarbeitungsfolge;* die Reihenfolge bei der Eingabe ist gleichgültig. In Abhängigkeit vom konkreten Interpreter ist der zulässige Wertebereich für die Zeilennummern unterschiedlich. Der kleinste Wert liegt bei 0 oder 1, der größte unter 65530. Um Platz für spätere Erweiterungen zu haben, erhöht man die Numerierung zwischen zwei aufeinanderfolgenden Zeilen meist um einen größeren Schritt; gebräuchlich sind Werte von 5 bis 20.

Bei der Kodierung eines BASIC-Programms wird meist so verfahren, daß auf jeder Zeile nur eine einzige Sprachanweisung steht. Das Programm ist dann übersichtlicher und läßt sich besser lesen. Man kann in diesem Fall auch – anstelle von der Zeilennummer – von der Anweisungsnummer sprechen. Erscheint es aber zweckmäßiger (oder ist es evtl. sogar notwendig), auf einer Zeile *mehrere* Sprachanweisungen unterzubringen, so ist das in gleicher Weise möglich, wie es im Abschnitt 3.2.2 für Kommandozeilen behandelt wurde; Trennzeichen ist auch hier der *Doppelpunkt.*

Erhält der *Editor* eine Zeile zur Bearbeitung, so führt er eine erste lexikale Analyse durch und *komprimiert* den Text, um Speicherplatz zu sparen und eine schnellere Arbeit des Interpreters zu ermöglichen. Dabei werden beispielsweise alle reservierten Wörter durch eine interne Kodierung von einem Byte Länge ersetzt. Der Programmierer merkt von dieser Verdichtung nichts, er sieht immer – auch bei einer späteren Inspektion des Programms – den vollständigen, von ihm eingegebenen Text.

Anschließend legt der Editor die bearbeitete Zeile im *Programmspeicher* ab und kettet sie an derjenigen Stelle in das bisher eingegebene Programm ein, wo sie entsprechend ihrer *Nummer* hingehört. Ist die betreffende Nummer bereits vorhanden, so wird die neue Zeile anstelle der alten eingebunden. (Die alte Zeile steht dann zwar immer noch physisch im Programmspeicher; sie gehört aber logisch nicht mehr zum Programm und läßt sich auch nie mehr wiederverwenden.)

Abschließend sollen noch zwei Sprachanweisungen eingeführt werden, die zur Strukturierung von BASIC-Programmen dienen. Die erste ist die *Kommentaranweisung:*

n **REM** *kommentartext*

Darin steht *n* für die jeweilige Zeilennummer, während **REM** (REMark) das Schlüsselwort ist. Manche Interpreter lassen an dieser Stelle auch das Ausrufungszeichen als Schlüsselwort zu:

n ! *kommentartext*

Andere wiederum erlauben statt dessen den Apostroph ' oder inverse Schrägstriche \ \.

Ein Kommentar ist für den Programmierer bestimmt, der einen BASIC-Quelltext zur Hand nimmt. Dieser Kommentar wird beim Auslisten des Programms mit dargestellt, beim Abarbeiten aber übergeht der Interpreter alle nach dem Schlüsselwort REM folgenden Zeichen bis zum Zeilenende. Daher können hier beliebige Zeichen benutzt werden, die sich eingeben und ausdrucken lassen. Es sind also auch Kleinbuchstaben und alle Sonderzeichen zulässig. Über die Zweckmäßigkeit dieser Kommentare wurde bereits im Abschnitt 2.6.1 gesprochen.

Der Text der REM-Anweisung wird zur Laufzeit nicht ausgegeben! Möchte man dem Bediener (und nicht dem Programmierer) etwas mitteilen, so muß man statt dessen eine PRINT-Anweisung (Abschn. 4.1.1) benutzen.

Die nächste zu diskutierende Sprachanweisung wird zwar vom Interpreter ausgeführt, hat aber ebenfalls keine Aktionen des Computers zur Folge. Es ist nämlich erforderlich, dem Interpreter mitzuteilen, an welcher Stelle der Programmlauf beendet werden soll. Dazu dient die Anweisung

n **END**

Sie kennzeichnet das *logische Ende* des Programms. (Das physische Ende wird durch die Anweisung mit der höchsten Zeilennummer gegeben.) Erkennt der Interpreter die END-Anweisung, so

beendet er die fortlaufende Interpretation des Programms und fordert vom Bediener das nächste Kommando an. Vergißt der Programmierer diese Anweisung, so führt das zu Fehlern, wenn nach dem logischen Ende noch weitere ausführbare Anweisungen folgen (z. B. Unterprogramme).

Ein Beispiel für ein vollständiges BASIC-Programm wurde bereits im Programm 2.4 gegeben. Hier tauchten auch die REM- und die END-Anweisung auf. Weitere Beispiele sollen erst gebracht werden, wenn noch mehr Sprachanweisungen bekannt sind.

3.3. BASIC-Elemente

Ziel dieses Abschnitts soll es sein, diejenigen Elemente vorzustellen, aus denen sich BASIC-Programme zusammensetzen. Dazu müssen zunächst die zulässigen Zeichen genannt werden. Aus ihnen bauen sich die Sprachelemente auf, und zwar sowohl vordefinierte (z. B. Schlüsselwörter, Operationssymbole) als auch vom Programmierer noch wählbare (u. a. Zahlen, Zeichenketten, Namen von Variablen).

3.3.1. Zeichenvorrat

Zum Aufbau von BASIC-Sprachelementen sind folgende *Textzeichen* zugelassen:

- die 26 *Großbuchstaben* (außer den Umlauten)
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- die 10 *Dezimalziffern*
  1 2 3 4 5 6 7 8 9 0
- mindestens die folgenden 17 *Sonderzeichen*
  " # ¤ ( ) * + , - . / : ; < = > ^

Manche BASIC-Systeme nutzen noch mehr Sonderzeichen. Außerdem sind alle über die Tastatur eingebbaren und auf dem Bildschirm darzustellenden Zeichen innerhalb der bereits behandelten Kommentartexte sowie in den im Abschnitt 3.3.4 zu besprechenden Zeichenketten (Textkonstanten) zulässig. Einen Überblick über alle möglichen Sonderzeichen gibt die Tafel A.3.

Eine besondere Rolle spielen die Leerzeichen:

- Innerhalb von BASIC-Schlüsselwörtern dürfen keine Leerzeichen eingeschoben werden, beispielsweise ist PR INT unzulässig.
- Leerzeichen sind als Trennzeichen meist dort erforderlich, wo solche Sprachelemente zusammenstoßen, die aus jeweils mehreren Buchstaben oder Ziffern gebildet werden müssen oder könnten:

```
PRINT 5
INPUT A
```

- Leerzeichen sind dann nicht erforderlich, wenn die obengenannten BASIC-Elemente durch (andere) Sonderzeichen getrennt werden:

```
PRINT"Seitenlaenge"
LET A=3 : LET B=4 : LET F=A+B : PRINT F
```

- Zwischen den einzelnen BASIC-Elementen dürfen beliebig viele Leerzeichen eingeschoben werden. Dadurch wird die Lesbarkeit verbessert, allerdings auch die Ausführungsdauer vergrößert:

```
IF V ( I ) < = V ( J ) THEN GOSUB 220
```

3.3.2. Schlüsselwörter

Bereits bei der Behandlung des Aufbaus von BASIC-Anweisungen (Abschn. 3.2.2) wurden die Schlüsselwörter erwähnt, die jeweils am Anfang jeder Anweisung stehen:

- Ein *Schlüsselwort* ist eine Folge von drei oder mehr Großbuchstaben, die die auszuführende Operation in symbolischer Form bezeichnen.

Dabei werden englische Benennungen bzw. Abkürzungen für die Operationen verwendet. Eine Zusammenstellung von Systemanweisungen bringt die Tafel A.5, während die Schlüsselwörter von Sprachanweisungen in der Tafel A.6 enthalten sind.

Schlüsselwörter sind sog. *reservierte Wörter.* Sie dürfen vom Programmierer nicht als (selbstgewählte) symbolische Bezeichnungen für Variablen und anderes verwendet werden.

3.3.3. Zahlenkonstanten

Beim Einsatz eines Rechners benötigt man konkrete Verarbeitungsdaten; beispielsweise Zahlen. Auch wenn ein Algorithmus mit Hilfe allgemeiner Zahlsymbole formuliert wurde, so sind bei der Eingabe in den Computer direkte Zahlenwerte erforderlich, und auch die Resultate erwartet man natürlich in Form von Zahlen:

- *Zahlen* werden durch eine Folge von Ziffern dargestellt. Diese Zeichenfolge repräsentiert den konstanten Wert dieser Zahl selbst (und nicht die Adresse der Speicherzelle, in der sie abgelegt ist). Man spricht daher auch von einem *Zahlenliteral.*

Entsprechend den unterschiedlichen Anforderungen bietet BASIC mehrere *Typen* von Zahlen, die einen unterschiedlichen Wertebereich und eine unterschiedliche Darstellungsform haben:

- *Ganze Zahlen* (auch *INTEGER-Zahlen* genannt)
  Die Darstellungsform entspricht völlig der üblichen Schreibweise. Dabei darf ein positives Vorzeichen weggelassen werden:

  17 +33 −152 1024

  Wenn für die Abspeicherung zwei Byte verwendet werden, liegen die zulässigen Werte im Bereich

  −32768 . . . +32767

  Einige große BASIC-Systeme bieten die Möglichkeit, INTEGER-Zahlen *doppelter Genauigkeit* zu verarbeiten.
- *Reelle Zahlen* (auch *REAL-Zahlen* genannt)
  In diesem Fall muß man verschiedene *Darstellungsformen* unterscheiden, die aber keinen Unterschied im Typ der Zahl bedeuten. Es handelt sich dabei nur um Varianten, die beliebig gewählt werden können:
  Der üblichen Schreibweise sehr ähnlich ist die Darstellung von *Dezimalbrüchen.* Es ist lediglich zu beachten, daß ein *Punkt* anstelle des Dezimalkommas zu setzen ist:

  3.14159 −0.7 25 70.105 −.01

  Dabei können (vor dem Dezimalpunkt) führende Nullen und (nach dem Dezimalpunkt) nachgestellte Nullen sowie das Pluszeichen weggelassen werden:

  +000.010 .01

  Komplizierter ist die *halblogarithmische Darstellung,* auch *Exponentialschreibweise* genannt. Sie ist aber bei sehr großen und sehr kleinen Werten recht zweckmäßig. Dabei wird die Zahl als Produkt einer ganzen Zahl oder eines Dezimalbruchs mit einer ganzzahligen Potenz zur Basis 10 angegeben, z. B. entsprechend

  $53{,}27 \cdot 10^7$

  In dieser Form kann man die Zahl allerdings noch nicht in den Rechner geben:

  statt des Kommas ist ein Punkt zu schreiben, der Multiplikationspunkt ist wegzulassen,

statt der Basis 10 wird der Großbuchstabe E verwendet, und der Exponent ist auf der Zeile zu schreiben.

Mit diesen Änderungen erhält man die richtige Schreibweise

53.27 E7

Dabei darf das Leerzeichen zwischen Dezimalbruch und Basis auch entfallen. Weitere Beispiele sind:

3.25 E-3 für $3{,}25 \cdot 10^{-3}$
-.7 E6 für $-0{,}70 \cdot 10^{6}$
9 E13 für $9 \cdot 10^{13}$

Das Pluszeichen ist auch beim Exponenten wahlweise möglich:

+.7 E+6

Die Zahl vor dem Symbol E darf nicht weggelassen werden, weil das dann entstehende Sprachelement vom Interpreter als der Name einer Variablen angesehen würde. Daher ist beispielsweise zu schreiben

1E6 für 10^{6}

Bei Ausgaben wählt der BASIC-Interpreter in der Regel einen solchen Exponenten, daß genau eine (von Null verschiedene) Ziffer vor dem Dezimalpunkt steht. Man bezeichnet diese Form auch als *normierte* Schreibweise.

Bisher wurde noch nichts über die *Genauigkeit* und den *Wertebereich* von REAL-Zahlen gesagt. Beide Angaben hängen von der Anzahl der Bytes ab, die für die Speicherung benutzt werden. Zur Erfüllung der praktischen Mindestforderung werden vier Byte gebraucht. Sie ermöglichen sechs gültige Dezimalstellen und einen ungefähren Wertebereich des Betrags der Zahl von

$10^{-38} \ldots 10^{+38}$

Da diese Genauigkeit nicht sehr hoch ist, beschreiten die einzelnen BASIC-Interpreter verschiedene Lösungswege:

- Es wird von vornherein mehr Platz für die Speicherung einer Zahl benutzt, z. B. sechs Byte.
- Es werden zwei Typen von REAL-Zahlen mit einfacher (4 Byte) und doppelter Genauigkeit (8 Byte) vorgesehen.

Auf diese Probleme wird aber in diesem Buch nicht weiter eingegangen.

3.3.4. Textkonstanten

Ein Computer kann nicht nur mit Zahlen arbeiten (wie z. B. ein Tisch- oder Taschenrechner), er kann auch Daten in Form von Texten manipulieren. Dazu benötigt man konkretes Ausgangsmaterial und erwartet konkrete Texte als Resultat. Und genauso, wie man eine Zahl (genauer: eine Zahlenkonstante) durch eine Folge von Ziffern darstellt, bildet man eine Textkonstante durch Aneinanderreihen von Textzeichen.

- *Textkonstanten* (auch *Zeichenketten* genannt) werden durch eine Folge von Textzeichen dargestellt, die zur Abgrenzung in Anführungszeichen einzuschließen sind. Diese Zeichenfolge repräsentiert den Wert dieser Textkonstanten selbst. Man spricht daher auch von einem *Textliteral*.

Es gibt nur *einen* Typ von Text(-konstanten), der auch als *STRING* bezeichnet wird. Eine solche Zeichenkette kann alle Zeichen enthalten, die sich überhaupt in den Computer eingeben lassen. Allerdings darf verständlicherweise kein Anführungszeichen darin vorkommen, denn es dient ja als Begrenzer:

"BASIC-Interpreter"
"8027 Dresden, Mommsenstr. 13"
"kleinstes Element V(1) ="

Die zulässige Maximalanzahl von Zeichen in einer Textkonstanten hängt vom Interpreter ab; sie liegt im Bereich von 15...255.

In den folgenden Abschnitten werden Zeichenketten zunächst nur für Bildschirmausgaben eingesetzt. Die Probleme der Textverarbeitung werden geschlossen im Abschnitt 8 behandelt.

3.3.5. Variablen

Wie bereits im Abschnitt 1.1.2 gezeigt wurde, ist der Begriff der Variablen eng mit dem Rechnerkonzept *John von Neumann*'s verbunden. Man versteht dort unter einer Variablen eine Speicherzelle, in der nacheinander unterschiedliche Werte enthalten sein können. Es gibt aber auch in mathematischen Formeln die allgemeinen Zahlsymbole, die verschiedene Werte repräsentieren können. Aus diesen Überlegungen heraus wird folgendes festgelegt:

- Unter einer *Variablen* wird ein Speicherbereich verstanden, in dem unterschiedliche Werte eines festgelegten Wertebereichs abgelegt werden können. Dieser Speicherbereich wird durch einen *symbolischen Namen* adressiert.

Dieser Name wird in BASIC in Analogie zu den allgemeinen Zahlsymbolen der Mathematik gebildet:

- Der *Name* einer *Zahlenvariablen* besteht aus einem Großbuchstaben, der von einer Ziffer gefolgt sein darf:

  X X9 D7 A I S0

Die Namen von Variablen werden dem Interpreter durch die konkrete Verwendung bekanntgemacht. Sie brauchen also *nicht* (wie in anderen höheren Programmiersprachen) *deklariert* zu werden. Sie gelten dann im gesamten restlichen Teil des Programms.

Aus dem Gesagten ergibt sich, daß in einem BASIC-Programm nur 286 verschiedene Zahlenvariablen benutzt werden können. Wenn vielleicht auch diese Anzahl ausreichend ist, so stört doch die Tatsache, daß man die Variablen nicht hinreichend verständlich durch ihre Namen unterscheiden kann. Daher lassen manche BASIC-Interpreter auch an der zweiten Position einen Buchstaben zu, oder es sind überhaupt mehr als zwei Zeichen erlaubt.

Ein BASIC-Interpreter kennt meist nur *einen einzigen* Typ von Zahlenvariablen, nämlich REAL-Variablen. Diese sind aber auch zur Aufnahme von INTEGER-Zahlen geeignet. Beim Überschreiten des zulässigen Zahlenbereichs oder beim Entstehen von Dezimalbrüchen erfolgt dann ein automatischer Übergang zu einer REAL-Darstellung. Manche Interpreter lassen sogar mehrere Typen von Zahlenvariablen zu. In diesen Fällen müssen hinter dem Namen vorgeschriebene Sonderzeichen angegeben werden.

Außer den Zahlenvariablen gibt es auch *Textvariablen*. Dabei handelt es sich um Variablen im obengenannten Sinn (d. h. Speicherbereiche), denen als Werte *Zeichenketten* zugewiesen werden können. Auch in diesem Fall wird der Speicherbereich durch einen symbolischen Namen adressiert:

- Der *Name* einer *Textvariablen* besteht aus einem Großbuchstaben, der von einer Ziffer gefolgt sein darf. Am Ende des Namens muß das *Währungszeichen* (Dollarzeichen) stehen.

Im vorliegenden Buch wird für dieses Zeichen stets das Symbol ¤ benutzt:

T¤ C5¤ V¤ H¤ N7¤

Auch für Textvariablen lassen einige Interpreter an zweiter Position einen Buchstaben zu, oder sie erlauben überhaupt mehr als zwei Zeichen.

3.3.6. Funktionen

In höheren Programmiersprachen spielen Funktionen eine große Rolle. In einer einfachen Form sind sie auch in BASIC vorhanden:

- Eine *Funktion* ist eine Transformationsvorschrift, die jedem zulässigen Wert eines Eingabepa-

rameters, des sog. *Arguments*, eindeutig einen zweiten Wert zuordnet, der *Funktionswert* genannt wird.

Funktionen werden in BASIC immer in dem Format

funktionsname (argument)

geschrieben. Dabei unterscheidet man zwei Klassen von Funktionen:

- *Standardfunktionen* sind solche Funktionen, die bereits im Interpreter enthalten sind. Sie werden in der Regel mit drei Buchstaben bezeichnet:

  ABS(X) SIN(X) LOG(X) LEN(T□)

- *Nutzerfunktionen* müssen in dem betreffenden BASIC-Programm erst definiert und auf diesem Wege dem Interpreter bekanntgemacht werden (Abschn. 6.3.2). Sie beginnen stets mit den Buchstaben FN, ein weiterer ist vom Nutzer noch wählbar:

  FNA(X) FNU(X) FNT(C□)

  Manche BASIC-Systeme lassen auch an dieser Stelle längere Namen zu.

Diejenigen Funktionen, die einen Wert vom Typ STRING liefern (also eine Zeichenkette), erhalten nach dem Namen noch zusätzlich das Währungssymbol (Dollarzeichen):

CHR□(N) INPUT□(I) FNC□(X□)

Die Angabe des Arguments in Klammern gehört zum Format einer Funktion. Es ist deshalb sogar dann anzuführen, wenn die betreffende Funktion überhaupt kein Argument benötigt. Man spricht dann von einem *Blindargument*, das völlig beliebig ist. Verschiedene Interpreter erlauben es jedoch, bei manchen Funktionen solche Blindargumente wegzulassen oder sehen überhaupt keine vor:

RND INKEY□

3.4. Editieren von BASIC-Programmen

Bisher wurde nur allgemein darüber gesprochen, wie ein BASIC-Programm in den Computer einzugeben ist. Jetzt sollen die Eigenschaften des im BASIC-System integrierten Editors ausführlicher behandelt werden. Im Mittelpunkt stehen dabei Hilfsmittel zur Erfassung und zur Korrektur von Programmen.

3.4.1. Vorbereiten der Programmeingabe

Wenn das BASIC-System mit Hilfe des jeweiligen Betriebssystems neu geladen und gestartet wurde, befinden sich alle seine Teile in ihrem Anfangszustand. So sind beispielsweise der Programmspeicher und der für die Variablen vorgesehene Bereich leer. Hat man dagegen bereits gearbeitet und möchte nun zu einem neuen Programm übergehen, so muß man folgende Systemanweisung benutzen:

NEW

Der Interpreter *löscht* daraufhin den *Programmspeicher* und den *Variablenbereich*. Manche Interpreter verwenden statt NEW das Schlüsselwort SCRATCH.

Soll dagegen nur der Variablenbereich gelöscht werden, ein vorhandenes Programm aber erhalten bleiben, so ist eine andere Systemanweisung einzusetzen:

CLEAR [*umfang zeichenkettenspeicher*] [,*obere grenze arbeitsbereich*]

Dieses Kommando *löscht* alle bisher (z. B. in einem vorhergehenden Programmlauf) benutzten *Variablen*. Außerdem gestattet es, die Standardfestlegungen des BASIC-Systems zu folgenden Punkten zu modifizieren:

- Wenn in einem Programm viele Zeichenketten hantiert werden, empfiehlt sich eine Erweite-

rung des Umfangs des vom BASIC-System standardmäßig dafür vorgesehenen Speicherbereichs.

- Sollen Unterprogramme im Maschinenkode eingesetzt werden (Abschn. 10.5.1), so wird dafür Speicherplatz benötigt. Deshalb könnte es erforderlich sein, die höchste Speicheradresse festzulegen, die vom BASIC-System noch benutzt werden darf.

Einige BASIC-Varianten gestatten noch mehr Spezifizierungen. Diese Einzelheiten sowie das Format der Parameterangaben müssen daher der Bedienungsanleitung entnommen werden.

Nunmehr kann der Programmierer sein Programm zeilenweise eingeben. Erfahrungsgemäß macht dabei das fortlaufende Numerieren einige Umstände. Eine spürbare Entlastung bringt die Systemanweisung

AUTO [*zeilennummer*] [,*schrittweite*]

Sie bewirkt, daß nach der Eingabe einer Programmzeile *automatisch* die Nummer der *nächsten Zeile* in den Eingabepuffer und auf den Bildschirm geschrieben wird. Dabei beginnt die Numerierung bei derjenigen Zahl, die in der AUTO-Anweisung benutzt wurde, und sie erfolgt in den ebendort angegebenen Schritten:

AUTO 1000,5 erzeugt 1000 1005 1010 usf.

In der Definition wurde aber durch die eckigen Klammern symbolisiert, daß beide Angaben wahlfrei sind. Fehlt die Schrittweite, so wird dafür der Wert 10 angenommen:

AUTO 500 erzeugt 500 510 520 usf.

Fehlt dagegen der Anfangswert, so wird er gleich null gesetzt:

AUTO ,20 erzeugt 0 20 40 usf.

Werden beide Werte nicht angegeben, liegt der Standardfall vor:

AUTO erzeugt 10 20 30 usf.

Diese größeren Schrittweiten sind, wie bereits erwähnt, für das nachträgliche Einschieben von Zeilen günstig.

Wird bei der automatischen Numerierung eine *bereits vorhandene* Zeile erreicht, so warnen manche Editoren durch einen zusätzlich ausgegebenen Stern:

130*

Drückt man hier sofort die Zeilenendetaste, so bleibt der bisherige Inhalt unversehrt. Gibt man dagegen Zeichen ein, so wird die alte Zeile gelöscht und durch die neu eingegebene ersetzt.

Möchte man die automatische Numerierung *beenden*, um aus dem Lernmodus wieder in den Kommandomodus zu gelangen, so muß man eine für den jeweiligen Interpreter spezifische Eingabe machen, z. B. die Tastenkombination CTRL-C (oder die STOP-Taste) drücken.

3.4.2. Eingeben des Programms

Alle mit einer Zahl beginnenden Eingabezeilen werden vom Editor im Programmspeicher abgelegt und entsprechend dieser Zeilennummer in die Folge der bisher darin enthaltenen Zeilen eingeordnet.

Während dieser Eingaben könnte die Frage auftauchen, wieviel Speicherplatz noch frei ist, um die weitere Arbeit geeignet planen zu können. Hierfür stellen viele Interpreter eine Standardfunktion zur Verfügung:

FRE(*x*)

Diese Funktion liefert die *Anzahl von Bytes*, die insgesamt für Programmzeilen, Variablen und Zeichenketten noch verfügbar sind. Das Argument *x* wird von manchen Interpretern als *Blindargument* betrachtet und gar nicht ausgewertet. Es ist dann willkürlich. Bei anderen ist es dagegen von Bedeutung, ob eine Zahl oder eine Zeichenkette als Argument angegeben ist:

- Wird als Argument eine *Zahl* oder eine *Zahlenvariable* angegeben, so liefert die FRE-Funktion den Umfang des *insgesamt* noch freien Speichers.

- Verwendet man dagegen eine *Zeichenkette* oder eine *Textvariable*, so erhält man die Anzahl der noch freien Bytes im Zeichenkettenspeicher.

Funktionen liefern einen Wert, im vorliegenden Fall eine Zahl. Will sie der Programmierer auf dem Bildschirm sehen, so muß er sie ausschreiben lassen, beispielsweise mit Hilfe des Kommandos

PRINT FRE(A)

Manche Interpreter sehen für diese Funktion das Schlüsselwort MEM (MEMory) vor oder bieten die Systemanweisung SIZE an.

Spätestens am Ende der Erfassung wird sich der Bediener sein Programm noch einmal im Zusammenhang ansehen wollen. Er hat dazu zwei Systemanweisungen zur Verfügung, LIST und LLIST:

LIST

Dieses Kommando veranlaßt, daß das gesamte Programm (die *Programmliste*) auf dem *Bildschirm* ausgegeben wird. Ist das Programm länger als der Bildschirm, so läuft es durch, und der Bediener sieht nur noch den Schluß. Hier gibt es verschiedene Hilfen. So könnte die Bedienungsroutine für das Bildschirmgerät ein *Blättern* erlauben (Abschn. 1.3.2). Solche BASIC-Systeme zeigen jeweils eine bestimmte Anzahl von Programmzeilen an, beispielsweise 10. Nach jedem Drücken der Zeilenendetaste werden die nächsten 10 Zeilen angezeigt. Möchte man diese Anzahl verändern, so steht eine Systemanweisung zur Verfügung:

LINES *anzahl*

Durch dieses Kommando kann der Bediener festlegen, wie viele Programmzeilen er bei folgenden LIST-Kommandos jeweils neu angezeigt haben möchte. In anderen Fällen läßt sich der Durchlauf *anhalten*, z. B. durch Eingabe von CTRL-S. Das Fortsetzen wäre dann ebenfalls durch CTRL-S möglich. Will man die Ausführung des Kommandos dagegen überhaupt abbrechen, so kann man das auch an dieser Stelle durch Eingaben von CTRL-C (STOP-Taste) erreichen.

Möchte man die Programmliste dagegen über den Drucker (*lineprinter*) ausgeben, so ist die folgende Systemanweisung einzusetzen:

LLIST

Durch dieses Kommando wird das gesamte Programm *ausgedruckt.* Dabei läßt sich die Ausgabe ebenfalls in der besprochenen Form steuern.

In vielen Fällen interessiert nur ein Teil des Programms. Dann ist es erforderlich, zusätzliche Parameter anzugeben. Dabei ist eine Reihe von Varianten möglich:

- [L]**LIST** *zeilennummer*

  Es wird nur die spezifizierte Zeile ausgegeben:

  LIST 300
  LLIST 170

- [L]**LIST** *zeilennummer-*

  Von der angegebenen Zeile ab wird das restliche Programm gedruckt:

  LIST 340-
  LLIST 550-

- [L]**LIST** *-zeilennummer*

  Der Anfang des Programms wird bis zur genannten Zeile ausgelistet:

  LIST -280
  LLIST -460

- [L]**LIST** *zeilennummer-zeilennummer*

  Der spezifizierte Programmabschnitt wird ausgegeben:

  LIST 710-800
  LLIST 1000-1500

Meist müssen die benutzten Zeilennummern im Programm tatsächlich vorhanden sein.

3.4.3. Korrigieren des Programms

Bereits bei der Besprechung der allgemeinen Verfahrensweise (bei der Eingabe von Programmzeilen) wurde darauf hingewiesen, wie man nachträglich neue Zeilen zusätzlich oder anstelle alter Zeilen in ein Programm hineinbringen kann. Das soll hier noch einmal systematisch zusammengestellt werden:

- Möchte man eine neue Zeile *zwischen* zwei bereits vorhandenen *einfügen*, so muß man ihr eine dazwischenliegende Nummer zuweisen und den Anweisungstext eingeben. Der Editor ordnet diese Zeile dann an der entsprechenden Stelle richtig ein.
- Soll eine alte Zeile durch eine neue *ersetzt* werden, so ist der neue Text unter der alten Nummer einzugeben. Der Editor kettet dann die alte Zeile aus und die neue an dieser Stelle ein.
- Das *Löschen* einer Zeile kann man als Sonderfall einer Änderung betrachten: Die neue Zeile enthält *nichts*. Man gibt nur die alte Nummer ein, gefolgt vom Zeilenendezeichen.

Möchte man aber mehrere hintereinanderliegende Zeilen löschen, so kann die folgende Systemanweisung den Schreibaufwand etwas reduzieren:

DELETE [*zeilennummer*] [*-zeilennummer*]

Dieses Kommando *löscht alle Zeilen* des spezifizierten Programmabschnitts einschließlich der beiden direkt angegebenen Zeilen:

```
DELETE 220-390
```

Möchte man nur eine einzige Zeile löschen oder alle Zeilen vom Anfang des Programms an bis einschließlich einer bestimmten Zeile streichen, so sind auch Parameterangaben wie

```
DELETE 320
DELETE -270
```

zulässig.

Nach solchen Programmkorrekturen ist natürlich die ursprüngliche Numerierung gestört; es sind größere und kleinere Abstände entstanden. Das kann weitere Ergänzungen erschweren oder – wenn keine Zwischenräume mehr vorhanden sind – unmöglich machen. Hier ist folgende Systemanweisung sehr nützlich:

RENUM

Das Kommando RENUM bewirkt eine *Neunumerierung* (RENUMber) des gesamten Programms, und zwar ab 10 (als niedrigster neuer Nummer) in Zehnerschritten. Möchte man nur einen Teil neu numerieren, wünscht man eine andere Anfangsnummer oder eine andere Schrittweite, so bieten die verschiedenen BASIC-Systeme Modifikationen des RENUMBER-Kommandos an. Die konkreten Implementierungen sind aber so unterschiedlich, daß man sich genau anhand der jeweiligen Bedienungsanleitung über Besonderheiten des benutzten Editors informieren muß.

Bei der Neunumerierung werden natürlich auch alle Zeilenangaben in Sprunganweisungen entsprechend geändert. Daher merkt der Editor, wenn Sprünge nach Zeilen vorhanden sind, die überhaupt nicht (mehr) existieren. Er bringt dann eine Fehlermeldung, die sich allerdings auf die alte Zeilennummer bezieht. Man hat durch diese Kontrolle die Möglichkeit, solche – meist beim Korrigieren entstandenen – Fehler bereits in der Editierungsphase aufzudecken.

3.4.4. Editieren einzelner Zeilen

Bereits im vorigen Abschnitt wurde eine Möglichkeit behandelt, wie man fehlerhafte Programmzeilen ändern kann. Häufig ist aber in einer längeren Zeile nur sehr wenig zu korrigieren. Um das erneute Eingeben der bereits richtigen Zeichen zu umgehen, bieten manche BASIC-Editoren die Möglichkeit für die einfache Überarbeitung einer einzelnen Zeile. Die konkrete Realisierung ist stark vom jeweiligen BASIC-System abhängig und muß der betreffenden Bedienungsanleitung entnommen werden.

Um eine Vorstellung von den angebotenen Funktionen zu ermitteln, soll hier kurz eine Variante vorgestellt werden.

Bemerkt man bei der Eingabe einer Programmzeile bereits vor dem Drücken der Endetaste einen Fehler, so kann man den Zeileneditor durch das Steuerzeichen CTRL-A aufrufen, ansonsten ist eine Systemanweisung erforderlich:

EDIT *zeilennummer*

Durch dieses Kommando wird der Zeileneditor aktiviert und schreibt den Anfang der zu überarbeitenden Zeile, die Zeilennummer, aus. Dann wartet er auf Bedienerkommandos, die aus jeweils einem Buchstaben oder Steuerzeichen bestehen. **Tafel 3.1** vermittelt dazu einen Überblick.

Tafel 3.1. Funktionen eines Zeileneditors

Befehl	Bedeutung	Funktion
L	List	Ausgabe der betreffenden Zeile
(SP) (Leertaste)	SPace	Schreibmarke eine Position nach rechts
(BS) (CTRL-H)	Back-Space	Schreibmarke eine Position nach links
S*z*	Search	Schreibmarke auf nächstes Zeichen *z*
I*text*	Insert	Einfügen des angegebenen Textes *text* vor der aktuellen Position der Schreibmarke. Endezeichen: (ESC)
X*text*	eXtend	Schreibmarke hinter das Ende der Zeile, dann weiter wie bei I
H*text*	Hack	Löschen der Zeile ab Position der Schreibmarke, dann wie bei I
C*z*	Change	Einsetzen des Zeichens *z* anstelle des durch die Schreibmarke angegebenen
D	Delete	Löschen des durch die Schreibmarke angegebenen Zeichens
K*z*	Kill	Schreibmarke auf nächstes Zeichen *z*, Löschen aller dabei überstrichenen Zeichen
A	Abort	bisherige Korrekturen ignorieren, Editierung der Zeile neu beginnen
Q	Quit	bisherige Korrekturen ignorieren, Editierung der Zeile abbrechen
E	End	bisherige Korrekturen in das Programm übernehmen, Editierung der Zeile abschließen
(Zeilen-endetaste)		wie bei E, aber Rest der Zeile noch ausschreiben

Diese Befehle werden über die Tastatur eingegeben, erscheinen aber nicht auf dem Bildschirm. Hier sieht man nur die Auswirkung auf die zu korrigierende Zeile. Soll ein Befehl mehrfach wiederholt ausgeführt werden, beispielsweise das Löschen mehrerer Zeichen, so ist *vor* dem Buchstaben (Sonderzeichen) die Anzahl der Wiederholungen einzugeben. Manche Befehle haben Parameter, z. B. ein in der betreffenden Zeile zu suchendes Zeichen. Diese Eingabe erfolgt *nach* dem Buchstaben (Sonderzeichen).

Eine andere Variante, die mit zusätzlichen Steuertasten arbeitet, wurde bereits im Abschnitt 3.2.1 erwähnt.

3.5. Abarbeiten von BASIC-Programmen

In den bisherigen Abschnitten wurden Entwurf und Editieren von BASIC-Programmen behandelt. Jetzt soll darüber gesprochen werden, wie der Programmierer ein Programm starten kann, und welche programmtechnischen und operativen Möglichkeiten er hat, den Lauf seines Programms zu unterbrechen.

3.5.1. Starten

Das bereitgestellte BASIC-Quellprogramm läßt sich mit Hilfe des Interpreters sofort ausführen. Dazu dient die Systemanweisung

RUN [*startzeilennummer*]

Durch dieses Kommando wird der BASIC-Interpreter damit beauftragt, die im Programmspeicher stehenden Zeilen zu *interpretieren*, und zwar von der angegebenen Zeile an fortlaufend in Richtung steigender Zeilennummern:

```
RUN 2000
```

Fehlt die Parameterangabe, so wird bei der Zeile mit der niedrigsten Nummer begonnen:

```
RUN
```

Erreicht der Interpreter bei der Übersetzung die END-Anweisung, so schließt er seine Arbeit ab. Das System kehrt in den Kommandomodus zurück, was an der entsprechenden Aufschrift (beispielsweise READY, OK, DONE) zu erkennen ist.

Das RUN-Kommando enthält implizit die im Abschnitt 3.4.1 behandelte Systemanweisung CLEAR, und zwar ohne Parameter.

3.5.2. Abschnittsweises Abarbeiten

Mitunter wird von vornherein gewünscht, ein BASIC-Programm in bestimmten Teilschritten abzuarbeiten. In anderen Fällen ist es in der Testphase erforderlich, an bestimmten Stellen einen *Haltepunkt* zu haben, wo man in Ruhe die bisher erreichten Resultate analysieren und Entscheidungen über den weiteren Ablauf fällen kann. An solchen Stellen läßt sich die Sprachanweisung STOP einsetzen:

n **STOP**

Durch diese Anweisung wird die fortlaufende Interpretation des BASIC-Programms *unterbrochen* und die Steuerung wieder an das Dialogprogramm übergeben. Insofern ähnelt die STOP-Anweisung der END-Anweisung. Der Interpreter rechnet aber mit einer Fortsetzung; er schließt z. B. die benutzten Dateien nicht ab (Abschn. 9.3.3). Außerdem wird die Unterbrechungsstelle auf dem Bildschirm ausgeschrieben.

Verschiedentlich ergibt sich der Anlaß zum Stoppen aber erst, wenn das Programm bereits gestartet wurde. Beispielsweise könnte man durch die bisherigen Programmausgaben oder durch eine überlange Laufzeit erkennen, daß das BASIC-Programm angehalten werden müßte. Hier hilft die Stopptaste mit dem Zeichen

```
CTRL-C
```

Durch die Eingabe dieses Steuerzeichens wird die Programmbearbeitung in gleicher Weise wie durch die STOP-Anweisung *unterbrochen*.

Für das Fortsetzen eines gestoppten BASIC-Programms steht eine Systemanweisung zur Verfügung:

CONT

Durch dieses Kommando wird die Interpretation eines angehaltenen Programms bei der auf die Unterbrechungsstelle (STOP-Anweisung, Tastendruck CTRL-C) folgenden Anweisung *weitergeführt*. (In einer Reihe von Fällen ist übrigens auch die Fortsetzung eines mit der END-Anweisung abgeschlossenen Programms möglich.)

Möchte man dagegen an einer anderen Stelle weiterarbeiten, so müßte man die Sprachanweisung GOTO (Abschn. 6.1) mit der gewünschten Zeilennummer in Form eines Kommandos einsetzen:

```
GOTO 2170
```

3.6. Auffinden von Fehlern in BASIC-Programmen

Der Testprozeß am Abschluß einer Implementierung wurde bereits im Abschnitt 2.5 ausführlich behandelt. Die allgemeine Vorgehensweise beim Aufdecken von Fehlern kann an dieser Stelle nachgelesen werden. Im folgenden Abschnitt soll hingegen darauf eingegangen werden, welche Hilfsmittel in BASIC-Systemen für das Feststellen und Lokalisieren von Fehlern zur Verfügung stehen.

3.6.1. Melden von syntaktischen Fehlern

Bereits bei der Eingabe einer Quellzeile kann eine gewisse Fehleranalyse durch das BASIC-System erfolgen. Zu diesem Zweck müßte das Dialogprogramm diese Zeile nicht nur dem Editor zum Abspeichern übergeben, sondern auch – evtl. nach einer Aufforderung durch den Bediener – dem Interpreter zur Ausführung. Dabei ist allerdings zu beachten, daß eine Reihe von Sprachanweisungen gewisse Fernwirkungen hat, die über den Rahmen einer einzigen Zeile hinausgehen können, z. B. die Schleifenanweisung. Daher spiegeln die Fehlermeldungen bei der Interpretation einzelner Zeilen nicht die tatsächliche Situation richtig wider.

Im wesentlichen werden *syntaktische Fehler* aber erst bei der Abarbeitung des vollständigen Programms festgestellt. Die BASIC-Interpreter differenzieren dabei zwischen ungefähr 20 bis 50 und mehr Fehlern, die durch Nummern gekennzeichnet werden. Anschließend wird zu einem *Fehlerbehandlungsprogramm* verzweigt, das zu den einzelnen Nummern *Fehlerausschriften* bringt (Tafel A.9). Je nach dem Charakter des Fehlers wird dieses Systemprogramm dann entweder die Interpretation des BASIC-Programms fortsetzen lassen oder es (entsprechend der STOP-Anweisung) anhalten. Jetzt kann der Bediener entscheiden, ob er trotz des Fehlers eine Fortsetzung (mit dem CONT-Kommando) versucht oder ob er die Ursache der aufgetretenen Fehlermeldung sofort korrigiert.

Die Fehlerausschriften sind zwar systemabhängig, aber meist ohne weitere Erläuterungen verständlich. Manche Interpreter geben außer der Fehlermeldung auch gleich den Text der betreffenden Zeile mit aus. Als weitere Unterstützung ist zuweilen eine Systemanweisung

HELP

vorhanden. Durch dieses Kommando können weitere *Erläuterungen* zu dem aufgetretenen Fehler, seinen Ursachen und seiner Beseitigung abgefordert werden.

3.6.2. Lokalisieren von Fehlern

Meldungen zu syntaktischen Fehlern sind meist so klar, daß man den Fehler sehr schnell erkennen und beseitigen kann, z. B. Undefined line (nicht definierte Zeile) bei einem Sprungbefehl oder Missing operand (fehlender Operand) bei einer arithmetischen Berechnung. Es gibt aber auch *Ablauffehler*, deren Ursache nicht in derjenigen Zeile zu liegen braucht, in der sie gemeldet werden, beispielsweise Division by zero (Division durch Null) oder String too long (Zeichenkette zu lang). Es ist verständlich, daß sich auch die Auswirkungen eines logischen Fehlers erst – früher oder später – im weiteren Programmablauf bemerkbar machen werden. Über Ursachen und Folgen solcher Fehler wurde bereits im Abschnitt 2.5.4 allgemein gesprochen.

Wie findet man sich nun von derjenigen Stelle, an der man die falschen Resultate feststellt, rückwärts zu der betreffenden Anweisung, in der die Fehlerursache liegt? Ein günstiges Verfahren ist hier die *Spurverfolgung*. Die meisten Interpreter bieten hierfür zwei komplementäre Sprachanweisungen:

n TRON

Erkennt der Interpreter diese Anweisung (TRace ON), so schreibt er von dieser Stelle an die Nummern aller nacheinander abgearbeiteten Zeilen aus, in Klammern [...] oder <...> eingeschlossen, und zwar so lange, bis die Sprachanweisung

n TROFF

(TRace OFF) erreicht wird. Diese Anweisungen können auch in Form von Bedienerkommandos erteilt werden. Dann erfolgt allerdings die Protokollierung des durchlaufenen Pfades mindestens bis zum nächsten Programmstopp (durch END, STOP, CTRL-C oder eine Fehlerbehandlung).

Der ausgedruckte Weg führt den Programmierer bei der Fehleranalyse von der Anweisung TRON bis zu derjenigen Stelle, wo die ersten Auswirkungen eines Fehlers entdeckt wurden. Auf diesem Pfad muß der zu lokalisierende Fehler liegen! Insofern ist eine solche Spurverfolgung sehr nützlich. Leider überschüttet der Interpreter den Bediener bei unglücklicher Wahl des TRON-TROFF-Bereichs mit einer unübersehbaren Informationsflut. So ist zweckmäßigerweise darauf zu achten, daß wiederholt abgearbeitete Programmteile (Schleifen) nicht jedesmal wieder protokolliert werden. Hier helfen bedingte Anweisungen (Abschn. 6.1.2). Zum Beispiel könnten im Programm 2.4 die Zeilen

```
1245 TRON
1255 IF I > 1 THEN TROFF
```

eingeschoben werden. Damit würde nur der erste Durchlauf durch die äußere Schleife des Sortierprogramms protokolliert.

Andere Interpreter reduzieren die Ausgaben dadurch, daß nur die Zeilennummern von GOTO-Anweisungen und die dabei verwendeten Sprungziele auf dem Bildschirm angezeigt werden. Diese Angaben reichen zur Rekonstruktion des Pfades vollkommen aus, da zwischen den Sprüngen bekanntlich eine sequentielle Abarbeitung erfolgt.

Eine weitere Methode zur Lokalisierung von Fehlern ist das Setzen von *Haltepunkten* mit Hilfe der im Abschnitt 3.5.2 besprochenen STOP-Anweisung. Durch die Unterbrechung der Interpretation erhält der Bediener die Möglichkeit, sich an dieser Stelle durch geeignete Kommandos den Bearbeitungsstand anzeigen zu lassen. So kann man beispielsweise im Programm 2.4 die Zeile

```
1295 STOP
```

einfügen. Sie hat einen Programmhalt nach jedem Durchlauf durch die innere Schleife des Sortierprogramms zur Folge.

Manche Interpreter bieten eine Systemanweisung

```
STEP
```

an, die die Ausführung der jeweils nächsten Programmzeile veranlaßt. Der Test geht aber dabei sehr langsam vor sich. Daher wird man ein solches Kommando nur in Ausnahmefällen einsetzen.

3.6.3. Individuelle Fehlerbehandlung

Es wurde bereits erwähnt, daß der BASIC-Interpreter einen erkannten Fehler durch eine Nummer kennzeichnet und dann zu einem *Fehlerbehandlungsprogramm* verzweigt, das diese Nummer oder einen entsprechenden Text ausschreibt sowie einige für den betreffenden Fehler charakteristische Aktionen veranlaßt. Für viele Zwecke ist das völlig ausreichend. In Sonderfällen könnten aber andere Maßnahmen zweckmäßiger sein. Daher bieten einige Interpreter die Möglichkeit, anstelle des standardmäßig vorhandenen Programms selbstentwickelte BASIC-Moduln zur individuellen Fehlerbehandlung einzusetzen.

Dieser Problemkreis ist für das vorliegende Buch zu speziell. Der interessierte Leser müßte sich anhand der Sprachbeschreibung seines BASIC-Systems über die bei ihm gültigen Einzelheiten informieren. Hier sollen nur einige allgemeine Hinweise folgen, die der Anfänger getrost überschlagen kann.

Für den Einsatz individueller Fehlerbehandlungsanweisungen innerhalb eines bestimmten BASIC-Programms sind folgende Schritte erforderlich, geordnet nach der zeitlichen Reihenfolge der Abarbeitung durch den Interpreter:

- *Anbinden* der individuellen Fehlerbehandlungsanweisungen an den Interpreter
  Es ist dem Interpreter im Verlauf der Abarbeitung eines BASIC-Programms rechtzeitig mit-

zuteilen, daß von jetzt ab beim Auftreten eines Fehlers nicht mehr das Standardprogramm einzusetzen ist:

n **ON ERROR GOTO** *zeilennummer*

Der Interpreter merkt sich die angegebene Zeilennummer und springt bei jedem erkannten Fehler an diese Stelle, an der die individuellen Fehlerbehandlungsanweisungen beginnen müssen.

Es ist durchaus möglich, in demselben BASIC-Programm nacheinander verschiedene Fehlerbehandlungen vorzusehen. Dabei ist jeweils diejenige ON-ERROR-GOTO-Anweisung gültig, die als letzte interpretiert wurde.

Mit Hilfe der Anweisung

n **ON ERROR GOTO 0**

wird dem Interpreter mitgeteilt, daß nunmehr wieder das Standardprogramm einzusetzen ist.

- Der Interpreter schreibt die *Fehlernummer* (ERRor code) in eine Zelle mit dem Namen

  ERR

  Von dort kann diese Variable gelesen und individuell ausgewertet werden. Die Nummer der fehlerhaften *Zeile* (ERror Line) kann über die Funktion (ohne Blindargument)

  ERL

  abgefragt werden.
- Am Ende der individuellen Fehlerbehandlung ist festzulegen, an welcher Stelle des fehlerhaften BASIC-Programms *weiterzuarbeiten* ist. Die Sprachanweisung

  RESUME

  setzt die Interpretation an der Fehlerstelle selbst fort. Dagegen läßt

  RESUME NEXT

  die fehlerhafte Zeile aus und geht zur nächsten Zeile über, während

  RESUME *zeilennummer*

  bei der angegebenen Zeile fortsetzt.
- Beim Testen der Fehlerbehandlungsanweisungen hilft die Sprachanweisung

  ERROR *fehlernummer*

  Sie *simuliert* das Auftreten eines Fehlers mit der angegebenen Nummer.

Diese kurzen Bemerkungen zur individuellen Fehlerbehandlung sollen genügen. Zur Demonstration mag das **Programm 3.1** dienen. Hier wird eine problemspezifische Fehlerausschrift gebracht, falls man versucht, aus einer negativen Zahl die Wurzel zu ziehen. (Dieses spezielle Pro-

Programm 3.1. Individuelle Fehlerbehandlung

```
100 REM <<<<<<<<<<<<<<<< FEHLERBEHANDLUNG >>>>>>>>>>>>>>>>
110 :
120 ON ERROR GOTO 200
130 :
140 INPUT ; "Argument X = " , X
150 IF X=0 THEN GOTO 240
160 :
170 PRINT " -> SQR(X) =" ; SQR(X)
180 GOTO 140
190 :
200 IF ERL<>170 OR ERR<>5 THEN GOTO 240
210   PRINT " imaginaer!" : REM individ. Fehlerausschrift
220   RESUME NEXT
230 :
240 ON ERROR GOTO 0 : REM Standardbehandlung
260 :
270 END
280 REM ================================================
```

blem läßt sich natürlich auch mit weniger Aufwand lösen! Man vergleiche dazu Abschnitt 5.2.2 und Programm 5.5).

3.7. Optimieren von BASIC-Programmen

Als Abschluß dieses einführenden Abschnitts, in dem allgemeine Aussagen über die Sprache BASIC zusammengestellt wurden, sollen einige Bemerkungen zur Effizienz von BASIC-Programmen gebracht werden. Bei einem ersten Lesen dieses Buches kann man sie sicher zunächst überschlagen. Werden die entwickelten Programme aber immer umfangreicher, findet der Interessent hier vielleicht einen Hinweis zur Verbesserung seiner Arbeit.

3.7.1. Verkürzen der Laufzeit

Die interpretative Abarbeitung von BASIC-Programmen bringt bei der Programmentwicklung große Vorteile. Die Laufzeit ist aber wesentlich größer als bei Maschinenprogrammen. Dabei unterscheiden sich die verschiedenen Interpreter noch stark in ihren Geschwindigkeiten. Um diese Kenngröße einschätzen zu können, wurden in der Literatur verschiedene Musterprogramme (*Bench-mark*-Programme) vorgeschlagen [3.3]. Eines davon ist im **Programm 3.2** dargestellt. **Tafel 3.2** bringt eine Zusammenstellung von Laufzeiten auf verschiedenen Computern mit ihren individuellen Interpretern.

Programm 3.2. Musterprogramm für den Vergleich von BASIC-Interpretern

```
100 K=0
110 DIM M(5)
120 K=K+1
130 A=K/2*3+4-5
140 GOSUB 200
150 FOR L=1 TO 5
160 M(L)=A
170 NEXT
180 IF K<1000 THEN 120
190 END
200 RETURN
```

Tafel 3.2. Vergleich der Geschwindigkeit verschiedener BASIC-Interpreter anhand des Programms 3.2.

Computer	Mikroprozessor	Interpreter	Laufzeit in s
Apple	6502	Apple II Integer	28
A5130	Z80	MBASIC (24K)	44
RM380Z	Z80	TDL-BASIC (8K)	45
Apple	6502	Extended BASIC	45
A5120	Z80	MBASIC (24K)	50
Z9001	Z80	resident	51
PET2001	6502	resident	51
K1520	Z80	TDL-BASIC (12K)	52
MPS4944	Z80	RDK-BASIC (3K)	83
ZX81	Z80	resident	286

In vielen Fällen mag der hohe Laufzeitbedarf von BASIC unwesentlich sein. Manchmal macht er sich aber störend bemerkbar:

- Es gibt *verarbeitungsintensive* Programme, die viele Entscheidungen fällen müssen und bei denen Schleifen häufig durchlaufen werden.

- Bei Regelungsaufgaben *(Echtzeitverarbeitung)* wird eine kurze Antwortzeit des Mikrorechners gefordert. Hier muß man vielleicht sogar Unterprogramme in der Maschinensprache einsetzen (Abschn. 10.5).

Oft hilft auch ein geeigneter Aufbau der BASIC-Programme. Es sollten *zeiteffektive Algorithmen* gesucht und zweckmäßig programmtechnisch umgesetzt werden. Dabei ist anzustreben, vor allem in Schleifen *keine zeitaufwendigen Operationen* auszuführen. Die folgenden Hinweise sind an diesen Stellen besonders ernst zu nehmen.

- Es ist sehr günstig, wenn das zu bearbeitende Problem und der verfügbare Interpreter das Arbeiten mit *ganzen Zahlen* zulassen, weil für den Zahlentyp INTEGER die Rechenoperationen schneller ausgeführt werden können als für den REAL-Typ.
- Die BASIC-Zeilen sind im Programmspeicher häufig nur in Richtung aufsteigender Nummern miteinander verkettet. Entfernt liegende Zeilen müssen schrittweise gesucht werden. Dabei verlängern *Kommentar- und Leerzeilen*, die zur Erhöhung der Übersicht eingeschoben werden, die Abarbeitung. Insbesondere ist bei einem Rücksprung nach einer Zeile mit niedrigerer Nummer diese Zeile vom Anfang des Programms an zu suchen. Dadurch kann die Abarbeitung von häufig durchlaufenden Schleifen verzögert werden.
- Die in einem Programm benutzten Namen von Variablen werden in der Reihenfolge ihrer ersten Verwendung in eine Tabelle eingetragen. Bei jedem weiteren Einsatz werden sie dort wieder gesucht. Häufig benutzte *Variablen* – z. B. Laufvariablen in Schleifen – sollten daher möglichst weit vorn in der Tabelle stehen. Das kann man durch eine *rechtzeitige Verwendung* erreichen; und sei es in einer zusätzlichen, logisch überflüssigen Anweisung!
- Textvariablen haben Werte veränderlicher Länge. Bei jeder Wertzuweisung wird daher eine Neueintragung in den Zeichenkettenspeicher notwendig, wodurch – spätestens bei dessen Überfüllung – eine Verdichtung der gültigen Eintragungen erforderlich wird. Das kostet einen gewissen Zeitaufwand. Daher sollten *Textvariablen* mit häufig wechselnden Längen der Werte *möglichst spät* im Programm benutzt werden, damit diese Werte am Ende des Zeichenkettenspeichers stehen.
- Wird in einer arithmetischen Anweisung (mit REAL-Variablen) eine *Zahlenkonstante* angegeben, beispielsweise

  ```
  380 LET X=X+.1
  ```

  so muß dieses Literal vom Interpreter zunächst in das REAL-Format umgewandelt werden, ehe die Addition erfolgen kann. Bei häufig durchlaufenden Schleifen empfiehlt es sich daher, den Zahlenwert vor dem Eintritt in die Schleife einer *Variablen zuzuweisen*, z. B.

  ```
  110 LET H=.1
  ```

  Damit ergibt sich die schnellere Anweisung

  ```
  380 LET X=X+H
  ```

- Die *Multiplikation* mit kleinen natürlichen Zahlen läßt sich durch mehrfache Additionen beschleunigen, die *Potenzierung* mit kleinen natürlichen Zahlen durch mehrfache Multiplikationen.
- Der Interpreter liest und analysiert eine Zeile zeichenweise, ehe er sie ausführt. Enthält die Zeile überflüssige Zeichen, so erfordert ihre Abarbeitung mehr Zeit. Insbesondere bei Anweisungen in häufig durchlaufenden Schleifen sollte man daher z. B. *Leerzeichen vermeiden*.

Zur Demonstration soll ein Beispiel dienen. So dauert die Abarbeitungszeit des Kommandos

```
FOR I=1 TO 670:NEXT
```

auf dem Bürocomputer A 5120 mit dem Interpreter MBASIC genau 1,00 Sekunden. Dieselbe Zeit erfordert der Lauf eines Programms, das nur aus der Zeile

```
100 FOR I=1 TO 670:NEXT
```

besteht. Fügt man hier an jeder zulässigen Position 10 zusätzliche Leerzeichen ein, so erhöht sich die Laufdauer auf 1,13 Sekunden. Stehen vor der oben angegebenen Zeile 100 leere Anwei-

sungszeilen (jeweils nur ein Doppelpunkt), so beträgt die Laufdauer 1,04 Sekunden. Schreibt man das Programm in der Form

```
100 FOR I=1 TO 670
110 NEXT
```

so läuft es ebenfalls 1,04 Sekunden. Und fügt man die oben angegebene Zeile in einen Rahmen ein, wie es im **Programm 3.3** geschehen ist, so beträgt seine Laufdauer 1,28 Sekunden! Aus diesem Grund müßte in diesem Programm der Endwert der Schleifenvariablen auch auf 524 festgelegt werden, um unter den gegebenen Randbedingungen wieder eine Laufzeit von 1,00 Sekunden zu erreichen.

Programm 3.3. Zeitverzögerungsschleife

```
100 REM <<<<<<<<<<<<<<<< WARTESCHLEIFE >>>>>>>>>>>>>>>>>
110 :
120 INPUT "Anzahl der Sekunden" ; S
130 FOR I=1 TO S
140   FOR J=1 TO 524 : NEXT J
150 NEXT I
160 :
170 END
180 REM ================================================
```

Und noch ein anderes Beispiel. Das Musterprogramm 3.2 zum Vergleich von BASIC-Interpretern benötigt auf dem angegebenen Mikrocomputer eine Laufzeit von 49,8 Sekunden. Ergänzt man es mit einigen Kommentaren, wie es das **Programm 3.4** zeigt, so erfordert es 54,4 Sekunden!

Programm 3.4. Kommentierte Variante des Programms 3.2

```
100 REM <<<<<<<<<<<<<< BASIC BENCHMARK 7 >>>>>>>>>>>>>>>
110 :
120 LET K = 0
130 DIM M(5)
140 LET K = K+1
150 LET A = K/2*3+4-5
160 GOSUB 230
170 FOR L=1 TO 5
180   LET M(L) = A
190 NEXT L
200 IF K<1000 THEN GOTO 140
210 END
220 :
230 RETURN
240 :
250 REM ================================================
```

Als weiteres Beispiel soll das sog. *Sieb des Eratosthenes* dienen. Es ermittelt alle Primzahlen (bis zu einer gewünschten oberen Grenze) dadurch, daß es – systematisch vorgehend – alle teilbaren Zahlen streicht. Dieser Algorithmus ist im **Bild 3.2** dargestellt. Man braucht dabei als Teiler nur alle Primzahlen bis zu einer oberen Grenze zu verwenden, die sich aus der Quadratwurzel der größten Zahl des zu untersuchenden Bereichs ergibt.

Dieser Algorithmus wurde nun in zwei Varianten in BASIC umgesetzt. **Programm 3.5** prüft direkt nach, ob die zu untersuchende Zahl ohne Rest teilbar ist. Die Laufzeit liegt (mit MBASIC auf dem Bürocomputer A5120) bei 134,4 Sekunden. Im **Programm 3.6** dagegen werden die teilbaren Zahlen durch fortlaufende Addition des Teilers ermittelt; die Laufzeit beträgt hier nur 10,5 Sekunden! Verwendet man außerdem noch Variablen vom Typ INTEGER (statt des Standardtyps REAL) und beachtet alle weiteren oben gegebenen Hinweise, so verringert sich die Laufzeit sogar auf 5,7 Sekunden.

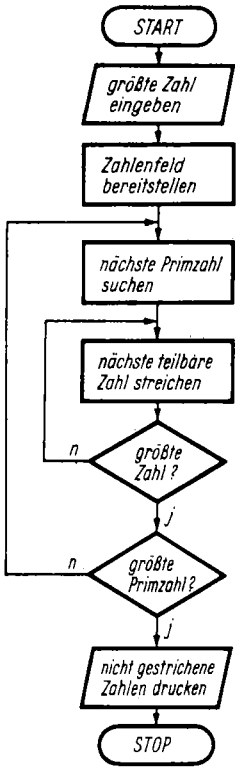


Bild 3.2. Sieb des *Eratosthenes*

Programm 3.5. Sieb des *Eratosthenes* (uneffiziente Variante)

```
100 REM <<<<<<<<<<<< SIEB DES ERATOSTHENES >>>>>>>>>>>>>
110 :
120 INPUT "Groesste Zahl" ; N
130 DIM P(N)
140 FOR I=2 TO N
150   LET P(I) = 1
160 NEXT I
170 :
180 FOR I=2 TO SQR(N)
190   IF P(I)=0 THEN GOTO 230 : REM NEXT I
200   FOR J=I+1 TO N : REM teilbare Zahlen streichen
210     IF (J MOD I)=0 THEN LET P(J) = 0
220   NEXT J
230 NEXT I
240 :
250 PRINT "Primzahlen: " ;
260 FOR I=2 TO N
270   IF P(I)=1 THEN PRINT I ;
280 NEXT I
290 :
300 END
310 REM ================================================
```

Programm 3.6. Sieb des *Eratosthenes* (effizientere Variante)

```
100 REM <<<<<<<<<<<< SIEB DES ERATOSTHENES >>>>>>>>>>>>>
110 :
120 INPUT "Groesste Zahl" ; N
130 DIM P(N) : REM Anfangswerte = 0
140 :
150 FOR I=2 TO SQR(N)
160   IF P(I) THEN 200
170   FOR J=2*I TO N STEP I
180     LET P(J) = 1
190   NEXT J
200 NEXT I
210 :
220 PRINT "Primzahlen: " ;
230 FOR I=2 TO N
240   IF P(I)=0 THEN PRINT I ;
250 NEXT I
260 :
270 END
280 REM ================================================
```

3.7.2. Verringern des Speicherbedarfs

Gerade bei Heimcomputern wird man schnell an die Grenzen der Speicherkapazität kommen. Die Möglichkeiten zur Verringerung des Speicherbedarfs von BASIC-Programmen sind allerdings begrenzt:

- Bei Korrekturen wird die alte Zeile aus dem Programm ausgekettet. Sie bleibt aber im Speicher stehen. Falls der Interpreter keine automatische Verdichtung durchführt, kann es ein Ausweg sein, das Programm auf den externen Zusatzspeicher auszulagern und wieder einzulesen. Diese Maßnahme hilft allerdings nur dann, wenn nicht nur eine Speicherkopie ausgelagert wird, sondern die Programmzeilen entsprechend ihrer fortlaufenden Numerierung.
- Ganze Zahlen benötigen nur zwei Byte für ihre Speicherung. Daher sollte – wenn möglich – mit dem *Zahlentyp INTEGER* gearbeitet werden.
- Wiederholt benötigte, längere Texte für Ausschriften sollte man nicht mehrmals als Textkonstante definieren, sondern einer *Textvariablen* zuweisen. Dadurch wird Platz im Zeichenkettenspeicher eingespart.

- Eine mehrfach im Programm auftretende Anweisungsfolge ist in Form eines *Unterprogramms* zu schreiben (Abschn. 6.3.1). Sie belegt dann den entsprechenden Platz im Programmspeicher nur ein einziges Mal.
- Arithmetische und logische Ausdrücke sollten unter Nutzung der Rangfolge der Operatoren (Tafel A.8) so kodiert werden, daß *keine Klammern* benötigt werden.
- *Datenfelder* sind sparsam zu dimensionieren (Abschn. 7.1).

Alle weiteren Vorschläge gehen auf Kosten der Übersichtlichkeit des Programms. Sie sollten daher nur in Ausnahmefällen angewendet werden:

- Weglassen aller *Kommentare, Leerzeilen* und *Leerzeichen* (soweit sie keine Trennzeichen sind) sowie der überflüssigen *Semikolons* in Ausgabelisten
- Verwendung *kurzer Namen* (je ein Zeichen) für Variablen, Felder und Funktionen
- Einsparen von Platz für Zeilennummern und Verkettungszellen durch Kodierung *mehrerer Anweisungen* je Zeile
- *mehrfache* Benutzung *derselben* Variablen *nacheinander* für verschiedene Zwecke.

4. Eingeben und Ausgeben von Daten

Am Anfang der ausführlichen Besprechung von BASIC-Sprachelementen werden zunächst die Anweisungen für die Eingabe und Ausgabe von Daten behandelt. Anlaß dazu ist die besondere Rolle dieser Aktivitäten bei der Nutzung von Rechenautomaten; schließlich beginnt jede Verarbeitung von Daten mit deren Eingabe in den Computer und endet mit der Ausgabe der Resultate. Außerdem wird es dadurch möglich, bei der differenzierten Besprechung der Verarbeitungsanweisungen jeweils vollständige Programmbeispiele bringen zu können, ohne dem Leser – wie bisher – noch nicht Bekanntes anzubieten.

Eingabe/Ausgabe-Anweisungen nehmen in allen Programmiersprachen eine besondere Stellung ein. Das liegt daran, weil hier gerätetechnische Probleme mit algorithmischen zusammenstoßen. Manche Sprachen definieren daher überhaupt keine konkreten Eingabe/Ausgabe-Operationen, sondern stellen nur eine Verbindung mit dem jeweiligen Betriebssystem her. BASIC hingegen bietet ein breites Spektrum von solchen Anweisungen. Dadurch wird allerdings die Darstellung für Lernende erschwert: Es gibt relativ einfache Standardanweisungen, aber auch recht komplexe, mit denen man eine gewünschte Druckaufbereitung realisieren kann. Aus methodischen Gründen sollen trotzdem alle Eingabe/Ausgabe-Anweisungen von BASIC in diesem Abschnitt zusammengefaßt dargestellt werden. Dem Lernenden wird empfohlen, sich beim ersten Lesen nur mit den Standardanweisungen zu befassen und sich die komplizierteren Elemente erst im Prozeß der konkreten Programmierarbeit schrittweise anzueignen.

Einige Ergänzungen werden im Abschnitt 9.3 zu den Eingabe/Ausgabe-Anweisungen bei der Nutzung externer Zusatzspeicher gebracht.

4.1. Ausgeben von Daten auf Bildschirm und über Drucker

Im folgenden soll zunächst die *Ausgabe* von Daten behandelt werden. Auch hierfür ist ein methodischer Grund maßgebend: Ein Programm kann durchaus so beschaffen sein, daß es keine Eingaben erfordert. Natürlich erhält man dann bei jedem Abarbeiten dieselben Ausgaben. Ein Programm jedoch, das zwar Eingaben fordert, aber keinerlei Ausgaben bringt, ist sinnlos! Es wird sich außerdem später zeigen, daß die Sprachanweisung für die Standardeingabe in BASIC bereits mit einer Ausgabe gekoppelt auftreten kann.

4.1.1. Standardausgabe

Für das Ausgeben von Daten auf den *Bildschirm* steht eine einzige Sprachanweisung zur Verfügung:

n **PRINT** *ausgabeliste*

Die *Ausgabeliste* besteht aus Ausgabeelementen, die durch vorgeschriebene Sonderzeichen voneinander getrennt werden. Als *Ausgabeelemente* können auftreten:

arithmetische Ausdrücke
Textausdrücke
Formatsteuerfunktionen.

Als *Trennzeichen* sind möglich:

Kommas und Semikolons.

Diese Vielfalt ist für den Neuling verwirrend. Die gebotenen Möglichkeiten erweisen sich aber im praktischen Einsatz als recht wirkungsvoll. Sie sollen in den folgenden Abschnitten schrittweise eingeführt und erläutert werden.

Manche Interpreter gestatten es, statt des Schlüsselworts PRINT ein *Fragezeichen* zu verwenden:

n **?** *ausgabeliste*

Diese Form ist vor allem dann bequem, wenn man sich – z. B. beim Programmtest – im Kommandomodus Werte von Variablen ausdrucken lassen möchte. In Programmen ist zweifellos PRINT übersichtlicher.

Die Ausgabe von Informationen über einen *Drucker* ist der Ausgabe über Bildschirm analog; lediglich ist PRINT durch LPRINT zu ersetzen:

n **LPRINT** *ausgabeliste*

Daher wird auf das Ausdrucken an dieser Stelle nicht speziell eingegangen. Im Text wird meist nur über PRINT gesprochen. In den Programmbeispielen wird dagegen häufig LPRINT verwendet, um Originalausdrucke für dieses Buch zu erhalten.

4.1.2. Standardausgabe einzelner Werte

Einleitend wird der Fall behandelt, daß die Ausgabeliste der PRINT-Anweisung nur ein einziges Element enthält:

n **PRINT** *arithmetischer ausdruck*

Die Bildung arithmetischer Ausdrücke erfolgt wie in der Mathematik gewohnt und wird im Abschnitt 5.1 behandelt. Meist besteht ein solcher Ausdruck in einer PRINT-Anweisung nur aus einer Zahl (Zahlenkonstanten) oder einer Zahlenvariablen:

```
330 PRINT +32
950 PRINT -10.70
270 PRINT Z
```

Die Wirkung der PRINT-Anweisung besteht nun darin, daß der Zahlenwert des Ausgabeelements am Beginn der Zeile *linksbündig* ausgegeben wird. Anschließend wird zum Anfang der nächsten Zeile weitergegangen.

Für die *Zahlendarstellung* gelten folgende Regeln, die aber noch von Interpreter zu Interpreter variieren:

- In Abhängigkeit vom auszugebenden Wert wird entweder die Form einer ganzen Zahl (also ohne Dezimalpunkt) oder eines Dezimalbruchs gewählt. Dem Betrag nach sehr große oder sehr kleine Zahlen werden in normierter Schreibweise ausgegeben (Abschn. 3.3.3).
- Führende Nullen (des ganzzahligen Anteils) und die einem Dezimalbruch folgenden Nullen werden unterdrückt.
- Vor der Zahl steht entweder das Minuszeichen (bei negativen Zahlen) oder ein Leerzeichen (bei positiven Zahlen).
- Nach der Zahl wird ein Leerzeichen ergänzt (das man allerdings bei der Ausgabe einer einzelnen Zahl nicht sehen kann!).

Mit Hilfe der PRINT-Anweisung läßt sich auch ein Textausdruck auf dem Bildschirm ausgeben:

n **PRINT** *textausdruck*

Textausdrücke werden im Abschnitt 8 behandelt. In den meisten Fällen besteht der Textausdruck in einer PRINT-Anweisung aus einer Zeichenkette (Textkonstanten), mitunter aus einer Textvariablen:

```
450 PRINT "Uebungsbeispiel:"
730 PRINT "Resultat ="
180 PRINT T$
```

Hier besteht die Wirkung der PRINT-Anweisung darin, daß nur der Wert der Zeichenkette bzw. der Textvariablen – also die Folge der darin enthaltenen Zeichen (einschließlich der Leerzeichen) – linksbündig auf dem Bildschirm erscheint. Dabei wird *nichts modifiziert;* es wird also z. B. kein Leerzeichen angefügt.

Gibt man in einer PRINT-Anweisung keinen Parameter an, so wird eine *leere Zeile* ausgegeben:

```
920 PRINT
```

4.1.3. Standardausgabe mit Ausgabelisten

Wünscht man mehrere Ausgaben nacheinander, so können entsprechend viele PRINT-Anweisungen benutzt werden. Die entsprechenden Ausschriften stehen dann aber auf verschiedenen Zeilen. Möchte man dagegen *mehrere* Ausgaben auf *derselben* Zeile haben, muß die PRINT-Anweisung in modifizierter Form eingesetzt werden.

Wie bereits erwähnt, sind dabei zwei verschiedene *Trennzeichen* möglich, die in derselben Ausgabeliste auch gemischt auftreten dürfen:

- Wird hinter einem Ausgabeelement der PRINT-Anweisung ein *Semikolon* angegeben, so wird der Zeiger für die Position des Anfangs der nächsten Ausgabe *(Schreibmarke) direkt hinter* das letzte geschriebene Zeichen gesetzt. Es wird auch dann nicht zur nächsten Zeile übergegangen, wenn das betreffende Element das letzte in der Ausgabeliste einer PRINT-Anweisung ist. Die nächste Ausgabe erfolgt also unmittelbar hinter der vorhergehenden, und zwar unabhängig davon, ob sie durch ein Ausgabeelement derselben PRINT-Anweisung oder durch eine später folgende PRINT-Anweisung verlangt wird. Die Anweisungsfolgen

  ```
  510 PRINT "Resultate:" ; X ; Y
  ```

  bzw.

  ```
  400 PRINT "Resultate:" ;
  ...
  460 PRINT X ;
  ...
  510 PRINT Y
  ```

  ergeben dieselben Ausschriften, z. B.

  ```
  Resultate: 32  56
  ```

  Die Leerzeichen zwischen den Zahlen entstehen durch das positive Vorzeichen und das stets nachgestellte, oben bereits erwähnte Leerzeichen.
  Reicht die Ausgabezeile nicht mehr für den Wert des nächsten Ausgabeelements aus, so wird er vollständig auf der nächsten Zeile ausgeschrieben.

Zur anschaulichen Darstellung soll das **Programm 4.1** dienen. Dabei wurden übrigens die in den Ausgabelisten stehenden Zahlen bereits bei der Eingabe durch den Editor in die Standarddarstellung umgeformt. Im Gegensatz dazu werden die Zeichenketten unverändert ausgedruckt; außerdem werden zwischen ihnen keine Leerzeichen eingeschoben. Der Variablen X wird im Programm kein Wert zugewiesen. Daher ist hier der Wert 0 ausgedruckt, der vom Interpreter automatisch jeder Zahlenvariablen bei deren erstmaliger Verwendung zugeordnet wird. (Andere Werte müssen explizit zugewiesen werden.)
Manche Interpreter erlauben es, das Semikolon vor und nach Zeichenketten wegzulassen. Darunter leidet allerdings die Übersichtlichkeit der Ausgabeliste.

- Bei Verwendung von *Kommas* erzielt man eine *spaltenorientierte* Ausgabe. Vergleichbar damit ist die Nutzung von Büroschreibmaschinen, die eine Spalteneinteilung durch Tabulatoren gestatten. Der Abstand dieser Tabulatoren hängt von der größten möglichen Länge einer Zahl ab, häufig beträgt er 13 bis 16 Zeichenpositionen. Nimmt man den Wert 14 an, so stehen die

Tabulatoren (bei einer Zählung ab 1) also auf den Positionen 1, 15, 29, 43, . . ., usf., je nach der maximalen Ausgabebreite (**Bild 4.1**).

Die Wirkungsweise ist nun folgende:
Gibt man hinter einem Ausgabeelement der PRINT-Anweisung ein Komma an, so wird die Schreibmarke nach erfolgter Ausgabe auf die Position des nächsten Tabulators weitergestellt; erforderlichenfalls auf den Anfang der nächsten Zeile. In die so gebildeten Spalten werden die Werte der Ausgabeelemente *linksbündig* eingeordnet. Auch hier ist das Druckbild unabhängig davon, ob die Ausgabeelemente in derselben oder in verschiedenen PRINT-Anweisungen auftreten.

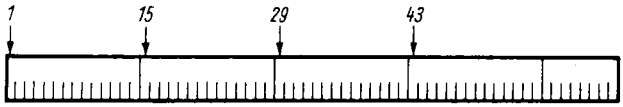


Bild 4.1. Spalteneinteilung der Bildschirmausgaben durch Tabulatoren

Programm 4.1. PRINT-Anweisungen mit Semikolons

```
100 REM <<<<<<<<<< AUSGABELISTEN MIT SEMIKOLONS >>>>>>>>>>
110 :
120 LPRINT -1 ; +12.34 ; 123 ;
130 LPRINT -1234.56 ; 1.23456E+07
140 LPRINT "-1" ; "+12.34" ; "+123" ;
150 LPRINT "-1234.56" ; "12345600"
160 LPRINT "3 + 7 =" ; 3 + 7
170 LPRINT "Gesamtergebnis:" ; "X" ; X
180 :
190 END
200 REM ================================================


-1  12.34  123 -1234.56  1.23456E+07
-1+12.34+123-1234.5612345600
3 + 7 = 10
Gesamtergebnis:X 0
```

Programm 4.2. PRINT-Anweisungen mit Kommas

```
100 REM <<<<<<<<<<< AUSGABELISTEN MIT KOMMAS >>>>>>>>>>>
110 :
115 WIDTH LPRINT 64
120 LPRINT "|" , "|" , "|" , "|"
130 LPRINT -1 , +12.34 ,
140 LPRINT +123 , -1234.56 , 1.23456E+07
150 LPRINT "|" , "|" , "|" , "|"
160 LPRINT "-1" , "+12.34" , "+123" ,
170 LPRINT "-1234.56" , "12345600"
180 LPRINT "|" , "|" , "|" , "|"
190 LPRINT "3 + 7 =" , 3 + 7
200 LPRINT "Gesamtergebnis:" , "X" , X
210 LPRINT "|" , "|" , "|" , "|"
220 :
230 END
240 REM ================================================


|             |             |             |
-1             12.34          123          -1234.56
 1.23456E+07
|             |             |             |
-1            +12.34        +123          -1234.56
12345600
|             |             |             |
3 + 7 =         10
Gesamtergebnis:             X              0
|             |             |             |
```

Programm 4.3. PRINT-Anweisungen mit Kommas und Semikolons

```
100 REM <<<< AUSGABELISTEN MIT KOMMAS UND SEMIKOLONS >>>>>
110 :
120 LPRINT "|" , "|" , "|" , "|"
130 LPRINT "Koordinaten:" ,
140 LPRINT "X =" ; X , "Y =" ; Y , "Z =" ; Z
150 LPRINT "|" , "|" , "|" , "|"
160 LPRINT "HOCH" , "HAUS" , "-->" , "HOCH" ; "HAUS"
170 LPRINT "|" , "|" , "|" , "|"
180 LPRINT ">" , T¤ , "<" ,
190 LPRINT ">" ; T¤ ; "<"
200 LPRINT "|" , "|" , "|" , "|"
210 :
220 END
230 REM ================================================
```

```
|               |               |               |
Koordinaten:    X = 0           Y = 0           Z = 0
|               |               |               |
HOCH            HAUS            -->             HOCHHAUS
|               |               |               |
>                               <               ><
|               |               |               |
```

Programm 4.4. Druck einer Potenztafel

```
100 REM <<<<<<<<<<<<<<<<<< POTENZTAFEL >>>>>>>>>>>>>>>>>>
110 :
120 LPRINT "N" , "N^2" , "N^3" , "N^4"
130 LPRINT
140 LET N = 0
150 :
160 LET Q = N*N
170 LPRINT N , Q , N*Q , Q*Q
180 LET N = N+1
190 IF N<=10 THEN GOTO 160
200 :
210 END
220 REM ================================================
```

```
N             N^2           N^3           N^4

 0             0             0             0
 1             1             1             1
 2             4             8             16
 3             9             27            81
 4             16            64            256
 5             25            125           625
 6             36            216           1296
 7             49            343           2401
 8             64            512           4096
 9             81            729           6561
 10            100           1000          10000
```

Zur Demonstration der Unterschiede zu Ausgabelisten mit Semikolons dienen die **Programme 4.2 und 4.3.** Bei ihnen wurden die Positionen der Tabulatoren bei der Ausgabe mit Hilfe von Strichen markiert. Man erkennt, wie beim Überschreiten der Spaltenbreite selbständig zum nächsten Tabulator weitergegangen wird, falls ein Komma in der Ausgabeliste steht.

Der Variablen T¤ im Programm 4.3 wurde bis zum Drucken noch kein Wert zugewiesen. Daher wird der Wert "" ausgegeben, mit dem der Interpreter sie bei der erstmaligen Verwendung initialisiert: die *leere Zeichenkette.* Es wird also an den betreffenden Stellen *nichts* gedruckt. (Andere Werte sind explizit zuzuweisen.)

Ein weiteres Beispiel vermittelt das **Programm 4.4.** Es druckt eine Tafel der Potenzen n^2, n^3 und n^4 für alle ganzen Zahlen n von 0 bis 10 aus. Der Leser sieht hier nochmals die Positionierung von Zeichenketten und Zahlen verschiedener Länge bei Ausgabelisten mit Kommas. In diesem Programm treten einige Anweisungen auf, die noch nicht besprochen wurden. Trotzdem dürften aber wohl an dieser Stelle keine Verständnisschwierigkeiten auftreten.

4.1.4. Erweiterte Druckbildgestaltung

Die bisher besprochenen Mittel gestatten bereits eine recht flexible Druckbildgestaltung. Damit sind die von BASIC gebotenen Möglichkeiten aber noch nicht erschöpft:

- Standardmäßig wird (bei Anwendung des Semikolons) kein Leerzeichen zwischen den ausgeschriebenen Werten von Ausgabeelementen eingeschoben; das Leerzeichen hinter Zahlen ist ja mit zu dieser Ausgabe zu rechnen. Möchte man dagegen zwischen zwei Ausschriften (mehr) Platz lassen, so muß man ein zusätzliches Element in der Ausgabeliste vorsehen. Handelt es sich um eine konstante Anzahl von Leerzeichen, so könnte eine entsprechende Zeichenkette benutzt werden:

  130 PRINT X ; " " ; Y

  Dabei läßt sich allerdings im Programm schwer erkennen, wie viele Leerzeichen in Anführungszeichen eingeschlossen wurden. Soll der konkrete Abstand erst während des Programmlaufs berechnet werden, ist auf dieser Basis keine einfache Lösung möglich. Hier hilft die Formatfunktion

  SPC(*anzahl*)

  Sie kann nur als *Ausgabeelement* in einer PRINT-Anweisung auftreten und bewirkt das (zusätzliche) Einschieben der angegebenen Anzahl von *Leerzeichen* (SPaCes) an der betreffenden Stelle der Ausgabezeile:

  130 PRINT X ; SPC(3) ; Y

Das Trennzeichen nach der SPC-Funktion darf bei manchen Interpretern beliebig sein, evtl. auch ganz entfallen. Andere Interpreter reagieren auf Semikolon und Komma entsprechend den Erläuterungen im letzten Abschnitt. Es empfiehlt sich daher, stets ein Semikolon zu verwenden, um den Überblick über die Druckbildgestaltung zu behalten. Beispiele sind im **Programm 4.5** dargestellt.

- Bei der Verwendung von Kommas in der Ausgabeliste werden bestimmte, standardmäßige Tabulatorpositionen benutzt (z. B. 15, 29, 43, . . .,). In zahlreichen Einsatzfällen benötigt man aber Spalten anderer Breite. Da die Zahlenausgaben linksbündig erfolgen und eine unter-

Programm 4.5. PRINT-Anweisungen mit SPC-Funktionen

```
100 REM <<<<<<<< AUSGABELISTEN MIT SPC-FUNKTIONEN >>>>>>>>
110 :
120 LPRINT "ZWISCHEN" ; SPC(2) ; "RAUM"
130 LPRINT "|" , "|" , "|" , "|" , "|"
140 LPRINT "|" ; SPC(13) ; "|" ; SPC(13) ; "|" ;
150 LPRINT SPC(13) ; "|" ; SPC(13) ; "|"
160 LPRINT "|" ; SPC(13) , "|" , SPC(13) ; "|"
170 :
180 END
190 REM ================================================


ZWISCHEN  RAUM
|             |             |             |
|
|             |             |             |             |
|                           |                         |
```

schiedliche Anzahl von Zeichen enthalten können, ist eine entsprechende Druckbildgestaltung auch mit Hilfe der SPC-Funktion recht umständlich. Für solche Fälle bietet BASIC eine weitere Möglichkeit:

TAB(*position*)

Auch diese Funktion ist nur als *Ausgabeelement* in einer PRINT-Anweisung zu verwenden. Sie bewirkt, daß die Ausschrift des Wertes des nächsten Elements auf der angegebenen *Druckposition* beginnt. (Positive Zahlen fangen mit einem Leerzeichen an!) Wurde diese Position bereits durch eine vorhergehende Ausgabe (unter Beachtung des abschließenden Trennzeichens!) überschritten, so erfolgt die Ausschrift zwar an der gewünschten Stelle, aber auf der nächsten Zeile.

Für das Trennzeichen nach der TAB-Funktion gilt das bereits bei der SPC-Funktion Gesagte in gleichem Maß. Es empfiehlt sich auch hier, stets ein Semikolon zu verwenden. Zur anschaulichen Demonstration sollen die Ausschriften des **Programms 4.6** dienen.

Programm 4.6. PRINT-Anweisungen mit TAB-Funktionen

```
100 REM <<<<<<<< AUSGABELISTEN MIT TAB-FUNKTIONEN >>>>>>>>
110 :
120 LPRINT "|" , "|" , "|" , "|"
130 :
140 LET I = 1
150 LPRINT TAB(I) ; "|" ; I ;
160 LET I = I+5
170 LPRINT TAB(I) ; "." ;
180 LET I = I+5
190 IF I<60 THEN GOTO 150
200 :
210 LPRINT TAB(1) ;+1234 ; TAB(15) ; "+1234" ;
220 LPRINT TAB(29) ; -1234 ; TAB(43) ; "-1234"
230 :
240 LPRINT TAB(1) ; "|" ; TAB(15) ; "|" ;
250 LPRINT TAB(29) ; "|" ; TAB(43) ; "|"
260 :
270 END
280 REM ================================================

|              |              |              |
| 1  .    | 11 .    | 21 .    | 31 .    | 41 .    | 51 .
 1234          +1234         -1234          -1234
|              |              |              |
```

Eine weitere Anwendungsmöglichkeit wird im **Programm 4.7** gezeigt. Hier geht es um die grafische Darstellung der Funktion $y = x^2$. Dabei liegt die x-Achse senkrecht nach unten, die y-Achse waagerecht nach rechts. Die ausgegebene Grafik ist daher von links zu betrachten.

In beiden Funktionen darf als *Argument* ein arithmetischer Ausdruck im Wertebereich von 0 . . . 255 verwendet werden, als Sonderfall eine einzelne Zahlenvariable oder Zahlenkonstante. Erforderlichenfalls wird der Wert des angegebenen Ausdrucks vom Interpreter selbständig gerundet. Ist das Argument größer als die Länge der Ausgabezeile, wird an den Anfang der nächsten Zeile gegangen. Weiterhin gilt auch an dieser Stelle das Prinzip, daß die Ausschriften am Zeilenende nie getrennt werden. Reicht die Länge der Ausgabezeile nicht, so erfolgt die vollständige Ausschrift am Anfang der nächsten Zeile.

Wie bereits erwähnt, können die variierenden Werte der Ausgabeelemente unterschiedlich lange Ausschriften zur Folge haben. In Abhängigkeit davon muß möglicherweise die Druckspezifikation verändert werden. Dazu ist es aber erforderlich, während des Programmlaufs abzufragen, welche aktuelle Druckposition inzwischen erreicht wurde. Dazu steht in BASIC eine Funktion zur Verfügung:

POS(*x*)

Diese Funktion liefert den Wert der *nächsten Druckposition.*

Es handelt sich hier um keine weitere Druckspezifikation, sondern um eine Standardfunktion, die allerdings meist im Zusammenhang mit der PRINT-Anweisung benutzt wird. Das Argument *x* ist wieder ein beliebiges Blindargument, da – wie bereits erwähnt – jede BASIC-Funktion ein Argument haben muß.

Eine Anwendung der drei bisher in diesem Abschnitt behandelten Funktionen wird im **Programm 4.8** demonstriert, das allerdings auch einige noch nicht besprochene Sprachelemente enthält. Trotzdem sollte der Leser selbständig die Frage beantworten, wie die angegebenen Ausschriften zustande kommen.

Programm 4.7. Grafische Darstellung der Funktion $y = x^2$

```
100 REM <<<<<<<<<<<<<<< KURVENDARSTELLUNG >>>>>>>>>>>>>>>>
110 :
120 LET X = -7
130 IF X <> 0 THEN LPRINT "|"; TAB(X*X+1); "*" : GOTO 160
140   LPRINT "*-------------------------" ;
150   LPRINT "------------------------>"
160 LET X = X+1
170 IF X<=7 THEN GOTO 130
180 LPRINT "V"
190 :
200 END
210 REM ================================================
```

```
|                                                 *
|                                     *
|                         *
|                 *
|         *
|    *
|*
*-------------------------------------------------->
|*
|    *
|         *
|                 *
|                         *
|                                     *
|                                                 *
V
```

Programm 4.8. Einsatz der POS-Funktion

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION POS >>>>>>>>>>>>>>>>>>
110 :
120 LPRINT "|" ; TAB(19) ; "|"
130 LET T¤ = "Zimmermann, Wilhelm"
140 LET N = 4711
150 LPRINT T¤ ; TAB(19) ; N
160 LPRINT T¤ ;
170 IF POS(X)>20 THEN LPRINT TAB(19) ;  : GOTO 190
180   LPRINT SPC(1) ;
190 LPRINT N
200 LPRINT "|" ; TAB(19) ; "|"
210 :
220 END
230 REM ================================================
```

```
|                  |
Zimmermann, Wilhelm
                  4711
Zimmermann, Wilhelm  4711
|                  |
```

4.1.5. Formatschablonen

Die TAB-Funktion bietet die Möglichkeit, Tabellen mit unterschiedlich breiten Spalten aufzubauen. Nachteilig macht sich aber die Eigenschaft der PRINT-Anweisung bemerkbar, alle Ausgaben linksbündig in die Spalten einzuordnen. Außerdem werden – je nach dem Zahlenwert – unterschiedliche Darstellungsformen gewählt. Dadurch wird beispielsweise der Einsatz in der ökonomischen Datenverarbeitung erschwert, wo eine positionsgerechte Ausgabe erforderlich ist. Aus diesem Grund gibt es noch eine weitere Sprachanweisung, deren konkrete Schreibweise allerdings stark vom jeweiligen Interpreter abhängt. In diesem Buch wird folgende Form gewählt:

n **PRINT USING** *formatschablone* ; *ausgabeliste*

Diese Anweisung bewirkt eine Ausgabe der in der Liste enthaltenen Elemente unter Beachtung der Formatschablone.

Die *Ausgabeliste* darf arithmetische Ausdrücke und Textausdrücke enthalten, deren Reihenfolge den Angaben in der Formatschablone entsprechen muß. Als Trennzeichen sind gleichberechtigt meist Komma und Semikolon zulässig. Ein Semikolon bzw. ein Komma am Ende der Liste hat dieselbe Bedeutung wie bei der einfachen PRINT-Anweisung: Nach der Ausgabe wird keine neue Zeile begonnen, sondern die Schreibmarke überhaupt nicht oder bis zum nächsten Tabulator vorgerückt.

Die *Formatschablone* ist vom Typ STRING, also eine Textkonstante oder eine Textvariable. Alle in dieser Zeichenkette angegebenen Zeichen werden mit Ausnahme der folgenden unverändert ausgegeben:

Nummernzeichen **#**
Punkt **.**
Potenzierungszeichen **^**
inverser Schrägstrich \ (oder Prozentzeichen %)
Ausrufungszeichen **!**
kommerzielles Und-Zeichen **&**

In diejenigen Felder, die durch diese Zeichen gewissermaßen *freigehalten* wurden, werden die Werte der Elemente der Ausgabeliste *eingetragen,* und zwar in der Reihenfolge ihres Auftretens. Sind mehr Elemente als Felder vorhanden, wird die Formatschablone wieder von vorn benutzt.

Programm 4.9. PRINT-USING-Anweisungen mit ganzen Zahlen

```
100 REM <<<<<<< FORMATSCHABLONEN MIT GANZEN ZAHLEN >>>>>>>
110 :
120 LET M1¤ = "MOTOR"     : LET T1 = 3
130 LET M2¤ = "DYNAMO"    : LET T2 = 1
140 LET N = 9500
150 :
160 LPRINT "An dem Fussballspiel "; M1¤ ;" gegen "; M2¤ ;
170 LPRINT " nahmen" ; N ; "Zuschauer teil."
180 LPRINT "Das Resultat lautete " ; T1 ; ":" ; T2 ; "."
190 :
200 LPRINT "An dem Fussballspiel "; M1¤ ;" gegen "; M2¤ ;
210 LPRINT USING " nahmen ##### Zuschauer teil." ; N
220 LPRINT USING "Das Resultat lautete ## : ## ."; T1, T2
230 :
240 END
250 REM ================================================
```

```
An dem Fussballspiel MOTOR gegen DYNAMO nahmen 9500
Zuschauer teil.
Das Resultat lautete  3 : 1 .
An dem Fussballspiel MOTOR gegen DYNAMO nahmen  9500 Zuschauer
teil.
Das Resultat lautete  3 :  1 .
```

Das letzte Element der Liste muß aber stets gerade in das letzte Feld der Schablone kommen. Es gibt verschiedene Typen von solchen *Feldern* in Formatschablonen:

- *ganze Zahlen:* **####**
  Das zugeordnete Element der Liste wird auf eine ganze Zahl gerundet und *rechtsbündig* in das Feld eingeordnet. Führende Nullen werden durch Leerzeichen ersetzt. Die Größe des Felds muß für die auftretenden Werte ausreichend sein. Dabei ist (mindestens bei negativen Zahlen) für das Vorzeichen eine Stelle vorzusehen. In dem **Beispielprogramm 4.9** wurde dieselbe Ausschrift zum Vergleich sowohl mit PRINT-Standardanweisungen als auch mit PRINT-USING-Anweisungen realisiert. Einen weiteren Vergleich ermöglicht das **Programm 4.10.** Hier wird wieder eine Potenztafel gedruckt. Im Gegensatz zum Programm 4.4 werden dabei die Zahlen positionsgerecht eingesetzt.
- *Dezimalbrüche:* **###.###**
  Das Listenelement wird seinem Wert entsprechend in das Feld eingeordnet. Eine evtl. unmittelbar vor dem Dezimalpunkt stehende Null sowie alle abschließenden Nullen werden ausgegeben. Vor dem Punkt muß (zumindest bei negativen Zahlen) eine Position für das Vorzeichen reserviert werden. Einige Beispiele sind im **Programm 4.11** angeführt.

Programm 4.10. Druck einer Potenztafel mit Hilfe von PRINT-USING-Anweisungen

```
100 REM <<<<<<<<<<<<<<<<< POTENZTAFEL >>>>>>>>>>>>>>>>>
110 :
120 LPRINT " N    N^2     N^3      N^4"
130 LPRINT
140 FOR N=0 TO 10
150   LET Q = N*N
160   LPRINT USING "##    ###    ####    #####"; N,Q,N*Q,Q*Q
170 NEXT N
180 :
190 END
200 REM ================================================
```

```
 N    N^2     N^3      N^4

 0      0       0        0
 1      1       1        1
 2      4       8       16
 3      9      27       81
 4     16      64      256
 5     25     125      625
 6     36     216     1296
 7     49     343     2401
 8     64     512     4096
 9     81     729     6561
10    100    1000    10000
```

Programm 4.11. PRINT-USING-Anweisung mit Dezimalbruch

```
100 REM <<<<<<<< FORMATSCHABLONE MIT DEZIMALBRUCH >>>>>>>>
110 :
120 LET N$ = "Lehmann"
130 LET W = 25.7
140 :
150 LPRINT "Der Sportler " ; N$ ; " erreichte eine Weite";
160 LPRINT USING " von ##.## Metern." ; W
170 :
180 END
190 REM ================================================
```

```
Der Sportler Lehmann erreichte eine Weite von 25.70 Metern.
```

Programm 4.12. PRINT-USING-Anweisung mit Exponententeil

```
100 REM <<<<<<< FORMATSCHABLONE MIT EXPONENTENTEIL >>>>>>>
110 :
120 LET O$ = "zum Mond"
130 LET E = 384400!
140 :
150 LPRINT "Die Entfernung " ; O$ ; " betraegt" ;
160 LPRINT USING "###.#^^^^ Kilometer." ; E
170 :
180 END
190 REM ================================================
```

```
Die Entfernung zum Mond betraegt 38.4E+04 Kilometer.
```

Programm 4.13. Vergleich der PRINT-USING-Anweisungen für Zahlen

```
100 REM <<<<<<<<<<<<<<< FORMATSCHABLONEN >>>>>>>>>>>>>>>
110 :
120 LPRINT "|" , "|" , "|" , "|"
130 LPRINT USING "####           "; -1, 12.3, 123.45, -1234
140 LPRINT USING "####.##        "; -1, 12.3, 123.45, -1234
150 LPRINT USING "####.##^^^^    "; -1, 12.3, 123.45, -1234
160 LPRINT "|" , "|" , "|" , "|"
170 :
180 END
190 REM ================================================
```

```
|              |              |              |
  -1             12            123          %-1234
  -1.000         12.300        123.450      %-1234.000
-100.000E-02    123.000E-01    123.450E+00  -123.400E+01
|              |              |              |
```

Programm 4.14. PRINT-USING-Anweisungen mit Vorzeichen und Scheckstern

```
100 REM <<<<<<<<<< VORZEICHEN UND SCHECKSTERNE >>>>>>>>>>
110 :
120 LET S1 = +100
130 LET S2 = -30.7
140 LET S3 = +133
150 LET S4 = -.137
160 :
170 LPRINT USING "M +###.##      " ; S1 , S2 , S3 , S4
180 LPRINT USING "M ####.##+     " ; S1 , S2 , S3 , S4
190 LPRINT USING "M ####.##-     " ; S1 , S2 , S3 , S4
200 LPRINT USING "M **##.##      " ; S1 , S2 , S3 , S4
210 LPRINT USING "M +**#.##      " ; S1 , S2 , S3 , S4
220 LPRINT USING "M **##.##+     " ; S1 , S2 , S3 , S4
230 LPRINT USING "M **##.##-     " ; S1 , S2 , S3 , S4
240 :
250 END
260 REM ================================================
```

```
M +100.00      M  -30.70      M +133.00      M   -0.14
M  100.00+     M   30.70-     M  133.00+     M    0.14-
M  100.00      M   30.70-     M  133.00      M    0.14-
M *100.00      M *-30.70      M *133.00      M **-0.14
M +100.00      M *-30.70      M +133.00      M **-0.14
M *100.00+     M **30.70-     M *133.00+     M ***0.14-
M *100.00      M **30.70-     M *133.00      M ***0.14-
```

- *halblogarithmische Darstellung (Exponentialschreibweise):* ##.##^^^^
  Die Darstellung der *Mantisse* durch eine ganze Zahl oder einen Dezimalbruch erfolgt wie oben. Dabei wird in der Regel der Exponent so gesucht, daß die vorderste Ziffer der Mantisse (an der Position des *zweiten* Nummernzeichens) von null verschieden ist. Ansonsten gilt für die Darstellung des Dezimalbruchs das oben bereits Gesagte: Mindestens bei negativen Zahlen wird vor dem Dezimalpunkt eine Stelle für das Vorzeichen benötigt.
  Die Potenzierungszeichen halten den Platz für den *Exponententeil* frei. Dabei ist zu beachten, daß (meist) genau vier Stellen vorhanden sein müssen:

  1. Zeichen: Basiszeichen E
  2. Zeichen: Vorzeichen des Exponenten
  3. Zeichen: } Exponent
  4. Zeichen: }

  Als Beispiel kann das **Programm 4.12** dienen.

Im **Programm 4.13** wurden weitere Beispiele zur PRINT-USING-Anweisung zusammengestellt. Man erkennt daran folgendes: Wenn ein Element der Ausgabeliste nicht ohne Verlust signifikanter Stellen in das angegebene Format umzuformen ist, so druckt der Interpreter diese Zahl zwar aus, bringt aber eine *Fehlermeldung* (im vorliegenden Fall das Prozentzeichen %).

Es gibt noch Sonderformen der PRINT-USING-Anweisung, die insbesondere in der *ökonomischen Datenverarbeitung* von Interesse sind. Sie sollen hier nur stichwortartig erwähnt werden. Die Beispiele sind im **Programm 4.14** enthalten:

- unbedingte Angabe des *Vorzeichens:*
  +## Vorzeichen *vor* der Zahl
  ##+ Vorzeichen *nach* der Zahl
  ##− nur *Minus*zeichen *nach* der Zahl
- Ersatz führender Nullen durch sog. *Scheckstern:*
  Anstelle der *ersten beiden* Nummernzeichen sind Sterne * anzugeben.

Darüber hinaus bieten einige BASIC-Interpreter PRINT-USING-Anweisungen für das Einordnen von *Texten* in Formatschablonen an:

- *unverändertes* Einfügen des Wertes eines Textausdrucks: &
  An der angegebenen Position der Formatschablone wird die betreffende Zeichenkette *voll-*

Programm 4.15. PRINT-USING-Anweisungen mit Zeichenketten

```
100 REM <<<<<<< FORMATSCHABLONE FUER ZEICHENKETTEN >>>>>>>
110 :
120 LET M¤ = "\        \, !. ###   &"
130 :
140 LET N¤ = "Lehmann" : LET V¤ = "Kurt"
150 LET A = 32 : LET B¤ = "Automateneinsteller"
160 LPRINT USING M¤ ; N¤ , V¤ , A , B¤
170 :
180 LET N¤ = "Unterdoerfer" : LET V¤ = "A."
190 LET A = 8 : LET B¤ = "Schueler"
200 LPRINT USING M¤ ; N¤ , V¤ , A , B¤
210 :
220 LET N¤ = "Meier" : LET V¤ = "Hans-Juergen"
230 LET A = 112 : LET B¤ = "Rentner"
240 LPRINT USING M¤ ; N¤ , V¤ , A , B¤
250 :
260 END
270 REM ================================================
```

```
Lehmann    , K.    32   Automateneinsteller
Unterdoerf, A.      8   Schueler
Meier      , H.   112   Rentner
```

ständig eingefügt. Dadurch kann die Ausgabezeile länger werden als die Schablone! Es handelt sich hier um das einzige Formatsteuerzeichen, bei dem die Formatschablone nicht eindeutig die Länge der Ausgabe bestimmt.

- *(linksbündiges)* Einordnen des Wertes eines Textausdrucks: \ \
  Dabei werden die erste und die letzte Position des für die betreffende Zeichenkette in der Formatschablone reservierten Bereichs durch inverse Schrägstriche markiert, bei manchen Interpretern durch Prozentzeichen. Zwischen diesen Marken ist die erforderliche Anzahl von Leerzeichen anzugeben. Ist die einzuordnende Zeichenkette länger als der vorgesehene Bereich, so werden nur die vorderen Zeichen eingesetzt und die überzähligen, hinteren weggelassen.
- Einsetzen des *ersten* Zeichens eines Textausdrucks in die entsprechende Position der Formatschablone: !

Zur Demonstration der Wirkung dieser Formatsteuerzeichen für die Ausgabe von Zeichenketten wird das **Programm 4.15** angegeben.

4.1.6. Festlegen der Ausgabebreite

Für die Ausgabe auf dem *Bildschirm* wird vom Interpreter eine Standardbreite, meist 64 oder 80 Zeichen, vorausgesetzt. In einigen Fällen mag eine individuelle Festlegung der maximalen

Programm 4.16. Festlegung der maximalen Ausgabebreite

```
100 REM <<<<<<<<<< FESTLEGUNG DER AUSGABEBREITE >>>>>>>>>>
110 :
120 LET T¤ = "1234567890"
130 LPRINT T¤ ; T¤ ; T¤ ; T¤ ; T¤ ; T¤
140 LPRINT
150 WIDTH LPRINT 43
160 LPRINT T¤ ; T¤ ; T¤ ; T¤ ; T¤
170 LPRINT
180 WIDTH LPRINT 23
190 LPRINT T¤ ; T¤ ; T¤
200 LPRINT
210 WIDTH LPRINT 13
220 LPRINT T¤ ; T¤
230 LPRINT
240 WIDTH LPRINT 3
250 LPRINT T¤
260 LPRINT
270 LPRINT 12345
280 :
290 WIDTH LPRINT 64
300 END
310 REM ================================================


12345678901234567890123456789012345678901234567890123456789012345678901234567890

12345678901234567890123456789012345678901234567890
1234567890

123456789012345678901234567890
1234567890

1234567890
1234567890

123
4567
890

 12
345
```

Länge einer Ausgabezeile von Interesse sein. Dafür bietet BASIC eine Sprachanweisung:

n **WIDTH** *ausgabebreite*

Durch diese Anweisung wird für alle folgenden Ausgaben über PRINT-Anweisungen eine neue *Maximalbreite* festgelegt. Der als Parameter angegebene, arithmetische Ausdruck wird auf einen ganzzahligen Wert gerundet.

Manche Interpreter gestatten es, die Ausgabebreiten für Bildschirm und *Drucker* getrennt festzulegen. Die entsprechenden Sprachanweisungen sind unterschiedlich, beispielsweise

n **WIDTH LPRINT** *ausgabebreite*

Andere Interpreter verwenden statt dessen das Schlüsselwort LWIDTH.

Ein Beispiel wird im **Programm 4.16** gegeben. Man sieht hier, daß die Ausgabeelemente erst dann geteilt werden, wenn die festgelegte Breite so klein wird, daß sich der Wert nicht mehr anders ausgeben läßt.

4.2. Zuordnen von Eingabedaten innerhalb des Programms

Eine einfache, aber wenig flexible Vorgehensweise ist es, die zu verarbeitenden Daten im Programm selbst bereitzustellen. Es gibt dabei zwei Formen, die im folgenden noch detailliert behandelt werden:

- Die benutzten Zahlen und Zeichenketten sind im Programm *verstreut* an denjenigen Stellen angegeben, an denen sie benötigt werden. Sie lassen sich daher nur schwer ändern.
- Die benötigten Daten werden *zusammengefaßt* definiert. Aus diesem Vorrat wird bei Bedarf fortlaufend gelesen. Daher lassen sich die Datenbestände leicht austauschen.

Es ist allerdings klar, daß die berechneten Resultate in beiden Fällen für jeden Programmlauf identisch sind, falls keine zusätzlichen Bedienereingaben über die Tastatur erfolgen.

4.2.1. Wertzuweisung

Wie bereits erwähnt, werden Variablen in BASIC nicht deklariert, bevor man sie benutzt. Sie werden dem Interpreter während der Übersetzungsarbeit einfach dadurch bekannt, daß sie verwendet werden. Er trägt sie beim erstmaligen Auftreten in seine Symboltabelle ein und initialisiert sie. An den Beispielprogrammen im Abschnitt 4.1.3 wurde bereits festgestellt, daß dieser *Anfangswert* bei Zahlenvariablen 0 (null) ist, bei Textvariablen "" (leere Zeichenkette).

Ist jedoch ein anderer Wert für eine Variable erforderlich, wie z. B. in den Programmen des Abschnitts 4.1, so kann man eine dafür vorgesehene Sprachanweisung benutzen:

n **[LET]** *variablenname* = *ausdruck*

Als Ausdruck kann ein arithmetischer Ausdruck (Abschn. 5.1) oder ein Textausdruck (Abschn. 8) eingesetzt werden. Durch die LET-Anweisung wird der links angegebenen *Variablen* der *Wert* des rechts stehenden Ausdrucks *zugewiesen.* In den Programmablaufplänen der Bilder 2.8 und 2.9 wurde dafür das Symbol ← benutzt.

Häufig auftretende Spezialfälle von Ausdrücken sind einzelne Zahlen- und Textkonstanten sowie Zahlen- und Textvariablen. Bei Konstanten wird der Wert der Zahl bzw. der Zeichenkette in dem entsprechenden Speicherbereich der Variablen abgelegt:

```
110 LET I = 2
130 LET T□ = "Zimmermann, Wilhelm"
140 LET N = 4711
```

Es ist weiterhin möglich, mit Hilfe der LET-Anweisung den Wert einer Variablen auf eine andere zu übertragen. Nach Ausführung dieser Operation haben beide Variablen denselben Wert:

```
120 LET J = I
190 LET Z = V(I)
```

Das Schlüsselwort LET darf übrigens auch weggelassen werden. Das ist beispielsweise im Testprogramm 3.2 geschehen. Der Interpreter erkennt die auszuführende Operation dann am Gleichheitszeichen. Es handelt sich dabei um den einzigen Fall, in dem ein Schlüsselwort überflüssig ist!

Manche Interpreter erlauben *Mehrfachzuweisungen* der Form

```
350 LET I = J = 3
```

Bei anderen wiederum führt eine solche Angabe zu einer Verwechslung, weil das Gleichheitszeichen eine Doppelbedeutung hat: Es wird auch bei Vergleichen eingesetzt (Abschn. 5.3.1).

Durch die Verwendung des Gleichheitszeichens wird übrigens verschleiert, daß die Wertzuweisung ein *Prozeß* ist, keine statische Gleichheit. Man erkennt das z. B. an der Programmzeile

```
130 LET J = J + 1
```

Der Interpreter geht hier folgendermaßen vor:

- Zunächst wird der *Wert* des Ausdrucks auf der rechten Seite ermittelt. Dazu liest der Interpreter den in der Speicherzelle J stehenden Wert und erhöht ihn um eins.
- Dann wird dieser Wert in den Speicherbereich der links stehenden *Variablen* eingetragen. Das ist hier dieselbe Zelle J, deren Inhalt also durch die gegebene Anweisung inkrementiert wird.

Bei der Wertzuweisung ist darauf zu achten, daß den Zahlenvariablen nur Zahlenwerte, den Textvariablen nur Zeichenkettenwerte zugewiesen werden können. Größere Interpreter mit mehreren Typen von Zahlenvariablen führen bei arithmetischen Ausdrücken selbständig Umwandlungen durch.

Übrigens bieten einige BASIC-Systeme Sprachanweisungen zum *Austausch* der Werte zweier Variablen an (SWAP, EXCHANGE), die sich günstig bei Sortierverfahren einsetzen lassen.

4.2.2. Programminterne Datenbestände

Im Abschnitt 1.5.1 wurde über den Stapelbetrieb gesprochen, bei dem Lochkartenstapel als sog. Jobs zur Abarbeitung an Großrechner übergeben werden. Dort ist es üblich, nach den Lochkarten mit dem (Quell-)Programm noch weitere Karten mit den Eingabedaten bereitzustellen, die bei Bedarf nacheinander gelesen werden.

Das gleiche Prinzip wurde auch in BASIC vorgesehen, obwohl diese Programmiersprache für den interaktiven Betrieb bestimmt ist. Dabei ist folgendermaßen vorzugehen:

- Die für das Programm erforderlichen *Daten* werden in besonderen Sprachanweisungen *definiert*:

  n **DATA** *konstantenliste*

  Die *Konstantenliste* enthält Zahlen und Zeichenketten, die voneinander durch Kommas getrennt sind. Zeichenketten müssen dabei nur dann in Anführungszeichen eingeschlossen werden, wenn sie selbst Kommas enthalten oder mit Leerzeichen beginnen bzw. enden:

  ```
  7240 DATA Lehmann, 5348
  7320 DATA "Meier, Hans", 8404
  ```

  DATA-Anweisungen dürfen an *beliebigen Stellen* des Programms auftreten. Meist bringt man sie aber – wie beim Lochkartenstapel! – am Ende des Programms unter. Sie lassen sich dort auch relativ leicht austauschen; bei Erweiterungen ist hinreichend Platz vorhanden.

Nach dem Schlüsselwort DATA dürfen in der betreffenden Zeile *nur noch Konstanten* enthalten sein, die durch Kommas getrennt sind! So ist es z. B. nicht möglich, dort Ausdrücke anzugeben oder Erläuterungen (REM-Anweisungen) anzufügen.

Vor Beginn des Programmlaufs werden alle in DATA-Anweisungen stehenden Konstanten unter Beachtung der Zeilennummern und der Reihenfolge in den einzelnen Listen *hintereinander* in einen dafür vorgesehenen Speicherbereich geladen; sie bilden eine sequentiell organisierte *Datei* (Abschn. 9.3.1).

- In dieser Datei kann man, am Anfang beginnend, fortlaufend Werte lesen. Dazu steht eine Sprachanweisung zur Verfügung:

  n **READ** *eingabeliste*

  Diese Anweisung entnimmt (von der aktuellen Position ab) der DATA-Datei *nacheinander* Werte und weist sie den Elementen der Eingabeliste in der Reihenfolge ihres Auftretens zu. Die *Eingabeliste* enthält Variablen(-namen), die durch Kommas voneinander getrennt sind. Dabei ist zu beachten, daß Zahlen zu Zahlenvariablen, Zeichenketten zu Textvariablen zugeordnet werden müssen.

Anschaulich kann man sich vorstellen, daß ein *Zeiger* fortlaufend durch die sequentielle DATA-Datei geführt wird **(Bild 4.2).** Er zeigt ständig auf diejenige Konstante, die als nächste zu lesen ist. Insgesamt muß eine für alle READ-Anweisungen ausreichende Anzahl von Konstanten in DATA-Anweisungen definiert werden, sonst erfolgt eine Fehlermeldung.

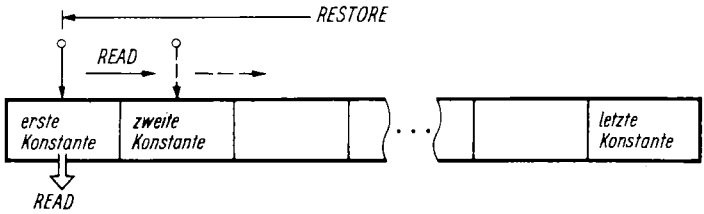


Bild 4.2. Sequentielle Organisation der durch DATA-Anweisungen definierten Datei

- In einem Programm könnte es erforderlich sein, dieselben Daten mehrfach zu verwenden. Hier hilft die Sprachanweisung

  n **RESTORE** [*zeilennummer*]

  Durch diese Anweisung wird der Datenzeiger *zurückgesetzt*, und zwar auf diejenige Konstante, die am Anfang der durch die angegebene Zeilennummer spezifizierten DATA-Anweisung steht. Läßt man diese Nummer weg, so wird die DATA-Datei wieder von vorn gelesen (Bild 4.2). Die Anweisungen RUN und CLEAR enthalten implizit die RESTORE-Anweisung.

Ein einfaches Beispiel für den kombinierten Einsatz der Anweisungen READ und RESTORE vermittelt das **Programm 4.17**, das sicher ohne weitere Erläuterungen verständlich ist. Die Anordnung der Werte im Speicher zeigt **Bild 4.3.**

Programm 4.17. Einsatz der Anweisungen DATA, READ und RESTORE

```
100 REM <<<<<<<<<<<<< DATA-READ-ANWEISUNGEN >>>>>>>>>>>>>
110 :
120 READ A , B , C
130 RESTORE
140 READ D , E , F , G
150 RESTORE 210
160 READ H , I
170 LPRINT A ; B ; C ; D ; E ; F ; G ; H ; I
180 END
190 :
200 DATA 1 , 2
210 DATA 3 , 4
220 :
230 REM ================================================
```

```
1  2  3  1  2  3  4  3  4
```

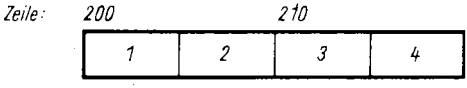


Bild 4.3. DATA-Datei des Programms 4.17

Im **Programm 4.18** wird die Fläche mehrerer Rechtecke berechnet, deren Kantenlängen durch DATA-Anweisungen gegeben sind. Von Interesse ist der Abbruchtest: Tritt (mindestens) eine Länge Null auf, so wird das Programm beendet.

Programm 4.18. Berechnung von Rechteckflächen

```
100 REM <<<<<<<<<<<< RECHTECKSBERECHNUNGEN >>>>>>>>>>>>>
110 :
120 LPRINT "Seite A" , "Seite B" , "Flaeche F"
130 LPRINT
140 :
150 READ A , B
160 LET F = A*B
170 IF F=0 THEN END
180 :
190 LPRINT A , B , F
200 GOTO 150
210 :
220 DATA 3 , 2
230 DATA 4 , 9
240 DATA 1 , 5
250 DATA 6 , 7
260 DATA 8 , 0
270 :
280 REM ================================================
```

```
Seite A       Seite B       Flaeche F

 3             2             6
 4             9             36
 1             5             5
 6             7             42
```

4.3. Eingeben von Daten über die Tastatur

Der dialogorientierten Arbeitsweise mit der Programmiersprache BASIC ist es angemessen, erforderliche Daten erst zur Laufzeit des Programms auf Anforderung hin einzugeben. Hierfür bietet BASIC eine einfache Standardanweisung, die für die meisten Anwendungen ausreichend ist. In manchen Fällen sind aber auch Eingabemöglichkeiten nützlich, die darüber hinausgehen. Bei einem ersten Durcharbeiten dieses Buches kann man sie durchaus zunächst überschlagen. Ein Leser, der bereits einige Fertigkeiten im Programmieren mit BASIC erworben hat, wird sich sicher auch diesen Abschnitten zuwenden.

4.3.1. Standardeingabe

Die Anweisung für die Standardeingabe über die Tastatur ist in ihrem Aufbau der READ-Anweisung gleich:

n **INPUT** [;] *eingabeliste*

Auch hier enthält die Eingabeliste Variablen(-namen), die durch Kommas voneinander getrennt sind. Die *Wirkungsweise* ist folgende:

- Erreicht der Interpreter die INPUT-Anweisung, so gibt er auf der aktuellen Position der Schreibmarke ein *Fragezeichen* und ein folgendes Leerzeichen aus. Dann wartet er auf eine Eingabe des Bedieners.
- Der Bediener muß nun für *alle* in der Liste aufgeführten Variablen in der richtigen Reihenfolge Werte festlegen. Dazu hat er zulässige *Zahlen-* bzw. *Textkonstanten,* durch Kommas getrennt, einzugeben. Auch hier dürfen die Anführungszeichen entfallen, soweit die Zeichenketten keine Kommas bzw. keine führenden oder abschließenden Leerzeichen enthalten.

- Am Ende seiner Eingabe muß der Bediener die *Zeilenendetaste* betätigen. Daraufhin arbeitet der Interpreter weiter. Er entfernt bei Zahlen alle evtl. enthaltenen Leerzeichen. Bei Textkonstanten werden die führenden und die abschließenden Leerzeichen eliminiert. Lediglich bei der Verwendung von Anführungszeichen bleibt die Zeichenkette – wie bereits erwähnt – unverändert.
  Die erhaltenen Werte weist der Interpreter in der entsprechenden Reihenfolge den *Variablen der Eingabeliste* zu. Wurden zu viele, zu wenige oder nicht typgerechte Konstanten eingegeben, so wird meist die gesamte Eingabe mit einer Fehlermeldung erneut angefordert.

Wurde vom Bediener nichts (außer den erforderlichen Kommas) eingegeben, so bleiben bei manchen BASIC-Interpretern die bisher für die betreffenden Variablen gültigen Werte erhalten, bei anderen dagegen werden diese Variablen gelöscht.

Nach der Eingabe wird die *Schreibmarke* an den Anfang der nächsten Zeile gesetzt. Das ist nicht immer günstig, wenn man ein bestimmtes Format auf dem Bildschirm erreichen möchte. Daher bieten einige Interpreter die Möglichkeit, diese Zeilenschaltung zu umgehen. Dazu ist hinter dem Schlüsselwort ein *Semikolon* anzugeben:

```
250 INPUT ; A,B
```

In diesem Fall steht die Schreibmarke hinter dem letzten eingegebenen Zeichen. Das entspricht der Situation nach einer PRINT-Anweisung, deren Ausgabeliste mit einem Semikolon abgeschlossen wurde. Folgt auf die obige Eingabe beispielsweise die Anweisung

```
260 PRINT "  ->  Flaeche: A*B =" ; A*B
```

so ergibt sich auf dem Bildschirm

```
? 3,4   ->  Fläche: A*B = 12
```

wobei die Bedienerangaben wieder eingerahmt wurden.

In den Programmbeispielen dieses Buches wird von dieser Variante Gebrauch gemacht. Bietet ein konkreter Interpreter diese Möglichkeit nicht, so läßt man das Semikolon einfach weg.

4.3.2. Anforderungstext

Sieht ein Programm mehrere Eingaben vor, so ist es für den Bediener schwierig, zu wissen, welche (und wie viele!) Werte beim Auftauchen eines Fragezeichens jeweils einzugeben sind. Möglich wäre eine Druckausgabe *vor* der INPUT-Anweisung. Das Beispiel im Abschnitt 4.3.1 könnte durch die Anweisung

```
249 PRINT "Kantenlaengen: A,B" ;
```

ergänzt werden. Dann ergäbe sich auf dem Bildschirm

```
Kantenlaengen: A,B? 3,4   ->  Flaeche: A*B = 12
```

Dasselbe läßt sich mit einer Modifikation der INPUT-Anweisung erreichen:

n **INPUT** [;] *textkonstante* ; *eingabeliste*

Diese Anweisung schreibt die angegebene Zeichenkette noch *vor* dem Fragezeichen aus:

```
250 INPUT ; "Kantenlaengen: A,B" ; A,B
```

Es entsteht das gleiche Schriftbild, wie es oben angegeben wurde. Wird das Semikolon nach dem Schlüsselwort INPUT weggelassen, so erfolgt die Ausgabe durch Anweisung 260 auf der neuen Zeile.

In einigen Fällen mag die Kennzeichnung einer Anforderung durch die Textkonstante allein bereits ausreichen, das Fragezeichen und das folgende Leerzeichen vielleicht sogar stören. Hier bieten manche Interpreter die Möglichkeit, nach der Zeichenkette anstelle des Semikolons ein Komma anzugeben:

n **INPUT** [;] *textkonstante* , *eingabeliste*

Bei dieser Anweisung wird nach der angegebenen Textkonstanten *nichts* weiter ausgeschrieben,

sondern die Schreibmarke für die Eingabe steht hinter dem letzten Zeichen der Textkonstanten:

```
310 INPUT ; "Name: " , N¤
320 INPUT "  Vorname: " , V¤
330 INPUT ; "PLZ: " , P
340 INPUT "  Ort: ", O¤
350 INPUT "Straße/Hausnummer: " , S¤
```

Nach der Abarbeitung steht auf dem Bildschirm z. B.

Name: [Lehmann] Vorname: [Fritz]
PLZ: [8027] Ort: [Dresden]
Straße/Hausnummer: [Grenzallee 53]

Die Werte "Lehmann", "Fritz", 8027, "Dresden" und "Grenzallee 53" wurden den entsprechenden Variablen zugewiesen. Die Schreibmarke zeigt auf den Anfang der nächsten Zeile.

4.3.3. Zeilenorientierte Eingabe

Bei der Eingabe von Zeichenketten mit Hilfe der bisher besprochenen Standardanweisung entstehen u. U. Schwierigkeiten:

- Führende und abschließende Leerzeichen werden eliminiert.
- Im Text enthaltene Kommas werden als Trennzeichen betrachtet.

Will man diese Komplikationen umgehen, so muß man die Zeichenkette – wie bereits erwähnt – in Anführungsstriche setzen. Das kann aber beim Eingeben schnell vergessen werden und führt dann zu Fehlern. Deshalb bieten manche Interpreter eine modifizierte Eingabeanweisung an:

n **LINE INPUT** [;] [*textkonstante* ;] *textvariable*

Hier wird zunächst die wahlweise anzugebende Textkonstante als Aufforderung ausgeschrieben, und zwar *ohne* Fragezeichen. Fehlt diese Zeichenkette, so erfolgt *keinerlei* Ausschrift. Danach erwartet der Interpreter eine Eingabe des Bedieners, wobei *alle* Zeichen bis zum abschließenden *Zeilenendezeichen* übernommen werden. Diese Zeichenkette, die u. a. auch führende und abschließende Leerzeichen sowie Kommas enthalten darf, wird der angegebenen Textvariablen zu-

Programm 4.19. Eingaben mit der LINE-INPUT-Anweisung

```
100 REM <<<<<<<<<<<< INPUT  UND  LINE INPUT >>>>>>>>>>>>
110 :
120 LPRINT
130 LINE INPUT "Name, Vorname, Alter, Beruf? " ; T¤
140 LPRINT T¤
150 :
160 INPUT "Name, Vorname, Alter, Beruf" ; N¤ , V¤ , A , B¤
170 LET M¤ = "\         \, !.  ###   &"
180 LPRINT  N¤ , V¤ , A , B¤
190 LPRINT USING  M¤ ; N¤ , V¤ , A , B¤
200 :
210 GOTO 120
220 :
230 REM ================================================
```

```
Schneider,Konrad,28,Lehrer
Schneider     Konrad          28          Lehrer
Schneider , K.   28   Lehrer

Richter, Ursula, 51, Sachbearbeiterin
Richter       Ursula          51          Sachbearbeiterin
Richter   , U.   51   Sachbearbeiterin
```

gewiesen. Hier ist also *keine* Eingabeliste zulässig:

```
770 LINE INPUT "Naechste Zeile: " ; Z¤
```

Das wahlweise nach dem Schlüsselwort angebbare Semikolon hat dieselbe Wirkung wie bei der INPUT-Anweisung: Normalerweise steht die Schreibmarke nach Ausführung der Eingabe am Anfang der nächsten Zeile. Wird aber das Semikolon eingeschoben, so wird die Schreibmarke hinter das letzte eingegebene Zeichen positioniert.

Im **Programm 4.19** werden die LINE-INPUT-Anweisung und die INPUT-Anweisung miteinander verglichen.

Manche Interpreter benutzen bei der behandelten Anweisung das Schlüsselwort LINPUT.

4.3.4. Eingabe ohne Endezeichen

Bei den bisher behandelten Eingabeanweisungen muß der Bediener zum Schluß die Zeilenendetaste drücken. Das ist bei komplizierten Eingaben vorteilhaft; man kann noch einmal kontrollieren, ob alles in Ordnung ist, und evtl. korrigieren. Insbesondere bei sehr kurzen Eingaben von konstanter Länge (z. B. nur ein Zeichen) ist das zusätzliche Betätigen der Zeilenendetaste störend. Hier bieten einige Interpreter die BASIC-Funktion

INPUT¤(*anzahl*)

Diese Funktion liest die angegebene Anzahl von Zeichen ein. Der *Ablauf* ist dabei folgendermaßen:

- Wenn der Interpreter auf eine INPUT¤-Funktion stößt, so hält er an und erwartet eine Bedienereingabe. Es erfolgt *keinerlei* Ausschrift! Ist eine Anforderung zweckmäßig, muß sie durch eine zusätzliche PRINT-Anweisung *vor* der Nutzung der INPUT¤-Funktion erfolgen.
- Jetzt kann der Bediener die erforderliche Anzahl von Zeichen eingeben. Dabei dürfen durchaus auch Steuerzeichen (außer CTRL-C) eingegeben werden, selbst der Kode der Zeilenendetaste wird angenommen und führt nicht zum Abbruch der Eingabe! Die eingegebenen Zeichen werden *nicht* ausgeschrieben!

Die Zeichenkette kann anschließend einer Textvariablen zugewiesen

```
420 LET T¤ = INPUT¤(3)
```

oder in Text- und Vergleichsausdrücken benutzt werden. Natürlich läßt sie sich auch mit einer PRINT-Anweisung ausdrucken.

Für die *Anzahl* kann ein arithmetischer Ausdruck im Wertebereich 1 . . . 255 benutzt werden. Dezimalbrüche werden gerundet.

Im **Programm 4.20** wurde von der INPUT¤-Funktion Gebrauch gemacht. Dabei wird ange-

Programm 4.20. Einsatz der INPUT¤-Funktion

```
100 REM <<<<<<<<<<<<<<< FUNKTION INPUT¤ >>>>>>>>>>>>>>>>
110 :
120 PRINT ; "Wochentag: " ;
130 LET A¤ = INPUT¤(2)
140 IF A¤="mo" THEN PRINT "Montag" : GOTO 250
150   IF A¤="di" THEN PRINT "Dienstag" : GOTO 250
160     IF A¤="mi" THEN PRINT "Mittwoch" : GOTO 250
170       IF A¤="do" THEN PRINT "Donnerstag" : GOTO 250
180         IF A¤="fr" THEN PRINT "Freitag" : GOTO 250
190           IF A¤="sa" THEN PRINT "Sonnabend" : GOTO 250
200             IF A¤="so" THEN PRINT "Sonntag" : GOTO 250
210 PRINT
220 PRINT "Die Abkuerzung " ; A¤ ;
230 PRINT " bezeichnet keinen Wochentag!"
240 :
250 REM Es folgt die weitere Verarbeitung
260 :
270 REM ================================================
```

nommen, daß in einem Programm der Wochentag anzugeben ist. Der Bediener braucht nun nur eine der üblichen Abkürzungen eines Wochentags (hier in Kleinbuchstaben) einzutippen; das Programm schreibt statt dessen die volle Bezeichnung aus. (Die REM-Anweisung soll die weitere Verarbeitung andeuten, die sich jetzt anschließen könnte.)

Eine andere Anwendung ist im **Programm 4.21** dargestellt. Hier wird angenommen, daß ein BASIC-Programm nur von befugten Personen benutzt werden soll. Es fordert daher zunächst einen Kode an. Dabei ist die Anzahl der Kodezeichen ein Teil des Kodes selbst: Das erste Zeichen wird als Anzahl der noch folgenden Zeichen gedeutet. Dazu muß man das Textzeichen einer Ziffer in den entsprechenden Zahlenwert dieser Ziffer umrechnen, also z. B. das Zeichen "5" in die Zahl 5. Das realisiert die VAL-Funktion, die im Abschnitt 8.2.1 behandelt wird. Der Kode für Programm 4.21 lautet danach: 508-15. Bei jeder anderen Eingabe wird das vom externen Zusatzspeicher eingelesene Programm im Hauptspeicher wieder gelöscht.

Manche Interpreter haben statt der INPUT¤-Funktion eine Eingabeanweisung GET, mit deren Hilfe man genau ein Zeichen eingeben kann. Der Wert wird einer Zahlen- oder Textvariablen zugewiesen.

Programm 4.21. Prüfung eines Benutzerkodes

```
100 REM <<<<<<<<<<<<<<<<< BENUTZERKODE >>>>>>>>>>>>>>>>>
110 :
120 PRINT "Benutzerkode"
130 LET N = VAL(INPUT¤(1))
140 LET C¤ = INPUT¤(N)
150 IF C¤="08-15" THEN GOTO 200
160   PRINT "Sie sind unbefugt!"
170   NEW
180   END
190 :
200 PRINT "Sie sind zugelassen."
210 REM Beginn des geschuetzten Programms
220 :
230 REM ================================================
```

4.3.5. Abfrage einer Eingabe

Alle bisher für die Eingabe über Tastatur bestimmten Sprachelemente von BASIC führen dazu, daß der Interpreter auf eine entsprechende Reaktion des Bedieners wartet. Das ist aber mitunter überhaupt nicht erwünscht. Beispielsweise könnte man ein Programm geschrieben haben, das zunächst laufend Zwischenresultate ausdruckt. Diese Ausgaben verfolgt man und möchte gern beim Erreichen eines bestimmten Wertes diesen Programmteil verlassen und in einen anderen verzweigen. Stoppt der Rechner nach jeder Ausgabe, um dem Bediener die Möglichkeit zum Eingreifen zu geben, so muß man ständig eine Fortsetzungstaste betätigen. Schöner wäre es, wenn man nur im richtigen Moment einen Eingriff vorzunehmen brauchte. Hier hilft die BASIC-Funktion

INKEY¤

Diese Funktion *fragt ab,* ob in der Zwischenzeit, seit der letzten bereits übernommenen Eingabe, eine Taste gedrückt wurde:

- Wurde *keine* Taste gedrückt, liefert die Funktion als Wert die *leere Zeichenkette.*
- Wurde dagegen eine Taste betätigt, so übermittelt INKEY¤ das entsprechende *Zeichen.*

In beiden Fällen wird also die Interpretation *ohne Halt* fortgesetzt! Im Programm läßt sich dann analysieren, ob eine – und wenn ja, welche – Taste gedrückt wurde. Daran kann sich eine entsprechende Reaktion des Programms anschließen.

Es ist aber auch hier möglich, auf das Eingeben eines Zeichens zu warten. In diesem Fall müßte man eine Warteschleife aufbauen:

```
150 LET C¤ = INKEY¤
160 IF C¤ = "" THEN GOTO 150
```

Programm 4.22. Reaktionstest mit der INKEY¤-Funktion

```
100 REM <<<<<<<<<<<<<<<< REAKTIONSTEST >>>>>>>>>>>>>>>>>
110 :
120 FOR I=1 TO 3000 : NEXT
130 LET Z = 0
140 PRINT "jetzt!" ;
150 LET Z = Z+1
160 IF INKEY¤="" THEN GOTO 150
170 PRINT " >" ; 5.9*Z ; "Millisekunden"
180 GOTO 120
190 :
200 REM ================================================
```

Diese Kombination hat die gleiche Wirkung wie die Anweisung

```
150 LET C¤ = INPUT¤(1)
```

Als Beispiel bringt **Programm 4.22** einen Reaktionstest. Beim Erscheinen der Ausschrift „jetzt!" muß der Bediener so schnell wie möglich irgendeine Taste betätigen. Bis dahin inkrementiert das Programm laufend den Zähler Z. Der bis zum Tastendruck erreichte Wert von Z wird noch in Millisekunden umgerechnet und ausgegeben. Der entsprechende Faktor (hier 5.9) hängt von Computer und Interpreter ab und muß durch Messungen über ein längeres Zeitintervall ermittelt werden.

5. Rechnen mit BASIC

Bisher wurde noch nicht im einzelnen darüber gesprochen, welche Möglichkeiten BASIC für die Datenverarbeitung bietet. Trotzdem wurden schon Zahlenrechnungen ausgeführt und vom Leser sicher sofort verstanden. Das ist ein Zeichen dafür, wie einfach das Rechnen mit BASIC ist. Nunmehr soll diese Problematik detailliert behandelt werden. Dabei stehen auch die Vergleiche und die logischen Operationen mit zur Diskussion, weil BASIC – im Gegensatz zu anderen höheren Programmiersprachen – keinen besonderen Datentyp für logische Aussagen einführt, sondern hier mit Zahlenwerten arbeitet.

5.1. Arithmetische Ausdrücke

BASIC benutzt für die Darstellung von Formeln die gleiche Schreibweise, wie sie von der Mathematik her bekannt ist. Einige geringfügige Modifikationen sind z. B. dadurch bedingt, daß man in einer Programmiersprache alles fortlaufend in gleicher Höhe („auf der Zeile") schreiben muß; man kann Exponenten nicht hochsetzen, Indizes nicht tiefstellen. Multiplikationszeichen müssen explizit angegeben werden. Waagerechte Bruchstriche sind ebenfalls nicht möglich. Daher sehen manche Formeln doch etwas anders aus, als man sie in der Mathematik niederschreibt. Aber es ist nicht schwierig, sich darauf umzustellen.

- Ein *arithmetischer Ausdruck* besteht in BASIC aus
  (Zahlen-) Konstanten (Abschn. 3.3.4),
  (Zahlen-) Variablen (Abschn. 3.3.5) und
  (Zahlen-) Funktionen (Abschn. 3.3.6),
  die miteinander durch Rechenoperationen verknüpft sind.

Häufig enthält ein arithmetischer Ausdruck nur eine einzige Konstante, Variable oder Funktion.

5.1.1. Rechenoperationen

In BASIC sind die folgenden, in der **Tafel 5.1** zusammengestellten Rechenoperationen zulässig:

Tafel 5.1. Rechenoperationen in BASIC

Rang	Symbol	Bedeutung
7	( )	Klammerung
6	–	negatives Vorzeichen
5	^ bzw. ↑ oder * *	Potenzierung
4	*	Multiplikation
	/	Division
3	\	ganzzahlige Division
2	MOD	Modulo-Rechnung
1	+	Addition
	–	Subtraktion

Addition (Operatorsymbol +)

N + 1 A + B S + .5 COS(X) + 5

Subtraktion (Operatorsymbol −)

I − 4 X − Y 1.3 − C EXP(F) − 1

Multiplikation (Operatorsymbol *, nicht weglassen!)

7 * F A * B 2.5 * X 5 * LOG(Y)

Division (Operatorsymbol /)

P/3 D/H 3.3/Y TAN(X)/2

Potenzierung (Operatorsymbol ^ oder ↑ , manchmal auch **)

$a^{1,4}$ → A^1.4
x^y → X^Y
$4{,}5^n$ → 4.5^N
$\sin^5(x)$ → SIN(X)^5

Einige Interpreter bieten darüber hinaus noch weitere Rechenoperationen an:

ganzzahlige Division (Operatorsymbol \ )

K\4 M\N 1000\L INPUT□(I\2)

Bei der ganzzahligen Division werden zunächst beide Operanden auf ganze Zahlen gerundet. Dann wird eine (normale) Division ausgeführt. Das Resultat wird abschließend auf die *nächstkleinere* ganze Zahl abgerundet. Auf diese Weise ergeben sich bei den folgenden Beispielen die Werte:

11\3 → 3
16.7\7.3 → 17/7 → 2

Modulo-Rechnung (Operatorschlüsselwort **MOD**)

I MOD 3 N MOD K 370 MOD J TAB(L MOD 10)

Bei der Modulo-Verknüpfung werden die beiden Operanden ebenfalls zunächst gerundet. Dann wird der Rest bestimmt, der sich bei der anschließenden (normalen) Division ergibt. So erhält man beispielsweise folgende Resultate:

11 MOD 3 → 2
16.7 MOD 7.3 → 17 MOD 7 → 3

Im vorliegenden Buch wurden diese beiden Rechenoperationen nicht eingesetzt. Sie waren zwar bei dem für die Programmerprobung benutzten Interpreter vorhanden, fehlen aber bei zahlreichen anderen. Daher werden entsprechende Aufgaben so gelöst, wie es im Programm 5.3 gezeigt ist. Als weiteres Beispiel kann das **Programm 5.1** dienen. Hier wird nach dem *Euklid*ischen Algorithmus, der in mathematischen Nachschlagewerken zu finden ist, der größte gemeinsame Teiler zweier gegebener Zahlen gesucht. Der entsprechende Programmablaufplan ist im **Bild 5.1** dargestellt. Falls für A eine kleinere Zahl als für B eingegeben wurde, so werden im ersten Schritt selbsttätig N und M miteinander vertauscht.

Bei *arithmetischen Operationen* können folgende Probleme auftauchen:

- Überschreitet der berechnete Wert die Größe der höchstzulässigen Zahl, so erfolgt ein sog. *Zahlenüberlauf* (Overflow). Der Interpreter bringt dann eine Fehlermeldung und rechnet mit der größtmöglichen Zahl und dem richtigen Vorzeichen weiter.
- Dieselbe Reaktion erfolgt bei einer *Division durch Null* (Division by Zero).
- Unterschreitet der berechnete Wert die kleinste noch darstellbare Zahl, so erfolgt ein sog. *Zahlenunterlauf*. Der Interpreter rechnet dann mit dem Wert null weiter.

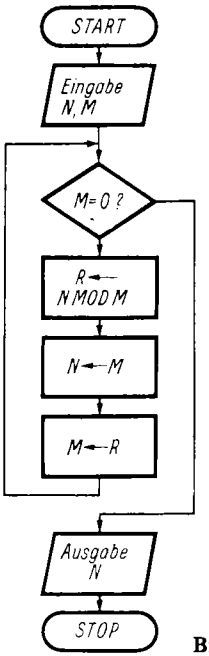


Bild 5.1. Euklidischer Algorithmus

Programm 5.1. Euklidischer Algorithmus

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"           EUKLIDISCHER ALGORITHMUS             "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Zahlen A,B" ; A , B
160 LET M = A
170 LET N = B
180 :
190 IF M=0 THEN GOTO 250
200   LET R = N - INT(N/M)*M : REM  R = N MOD M
210   LET N = M
220   LET M = R
230   GOTO 190
240 :
250 PRINT "-> groesster gemeinsamer Teiler =" ; N
260 PRINT "   kleinstes gemeinsames Vielfaches =" ; A*B/N
270 :
280 END
290 REM ==================================================
```

5.1.2. Aufbau arithmetischer Ausdrücke

Auf der beschriebenen Grundlage lassen sich jetzt komplizierte Berechnungen in BASIC ausdrücken. Dabei ist zu beachten, daß die verschiedenen Operationen in einer bestimmten *Reihenfolge* ausgeführt werden:

- Aus der Mathematik ist die Regel bekannt: Punktrechnung geht vor Strichrechnung. In BASIC ist das noch etwas differenzierter. Die entsprechenden *Vorränge* sind in der Tafel 5.1 dargestellt.
- Bei gleichem Vorrang wird i. allg. die in der betreffenden Zeile weiter *links* stehende Operation *vor* derjenigen ausgeführt, die *rechts* von ihr steht. Nur bei der Potenzierung wird zunächst der Exponent berechnet, dann erst die Basis.
- Es dürfen nie zwei Symbole für Rechenoperationen *unmittelbar* hintereinander stehen! (Beispielsweise führt der Ausdruck A*-B zu einem Fehler. Man kann statt dessen -A*B schreiben.) Die einzige Ausnahme bildet auch hier die Potenzierung; hier ist der Ausdruck A^-2 zulässig.

Läßt sich die zu kodierende Formel nach diesen Regeln noch nicht in BASIC ausdrücken oder ist man sich über die Wirkung der Vorränge im unklaren, so helfen (runde) *Klammern*. Klammerinhalte werden stets zuerst ausgewertet; bei geschachtelten Klammerausdrücken *von innen nach außen*.

Diese Regeln sollen jetzt in einer Reihe von Beispielen angewandt werden:

$\frac{a\,b}{c}$ → A*B/C oder A/C*B

$\frac{a}{b\,c}$ → A/B/C oder A/(B*C)

$x - \frac{y}{2}$ → X-Y/2

ab + cd → A*B + C*D

$\frac{a+b}{c}$ → (A+B)/C

$\frac{R_1 R_2}{R_1 + R_2}$ → R1*R2/(R1+R2)

$ax^2 + bx + c$ → A*X^2 + B*X + C oder (A*X+B)*X+C

$x^{-\frac{n}{3}}$ → X^(-N/3)

a^{b^c} → (A^B)^C

$cx^{(y^2)}$ → C*X^Y^2

$K\,(1 + \frac{p}{100})^n$ → K*(1+P/100)^N

$\tan^2(x) + 3$ → TAN(X)^2 + 3

Bei der Abarbeitung eines arithmetischen Ausdrucks verknüpft der BASIC-Interpreter die Werte der angegebenen Zahlenkonstanten, Variablen und Funktionen und ermittelt ein *Resultat.* Dieser Wert kann in verschiedener Weise benutzt werden:

- rechte Seite einer *LET-Anweisung:*

```
170 LET   Q = X*X + Y*Y
```

- Element in der Ausgabeliste einer *PRINT-(USING-)Anweisung:*

```
440 PRINT   COS(Y) ; (A+B)/(C-D)
```

- Argument einer *Funktion:*

```
TAN(3.14159*G/180)
```

- Operand in einem *Vergleich* (Abschn. 5.3.1):

```
X+5 > 3*A
```

Der *Umfang* eines Ausdrucks wird nur dadurch begrenzt, daß die BASIC-Anweisung, in der er benutzt wird, nicht länger als eine Zeile sein kann.

5.2. Arithmetische Standardfunktionen

Über Standardfunktionen wurde bereits im Abschnitt 3.3.6 gesprochen. Im folgenden Abschnitt sollen nun diejenigen Funktionen detailliert behandelt werden, die als Argument einen arithmetischen Ausdruck zulassen und als Resultat einen Zahlenwert liefern. Dabei ist es durchaus möglich, als Argument einer Funktion wiederum eine Funktion anzugeben.

5.2.1. Zahlenmanipulierung

Zunächst werden einfache Funktionen behandelt, mit denen man die Werte von Ausdrücken analysieren und manipulieren kann:

ABS(*argument*)

Diese Funktion bildet den *Absolutbetrag* des als Argument angegebenen Ausdrucks, indem es negative Zahlen mit dem Faktor −1 multipliziert. Sie entspricht also der mathematischen Schreibweise |x|.

SGN(*argument*)

Diese Funktion ermittelt das *Vorzeichen* (SIGN) des Arguments. Als Werte liefert sie

$$\mathrm{SGN}(x) = \begin{cases} -1 & x < 0 \\ 0 \quad \text{falls} & x = 0 \\ +1 & x > 0 \end{cases}$$

INT(*argument*)

Diese Funktion berechnet die *größte ganze Zahl* (INTeger), die kleiner oder gleich dem Wert des Arguments ist. Handelt es sich um eine positive Zahl, so werden einfach die Stellen hinter dem Dezimalpunkt weggelassen:

INT(2.3) → 2 INT(5.71) → 5

Bei negativen Zahlen muß man beachten, daß die Rundung immer nach der *nächstkleineren* ganzen Zahl führt:

INT(−2.3) → −3 INT(−5.71) → −6

Ein entsprechendes Beispiel zeigt das **Programm 5.2.** Hier werden ganzzahliger Anteil und echter Dezimalbruch positiver Zahlen voneinander getrennt und ausgedruckt. Was ergibt sich bei negativen Werten?

Wird statt dessen das übliche *Runden* eines berechneten Wertes auf die nächstgelegene ganze Zahl gewünscht, so ist vor dem Anwenden der INT-Funktion noch der Betrag .5 zu addieren. Ein solcher Fall ist im **Programm 5.3** dargestellt, bei dem die ganzzahlige Division und die Modulo-Rechnung simuliert werden.

Programm 5.2. Trennen des ganzzahligen Anteils einer Zahl vom echten Dezimalbruch

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION INT >>>>>>>>>>>>>>>>>>
110 :
120 INPUT "Dezimalbruch = " , X
130 PRINT " -> ganzzahliger Anteil =" ; INT(X) ;
140 PRINT SPC(4) ; "Rest =" ; X - INT(X)
150 GOTO 120
160 :
170 REM ================================================
```

Programm 5.3. Ganzzahlige Division und Modulo-Rechnung

```
100 REM <<< GANZZAHLIGE DIVISION UND MODULO-ARITHMETIK >>>
110 :
120 INPUT ; "Operanden: X = " , X
130 INPUT ; "  Y = " , Y
140 LET X0 = INT(X+.5)
150 LET Y0 = INT(Y+.5)
160 LET Q = X0/Y0
170 LET Q0 = SGN(Q)*INT(ABS(Q))
180 PRINT "  -> A\B =" ; Q0 ;
190 PRINT " A MOD B =" ; X0 - Q0*Y0
200 GOTO 120
210 :
220 REM ================================================
```

Programm 5.4. Runden von Dezimalbrüchen

```
100 REM <<<<<<<<<<< RUNDEN VON DEZIMALBRUECHEN >>>>>>>>>>>
110 :
120 INPUT "Zahl" ; Z
130 INPUT "Anzahl der Dezimalstellen" ; N
140 LET F = 10^N
150 PRINT " -> gerundete Zahl =" ; INT(Z*F +.5)/F
160 GOTO 120
170 :
180 REM ================================================
```

Möchte man dagegen nicht alle Ziffern hinter dem Punkt abschneiden, sondern auf eine vorgegebene Anzahl von Dezimalstellen runden, so kann man die im **Programm 5.4** angegebene Variante wählen. Dabei wird die zu rundende Zahl vor dem Anwenden der INT-Funktion um die erforderliche Anzahl von Stellen nach links verschoben. Am Schluß erfolgt eine entsprechende Verschiebung nach rechts.

5.2.2. Quadratwurzel

SQR(*argument*)

Diese Funktion bestimmt die *Quadratwurzel* (SQuare Root) aus dem angegebenen arithmetischen Ausdruck. Sie wird im **Programm** 5.5 angewendet, mit dem sich der Bediener die Wurzeln von eingegebenen Zahlen ausdrucken lassen kann. Dieses Programm kommt nie zum Ende, man muß es durch CTRL-C abbrechen.

Programm 5.5. Quadratwurzel

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION SQR >>>>>>>>>>>>>>>>>>
110 :
120 INPUT ; "Argument X = " , X
130 PRINT " -> SQR(X) =" ; SQR(X)
140 GOTO 120
150 :
160 REM ================================================
```

Bei negativen Werten beendet der Interpreter seine Arbeit mit einer Fehlermeldung. Man kann das durch Einsatz der ABS-Funktion vermeiden:

SQR(ABS(3+X−Y))

Programm 5.6. Berechnung von Dreiecksflächen

```
100 REM <<<<<<<<<<<<<< FLAECHENBERECHNUNGEN >>>>>>>>>>>>>>
110 :
120 LPRINT "Seite A" , "Seite B" , "Seite C" , "Flaeche F"
130 LPRINT
140 :
150 READ A,B,C
160 LET S = (A+B+C)/2
170 IF S=0 THEN END
180 :
190 LPRINT A , B , C , SQR(S*(S-A)*(S-B)*(S-C))
200 GOTO 150
210 :
220 DATA 3,4,5
230 DATA 8,5,9
240 DATA 10,7,6
250 DATA 7,11,9
260 DATA 8,8,12
270 DATA 0,0,0
280 :
290 REM ================================================
```

```
Seite A       Seite B       Seite C       Flaeche F

 3             4             5             6
 8             5             9             19.8997
 10            7             6             20.6625
 7             11            9             31.4195
 8             8             12            31.749
```

Ein Anwendungsbeispiel vermittelt das **Programm** 5.6. Es berechnet nach der bekannten Formel

$$F = \sqrt{s\,(s-a)\,(s-b)\,(s-c)} \quad \text{mit} \quad s = a+b+c$$

die Flächen von Dreiecken, deren Seitenlängen a, b, c bekannt sind. Die Eingabedaten sind in DATA-Anweisungen enthalten. Das Programm wird beendet, wenn alle drei Längen gleich null sind.

Programm 5.7. Lösung von quadratischen Gleichungen

```
100 REM +++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"            'QUADRATISCHE GLEICHUNG              "
120 PRINT"             A * X^2 + B * X + C = 0             "
130 REM +++++++++++++++++++++++++++++++++++++++++++++++++++
140 :
150 REM -------------------- EINGABE ----------------------
160 :
170 PRINT
180 INPUT "Koeffizienten A, B, C" ; A, B, C
190 :
200 REM -------------------- ANALYSE ----------------------
210 :
220 IF A<>0 THEN GOTO 330
230   IF B<>0 THEN GOTO 270
240     IF C=0 THEN PRINT : END
250       PRINT "  Eingabefehler!"
260       GOTO 180
270 LET X = -C/B
280 PRINT "Lineare Gleichung: X =" ; X
290 GOTO 170
300 :
310 REM -------------------- LOESUNG ----------------------
320 :
330 LET U = B/(2*A)
340 LET D = U*U - C
350 LET V = SQR(ABS(D))
360 :
370 REM -------------------- AUSGABE ----------------------
380 :
390 IF D<0 THEN GOTO 450
400 LET X1 = - U + V
410 LET X2 = - U - V
420 PRINT "Loesungen:  X1 =" ; X1 ; "  X2 =" ; X2
430 GOTO 170
440 :
450 PRINT "Loesungen:  X1 =" ; -U ; "+" ; V ; "i"
460 PRINT TAB(13) ; "X2 =" ; -U ; "-" ; V ; "i"
470 GOTO 170
480 :
490 REM ==================================================
```

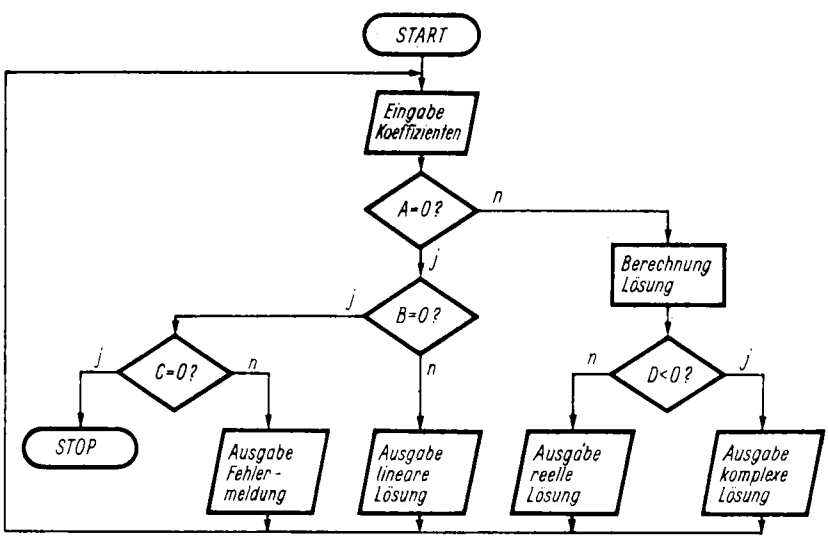


Bild 5.2. Berechnung der Wurzeln einer quadratischen Gleichung

Das **Programm 5.7** zeigt ein komplizierteres Beispiel. Es berechnet die Wurzeln der quadratischen Gleichung

$$a\,x^2 + b\,x + c = 0$$

bei beliebigen Werten der Koeffizienten a, b, c. Auch komplexe Resultate werden durch eine entsprechende Druckaufbereitung ausgegeben. Obwohl einige noch nicht besprochene BASIC-Anweisungen enthalten sind, dürfte dieses Programm dem Leser sicher verständlich sein. (Das Symbol <> bedeutet: verschieden von.) Der Programmablaufplan ist im **Bild 5.2** dargestellt.

5.2.3. Exponential- und Logarithmusfunktionen

EXP(*argument*)

Diese BASIC-Funktion entspricht der *Exponentialfunktion* e^x. Auf ihrer Grundlage lassen sich sowohl Exponentialfunktionen mit einer anderen Basis, wie z. B. 10^x, als auch die hyperbolischen Funktionen sinh x, cosh x usw. berechnen.

Mit Hilfe des **Programms 5.8** kann der Bediener Funktionswerte ermitteln lassen. Was geschieht bei großen Argumenten, z. B. bei der Eingabe der Zahl 90?

Programm 5.8. Exponentialfunktion

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION EXP >>>>>>>>>>>>>>>>>>
110 :
120 INPUT ; "Argument X = " , X
130 PRINT " -> EXP(X) =" ; EXP(X)
140 GOTO 120
150 :
160 REM ================================================
```

LOG(*argument*)

Diese Funktion liefert den Wert des *natürlichen Logarithmus* ln x; daher verwenden manche BASIC-Systeme an dieser Stelle auch das Schlüsselwort LN. Die Funktion ist für negative Werte des Arguments nicht definiert. Was geschieht, wenn im **Programm 5.9** dennoch ein negatives Argument benutzt wird?

Aus dem natürlichen Logarithmus lassen sich sowohl die Logarithmen zu einer anderen Basis, z. B. der dekadische Logarithmus lg x, als auch die Areafunktionen arsinh x, arcosh x usw. berechnen.

Programm 5.9. Logarithmusfunktion

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION LOG >>>>>>>>>>>>>>>>>>
110 :
120 INPUT ; "Argument X = " , X
130 PRINT " -> LOG(X) =" ; LOG(X)
140 GOTO 120
150 :
160 REM ================================================
```

5.2.4. Trigonometrische und zyklometrische Funktionen

SIN(*argument*)
COS(*argument*)
TAN(*argument*)

Diese bekannten Funktionen erwarten das Argument im Bogenmaß. Möchte man statt dessen mit (Alt-)Grad arbeiten, so muß man diese Gradangabe mit dem Faktor

3.14159/180

multiplizieren.

Ein einfaches Beispiel zeigt das **Programm 5.10**. Hier kann sich der Bediener den Sinuswert für eine gegebene Gradzahl berechnen lassen. Entsprechende Programme sind für Kosinus und Tangens möglich.

Im **Programm 5.11** wird nach dem Kosinussatz der ebenen Trigonometrie die dritte Seite eines Dreiecks berechnet, von dem zwei Seiten und der eingeschlossene Winkel bekannt sind.

Programm 5.10. Sinusfunktion

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION SIN >>>>>>>>>>>>>>>>>>
110 :
120 INPUT ; "Grad = " , G
130 LET X = 3.14159/180 * G
140 PRINT " -> X =" ; X ; " SIN(X) =" ; SIN(X)
150 GOTO 120
160 :
170 REM ================================================
```

Programm 5.11. Kosinussatz der ebenen Trigonometrie

```
100 REM <<<<<<<<<<<<<<< DREIECKSBERECHNUNG >>>>>>>>>>>>>>>
110 :
120 INPUT ; "A = " , A
130 INPUT ; "  B = " , B
140 INPUT ; "  GAMMA = " , G
150 PRINT " GRAD" ;
160 PRINT "  ->  C =" ;
170 PRINT SQR(A*A + B*B - 2*A*B*COS(3.14159/180*G))
180 GOTO 120
190 :
200 REM ================================================
```

Programm 5.12. Arcustangensfunktion

```
100 REM <<<<<<<<<<<<<<<<<< FUNKTION ATN >>>>>>>>>>>>>>>>>>
110 :
120 INPUT ; "Tangens = " , T
130 PRINT " -> ATN(T) =" ; 180/3.14159 * ATN(T) ; "Grad"
140 GOTO 120
150 :
160 REM ================================================
```

Programm 5.13. Genauigkeitstests

```
100 REM <<<<<<<<<<<<<<< GENAUIGKEITSTESTS >>>>>>>>>>>>>>>
110 :
120 INPUT "beliebige Zahl X>0 " ; X
130 LPRINT "X =" ; X
140 LPRINT "SQR(X)^2 =" ; SQR(X)^2
150 LPRINT "EXP(LOG(X)) =" ; EXP(LOG(X))
160 LPRINT "TAN(ATN(X)) =" ; TAN(ATN(X))
170 LPRINT "SIN(X)^2 + COS(X)^2 =" ;
180 LPRINT SIN(X)*SIN(X) + COS(X)*COS(X)
190 :
200 END
210 REM ================================================
```

```
X = 7
SQR(X)^2 = 7
EXP(LOG(X)) = 7
TAN(ATN(X)) = 7.00001
SIN(X)^2 + COS(X)^2 = 1
```

Zyklometrische Funktionen sind die Umkehrfunktionen zu den trigonometrischen Funktionen. In BASIC gibt es davon nur die Funktion

ATN(*argument*)

Sie liefert den Hauptwert der Funktion *arctan x* im Bogenmaß. Als Beispiel zum Probieren kann das **Programm 5.12** verwendet werden. Es formt gegebene Werte der Tangensfunktion in die entsprechenden Gradzahlen um.

Die anderen zyklometrischen Funktionen, wie arcsin x und arccos x, lassen sich aus dem Arcustangens berechnen. Darauf soll hier aber nicht eingegangen werden.

Ein abschließendes Beispiel zu den bisher besprochenen Funktionen ist das **Programm 5.13**. In ihm wird als Argument einer Funktion jeweils die entsprechende Umkehrfunktion verwendet; also müßte der ursprüngliche Wert reproduziert werden. Außerdem wird die bekannte Beziehung zwischen Sinus und Kosinus als Test für die Rechengenauigkeit benutzt. In dem ausgedruckten Zahlenbeispiel wurde ein Fall ausgewählt, bei dem sich beim eingesetzten Interpreter einmal eine Abweichung feststellen ließ.

5.2.5. Zufallszahlen

In einer Reihe von Fällen, z. B. bei der Lösung von Simulationsproblemen oder bei Computerspielen, benötigt man zufällig verteilte Zahlen. Nun arbeitet aber ein Computer (glücklicherweise!) nicht zufällig, kann also prinzipiell keine echt zufällig verteilten Zahlen liefern. Es ist jedoch möglich, sich einen solchen Algorithmus auszudenken, daß die damit berechneten Zahlen derart verteilt sind, daß sie zufällig zu sein scheinen. Man nennt derart erzeugte Resultate *Pseudozufallszahlen*. In Wirklichkeit wird aber eine ganz bestimmte, reproduzierbare Zahlenfolge erzeugt!

Möchte man diese determinierte Folge verlassen, so ist der Algorithmus anders zu *initialisieren*. Man muß ihm gewissermaßen einen neuen Anfangswert vorschreiben. Dabei entsteht zunächst die Frage, woher man diese Zahl gewinnt. Und wird der Algorithmus wieder gestartet, so ergibt sich erneut eine determinierte Zahlenfolge, wenn auch eine andere als zuvor.

Hier sollen zunächst diejenigen Sprachelemente behandelt werden, die BASIC für die Erzeugung von Zufallszahlen bereitstellt. Dabei ist zu beachten, daß sich die einzelnen Interpreter stark voneinander unterscheiden. Deshalb ist auf jeden Fall die jeweilige Bedienungsanleitung zu Rate zu ziehen.

RND(*argument*)

Diese Funktion liefert eine *Pseudozufallszahl* z im Intervall $0 \leqq z < 1$. Wenn speziell das Argument den Wert null hat, so wird i. allg. keine neue Zahl ermittelt, sondern die letzte nochmals geliefert. Darüber hinaus kann man zwei verschiedene Varianten von Interpretern unterscheiden:

- Zufallszahlenerzeugung mit *expliziter Initialisierung*
  Hier ist das Argument der RND-Funktion ohne Bedeutung, mindestens dann, wenn es positiv ist. Manche Interpreter gestatten daher, es überhaupt wegzulassen. Bei negativen Werten wird von verschiedenen Interpretern eine Folge konstanter Zahlen erzeugt, die vom Betrag des Arguments abhängen kann.
  Für die Initialisierung steht eine besondere Sprachanweisung

  RANDOMIZE[(*argument*)]

  zur Verfügung. Das angegebene Argument wird für die Ermittlung des Anfangswertes benutzt. Fehlt es, so wird eine Zahl vom Bediener angefordert. Einige Interpreter hingegen benötigen kein Argument. Statt RANDOMIZE ist auch RANDOM gebräuchlich.
- Zufallszahlenerzeugung mit *impliziter Initialisierung*
  In diesem Fall ist das Argument der RND-Funktion signifikant. Ist es positiv, so wird – wie oben – eine Zufallszahlenfolge erzeugt. Manche Interpreter benutzen dabei das (ganzzahlige) Argument N für die Bestimmung einer oberen Schranke für die Pseudozufallszahlen,

deren Werte dann im Bereich $0 \leqq z < N + 1$ liegen. Ist das Argument dagegen negativ, so wird der Algorithmus mit dem Betrag dieses Wertes initialisiert.

Wie erwähnt, erzeugt die RND-Funktion meist Zufallszahlen im Intervall $0 \leqq z < 1$. Möchte man statt dessen Werte im Bereich $m \leqq z < n$ haben, so muß man den Ausdruck

M + (N-M)*RND

berechnen. Er wurde im Programm 5.17 verwendet, das ein Würfelspiel simulieren soll. Da die Funktion INT stets nach unten abrundet, mußte der Faktor 6 bei der RND-Funktion benutzt werden. Dieses Programm kann zum Experimentieren dienen. Mit ihm läßt sich die Auswirkung der Initialisierung des Zufallszahlengenerators und des Arguments der RND-Funktion untersuchen.

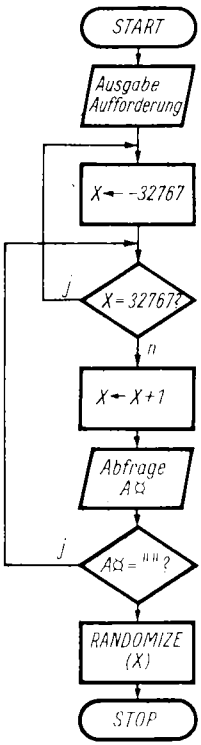
Programm 5.14. Ausgabe von Zufallszahlen

```
100 REM <<<<<<<<<<<< ZUFALLSZAHLENGENERATOR >>>>>>>>>>>>
110 :
120 INPUT "Argument von RANDOMIZE(X)" ; X
130 RANDOMIZE(X)
140 INPUT "Argument von RND(Y)" ; Y
150 LPRINT "X =" ; X , "Y =" ; Y
160 :
170 LPRINT RND(Y) ,
180 GOTO 170
190 :
200 END
210 REM ================================================
```

```
X = 0          Y = 0
 .594184        .594184        .594184        .594184
 .594184        .594184        .594184        .594184
 .594184        .594184        .594184        .594184
 .594184        .594184        .594184        .594184


X = 0          Y = 1
 .278959        .964895        .693033        .19737
 .974314        .174121        .0737999       .923051
 .764704        .319673        .201901        .982384
 .855908        .746172        .136499        .29834
 .770734        .910311        .843835


X = 0          Y =-1
 .308601        .308601        .308601        .308601
 .308601        .308601        .308601        .308601
 .308601


X = 1          Y = 1
 .58041         .128928        .928324        .901162
 .532818        .499882        .286114        .608704
 .342298        .163376        .843915        .883858
 .17126         .204987        .749476        .419742
 .963892        .768147        .321532        .260266
 .492917


X = 2          Y = 1
 .0438479       .891465        .801263        .656113
 .16401         .835579        .238398        .0730044
 .581113        .0987764       .516006        .567164
 .47412         .307071        .392471        .914768
 .544047
```

Möchte man die erzeugten Pseudozufallszahlen jedoch unmittelbar sehen, so kann man statt dessen das **Programm 5.14** benutzen. Hier werden laufend Werte erzeugt und ausgegeben; man muß das Programm mit der Taste CTRL-C anhalten.

Die bisher beschriebenen BASIC-Elemente reichen aber bereits aus, um eine *zufällige Initialisierung* des Zufallszahlengenerators zu erreichen. Dazu wird auf die Funktion INKEY$ zurückgegriffen. Im **Programm** 5.15 wird nach dem Start eine Zählschleife durchlaufen, die man durch das Drücken einer beliebigen Taste unterbrechen kann. Dabei erhält man eine zwar von der bisherigen Laufdauer abhängige, aber doch nicht vorher bekannte Zahl. Sie wird als Argument der RANDOMIZE-Anweisung benutzt. Der entsprechende Programmablaufplan ist im **Bild 5.3** dargestellt.



Programm 5.15. Zufällige Initialisierung der RANDOMIZE-Anweisung

```
100 REM <<<<<<<< INITIALISIERUNG VON RANDOMIZE(X) >>>>>>>>
110 :
120 PRINT "Taste druecken!"
130 LET X = -32767
140 IF X=32767 THEN GOTO 130
150 LET X = X+1
160 IF INKEY$="" THEN GOTO 140
170 RANDOMIZE(X)
180 :
190 REM ================================================
```

Bild 5.3. Zufällige Initialisierung des Zufallszahlengenerators

5.3. Logische Ausdrücke

In den vorangegangenen Abschnitten wurden BASIC-Elemente besprochen, mit deren Hilfe man Berechnungen durchführen kann. Der wesentlichste Vorzug eines Computers ist es aber, daß er Entscheidungen fällen kann. Dazu muß ihm jedoch im Algorithmus vorgeschrieben werden, unter welchen Bedingungen er jeweils in der einen oder anderen Weise weitergehen soll. Solche Bedingungen ergeben sich durch den Vergleich der Werte von arithmetischen Ausdrükken bzw. von Textausdrücken. Diese Vergleichsaussagen lassen sich dann noch durch logische Operationen zu komplexen Aussagen verknüpfen.

- Ein *logischer Ausdruck* besteht in BASIC aus Vergleichsaussagen, die miteinander durch logische Operationen verknüpft sein können.

Häufig enthält ein logischer Ausdruck nur einen einzigen Vergleich.

Logische Ausdrücke können zwei mögliche Werte haben: Sie sind entweder *wahr* (TRUE) oder *falsch* (FALSE).

5.3.1. Vergleiche

Um die Werte von zwei Ausdrücken miteinander vergleichen zu können, stellt BASIC sechs Operatoren zur Verfügung. Sie sind in der **Tafel 5.2** zusammengestellt. Hier taucht erneut das Gleichheitszeichen auf; es hat aber eine andere Bedeutung als in der LET-Anweisung.

x = 7	ist nur dann wahr, wenn x den Wert 7 hat.
I < J	ist immer dann wahr, wenn der Wert von I kleiner als der Wert von J ist.
A >= 10	ist für alle diejenigen Fälle wahr, in denen der Wert von A größer als oder gleich 10 ist.

Tafel 5.2. Vergleichsoperationen in BASIC

Symbol	Bedeutung
=	gleich
< > oder > < oder #	verschieden von
<	kleiner als
>	größer als
< =	kleiner als oder gleich
> =	größer als oder gleich

Der *Vorrang* von Vergleichen ist kleiner als derjenige von Berechnungen. Bei der Auswertung eines Vergleichsausdrucks werden also erst die Werte der beiden arithmetischen Ausdrücke ermittelt und danach der Vergleich ausgeführt:

I <> J+1	ist immer dann wahr, wenn der Wert von I nicht um eins größer als derjenige von J ist.
X^2+Y^2 > R^2	ist wahr für alle Punkte (X, Y), die außerhalb des Kreises mit dem Radius R liegen.

In BASIC ist aber – im Gegensatz zu anderen höheren Programmiersprachen – kein besonderer Datentyp für solche Aussagen vorhanden. Deshalb ermittelt der Interpreter *Zahlenwerte*, um das Ergebnis eines Vergleichs auszudrücken:

falsch → Zahlenwert = 0
wahr → Zahlenwert <> 0

Diesen Wert kann man z. B. einer Zahlenvariablen zuweisen. So verstehen viele Interpreter die bereits im Abschnitt 4.2.1 genannte Zeile

```
350 LET I = J = 3
```

nicht als eine Mehrfachzuweisung (I und J erhalten beide den Wert 3), sondern als einen Vergleich (J=3), dessen Resultat der Variablen I zugewiesen wird! Leider ist dieser Fakt nicht klar zu erkennen, weil das Gleichheitszeichen hier gleichzeitig in seinen beiden unterschiedlichen Bedeutungen auftritt.

Welcher konkrete Wert der wahren Aussage zugeordnet wird, hängt vom Interpreter und von den Werten der verglichenen Ausdrücke ab. Es gibt Interpreter, die hierfür stets den Wert −1 liefern. Wie das **Programm 5.16** zeigt, kann man in diesem Fall die größere von zwei gegebenen

Programm 5.16. Ermittlung der größeren von zwei Zahlen

```
100 REM <<<<<<<<<<<< MAXIMUM ZWEIER ZAHLEN >>>>>>>>>>>>>
110 :
120 INPUT : "Zahlen A,B" ; A , B
130 PRINT " -> MAX =" ; -(A>B)*A -(A<=B)*B
140 GOTO 120
150 :
160 REM ================================================
```

Variablen A und B mit Hilfe eines einzigen Ausdrucks berechnen. Dabei muß man Klammern setzen, damit zuerst die Zahlenwerte (0 oder −1) für die Vergleiche berechnet werden, bevor die Multiplikation mit den Werten der Variablen A und B ausgeführt wird. Andere Interpreter benutzen statt dessen den Wert +1 für eine wahre Aussage.

Wenn man eine Variable, die gebrochene Werte haben kann (Typ REAL), auf *Gleichheit* oder *Verschiedenheit* testen will, muß man besondere Vorsicht walten lassen. Durch die begrenzte Genauigkeit der Zahlendarstellung und der Rechenoperationen in Computern kann es durchaus dazu kommen, daß geringfügige Fehler auftreten. Das Programm 5.13 zeigt dafür ein Beispiel. In einem solchen Fall würde ein Test auf Gleichheit aber möglicherweise nicht erfüllt werden können. Es ist daher empfehlenswert, je nach den vorliegenden Umständen statt dessen die Vergleichsoperationen < oder <= bzw. > oder >= einzusetzen. Das ist in den bisher gezeigten Programmen, in denen Wiederholungsschleifen enthalten sind, bereits realisiert.

5.3.2. Aufbau logischer Ausdrücke

Der folgende Abschnitt enthält eine Reihe komplizierterer Überlegungen. Derjenige Leser, der sich noch nicht mit Problemen der Aussagenlogik beschäftigt hat, kann es ohne Bedenken zunächst überschlagen. Bei vielen Aufgaben benötigt man die hier vermittelten BASIC-Elemente nicht oder kann sie vermeiden. Andererseits ist aber gerade die schnelle Entscheidung bei komplexen logischen Verknüpfungen die große Stärke des Computers. Daher sollen hier wenigstens einige einführende Bemerkungen zu diesem Problemkreis gebracht werden.

Die Vergleichsaussagen (wahr und falsch) lassen sich noch weiter miteinander verknüpfen. Dafür stehen die in der **Tafel 5.3** zusammengestellten Operatoren zur Verfügung. Man bezeichnet sie in Anlehnung an die entsprechenden Verbindungen in der Aussagenlogik auch als logische Operatoren, die damit gebildeten Ausdrücke als logische Ausdrücke. Als Resultate von solchen Ausdrücken ergeben sich wieder die logischen Aussagen wahr und falsch mit den bereits genannten numerischen Werten.

Tafel 5.3. Logische Operationen in BASIC

Rang	Symbol	Bedeutung
4	NOT	Negation (logisches NICHT)
3	AND	Konjunktion (logisches UND)
2	OR	Disjunktion (logisches ODER)
1	XOR	Antivalenz (logisches ENTWEDER-ODER)

Für das *Negieren* einer logischen Aussage bietet BASIC die einstellige Operation:

NOT *logischer ausdruck*

Manche Interpreter benutzen als Operatorsymbol ~ (Tilde) oder ¬ (Überstreichung). Dieser Operator kehrt die Aussage um, auf die er angewendet wird:

I > 5 ist wahr für I = 6, 7, 8, ...
NOT I > 5 ist wahr für I = 5, 4, 3, ...

Die letzte Aussage ließe sich allerdings auch durch eine andere Vergleichsoperation erreichen:

I <= 5 ist wahr für I = 5, 4, 3, ...

Bei den bisherigen Vergleichen war der Zahlenbereich nach einer Seite hin offen. Braucht man dagegen einen abgeschlossenen Bereich, so muß man zwei Aussagen miteinander verknüpfen. Beispielsweise ist für die Zahlen 6, 7, 8 sowohl die oben gegebene Bedingung I > 5 erfüllt als auch die Bedingung I <= 8:

I <= 8 ist wahr für I = 8, 7, 6, ...

Es sind also I > 5 und I <= 8 beide wahr. In der Aussagenlogik heißt diese Verbindung **Konjunktion,** *logisches UND* (Operatorbezeichnung **AND**, manchmal auch **&**):

I > 5 AND I <= 8

Diese Aussage ist nur für die Zahlen 6, 7 und 8 wahr (sowie für alle zwischen 5 und 8 liegenden Dezimalbrüche).

Programm 5.17. Würfelspiel

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"           HAEUFIGKEITSVERTEILUNGEN              "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Argument von RANDOMIZE(X)" ; X
160 RANDOMIZE(X)
170 INPUT "Argument von RND(Y)" ; Y
180 LPRINT "RANDOMIZE(X) mit X ="; X;", RND(Y) mit Y ="; Y
190 LPRINT
200 :
210 DIM H(6) : REM Haeufigkeitsverteilung, Anfangswert = 0
220 :
230 LET M¤ = "####    ###   ###   ###   ###   ###   ###"
240 LPRINT   "   I      1     2     3     4     5     6"
250 LPRINT
260 LET I = 1
270 :
280 LET R = INT(6*RND(Y) +1)
290 LET H(R) = H(R) + 1
300 IF I<>10 AND I<>100 AND I<>1000 THEN GOTO 330
310 LPRINT USING M¤ ; I, H(1),H(2),H(3),H(4),H(5),H(6)
320 :
330 LET I = I+1
340 IF I<=1000 THEN GOTO 280
350 :
360 PRINT "Haeufigkeitsverteilungen ausgedruckt."
370 :
380 END
390 REM ================================================
```

```
RANDOMIZE(X) mit X = 0 , RND(Y) mit Y = 1

   I      1     2     3     4     5     6

  10      1     4     0     0     2     3
 100     23    14     9    15    16    23
1000    178   186   142   169   164   161


RANDOMIZE(X) mit X = 1 , RND(Y) mit Y = 1

   I      1     2     3     4     5     6

  10      2     1     2     3     0     2
 100     17    15    17    14    14    23
1000    170   167   169   175   153   166


RANDOMIZE(X) mit X = 2 , RND(Y) mit Y = 1

   I      1     2     3     4     5     6

  10      4     1     0     2     1     2
 100     15    15    13    24    16    17
1000    168   156   174   175   170   157
```

Eine Anwendung ist im **Programm 5.17**, das bereits im Abschnitt 5.2.5 genannt wurde, dargestellt. Dort wird in denjenigen Fällen, wenn die Variable I weder gleich 10 noch gleich 100, noch gleich 1000 ist, die Ausgabe der bisher erreichten Häufigkeitsverteilung ausgelassen.

In anderen Fällen kann eine resultierende Bedingung bereits als erfüllt gelten, wenn nicht alle, sondern sogar nur eine einzige der Teilbedingungen erfüllt ist. Beispielsweise könnte man bei der Kodierung des letzten Programms auch sagen: Wenn entweder N=10 oder N=100 oder N=1000 ist, dann soll gedruckt werden. Diese Verbindung heißt in der Aussagenlogik **Disjunktion**, *logisches ODER* (Operatorbezeichnung **OR**, manchmal auch !):

IF N=10 OR N=100 OR N=1000 THEN LPRINT ...

Diese Variante konnte aber im Programm 5.17 nicht verwendet werden, weil dort die LPRINT-Anweisung zu lang ist. Allerdings wäre der Einsatz eines Unterprogramms (Abschn. 6.3.1) möglich gewesen.

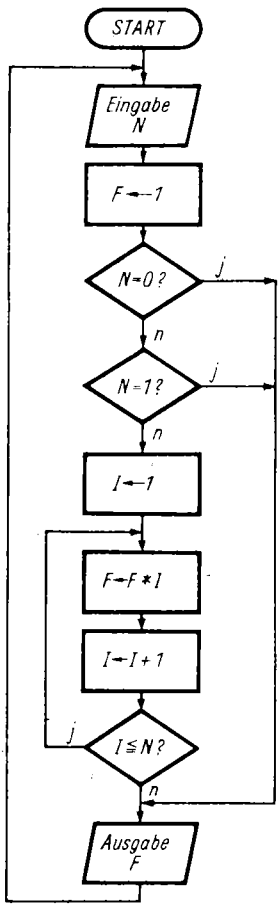
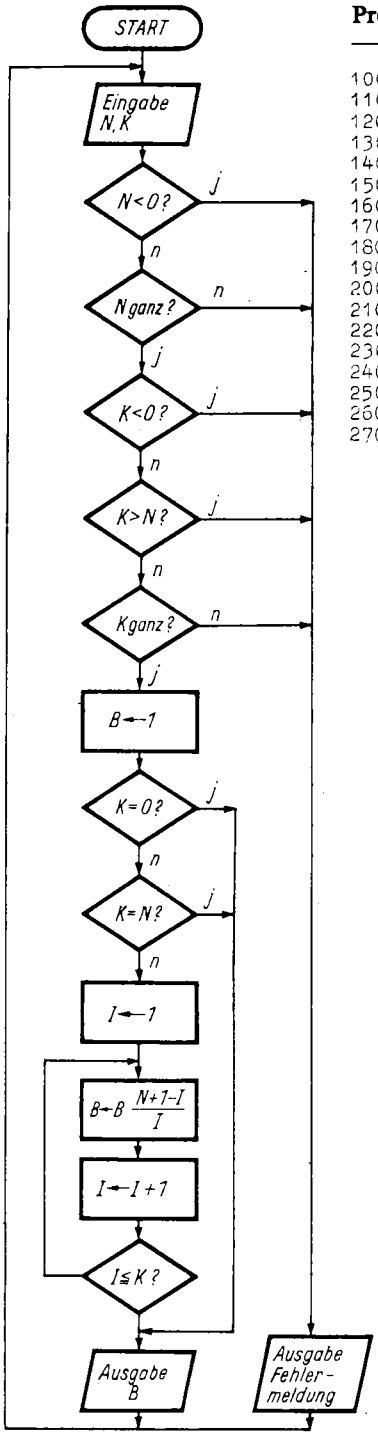


Bild 5.4. Berechnung der Fakultät N!

Programm 5.18. Berechnung der Fakultät

```
100 REM <<<<<<<<<<<<<<<<<<< FAKULTAET >>>>>>>>>>>>>>>>>>>
110 :
120 INPUT ; "natuerliche Zahl" ; N
130 LET F = 1 : REM Anfangswert
140 IF N=0 OR N=1 THEN GOTO 190
150   LET I = 1
160   LET F = F*I
170   LET I = I+1
180   IF I<=N THEN GOTO 160
190 PRINT " -> " ;N ; CHR¤(8) ; "! =" ; F
200 GOTO 120
210 :
220 REM ================================================
```

Anwendungen der OR-Verbindung werden in den Programmen 5.18 und 5.19 gezeigt. Es handelt sich hier um die Berechnung der Fakultät **(Bild 5.4)** bzw. eines Binomialkoeffizienten **(Bild 5.5)** durch fortlaufende Multiplikationen. Einige Bemerkungen sollen zur Erläuterung von Besonderheiten dienen: In der Ausgabeliste des **Programms 5.18** ist das Element CHR¤(8) enthalten. Es erzeugt einen Rückwärtsschritt *(Back-Space)* des Druckers; die Funktion CHR¤(N) wird im Abschnitt 8.2.2 besprochen. Dadurch kommt das Ausrufungszeichen unmittelbar hinter



Programm 5.19. Berechnung von Binomialkoeffizienten

```
100 REM <<<<<<<<<<<<< BINOMIALKOEFFIZIENT >>>>>>>>>>>>
110 :
120 INPUT "Natuerliche Zahlen N,K " ; N , K
130 LET B1 = N<0 OR NOT N=INT(N)
140 LET B2 = K<0 OR K>N OR NOT K=INT(K)
150 IF B1 OR B2 THEN PRINT "Fehleingabe!" : GOTO 120
160 :
170 LET B = 1 : REM Anfangswert fuer Resultat
180 IF K=0 OR K=N THEN GOTO 240
190 LET I = 1
200 LET B = B*(N+1-I)/I
210 LET I = I+1
220 IF I<=K THEN GOTO 200
230 :
240 PRINT " ->" ; N ; "ueber" ; K ; " = " ; B
250 GOTO 120
260 :
270 REM ================================================
```

Bild 5.5. Berechnung des Binomialkoeffizienten $\binom{N}{K}$

die ausgegebene Zahl. Im **Programm 5.19** sorgen die komplexen logischen Ausdrücke dafür, daß keine ungeeigneten Werte für N und K eingegeben werden können. (Im Programm 5.18 ist keine solche Kontrolle vorgesehen!) Mitunter muß geprüft werden, ob von zwei gegebenen Bedingungen genau eine erfüllt ist. Hier hilft die logische Verbindung

Antivalenz, *logisches exklusives ODER* (Operatorbezeichnung **XOR**):
Diese Verknüpfung ergibt genau dann das Resultat wahr, wenn irgendeine von beiden Bedingungen wahr, die andere dann aber jeweils falsch ist:

X = 0 XOR Y = 0 ist wahr für X = 0 und Y<>0
oder für X<>0 und Y = 0

Häufig verwendet man die Antivalenz, um eine Aussage A durch einen logischen Schalter S wahlweise beizubehalten oder durch das Programm zu *negieren*:

A XOR S ergibt den Wert von A, wenn S = falsch
und den Wert NOT A, wenn S = wahr.

Die Wirkung der beschriebenen logischen Operationen kann dem Ergebnisausdruck des Programms 6.13 entnommen werden. In Modifikation der üblichen Wahrheitstafeln wird hier (in Anlehnung an die oben genannten BASIC-Interpreter) der Zahlenwert −1 für die wahre Aussage verwendet; für die falsche steht der Wert 0.
Bei der Auswertung eines komplexen Ausdrucks geht der Interpreter folgendermaßen vor:

- Zuerst werden die Werte der *arithmetischen Ausdrücke* in der behandelten Reihenfolge der Rechenoperationen (Tafel 5.1) ermittelt.
- Danach werden die Werte der *Vergleiche* (Tafel 5.2) berechnet.
- Zuletzt werden die *Verbindungen* der Vergleichsaussagen bearbeitet. Dabei wird die in Tafel 5.3 angegebene Rangfolge beachtet, bei mehreren Operationen gleichen Ranges erfolgt die Auswertung *von links nach rechts*. Bei manchen Interpretern wird allen logischen Operatoren derselbe Rang zugeordnet.

Auf jeden Fall werden die Werte von Klammerausdrücken zuerst berechnet, bei Schachtelungen von innen nach außen.

6. Programmstrukturen in BASIC

Ein Rechenautomat führt die hintereinander im Hauptspeicher abgelegten Anweisungen sequentiell aus. In analoger Weise arbeitet ein BASIC-Interpreter ein Programm in der Reihenfolge der Zeilennummern ab. Für die Implementierung von Algorithmen sind daher noch weitere Strukturelemente erforderlich, um den Ablauf der Bearbeitung in Abhängigkeit von berechenbaren Bedingungen steuern zu können. Darüber wurde bereits ausführlich im Abschnitt 2.2.3 gesprochen. Zur Realisierung der dort diskutierten Steuerstrukturen stellt BASIC allerdings nur relativ maschinennahe Hilfsmittel zur Verfügung. Sie erlauben natürlich ein *strukturiertes Programmieren,* erzwingen es aber genau so wenig, wie es beispielsweise die Assemblersprache tut. Daher ist eine disziplinierte Arbeit des Programmierers erforderlich, wenn übersichtliche, gut strukturierte Programme entstehen sollen.

Das Grundelement von BASIC zum Aufbau von Programmstrukturen ist der auch in der Maschinensprache vorhandene *Sprung,* durch den die sequentielle Abarbeitung verlassen werden kann:

n **GOTO** *zeilennummer*

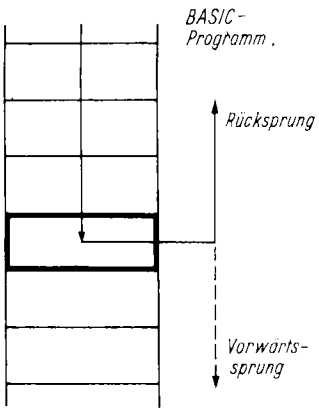


Bild 6.1. Unbedingter Sprung

Diese Sprachanweisung bewirkt, daß die Interpretation bei der angegebenen Zeile fortgesetzt wird **(Bild 6.1).** Sie wurde bereits in den vorstehenden Abschnitten mehrfach benutzt und vom Leser sicher intuitiv erfaßt. Sie wird meist im Zusammenwirken mit – noch zu behandelnden – bedingten Sprüngen eingesetzt.

6.1. Verzweigungen

Beim Aufbau von Programmen wird ein Strukturelement benötigt, durch das der Computer aufgefordert wird, auf der Grundlage von bisher erzielten Teilresultaten eine Entscheidung darüber zu fällen, in welcher von mehreren möglichen Richtungen weiterzuarbeiten ist. Meist handelt es sich dabei um eine Alternative, die auf der Grundlage einer Vergleichsaussage entschieden wird (Bild 2.11). Es gibt aber auch Verzweigungen mit mehr als zwei Ausgängen.

In den bisher dargestellten Programmbeispielen wurden solche Sprachkonstruktionen bereits häufig benutzt und vom Leser sicher verstanden. Im folgenden sollen ihr Aufbau und ihre Anwendung systematisch behandelt werden.

6.1.1. Bedingte Sprünge

In Analogie zur Maschinensprache gibt es auch in BASIC eine Sprunganweisung, die nur dann ausgeführt wird, wenn eine bestimmte Bedingung erfüllt (wahr) ist:

n **IF** *logischer ausdruck* **THEN** *zeilennummer*

Diese Sprachanweisung ermittelt zunächst den Wert des *logischen Ausdrucks* (Abschn. 5.3). Ergibt sich dabei das Resultat *wahr* (Zahlenwert <> 0), so wird die Interpretation bei der *angegebenen Zeile* fortgesetzt. Ist der logische Ausdruck dagegen *falsch* (Zahlenwert = 0), so hat diese Sprachanweisung *keinerlei Wirkung*; die Abarbeitung geht bei der nächsten Zeile weiter

```
720 IF A <= B THEN 740
730 LET Z = A : LET A = B : LET B = Z
```

Vergleicht man **Bild 6.2** mit der Steuerstruktur *Alternative* (Bild 2.11), so erkennt man die Forderung der strukturierten Programmierung, die zwei entstandenen Zweige (bald) wieder zusammenzuführen. Dazu dienen – wie bereits angedeutet – die unbedingten Sprünge.

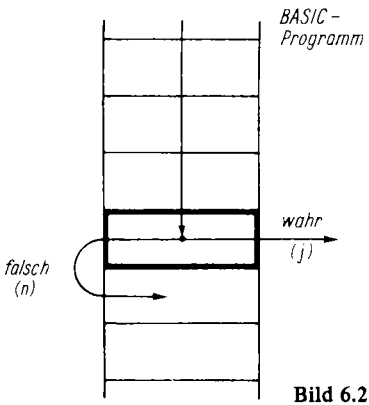


Bild 6.2. Bedingter Sprung

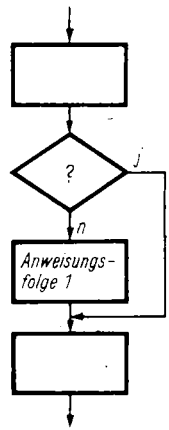


Bild 6.3. Einsatz eines bedingten Sprungs

Die bedingte Sprunganweisung kann man zweckmäßig dazu einsetzen, um eine Anweisungsfolge wahlweise anzuführen oder zu überspringen (**Bild 6.3**). In dieser Form wird sie auch im **Programm 6.1** (Programmablaufplan **Bild 6.4**) eingesetzt. Kern dieses Beispiels ist die Warteschleife in Zeile 180, die bereits im Abschnitt 3.7.1 diskutiert und im Programm 3.3 dargestellt ist; die betreffende Sprachanweisung selbst wird allerdings erst im Abschnitt 6.2.3 näher erläutert. Der obere Grenzwert 467 wurde mit dem Testcomputer experimentell so ermittelt, daß die fortlaufende Ausgabe der Uhrzeit etwa im Sekundenrhythmus erfolgt.
Einige Interpreter sehen für den bedingten Sprung die Schreibweise

n **IF** *logischer ausdruck* **GOTO** *zeilennummer*

vor. Weiterhin gibt es manchmal die Sprachanweisung

n **IF** *logischer ausdruck* **THEN** *zeilennummer1* **ELSE** *zeilennummer2*

Bei dieser Anweisung wertet der Interpreter zuerst den logischen Ausdruck aus. Ist er *wahr*, so wird ein Sprung nach der hinter dem Schlüsselwort THEN stehenden *Zeilennummer 1* ausgeführt. Ergibt sich dagegen das Resultat *falsch*, so erfolgt der Sprung zu der hinter ELSE angegebenen *Zeilennummer 2*. Wie man dem **Bild 6.5** entnehmen kann, erreicht man also mit dieser

Anweisung (ähnlich wie beim unbedingten Sprung GOTO und im Gegensatz zur IF-THEN-Anweisung) die nächste Zeile nicht unmittelbar!

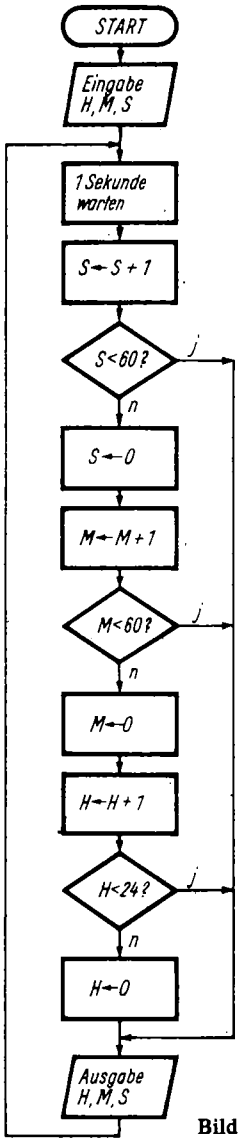


Bild 6.4. Uhrprogramm

Programm 6.1. Uhrprogramm

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                          UHR                   "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Stunde, Minute, Sekunde" ; H,M,S
160 :
170 PRINT
180 LET P = 1
190 FOR I=1 TO 467 : NEXT I
200 LET S = S+1
210 IF S<60 THEN 310
220   LET S = 0
230   LET M = M+1
240   IF M<60 THEN 310
250     LET M = 0
260     LET H = H+1
270     IF H<24 THEN 310
280       LET H = 0
290       PRINT : PRINT "Guten Morgen!"
300       LET P = 1
310 PRINT USING "##:##:##  " ; H , M , S ;
320 LET P = P+1
330 IF P>6 THEN 170
340 GOTO 190
350 :
360 REM ==================================================
```

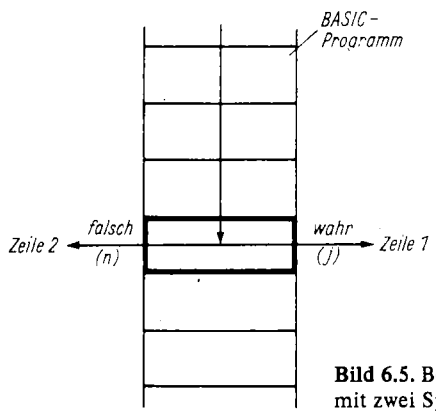


Bild 6.5. Bedingter Sprung mit zwei Sprungzielen

6.1.2. Bedingte Anweisungen

In Erweiterung von BASIC trat im Laufe der Entwicklung zu dem bedingten Sprung die bedingte Anweisung hinzu:

n **IF** *logischer ausdruck* **THEN** *anweisungsfolge*

Bei der Interpretation dieser Sprachanweisung wird zuerst der Wert des logischen Ausdrucks berechnet. Nur falls er *wahr* ist, werden alle nach dem Schlüsselwort THEN (bis zum Zeilenende) stehenden, in bekannter Weise durch Doppelpunkte getrennten *Anweisungen ausgeführt.* Auf je-

den Fall aber geht die Arbeit bei der *nächsten Zeile* weiter! Insofern wird dieselbe Programmstruktur erreicht, wie sie auch mit einem bedingten Sprung entsprechend Bild 6.3 zu erzielen ist. Man wird dann mit der IF-GOTO-Anweisung arbeiten müssen, wenn die bedingt auszuführende Anweisungsfolge so lang ist, daß man sie nicht hinter THEN auf der Zeile unterbringen kann. Dagegen ist die IF-THEN-Anweisung vorteilhaft anzuwenden, wenn nur eine oder zwei Anweisungen vorhanden sind:

```
530 IF INKEY$ <> "" THEN PRINT "ABBRUCH!" : END
```

Übrigens wird in den Programmbeispielen dieses Buches stets die Variante der bedingten Anweisungen benutzt, und zwar auch dann, wenn ein bedingter Sprung ausgedrückt werden soll. Dieses konsequente Vorgehen erfordert allerdings die zusätzliche Angabe des Schlüsselworts GOTO.

Manche Interpreter bieten in Anlehnung an andere höhere Programmiersprachen noch die folgende Sprachanweisung an:

IF *logischer ausdruck* **THEN** *anweisungsfolge1* **ELSE** *anweisungsfolge2*

Ergibt sich beim Auswerten des logischen Ausdrucks das Resultat *wahr,* so wird die *erste* Anweisungsfolge ausgeführt, *anderenfalls* die *zweite.* In beiden Fällen geht der Interpreter anschließend zur *nächsten Zeile* weiter. **Bild 6.6** zeigt die entsprechende Programmstruktur.

```
660 IF X1 = 0 OR X2 = 0 THEN LET Z = 0 ELSE LET Z = 1
```

Ein Anwendungsbeispiel ist im **Programm 6.2** dargestellt, das eine Reihe von bedingten Anweisungen enthält. **Bild 6.7** vermittelt den dazugehörigen Programmablaufplan.

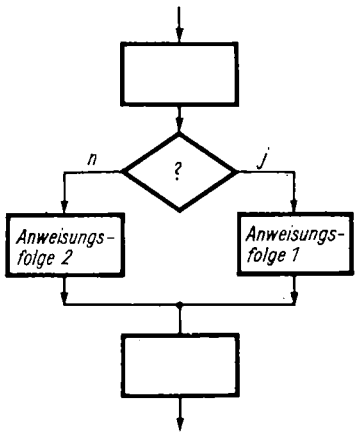


Bild 6.6. IF-THEN-ELSE-Anweisung

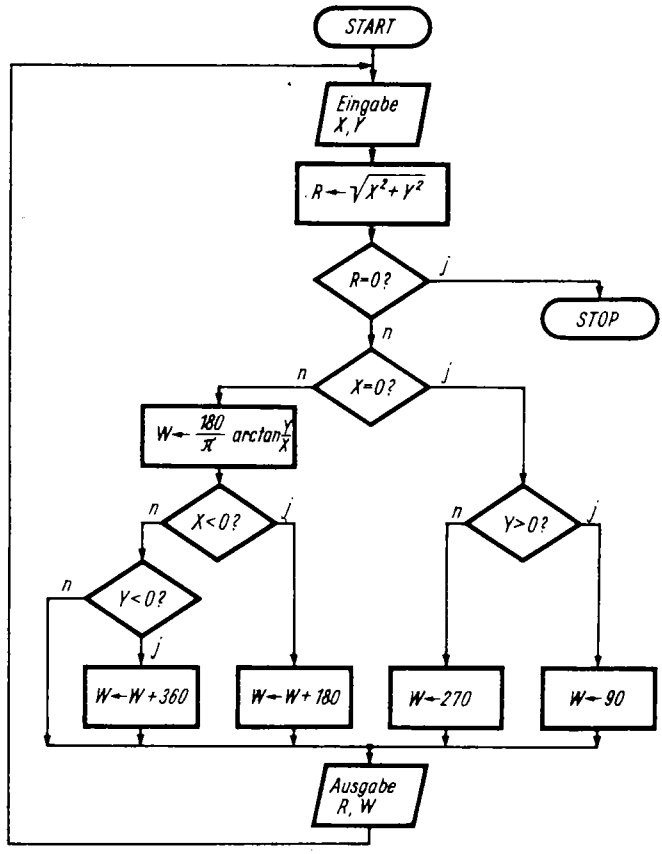


Bild 6.7. Umrechnung von kartesischen Koordinaten in Polarkoordinaten

Programm 6.2. Umrechnung von kartesischen Koordinaten in Polarkoordinaten

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"   UMRECHNUNG KARTESISCHE -> POLARKOORDINATEN  "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT ; "Kartesische Koordinaten: X = " , X
160 INPUT ",  Y = " , Y
170 LET R = SQR(X*X + Y*Y)
180 IF R=0 THEN END
190 :
200 IF X<>0 THEN GOTO 230
210   IF Y>0 THEN LET W = 90 ELSE LET W = 270
220   GOTO 260
230 LET W = 180/3.14159 * ATN(Y/X)
240 IF X<0 THEN LET W = W+180 ELSE IF Y<0 THEN W = W+360
250 :
260 PRINT "Polarkoordinaten:  R =" ; R ;
270 PRINT ", W =" ; W ; "Grad"
280 GOTO 150
290 :
300 REM ================================================
```

Bei der IF-THEN-ELSE-Anweisung macht sich die Beschränkung auf jeweils eine einzige Zeile recht nachteilig bemerkbar; häufig wird man nicht mehr als je eine Anweisung hinter den Schlüsselwörtern THEN und ELSE angeben können. Es gibt hier zwei *Auswege:*

- Die gewünschten beiden Anweisungsfolgen werden als *Unterprogramme* geschrieben. Hinter THEN und ELSE stehen dann nur die Unterprogrammaufrufe GOSUB. Beispiele dafür werden im Abschnitt 6.3.1 und in den Programmen 6.14 bis 6.16 gebracht.
- Verschiedene Interpreter gestatten es in Anlehnung an andere höhere Programmiersprachen, die bedingte Anweisung über *mehrere Zeilen* auszudehnen. Die Anweisungsfolgen hinter THEN und ELSE müssen dann geeignet eingeklammert werden; beispielsweise sind sie jeweils in die beiden Schlüsselwörter DO und DOEND einzuschließen. Solche Erweiterungen sind zwar nützlich, aber der ursprünglichen Konzeption von BASIC fremd.

6.1.3. Berechnete Sprünge

Beim Kodieren von Algorithmen tritt häufig der Fall auf, daß an einer Verzweigungsstelle nicht nur zwei weitere Wege möglich sind. Hier lassen sich beispielsweise *Testketten* verwenden:

```
210 IF X = 3 THEN GOTO 370
220   IF X = 5 THEN GOTO 510
230     IF X = 2 THEN GOTO 150
240       PRINT "FEHLER"
250       GOTO 100
```

Falls man den zur Verzweigung analysierten Ausdruck (Variable) aber in eine solche Form bringen kann, daß er als Resultate aufeinanderfolgende, ganze Zahlen erzeugt, so ist besser eine andere BASIC-Sprachanweisung einzusetzen, die auch als *Sprungverteiler* bezeichnet wird:

n **ON** *arithmetischer ausdruck* **GOTO** *sprungliste*

Diese Anweisung enthält hinter dem Schlüsselwort GOTO mehrere voneinander durch Kommas getrennte Zeilennummern, die als Sprungziele benutzt werden sollen. Ihre Anzahl ist nur durch die Länge der Zeile begrenzt. Erreicht der Interpreter die ON-GOTO-Anweisung, so berechnet er zunächst den Wert des *arithmetischen Ausdrucks* und rundet ihn auf die nächstgelegene ganze Zahl. Je nach dieser Zahl wählt er sich eine *Zeilennummer* aus: bei 1 die erste Nummer, bei 2 die zweite Nummer usw. Zu der so ermittelten Zeile verzweigt der Interpreter dann. Hat sich bei der Berechnung ein Wert kleiner als oder gleich null ergeben oder ist das Resultat größer als die An-

zahl der angegebenen Zeilennummern, so wird *kein* Sprung ausgeführt, sondern die Verarbeitung bei der folgenden Zeile fortgesetzt.

Als Beispiel wurde das einfache **Programm 6.3** gewählt, bei dem der entscheidende Zahlenwert vom Bediener eingegeben wird. Dadurch ist es möglich, Experimente mit negativen, mit zu großen oder gebrochenen Zahlen zu machen.

Programm 6.3. Sprungverteiler

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                    MONATSNAMEN                  "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT ; " Monatsnummer" ; M
160 ON M GOTO 190,200,210,220,230,240,250,260,270,280,290,300
170 PRINT " ist keine Monatsnummer!" : GOTO 150
180 :
190 PRINT " -> Januar" : GOTO 150
200 PRINT " -> Februar" : GOTO 150
210 PRINT " -> Maerz" : GOTO 150
220 PRINT " -> April" : GOTO 150
230 PRINT " -> Mai" : GOTO 150
240 PRINT " -> Juni" : GOTO 150
250 PRINT " -> Juli" : GOTO 150
260 PRINT " -> August" : GOTO 150
270 PRINT " -> September" : GOTO 150
280 PRINT " -> Oktober" : GOTO 150
290 PRINT " -> November" : GOTO 150
300 PRINT " -> Dezember" : GOTO 150
310 :
320 REM ==================================================
```

6.2. Schleifen

Bei der Auswahl von Algorithmen und ihrer Kodierung in Programmen wird stets angestrebt, bestimmte Anweisungsfolgen mehrfach einzusetzen, beispielsweise durch den Aufbau von Schleifen *(Zyklen)*. Eine einfache Variante stellt die *unendliche Schleife* dar, die in einer Reihe von Programmbeispielen bereits benutzt wurde **(Bild 6.8)**. Sie enthält einen Rücksprung mit einer GOTO-Anweisung und kann nur durch einen Bedienereingriff (CTRL-C) unterbrochen werden.

Bei anderen Schleifen erfolgt ein *selbsttätiger Abbruch* der Wiederholung in Abhängigkeit von einer geprüften Bedingung. Diese Schleifen werden nach zwei unterschiedlichen Merkmalen klassifiziert:

- nach der *Position des Abbruchtests* innerhalb der Schleife:
  Test vor dem Eintritt (*abweisende Schleife*, **Bild 6.9**)
  Test am Ende (*nicht abweisende Schleife*, **Bild 6.10**)
  Test *innerhalb* der Schleife (Bild 2.12)

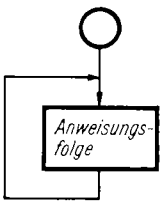


Bild 6.8. Unendliche Schleife

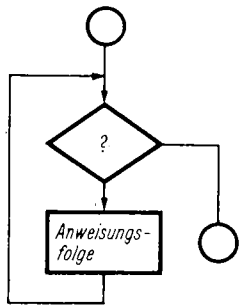


Bild 6.9. Abweisende Schleife

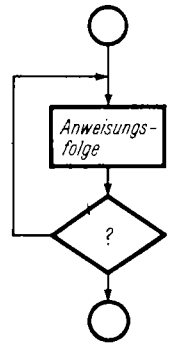


Bild 6.10. Nicht abweisende Schleife

- nach der *Anzahl der Durchläufe:*
  fest vorgegebene Anzahl
  variable, vorher unbekannte Anzahl.

Einige wesentliche Fälle werden im folgenden diskutiert.

6.2.1. Iterative Schleifen

Iterative Schleifen ermitteln ein gewünschtes Resultat durch schrittweise Verbesserung eines Ausgangswertes. Sie werden abgebrochen, wenn eine vorgegebene Genauigkeit erreicht wurde. Hier müssen also zunächst Berechnungen ausgeführt werden; dann erst kann man die Abbruchbedingung prüfen. Daher werden meist keine abweisenden Schleifen eingesetzt.

Die Programmbeispiele 6.4 und 6.5 haben eine Struktur entsprechend Bild 2.12. Im **Programm 6.4** wird nach der bekannten Reihenentwicklung

$$1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

ein Näherungswert für die Basis e der natürlichen Logarithmen berechnet. Das **Programm 6.5** ermittelt die Quadratwurzel aus einer eingegebenen Zahl r nach dem Newtonschen Iterationsverfahren

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{r}{x_n}\right)$$

Die Anzahl der Durchläufe, die in beiden Fällen erforderlich ist, um eine vorgegebene Genauig-

Programm 6.4. Reihenentwicklung der Zahl e

```
100 REM <<<<<<<<<< REIHENENTWICKLUNG DER ZAHL e >>>>>>>>>>
110 :
120 LET S = 1 : REM Anfangswert Summe
130 LET G = 1 : REM erstes Summenglied
140 LET I = 1
150 :
160 LET G = G/I
170 LET E = S+G
180 IF ABS((E-S)/S)<.000001 THEN GOTO 230
190   LET S = E
200   LET I = I+1
210   GOTO 160
220 :
230 LPRINT "Basis der natuerlichen Logarithmen:  e =" ; E
240 :
250 END
260 REM ================================================

Basis der natuerlichen Logarithmen:  e = 2.71828
```

Programm 6.5. Berechnung der Quadratwurzel nach dem Newtonschen Iterationsverfahren

```
100 REM <<<<<<<< NEWTONSCHES ITERATIONSVERFAHREN >>>>>>>>>
110 :
120 INPUT ; "Radikand = " , R
130 IF R<=0 THEN PRINT "Fehler: R<=0" : GOTO 120
140 LET X0 = 1 : REM Anfangswert Wurzel
150 LET X = (X0 + R/X0)/2
160 IF ABS((X0-X)/X)>.000001 THEN LET X0 = X : GOTO 150
170 PRINT "  -> Wurzel =" ; X
180 GOTO 120
190 :
200 REM ================================================
```

keit zu erreichen, ist vorher nicht bekannt. Der Bediener kann sie sich aber am Ende der Berechnung ausdrucken lassen.

6.2.2. Schleifen mit einer vorgegebenen Anzahl von Durchläufen

Bei Schleifen mit einer festen Anzahl von Durchläufen wird eine Variable eingeführt, die den Abbruch steuert. Diese Variable wird meist *Laufvariable* genannt. Sie muß vor dem Eintritt in die Schleife einen *Anfangswert* erhalten.

Innerhalb der Anweisungsfolge der Schleife (dem *Schleifenkörper)* darf die Laufvariable durchaus bei Berechnungen benutzt werden. Dabei muß aber mindestens eine Anweisung vorhanden sein, die den Wert der Laufvariablen *verändert.*

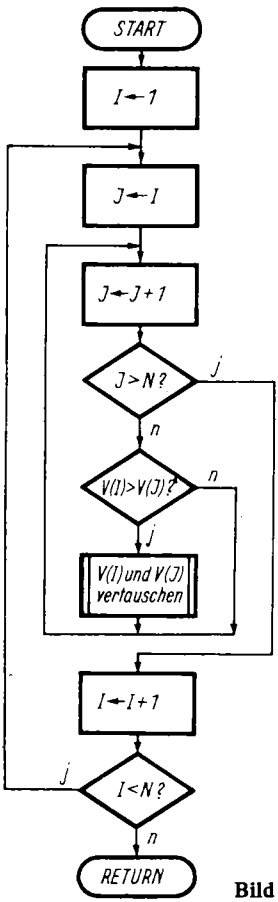


Bild 6.11. Modifiziertes Sortierverfahren

Programm 6.6. Modifiziertes Sortierverfahren

```
100 REM +++++++++++++++++++ SORTIEREN ++++++++++++++++++++
110 LET I = 1
120 LET J = I
130 LET J = J+1
140 IF J>N THEN GOTO 170
150 IF V(I)>V(J) THEN GOSUB 210
160 GOTO 130
170 LET I = I+1
180 IF I<N THEN GOTO 120
190 RETURN
200 REM ------------------ VERTAUSCHEN -------------------
210 LET Z = V(I)
220 LET V(I) = V(J)
230 LET V(J) = Z
240 RETURN
250 REM ==========================================================
```

An einer beliebigen Stelle im Schleifenkörper wird der aktuelle Wert der Zählvariablen mit einem vorgegebenen Endwert für den Laufbereich *verglichen* und in Abhängigkeit vom Resultat die Schleife evtl. *verlassen.*

Solche Schleifen traten bisher bereits in vielen Beispielprogrammen auf, beginnend beim Sortierverfahren im Abschnitt 2. Dabei wurden stets nicht abweisende Schleifen aufgebaut. Dieses Vorgehen ist aber nicht immer zweckmäßig.

Wird das Programm 2.3 z. B. zur Sortierung eines Datenfelds aufgerufen, das nur aus einem

Element besteht (also gar nicht sortiert zu werden brauchte), so bemerkt das Programm diese Tatsache erst dann, wenn es das einzige vorhandene Datenelement bereits mit dem zufälligerweise in der nächsten Speicherzelle stehenden Wert verglichen und möglicherweise sogar vertauscht hat. Daher ist hier (innen) eine abweisende Schleife zweckmäßig: Das **Programm 6.6** arbeitet auch in dem genannten Sonderfall einwandfrei. **Bild 6.11** zeigt den entsprechenden Programmablaufplan.

Abschließend soll noch auf das folgende Problem eingegangen werden: Verschiedentlich möchte man gern eine Laufvariable benutzen, die *nicht ganzzahlig* ist, z. B. beim Druck von Funktionstafeln. In diesen Fällen können beim *Abbruchtest* Schwierigkeiten auftreten, wenn der festgelegte Endwert infolge unvermeidlicher Ungenauigkeiten bei der Zahlendarstellung im Rechenautomaten *nicht exakt* erreicht wird. Es gibt hier zwei Auswege:

- Der Endwert wird etwas hinausgeschoben (z. B. um eine halbe Schrittweite) und der Abbruchtest nicht auf Gleichheit durchgeführt, sondern mit den Vergleichsoperatoren < oder > (unter anderem: Programme 6.12 und 6.19).
- Zum Zählen der Durchläufe wird eine ganzzahlige Laufvariable benutzt. Die außerdem benötigte nicht ganzzahlige Variable wird gesondert erhöht und nicht für den Abbruchtest eingesetzt (Programm 11.3).

6.2.3. Laufanweisungen

In Schleifen mit einer festen Anzahl von Durchläufen wird die Laufvariable meist bei jedem Durchlauf um einen bestimmten Betrag erhöht bzw. erniedrigt. Diese Schrittweite hat meist den Betrag eins. Man spricht auch von *inkrementellen* (bzw. *dekrementellen*) *Schleifen*. Für diesen sehr häufigen Fall bietet BASIC eine recht bequeme Sprachanweisung:

n **FOR** *variablenname* = *anfangswert* **TO** *endwert* [**STEP** *schrittweite*]

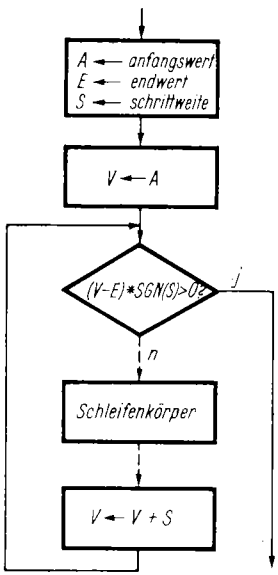


Bild 6.12. Struktur einer Laufanweisung

Diese Anweisung bestimmt den Anfang einer inkrementellen Schleife. Sie veranlaßt den Interpreter zu folgenden Aktionen **(Bild 6.12)**:

- Für Anfangs- und Endwert der Laufvariablen sowie für die Schrittweite sind arithmetische Ausdrücke zulässig. Die entsprechenden Werte werden *einmalig* beim ersten Erreichen der Anweisung berechnet.

- Der Anfangswert wird der Laufvariablen zugewiesen; Endwert und Schrittweite werden bis zum Abbruch der Schleife gespeichert.
- Es wird geprüft, ob

  Endwert ≧ Anfangswert bei Schrittweite > 0 bzw.
  Endwert ≦ Anfangswert bei Schrittweite < 0.

  Ist diese Bedingung nicht erfüllt, wird der Schleifenkörper nicht ausgeführt *(abweisende Schleife)*.

Bei allen weiteren Anweisungen dient die FOR-TO-Anweisung lediglich als Anfangsmarkierung der Schleife. Werden das Schlüsselwort STEP und die Schrittweite weggelassen, so nimmt der BASIC-Interpreter dafür den Wert eins an.

Nach der FOR-TO-Anweisung folgen die innerhalb der Schleife auszuführenden Anweisungen. Sie werden abgeschlossen durch die Sprachanweisung

NEXT [*variablenname*]

Diese Anweisung realisiert folgende Aktionen (Bild 6.12):

- Der Wert der Laufvariablen wird um die oben berechnete Schrittweite erhöht.
- Es erfolgt ein Sprung zur FOR-TO-Anweisung. Dort wird geprüft, ob der jetzt erreichte Wert der Laufvariablen noch im Bereich vom Anfangswert bis einschließlich des Endwertes liegt. Ist diese Voraussetzung erfüllt, so wird der Schleifenkörper erneut abgearbeitet. Anderenfalls wird diejenige Anweisung ausgeführt, die nach der NEXT-Anweisung steht.

Das Anführen der Laufvariablen ist eigentlich überflüssig, weil der Interpreter die NEXT-Anweisung automatisch auf die letzte FOR-TO-Anweisung bezieht. Durch diese Angabe erhöht man aber die Übersichtlichkeit des Programms und ermöglicht es dem Interpreter, Fehler aufzudecken.

Programm 6.7. Laufanweisung

```
100 REM <<<<<<<<<<<<<<<< LAUFANWEISUNG >>>>>>>>>>>>>>>>
110 :
120 LET A = 2
130 LET E = 10
140 LET S = 3
150 :
160 FOR I=A TO E STEP S
170   LPRINT I ;
180 NEXT I
190 LPRINT ">" ; I
200 :
210 END
220 REM ================================================

2  5  8 > 11
```

Ein einfaches Demonstrationsbeispiel zeigt das **Programm 6.7.** Die Anzahl der Durchläufe ist durch die Angaben in der FOR-TO-Anweisung eindeutig bestimmt; sie beträgt

$$N = \mathrm{INT}\left(\frac{\text{endwert} - \text{anfangswert}}{\text{schrittweite}}\right) + 1$$

Die Ausdrücke für Anfangs- und Endwert sowie für die Schrittweite werden – wie gesagt – einmalig vor dem Betreten der Schleife berechnet. Daher hat es keinerlei Einfluß auf die Abarbeitung, wenn innerhalb der Schleife die Werte derjenigen Variablen verändert werden, die für diese Berechnungen erforderlich waren. Das wird beispielsweise am **Programm 6.8** deutlich.

Dagegen ist es von Bedeutung, wenn der Wert der Laufvariablen innerhalb des Schleifenkörpers verändert wird, wie es im **Programm 6.9** gezeigt ist. Solche zusätzlichen Einflüsse machen

Programm 6.8. Änderung von Anfangswert, Schrittweite und Endwert innerhalb der Laufanweisung

```
100 REM <<<<<<<<<<<<<<<< LAUFANWEISUNG >>>>>>>>>>>>>>>>
110 :
120 LET A = 2
130 LET E = 10
140 LET S = 3
150 :
160 FOR I=A TO E STEP S
170   LPRINT I ;
173   LET A = 3
175   LET E = 20
177   LET S = 5
180 NEXT I
190 LPRINT ">" ; I
200 :
210 END
220 REM ================================================

 2  5  8 > 11
```

Programm 6.9. Änderung des Wertes der Laufvariablen innerhalb der Laufanweisung

```
100 REM <<<<<<<<<<<<<<<< LAUFANWEISUNG >>>>>>>>>>>>>>>>
110 :
120 LET A = 2
130 LET E = 10
140 LET S = 3
150 :
160 FOR I=A TO E STEP S
170   LPRINT I ;
180   LET I = I+1
190   LPRINT "-" ; I ;
200 NEXT I
210 LPRINT ">" ; I
220 :
230 END
240 REM ================================================

 2 - 3  6 - 7  10 - 11 > 14
```

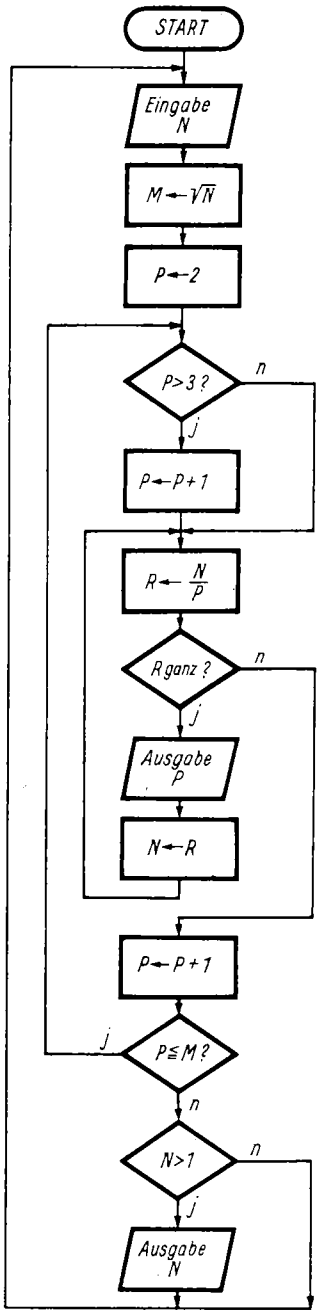
das Programm unübersichtlich und sollten vermieden werden, soweit es möglich ist. Eine Anwendung wird im **Programm 6.10** dargestellt. Hier soll erreicht werden, daß die Schrittweite von P=3 ab auf 2 erhöht wird, um nur die ungeraden Zahlen als mögliche Faktoren zu prüfen. **Bild 6.13** zeigt den entsprechenden Programmablaufplan. Ein anderes einfaches Beispiel vermittelt **Programm 6.11**, das wohl keiner weiteren Erläuterung bedarf.

Die innerhalb einer Schleife benutzte Laufvariable kann auch später weiterverwendet werden. Sie weist dann denjenigen Wert auf, der an der Austrittsstelle gültig war; beim Abbruch über die NEXT-Anweisung also den neuen, außerhalb des definierten Laufbereichs liegenden Wert.

Die Anweisungen innerhalb einer Schleife können nur über die FOR-TO-Anweisung erreicht werden. Es ist verboten, von außen her direkt in eine Schleife zu springen. Dagegen ist es durchaus zulässig, eine Schleife über einen bedingten Sprung vorzeitig zu verlassen.

Laufanweisungen dürfen ineinander *geschachtelt* werden. Dabei ist es allerdings erforderlich, daß die innere Schleife vollständig innerhalb der äußeren liegt (siehe z. B. Bild 3.2); die Laufbereiche dürfen sich also nicht überschneiden. Es ist als Grenzfall zulässig, daß mehrere Schleifen an derselben Stelle enden. Hier darf man für die Begrenzung aller Schleifen eine gemeinsame NEXT-Anweisung mit einer Liste der entsprechenden Laufvariablen schreiben, wobei die Varia-

ble der inneren Schleife vor derjenigen der äußeren anzugeben ist. Ein Demonstrationsbeispiel zeigt das **Programm 6.12.** Hier wurde der Endwert – wie im Abschnitt 6.2.2 empfohlen – um eine halbe Schrittweite erhöht, damit ein sicherer Abbruch gewährleistet ist. Getrennte NEXT-Anweisungen ergeben allerdings eine übersichtlichere Darstellung, wie das **Programm 6.13** zeigt. Dieses Programm druckt eine Wahrheitstabelle für die im Abschnitt 5.3 diskutierten logi-



Programm 6.10. Primfaktorenzerlegung einer Zahl

```
100 REM <<<<<<<<<<<< PRIMFAKTORENZERLEGUNG >>>>>>>>>>>
110 :
120 INPUT ; "natuerliche Zahl N = " , N
130 PRINT "  -> Primfaktoren:" ;
140 FOR P=2 TO SQR(N)
150   IF P>3 THEN LET P = P+1
160   LET R = N/P
170   IF R>INT(R) THEN GOTO 210 : REM NEXT P
180     PRINT P ;
190     LET N = R
200     GOTO 160
210 NEXT P
220 IF N>1 THEN PRINT N ;
230 PRINT
240 GOTO 120
250 :
260 REM ============================================
```

Programm 6.11. Druck einer Spartabelle

```
100 REM <<<<<<<<<<<<<<<<< SPARGUTHABEN >>>>>>>>>>>>>>>>
110 :
120 INPUT "Anfangsguthaben" ; G
130 INPUT "Anzahl der Jahre" ; N
140 INPUT "Zinssatz" ; Z
150 LPRINT "Laufzeit" , "Guthaben"
160 LPRINT
170 :
180 LET F = 1 + Z/100
190 FOR I=0 TO N
200   LPRINT USING "   ##          #####.##" : I , G
210   LET G = G*F
220 NEXT I
230 :
240 END
250 REM ============================================
```

```
Laufzeit       Guthaben

    0           1000.00
    1           1032.50
    2           1066.06
    3           1100.70
    4           1136.48
    5           1173.41
    6           1211.55
    7           1250.92
    8           1291.56
    9           1333.55
   10           1376.89
```

Bild 6.13. Zerlegung in Primfaktoren

schen Operationen aus. Hier wird die Darstellung der logischen Aussage *wahr* durch die Zahl −1 benutzt und die Tabelle mit Hilfe zweier verschachtelter Laufanweisungen mit der Schrittweite −1 realisiert. Der Programmablaufplan ist im **Bild 6.14** enthalten.

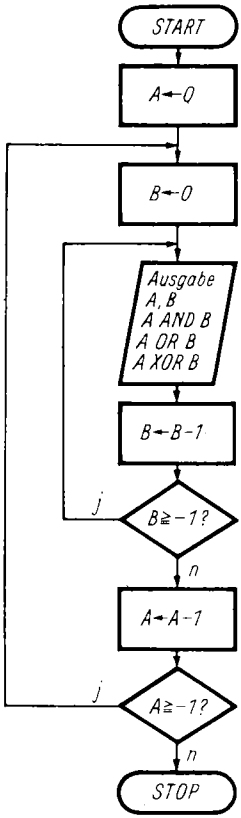


Bild 6.14. Druck einer Wahrheitstafel

Programm 6.12. Verschachtelte Schleifen

```
100 REM <<<<<<<<<<<< GESCHACHTELTE SCHLEIFEN >>>>>>>>>>>>
110 :
120 FOR I=1 TO 5
130   LPRINT
140   LPRINT "I =" ; I ; "  X =" ;
150   FOR X=0 TO .95 STEP .1
160     LPRINT I+X ;
170 NEXT X,I
180 :
190 END
200 REM ================================================
```

```
I = 1   X = 1  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9
I = 2   X = 2  2.1  2.2  2.3  2.4  2.5  2.6  2.7  2.8  2.9
I = 3   X = 3  3.1  3.2  3.3  3.4  3.5  3.6  3.7  3.8  3.9
I = 4   X = 4  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9
I = 5   X = 5  5.1  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9
```

Programm 6.13. Druck einer Wahrheitstafel

```
100 REM <<<<<<<<<<<<<<<< WAHRHEITSTAFEL >>>>>>>>>>>>>>>>
110 :
120 LET M$ = "|   ##    "
130 LPRINT STRING$(51,"-") : REM Strich
140 LPRINT "|    A    |    B    | A AND B | A OR B  |" ;
150 LPRINT " A XOR B |"
160 LPRINT STRING$(51,"-") : REM Strich
170 FOR A=0 TO -1 STEP -1
180   FOR B=0 TO -1 STEP -1
190     LPRINT USING M$ ; A, B, A AND B, A OR B, A XOR B;
200     LPRINT "|"
210   NEXT B
220 NEXT A
230 LPRINT STRING$(51,"-") : REM Strich
240 :
250 END
260 REM ================================================
```

```
-------------------------------------------------
|    A    |    B    | A AND B | A OR B  | A XOR B |
-------------------------------------------------
|    0    |    0    |    0    |    0    |    0    |
|    0    |   -1    |    0    |   -1    |   -1    |
|   -1    |    0    |    0    |   -1    |   -1    |
|   -1    |   -1    |   -1    |   -1    |    0    |
-------------------------------------------------
```

6.3. Funktionsmoduln

Bei den methodischen Überlegungen zum Entwurf von Programmen wurde darauf orientiert, die Struktur eines Problems schrittweise herauszuarbeiten. Als Ergebnis erhält man das Programm als ein System von Funktionsmoduln. BASIC bietet folgende Möglichkeiten zu ihrer Implementierung:

- *Unterprogramme*, die den Unterprogrammen der Assemblersprache vergleichbar sind, und

- *Funktionen*, die in der Anwendung den bisher diskutierten Standardfunktionen entsprechen, aber vom Programmierer selbst definiert werden können.

Auf dieser Basis läßt sich die Übersichtlichkeit der Programmsysteme verbessern. Außerdem kann man abgeschlossene Moduln, die eine bestimmte Aufgabe realisieren, in Bibliotheken für spätere Einsatzfälle aufheben und dann nachnutzen.

6.3.1. Unterprogramme

- Als *Unterprogramm* bezeichnet man eine Folge von Anweisungen, die eine bestimmte Teilaufgabe erfüllt und in einem Programmsystem nur einmal vorhanden ist. Sie ist dadurch charakterisiert, daß
- sie von allen anderen Teilen des Programmsystems durch besondere Unterprogrammsprünge (*Aufrufe*) erreicht werden kann;
- nach ihrer Abarbeitung zu derjenigen Anweisung zurückgesprungen wird (*Rückkehr*), die hinter dem jeweiligen Aufruf steht.

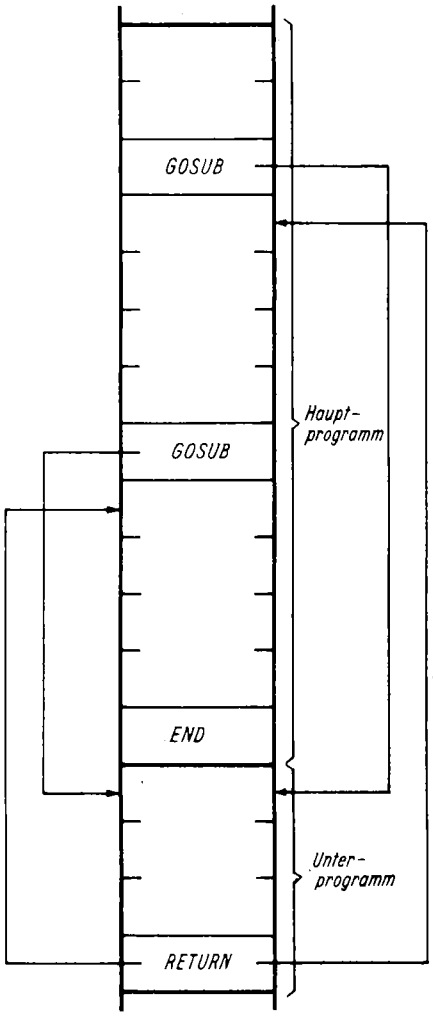


Bild 6.15. Einbinden eines Unterprogramms in den Ablauf des Hauptprogramms

Diese Zusammenhänge sind im **Bild 6.15** grafisch dargestellt. Ziel eines solchen Vorgehens ist es meist, Speicherplatz für das mehrfache Ablegen derselben Anweisungsfolge zu sparen. Bei BASIC bietet sich der Einsatz von Unterprogrammen außerdem noch deshalb an, weil für bedingte Anweisungen jeweils nur eine einzige Zeile zur Verfügung steht. Dieser Platz reicht aber meist nicht dazu aus, alle gewünschten Anweisungen zu kodieren; insbesondere dann nicht, wenn auch das Schlüsselwort ELSE verwendet wird. Das **Programm 6.14** bringt dazu ein Beispiel; der verwendete Algorithmus ist in der einschlägigen Literatur nachzulesen.

Programm 6.14. Berechnung des Ostersonntags

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                    OSTERFEST                    "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 DIM D(7) , M(7) , K(7)
150 FOR I=1 TO 7
160   READ D(I) , M(I) , K(I)
170 NEXT I
180 :
190 INPUT ; "Im Jahre " , J
200 IF J>=1700 THEN GOTO 240
210   PRINT " laesst sich keine Aussage machen. " ;
220.  PRINT "(Julianischer Kalender!)"
230   GOTO 190
240 :
250 LET I = INT(J/100) - 16
260 LET Q = INT(J/4)
270 LET A = J - INT(J/19)*19
280 LET B = M(I) - 11*A
290 LET B = B - INT(B/30)*30
300 IF B=28 OR B=29 THEN LET B = B-K(I)
310 LET C = J+Q+B-D(I)
320 LET C = C - INT(C/7)*7
330 LET T = 28+B-C
340 PRINT " liegt der Ostersonntag am " ;
350 IF T<=31 THEN GOSUB 380 ELSE GOSUB 390
360 END
370 :
380 PRINT USING "##. Maerz." ; T  : RETURN
390 PRINT USING "##. April." ; T-31  : RETURN
400 :
410 DATA 11,203,0
420 DATA 12,203,0
430 DATA 13,204,1
440 DATA 13,204,1
450 DATA 14,204,1
460 DATA 15,205,1
470 DATA 16,206,0
480 :
490 REM ==================================================
```

In BASIC wird (wie in der Assemblersprache, aber im Gegensatz zu anderen höheren Programmiersprachen) der Beginn eines Unterprogramms *nicht* durch eine besondere Anweisung gekennzeichnet. Beim *Aufruf* verwendet man einfach die Nummer der ersten Unterprogrammzeile als Adresse:

n **GOSUB** *zeilennummer*

Diese Anweisung veranlaßt einen *Sprung* zur angegebenen Zeile. Zusätzlich merkt sich der Interpreter aber noch die Adresse derjenigen *Anweisung*, die *nach* der GOSUB-Anweisung im aufrufenden (Haupt-)Programm steht.

Am Ende eines Unterprogramms ist die Sprachanweisung

n **RETURN**

erforderlich. Erreicht der Interpreter eine solche Anweisung, so greift er auf diejenige Adresse

zurück, die er sich bei der *letzten* GOSUB-Anweisung gemerkt hat, und er realisiert einen *Rücksprung* in das Hauptprogramm. Auf diese Weise wird mit Hilfe der Anweisungen GOSUB und RETURN die gesamte Anweisungsfolge des Unterprogramms an der gewählten Stelle in den Ablauf des Hauptprogramms eingeschoben.

Programm 6.15. Berechnung und Druck der Sinusfunktion

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                KURVENDARSTELLUNG                "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 LET F = 3.14159/180
150 LPRINT " G      SIN(G)"
160 LPRINT
170 FOR G=0 TO 360 STEP 15
180   LET S = SIN(F*G)
190   LPRINT USING "###    ##.####" ; G , S ;
200   IF G<>0 THEN GOTO 240
210     LPRINT TAB(15) ; STRING¤(20,"-") ; "*" ;
220     LPRINT STRING¤(20,"-") ; ">"
230     GOTO 260
240   IF G/180=INT(G/180) THEN LPRINT TAB(35);"*":GOTO 260
250     IF S>0 THEN GOSUB 330 ELSE GOSUB 340
260 NEXT G
270 LPRINT TAB(35) ; "|"
280 LPRINT TAB(35) ; "V"
290 PRINT "Funktion sin(x) ausgelistet " ;
300 PRINT "und grafisch dargestellt."
310 END
320 :
330 GOSUB 350 : GOSUB 360 : LPRINT : RETURN
340 GOSUB 360 : GOSUB 350 : LPRINT : RETURN
350 LPRINT TAB(35) ; "|" ;  : RETURN
360 LPRINT TAB(20*S +35) ; "*" ;  : RETURN
370 :
380 REM ==================================================
```

```
 G      SIN(G)

  0     0.0000 ----------------------*----------------------->
 15     0.2588                       |    *
 30     0.5000                       |         *
 45     0.7071                       |              *
 60     0.8660                       |                   *
 75     0.9659                       |                      *
 90     1.0000                       |                        *
105     0.9659                       |                      *
120     0.8660                       |                   *
135     0.7071                       |              *
150     0.5000                       |         *
165     0.2588                       |    *
180    -0.0000                       *
195    -0.2588                  *    |
210    -0.5000             *         |
225    -0.7071         *             |
240    -0.8660     *                 |
255    -0.9659   *                   |
270    -1.0000 *                     |
285    -0.9659   *                   |
300    -0.8660     *                 |
315    -0.7071         *             |
330    -0.5000             *         |
345    -0.2588                  *    |
360     0.0000                       *
                                     |
                                     V
```

Man muß sich nun genau überlegen, an welcher Stelle im BASIC-Programmsystem, d. h. bei welchen Zeilennummern, man die Unterprogramme unterbringt. Es ist zu berücksichtigen, daß ein BASIC-Programm entsprechend den Zeilennummern sequentiell abgearbeitet wird, aber dabei nicht ungewollt Unterprogramme erreicht werden dürfen! Im Bild 6.11 und im Programm 6.14 wird eine günstige Variante gezeigt: Hier stehen die Unterprogramme hinter der END-Anweisung, bei der der Interpreter bekanntlich die Abarbeitung des Programms beendet.

Bisher wurden strukturelle Fragen diskutiert. Ein weiteres wichtiges Problem ist die Übergabe von Daten (*Parametern*) vom Haupt- zum Unterprogramm und umgekehrt. Ein Unterprogramm liefert ja eine Zuarbeit, leistet Dienste. Es benötigt dazu individuelle Anfangswerte, die *Eingangsparameter*. Weiterhin erzeugt es gewünschte Resultate, die *Ausgangsparameter*. Im Gegensatz zu anderen höheren Programmiersprachen realisiert BASIC aber *keine* automatische Parametervermittlung. Außerdem arbeiten BASIC-Unterprogramme *nicht* mit eigenen, lokalen Varia-

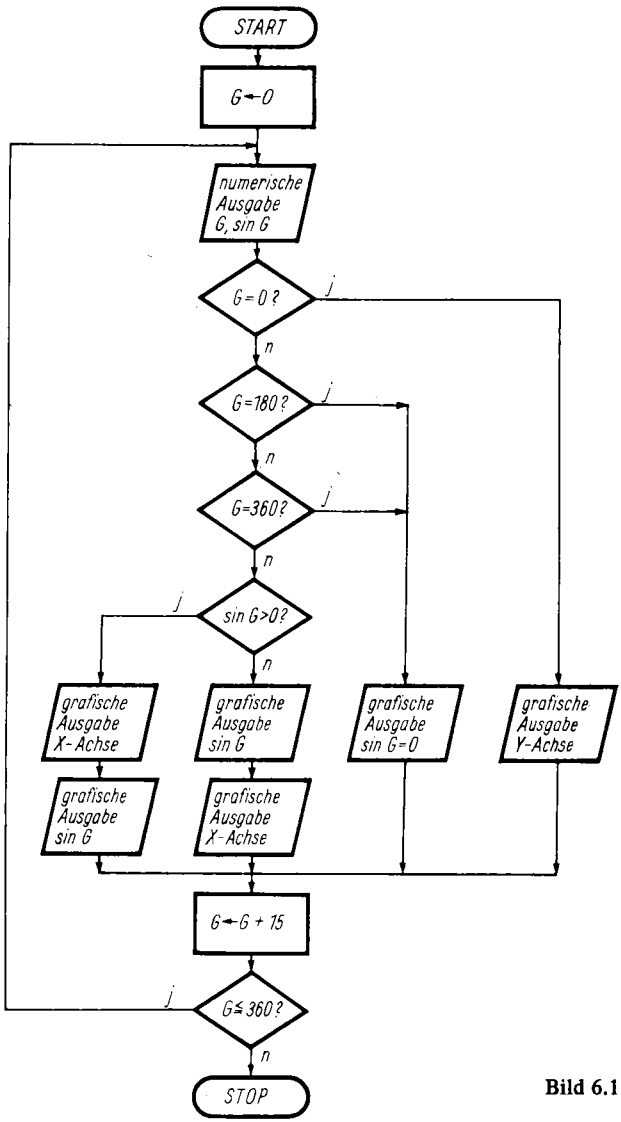


Bild 6.16. Grafische Darstellung der Funktion sin x

blen. Alle im Hauptprogramm benutzten Variablen sind auch im Unterprogramm erreichbar und umgekehrt.

Die *Parameterübergabe* erfolgt in BASIC über solche gemeinsamen (*globalen*) Variablen. Das bringt folgende Probleme mit sich:

- Der Programmierer muß bei jedem Aufruf selbst für die entsprechende Bereitstellung bzw. Auswertung der Parameter sorgen.
- Der Vorrat an möglichen Variablen wird eingeengt: Es wächst die Gefahr, aus Versehen Variablen doppelt zu benutzen. Besondere Schwierigkeiten entstehen dann, wenn der Interpreter nur Namen aus maximal zwei Zeichen zuläßt.
- Es entstehen Komplikationen durch Namensüberschneidungen, wenn man auf vorhandene Bibliotheken von Unterprogrammen zurückgreifen will.

Ein Beispiel zur Parameterübergabe enthält das Programm 6.14. Hier wird der Wert der Variablen T im Hauptprogramm berechnet und durch ein Unterprogramm ausgedruckt.

Es sind auch geschachtelte Unterprogrammaufrufe möglich, das heißt, aus einem Unterprogramm heraus kann in ein weiteres verzweigt werden usf. Ein Beispiel dafür vermittelt das **Programm 6.15**, das eine Tabelle und eine grafische Darstellung der Sinusfunktion ausdruckt. **Bild 6.16** zeigt den Programmablaufplan. Hier wird bei positiven Sinuswerten das Unterprogramm 330 aufgerufen (Zeile 250); von dort aus nacheinander die Unterprogramme 350 (senkrechter Strich) und 360 (Stern). Im **Bild 6.17** sind diese Aufrufe grafisch dargestellt.

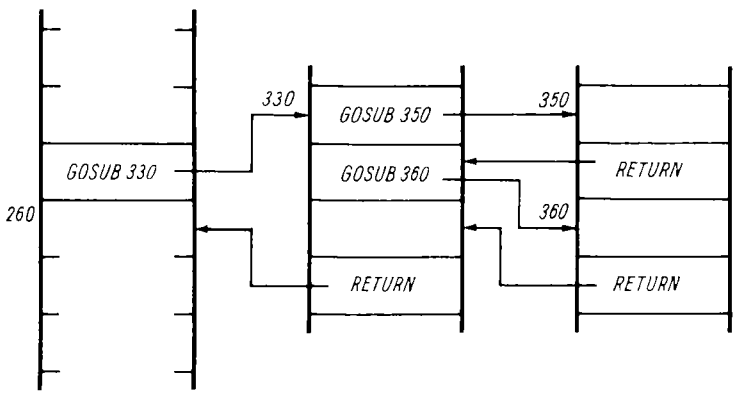


Bild 6.17. Schachtelung von Unterprogrammen

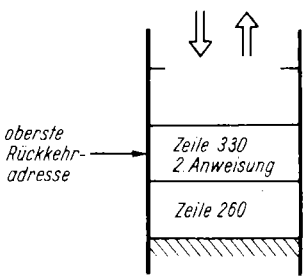


Bild 6.18. Kellerspeicher für Rückkehradressen

Der Interpreter merkt sich die Rückkehradressen in einem sog. *Keller* oder *Stapel*. Das ist ein Zwischenspeicher, bei dem man immer nur obenauf ablegen bzw. von oben her herausnehmen kann **(Bild 6.18)**. Die Tiefe eines solchen Kellers, also die maximale Anzahl der ineinanderzuschachtelnden Unterprogramme, hängt vom jeweiligen Interpreter ab.

Rekursive Aufrufe von Unterprogrammen sind in BASIC verboten, ein Unterprogramm darf sich deshalb selbst nie aufrufen, und zwar weder unmittelbar noch indirekt.

Es gibt – in Analogie zur ON-GOTO-Anweisung (Abschn. 6.1.3) – auch einen *berechneten Unterprogrammaufruf*:

n **ON** *arithmetischer ausdruck* **GOSUB** *sprungliste*

Bei dieser Anweisung berechnet der Interpreter zunächst den Wert des arithmetischen Ausdrucks und rundet ihn auf die nächstgelegene ganze Zahl. Diese Zahl verwendet er dann als Index bei der Auswahl einer Sprungadresse aus der angegebenen Liste. Die Rückkehr aus dem aufgerufenen Unterprogramm erfolgt stets zu der *nächsten Zeile* hinter der ON-GOSUB-Anweisung. Ein einfaches Beispiel zeigt das **Programm 6.16**. Ergibt sich als Wert des Ausdrucks aber null, eine negative oder eine zu große Zahl, so wird die Abarbeitung unmittelbar bei der nächsten Zeile fortgesetzt.

Programm 6.16. Unterprogrammrufverteiler

```
100 REM <<<<<<<<<<<<<<<< VORZEICHENTEST >>>>>>>>>>>>>>>>
110 :
120 INPUT ; "Gib eine Zahl ein!  " , Z
130 ON SGN(Z)+2 GOSUB 160,170,180
140 GOTO 120
150 :
160 PRINT " ist negativ." : RETURN
170 PRINT " ist null." : RETURN
180 PRINT " ist positiv." : RETURN
190 :
200 REM ================================================
```

Es ist zu beachten, daß beim berechneten Unterprogrammaufruf nicht – wie bei GOSUB – zur nächsten Anweisung, sondern zur nächsten *Zeile* weitergegangen wird. Hinter der Liste von Unterprogrammzeilennummern kann man also keine weiteren Anweisungen angeben!

6.3.2. Funktionen

Häufig kommen Unterprogramme vor, die (höchstens) einen Eingangsparameter, das *Argument*, und genau einen Ausgangsparameter, den *Resultatwert*, aufweisen. Es handelt sich dabei also um eine Analogie zu einer mathematischen Funktion mit einer Veränderlichen, die jedem zulässigen Wert des Arguments eindeutig einen Funktionswert zuordnet. Entsprechende Funktionen, im BASIC-Interpreter bereits integrierte arithmetische Standardfunktionen, wurden im Abschnitt 5.2 behandelt. Textfunktionen folgen im Abschnitt 8. Darüber hinaus bietet BASIC dem Programmierer aber noch die Möglichkeit, sich *eigene Funktionen* zu definieren. Die meisten BASIC-Systeme beschränken sich dabei darauf, für einen häufig benötigten arithmetischen Ausdruck oder Textausdruck eine Abkürzung in Form einer Funktion einzuführen (**Programm 6.17**):

n **DEF FN***name*(*argument*) = *ausdruck*

Der *Funktionsname* beginnt mit den Buchstaben FN. Der folgende, vom Programmierer frei wählbare Teil des Namens darf bei vielen Interpretern nur einen einzigen Buchstaben umfassen.

Programm 6.17. Definition von Nutzerfunktionen

```
100 REM <<<<<<<<<<<<<< FUNKTIONSDEFINITION >>>>>>>>>>>>>>>
110 :
120 DEF FNY(X) = X*X + X+X + 1
130 :
140 INPUT ; "Argument:  X = " , A
150 PRINT "  -> Funktion:  Y(X) =" ; FNY(A)
160 GOTO 140
170 :
180 REM ================================================
```

Programm 6.18. Verschachtelter Funktionsaufruf

```
100 REM <<<<<<<<<<<<<<<< VIERTE WURZEL >>>>>>>>>>>>>>>>>
110 :
120 DEF FNW(X) = SQR(SQR(ABS(X)))
130 :
140 INPUT "Zahl" ; Z
150 PRINT "Vierte Wurzel aus dem Betrag von " ; Z ; " = ";
160 PRINT FNW(Z)
170 GOTO 140
180 :
190 REM ================================================
```

Programm 6.19. Druck einer Tafel der Hyperbelfunktionen

```
100 REM <<<<<<<<<<<<<< HYPERBELFUNKTIONEN >>>>>>>>>>>>>>
110 :
120 DEF FNS(Z) = (EXP(Z)-EXP(-Z))/2
130 DEF FNC(Z) = (EXP(Z)+EXP(-Z))/2
140 DEF FNT(Z) = FNS(Z)/FNC(Z)
150 LET M¤ = "##.####^^^^    "
160 :
170 LPRINT TAB(3) ; "TANH(X)" ; TAB(21) ; "X" ; TAB(33) ;
180 LPRINT "SINH(X)" ; TAB(48) ; "EXP(X)-1"
190 LPRINT
200 FOR X=0 TO 1.05 STEP .1
210   LPRINT USING M¤ ; FNT(X) , X , FNS(X) , EXP(X)-1
220 NEXT X
230 :
240 END
250 REM ================================================
```

```
 TANH(X)          X          SINH(X)         EXP(X)-1

0.0000E+00    0.0000E+00    0.0000E+00    0.0000E+00
9.9668E-02    1.0000E-01    1.0017E-01    1.0517E-01
1.9738E-01    2.0000E-01    2.0134E-01    2.2140E-01
2.9131E-01    3.0000E-01    3.0452E-01    3.4986E-01
3.7995E-01    4.0000E-01    4.1075E-01    4.9182E-01
4.6212E-01    5.0000E-01    5.2110E-01    6.4872E-01
5.3705E-01    6.0000E-01    6.3665E-01    8.2212E-01
6.0437E-01    7.0000E-01    7.5858E-01    1.0138E+00
6.6404E-01    8.0000E-01    8.8811E-01    1.2255E+00
7.1630E-01    9.0000E-01    1.0265E+00    1.4596E+00
7.6159E-01    1.0000E+00    1.1752E+00    1.7183E+00
```

Andere erlauben zwei Zeichen, weitere noch mehr. Bei Textfunktionen ist der Name mit dem Währungssymbol abzuschließen.

Die in der Funktionsdefinition als *Argument* angegebene Variable hat den Charakter eines *formalen Parameters*. Das bedeutet, daß diese Variablenbezeichnung nur *innerhalb* der Definitionszeile von Bedeutung ist und nichts mit einer evtl. im Programm vorhandenen, gleichbenannten Variablen zu tun hat.

Der angegebene Ausdruck enthält die Berechnungsvorschrift für den Funktionswert. In ihm kann das Argument benutzt werden. Darüber hinaus ist es auch zulässig, *globale Variablen* zu verwenden. Das bietet sich häufig an, weil vielfach Funktionen von zwei und mehr Veränderlichen zu berechnen sind. Trotzdem ist von diesem Vorgehen abzuraten, weil man bei einer solchen Mischung von formalen und globalen Parametern in einer Funktion schnell den Überblick verlieren kann.

Die *Berechnung* des Funktionswerts wird dann angestoßen, wenn der Interpreter bei der Auswertung eines Ausdrucks auf die entsprechende Funktionsbezeichnung

FN*name*(*ausdruck*)

trifft (z. B. in der Ausgabeliste in Zeile 150 des Programms 6.17):

- Zunächst wird der Wert des Ausdrucks ermittelt, der als Argument (als sog. *aktueller Parameter*) angegeben ist (Variable A im Programm 6.17).
- Diesen Wert des Arguments setzt der Interpreter anstelle des formalen Parameters in denjenigen Ausdruck ein, der in der entsprechenden Funktionsdefinition angegeben ist. Damit wird dann der Wert dieses Ausdrucks berechnet und als Funktionswert benutzt (Zeile 120 im Programm 6.17).
- Dieser Funktionswert wird nunmehr in den Ausdruck übernommen, in dem die Funktionsbezeichnung auftrat, und dort weiterverarbeitet (Ausgabeliste in Zeile 150 des Programms 6.17).

Es ist übrigens erforderlich, daß der Interpreter bereits die Funktionsdefinition abgearbeitet hat, *bevor* er einen Aufruf dieser Funktion erreicht. Daher ist es zweckmäßig, alle Definitionen am Anfang des Programms zusammenzustellen.

Ein weiteres kleines Beispiel ist im **Programm 6.18** dargestellt. Hier wird eine Schachtelung von Funktionen demonstriert. Im **Programm 6.19** werden Funktionsdefinitionen benutzt, um das Programm übersichtlicher zu gestalten.

Manche Interpreter gestatten es, Funktionen mit mehreren Eingabeparametern zu definieren. Damit erweitern sich die Einsatzmöglichkeiten für solche Sprachmittel beträchtlich. Verschiedentlich ist es auch erlaubt, Blindargumente wegzulassen.

Weil die Funktionsdefinitionen in BASIC nur einen einzigen Ausdruck enthalten und nicht länger als eine Zeile sein dürfen, ist ihr Einsatz natürlich etwas eingeschränkt. Daher ist auch eine kompakte Darstellung der Berechnungsvorschrift sehr gefragt. In einigen BASIC-Systemen hingegen dürfen Funktionsdefinitionen – wie bei anderen höheren Programmiersprachen – über mehrere Zeilen hinweggehen. Der Abschluß wird dann durch ein Schlüsselwort (FNEND) markiert. Der interessierte Leser muß auch hier auf die Sprachbeschreibung seines speziellen Interpreters verwiesen werden.

7. Datenstrukturen in BASIC

Ein Computer verarbeitet Daten, rechnerintern verschlüsselte Informationen, die in Speicherzellen abgelegt sind. In den bisherigen Abschnitten wurden nur elementare Komponenten von Datenstrukturen erwähnt, nämlich einfache Zahlenvariablen und Textvariablen (Abschn. 3.3.5). Aus solchen Elementen lassen sich in höheren Programmiersprachen zusammengesetzte Datenstrukturen aufbauen. BASIC gestattet dabei nur die Definition von Datenfeldern.

7.1. Datenfelder

- Ein *Datenfeld (array)* ist eine Aneinanderreihung einfacher Variablen desselben Typs, die sequentiell im Speicher abgelegt sind. Sie haben einen gemeinsamen Namen, den *Feldnamen.*

Eine derartige Zusammenfassung ist dann zweckmäßig, wenn logisch zusammengehörige Variablen gemeinsam mit demselben Algorithmus bearbeitet werden sollen. Ein Beispiel ist das im Abschnitt 2 eingeführte Sortierverfahren.

- Unter einer *indizierten Variablen* versteht man das einzelne Element eines Datenfelds. Es wird durch eine oder mehrere Nummern (*Indizes*) gekennzeichnet.

In der Mathematik sind solche Datenfelder unter verschiedenen Bezeichnungen bekannt. *Vektoren* sind lineare, eindimensionale Reihungen. Hier genügt zur Adressierung einer Vektorkomponente ein einziger Index. *Matrizen* hingegen sind zweidimensional, das einzelne Element ist durch zwei Indizes zu kennzeichnen.

Tafel 7.1. Matrizenanweisungen

a, b und c stehen für beliebige Namen von Matrizen, u und v für Vektoren sowie x für skalare Größen

Anweisung	Bedeutung
n MAT *a* = ZER	Nullmatrix
n MAT *a* = IDN	Einheitsmatrix
n MAT *b* = *a*	Wertzuweisung
n MAT *b* = TRN(*a*)	Transponieren
n MAT *b* = INV(*a*)	Invertieren
n MAT *c* = *a* + *b*	Addieren
n MAT *c* = *a* − *b*	Subtrahieren
n MAT *c* = (*x*) * *a*	Multiplizieren mit skalarer Größe
n MAT *c* = *a* * *b*	Multiplizieren von Matrizen
n LET *x* = DET(*a*)	Determinante einer Matrix
n LET *x* = DOT(*u*,*v*)	Skalarprodukt zweier Vektoren

Es gibt übrigens BASIC-Interpreter, die die Matrizenrechnung durch eine Reihe spezieller Sprachanweisungen unterstützen, die jeweils mit ganzen Matrizen (und nicht nur mit deren Elementen) arbeiten. Eine Zusammenstellung ist in der **Tafel 7.1** enthalten. Im vorliegenden Buch wird allerdings nicht näher darauf eingegangen.

7.1.1. Dimensionierung von Feldern

Bei der erstmaligen Verwendung einer einfachen Variablen reserviert der BASIC-Interpreter bekanntlich automatisch den erforderlichen Speicherplatz. Beim Auftreten einer indizierten Variablen hingegen weiß er nicht, wieviel zusammenhängender Speicherplatz für das betreffende Feld reserviert werden soll. Daher müssen die *Dimensionen* eines Felds deklariert werden, bevor man eine indizierte Variable daraus benutzt:

n **DIM** *feldname(liste von feldgrenzen)*

Der *Feldname* ist nach denselben Regeln zu bilden wie die Namen einfacher Variablen. Derselbe Name darf nicht gleichzeitig für ein Feld und für eine Variable verwendet werden.

In der Liste sind Ausdrücke für die *Maximalwerte der Indizes* für die verschiedenen Dimensionen, durch Kommas getrennt, anzugeben. Gebrochene Werte werden automatisch auf die nächstgelegene ganze Zahl gerundet. Die maximal mögliche Anzahl von Dimensionen ist in BASIC häufig nur durch die Länge der Eingabezeile begrenzt. Es gibt aber auch Interpreter, die höchstens zweidimensionale Zahlenfelder und sogar nur eindimensionale Textfelder zulassen. Manche gestatten als Feldgrenzen nur Zahlen, keine Ausdrücke.

Der *kleinstmögliche Index* ist meist null. Manche BASIC-Systeme verwenden statt dessen den Anfangswert eins. Andere wiederum gestatten es, als untere Grenze wahlweise null oder eins zu verwenden. Alle deklarierten Elemente werden mit 0 beziehungsweise " " initialisiert.

Vektoren:

```
100 DIM A(N)
180 DIM K(9)
270 DIM C□(K+L)
```

Matrizen:

```
140 DIM L(H,B)
200 DIM W(1,20)
210 DIM A(M,N)
```

Es ist auch zulässig, in einer DIM-Anweisung mehrere Felder zu deklarieren:

```
110 DIM B(S) , H(Z)
100 DIM X(N) , Y(N)
```

Es wurde bereits gesagt, daß die DIM-Anweisung *rechtzeitig* im Programm angegeben sein muß. Andererseits ist bei der Verwendung von Variablen in den Ausdrücken für die oberen Indexgrenzen darauf zu achten, daß diesen Variablen bereits Werte zugewiesen wurden.

Wird die DIM-Anweisung *weggelassen*, so reserviert der Interpreter beim ersten Auftreten einer indizierten Variablen des betreffenden Felds den Speicherplatz unter der Annahme, daß der Index in jeder Dimension höchstens den Wert 10 erreicht.

Da Felder viel Speicherplatz belegen, ist es manchmal zweckmäßig, ihn noch während des Programmlaufs wieder *freizugeben*, wenn ein Feld nicht mehr benötigt wird. Verschiedene BASIC-Systeme stellen dafür eine Sprachanweisung zur Verfügung:

n **ERASE** *liste von feldnamen*

Arbeitet der BASIC-Interpreter diese Anweisung ab, so streicht er die angegebenen Felder wieder und disponiert über den frei gewordenen Speicherplatz neu:

```
720 ERASE A , B
```

7.1.2. Arbeit mit indizierten Variablen

Indizierte Variablen lassen sich in völlig gleicher Weise wie einfache Variablen verwenden; lediglich als Laufvariablen in der FOR-TO-Anweisung sind sie unzulässig. Bei ihrer Kodierung entsteht die Schwierigkeit, daß Indizes nicht tiefgestellt werden können. BASIC schreibt daher vor, Indizes in runden Klammern anzugeben. Bei mehrdimensionalen Datenfeldern sind die

einzelnen Indizes durch Kommas zu trennen:

feldname(liste von indizes)

Die Liste enthält Ausdrücke für die Indizes des gewünschten Feldelements, wobei gebrochene Werte gerundet werden.

Vektorkomponenten: $v_k \rightarrow$ V(K)
$x_7 \rightarrow$ X(7)

Matrixkomponenten: $a_{ij} \rightarrow$ A(I,J)
$b_{23} \rightarrow$ B(2,3)

Beim Auftreten einer indizierten Variablen berechnet der Interpreter zunächst die Indexausdrücke und vergleicht die erhaltenen Resultate mit den Maximalwerten, die in der DIM-Anweisung angegeben wurden. Liegen die aktuellen Indizes innerhalb des deklarierten Bereichs, so kann mit der betreffenden Variablen gearbeitet werden; anderenfalls erfolgt eine Fehlermeldung.

Es wurden bereits zweimal Felder eingesetzt, nämlich in den Programmen 2.3, 2.4 und 6.6 (Sortierverfahren) sowie in den Programmen 3.5 und 3.6 (Sieb des *Eratosthenes*). Nachfolgend wird eine Reihe weiterer Beispiele gebracht. Ihnen läßt sich entnehmen, daß die Laufanweisung für die Verarbeitung von Datenfeldern eine große Bedeutung hat.

Beim **Programm 7.1** kann man einen Vektor mit einer wählbaren Anzahl von Komponenten eingeben. Der Betrag dieses Vektors wird nach der Formel

$$\sqrt{\sum_{i=1}^{1} a_i^2}$$

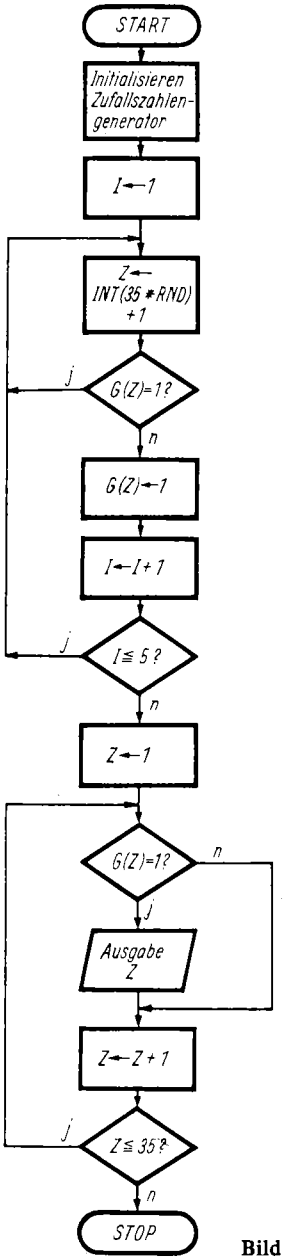
berechnet und ausgedruckt.

Programm 7.1. Berechnung des Betrags eines Vektors

```
100 REM <<<<<<<<<<<<< BETRAG EINES VEKTORS >>>>>>>>>>>>>
110 :
120 INPUT "Anzahl der Elemente" ; L
130 DIM A(L)
140 PRINT "Elemente eingeben:"
150 FOR I=1 TO L
160   INPUT ;"  "; A(I)
170 NEXT I
180 :
190 LET Q = 0 : REM Anfangswert Quadratsumme
200 FOR I=1 TO L
210   LET Q = Q + A(I)*A(I)
220 NEXT I
230 PRINT
240 PRINT " -> Betrag =" ; SQR(Q)
250 :
260 END
270 REM ================================================
```

Im **Programm 7.2** erfolgt zunächst eine (zufällige) Initialisierung des Zufallszahlengenerators (vgl. Programm 5.15). Dann werden insgesamt fünf unterschiedliche Pseudozufallszahlen im Bereich 1 . . . 35 ermittelt und in dem Zahlenfeld G („gezogen") markiert. Am Ende werden diese fünf Zahlen in geordneter Reihenfolge ausgedruckt. **Bild 7.1** zeigt den Programmablaufplan.

Das **Programm 7.3** beschäftigt sich mit einem Textfeld, das die Namen der zwölf Monate enthält. Diese Namen sind in DATA-Anweisungen angegeben und werden am Anfang des Programms in das Textfeld eingelesen. Aus diesem Feld wird dann diejenige Komponente ausgewählt und gedruckt, deren Index (Monatsnummer) der Bediener eingibt. Läßt man die Kontrolle des Eingabewertes in den Zeilen 200 bis 220 weg, so kann man Fehlermeldungen des Interpreters provozieren.



Programm 7.2. Ermittlung von LOTTO-Zahlen

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                  LOTTO  5 AUS 35                 "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM ----- ZUFALLSZAHLENGENERATOR INITIALISIEREN ------
150 :
160 PRINT
170 PRINT "Taste druecken!" ;
180 LET X = -32767
190 IF X=32767 THEN GOTO 180
200 LET X = X+1
210 IF INKEY$="" THEN GOTO 190
220 RANDOMIZE(X)
230 :
240 REM --------------- LOTTOZAHLEN ZIEHEN ---------------
250 :
260 DIM G(35)
270 FOR I=1 TO 5
280   LET Z = INT(35*RND) + 1
290   IF G(Z)=1 THEN GOTO 280
300   LET G(Z) = 1
310 NEXT I
320 :
330 REM --------------- RESULTATE AUSGEBEN ---------------
340 :
350 PRINT "-> LOTTO-Zahlen: " ;
360 FOR Z=1 TO 35
370   IF G(Z)=1 THEN PRINT Z ;
380 NEXT Z
390 :
400 END
410 REM ==================================================
```

Programm 7.3. Indizierte Textvariablen

```
100 REM <<<<<<<<<<<<<<<<<<<< TEXTFELD >>>>>>>>>>>>>>>>>>>>
110 :
120 DIM M$(12)
130 FOR I=1 TO 12
140   READ M$(I)
150 NEXT I
160 :
170 INPUT ; "Monatsnummer" ; N
180 IF N>=1 AND N<=12 THEN GOTO 210
190   PRINT " ist keine moegliche Monatsnummer!"
200   GOTO 170
210 PRINT " -> " ; M$(N)
220 GOTO 170
230 :
240 DATA Januar , Februar , Maerz , April
250 DATA Mai , Juni , Juli , August , September
260 DATA Oktober , November , Dezember
270 :
280 REM ==================================================
```

Bild 7.1. Ermittlung von LOTTO-Zahlen

Dieselbe Vorgehensweise wurde im **Programm 7.4** angewandt, um Dezimalzahlen in römische Zahlen umzurechnen. Es ist ein Textfeld vorhanden, das die Elemente der römischen Zahlen enthält. Um den Algorithmus einfach zu gestalten, werden die Sonderfälle CM, CD, XL, XC, IX und IV nicht berechnet, sondern bereits in dieser Form gespeichert (**Tafel 7.2**). In der Schleife 240 bis 270 wird jeweils geprüft, wie oft eine römische Zahleneinheit in der gegebenen Dezimalzahl enthalten ist. Wenn sie für den Rest der Zahl zu groß ist, so wird zur nächstkleineren Zahleneinheit übergegangen, usf. **Bild 7.2** zeigt den Programmablaufplan.

Programm 7.4. Berechnung römischer Zahlen

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"   UMRECHNUNG: DEZIMALZAHL -> ROEMISCHE ZAHL   "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 DIM Z¤(13) : REM Elemente roemischer Zahlen
150 DIM Z(13)  : REM dezimale Aequivalente zu Z¤
160 FOR I=1 TO 13
170   READ Z¤(I) , Z(I)
180 NEXT I
190 :
200 PRINT
210 INPUT ; "Dezimalzahl" ; D
220 PRINT " -> roemische Zahl = " ;
230 :
240 FOR I=1 TO 13
250   IF D<Z(I) THEN GOTO 290
260     PRINT Z¤(I) ;
270     LET D = D-Z(I)
280     GOTO 250
290 NEXT I
300 PRINT
310 GOTO 210
320 :
330 DATA  M,1000 , CM,900 , D,500 , CD,400 , C,100 , XC,90
340 DATA  L,50 , XL,40 , X,10 , IX,9 , V,5 , IV,4 , I,1
350 :
360 REM ==================================================
```

Tafel 7.2. Vektoren zur Umrechnung von Dezimalzahlen in römische Zahlen

I	Z(I)	Z¤(I)
1	1000	M
2	900	CM
3	500	D
4	400	CD
5	100	C
6	90	XC
7	50	L
8	40	XL
9	10	X
10	9	IX
11	5	V
12	4	IV
13	1	I

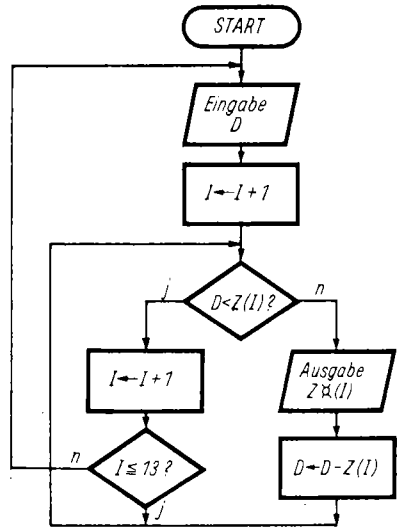


Bild 7.2. Umrechnung einer Dezimalzahl D in eine römische Zahl

8. Textverarbeitung mit BASIC

Die Verarbeitung von Zahlen mit technischen Hilfsmitteln ist für viele Menschen eine gewohnte Tätigkeit geworden. Es bereitet daher wohl kaum Schwierigkeiten, sich eine Automatisierung dieser Arbeitsschritte durch einen Computer vorzustellen. Ganz anders liegt die Situation bei der Verarbeitung von Texten. Hier sind möglicherweise bereits die zulässigen Operationen unklar. Ein Computer bietet aber gerade für solche Probleme günstige Voraussetzungen, weil er nicht nach numerischen, sondern nach logischen Prinzipien arbeitet. Und die Bedeutung einer automatisierten Bearbeitung von Texten steigt ständig!

Mit BASIC steht dem Programmierer eine Sprache zur Verfügung, die die einfache Kodierung von Textverarbeitungsalgorithmen gestattet. Es soll daher das Ziel dieses Abschnitts sein, einige Grundlagen zu erläutern und die Möglichkeiten anhand von Beispielen darzustellen.

Gegenstand der Textverarbeitung sind Daten des Typs STRING, also Textkonstanten – meist als Zeichenketten bezeichnet (Abschn. 3.3.4) – und Textvariablen, deren Werte Zeichenketten sind. Deshalb werden sie auch Zeichenkettenvariablen genannt (Abschn. 3.3.5).

- Unter einem *Textausdruck* versteht man eine Textkonstante, eine Textvariable oder eine Verknüpfung dieser Elemente.

Es gibt in BASIC folgende Klassen von *Textoperationen*, die mit solchen Daten arbeiten:

Analyse und *Synthese* von Texten
Vergleich von Texten
Konvertierungen zwischen Texten und Zahlen

Sie werden im folgenden ausführlicher behandelt.

8.1. Analyse und Synthese von Texten

Die möglichen Operationen kann man sich mit den Arbeitsschritten veranschaulichen, die beim Zusammenstellen von Texten durch Ausschneiden und Aufkleben von Zeitungsbuchstaben auftreten:

- *Untersuchen* eines Textes
  (z. B. Länge, Auftreten bestimmter Zeichen)
- *Zerlegen* eines Textes
  (Herausschneiden von Zeichenfolgen)
- *Zusammensetzen* eines neuen Textes
  (Aneinanderfügen von Zeichenfolgen).

8.1.1. Untersuchen von Texten

Vor der Verarbeitung eines Textes ist es meist erforderlich, sich über *Form* (das heißt über die Länge) und *Inhalt* (das heißt über die enthaltenen Zeichen) Klarheit zu verschaffen. Bei Textkonstanten kann das der Programmierer tun, indem er die Zeichenkette selbst liest und analysiert. BASIC arbeitet aber auch mit Textvariablen, die verschiedene Werte haben können. Hier muß der Computer diese Untersuchung während des Programmlaufs vornehmen. BASIC bietet dafür zunächst die Standardfunktion

LEN(*textausdruck*)

Diese Funktion liefert die *Anzahl* der Zeichen, die der Wert des Textausdrucks hat. Beispielsweise ergibt

LEN ("BASIC-80")

den Wert 8.

Eine Anwendung zeigt das **Programm 8.1**. Es zentriert eine eingegebene Zeichenkette T¤ auf die Mitte einer Bildschirmzeile wählbarer Länge. Vor bzw. nach dieser Überschrift wird die Zeile mit den Werten (je ein Zeichen) der Variablen V¤ bzw. N¤ aufgefüllt, die ebenfalls angefordert werden.

Programm 8.1. Zentrieren von Text in einer Zeile

```
100 REM <<<<<<<<<<<<<< ZENTRIEREN VON TEXT >>>>>>>>>>>>>>
110 :
120 LINE INPUT "Text? " , T¤
130 INPUT "Zeilenlaenge" ; L
140 INPUT "Zeichen vor/nach Text" ; V¤ , N¤
150 :
160 LET L = L - LEN(T¤)
170 LET V = INT(L/2)
180 LET N = L-V
190 :
200 PRINT STRING¤(V-1,V¤) ;
210 PRINT " " ; T¤ ; " " ;
220 PRINT STRING¤(N-1,N¤)
230 GOTO 120
240 :
250 REM ================================================
```

Für die inhaltliche Untersuchung einer Zeichenkette stellt BASIC eine Funktion (mit *mehreren* Parametern!) bereit, die das Auftreten einer bestimmten Zeichenfolge – meist eines einzelnen Zeichens – in einem gegebenen Text prüft:

INSTR([*position*,] *zu untersuchender text*, *gesuchter teiltext*)

Stößt der Interpreter auf diese Funktion, so berechnet er zunächst den arithmetischen Ausdruck (der für die *Position* des ersten Zeichens angegeben wurde, das in dem gegebenen Text zu untersuchen ist) und rundet das Resultat auf die nächste ganze Zahl. Dann ermittelt er die Werte der Textausdrücke für den *zu untersuchenden* Text und für die *aufzusuchende* Zeichenfolge. Anschließend mustert er die zu untersuchende Zeichenkette von der berechneten Position an schrittweise daraufhin durch, ob er die zu suchende Zeichenkette als Teilfolge darin vorfindet. Als *Resultat* liefert er

- die Position des *ersten* Zeichens dieser Teilfolge im Text, wenn er sie gefunden hat, oder
- den Wert *null*, wenn die Teilfolge nicht in dem Gesamttext enthalten ist oder falls dieser Text so kurz ist, daß die angegebene Anfangsposition schon nicht mehr darin auftritt.

So ergibt die Funktion

INSTR(3,S¤,"S")

den Wert 6, falls S¤ = "SCHLOSSER".

Dabei wird nur das jeweils erste Vorkommen der Teilfolge gemeldet. Möchte man jedes Auftreten wissen, muß die Inspektion fortgesetzt werden. Das ist im **Programm 8.2** der Fall, das eine eingegebene Zeile nach einer gewünschten Zeichenkette durchmustert und deren Positionen ausschreibt. Gibt man eine leere Textkette ein, so wird ein neuer Text angefordert. Wird auch an dieser Stelle nichts eingetippt, bricht das Programm ab.

Es ist möglich, die Anfangsposition in der INSTR-Funktion *wegzulassen*. Der Interpreter untersucht dann den angegebenen Text von Anfang an. Mit der obengenannten Variablen S¤ würde die Funktion

INSTR(S¤,"S")

Programm 8.2. Suchen eines Zeichens in einem Text

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"              ZEICHENKETTE SUCHEN              "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 LINE INPUT "Text? " , T¤
160 LET T = LEN(T¤)
170 IF T=0 THEN END
180 :
190 DIM P(T)
200 INPUT ; "Gesuchte Zeichenkette " , S¤
210 LET S = LEN(S¤)
220 IF S=0 THEN CLEAR : PRINT : GOTO 140
230 :
240 LET K = 0
250 LET K = INSTR(K+1,T¤,S¤)
260 IF K>0 THEN LET P(K) = 1 : GOTO 250
270 :
280 PRINT " steht an Position: " ;
290 FOR K=1 TO T
300   IF P(K)=1 THEN LET P(K) = 0 : PRINT K ;
310 NEXT K
320 PRINT
330 GOTO 200
340 :
350 REM ========================================
```

z. B. den Wert 1 liefern,

INSTR(S¤,"SS")

dagegen das Resultat 6. Diese Variante wird u. a. im Programm 8.4 benutzt.

Manche Interpreter verwenden statt INSTR das Schlüsselwort POS.

8.1.2. Zerlegen von Texten

Eine weitere Klasse von Funktionen bietet BASIC für das *Heraustrennen* von Teilen aus einem gegebenen Text an. Dabei handelt es sich um Textfunktionen mit *mehreren* Parametern. Sie stellen als Wert die herausgeschnittene *Teilzeichenkette* zur Verfügung. Dabei gibt es verschiedene Möglichkeiten:

- Abtrennen einer bestimmten Anzahl von Zeichen *am Textanfang*
- Herausschneiden von Zeichen *von einer festgelegten Position ab*, und zwar entweder eine bestimmte Anzahl von Zeichen oder alle restlichen
- Abtrennen einer bestimmten Anzahl von Zeichen *am Textende*.

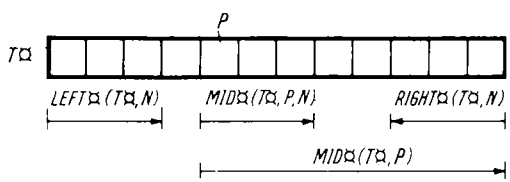


Bild 8.1. BASIC-Funktionen zur Zerlegung von Texten

Bild 8.1 gibt dazu eine Übersicht. Die entsprechenden BASIC-Funktionen sind die folgenden:

LEFT¤(*text,anzahl*)

Der Interpreter bestimmt zunächst den Wert des Textausdrucks. Weiterhin berechnet er den arithmetischen Ausdruck und rundet das Resultat. Als *Funktionswert* wird die am Anfang des gegebenen Textes (d. h. *links*) herausgeschnittene *Teilzeichenkette* mit der gewünschten Anzahl von Zeichen übergeben.

RIGHT¤(*text,anzahl*)

Diese Funktion liefert in völlig analoger Weise eine Teilzeichenkette vom Ende des Textes (d. h. von *rechts*). In beiden Fällen ergibt sich als Resultat

- die *leere* Zeichenkette, wenn die Anzahl mit null berechnet wurde
- der *volle* Text, wenn die ermittelte Anzahl größer als die Länge des Textes ist.

MID¤(*text,position*[,*anzahl*])

Bei dieser Funktion berechnet der Interpreter zusätzlich zum bisher Gesagten noch den arithmetischen Ausdruck (der für die *Position* des ersten zu übernehmenden Zeichens angegeben wurde) und rundet das Resultat. *Von dieser Stelle an* wird die gewünschte Anzahl von Zeichen aus dem Text herausgetrennt. Das Resultat ist gleich

- der *leeren* Zeichenkette, wenn die Anzahl null oder die berechnete Position größer als die Länge des Textes ist
- dem *vollen* Text, wenn die Anzahl größer ist als die restliche Zeichenzahl oder wenn dieser Parameter in der MID¤-Funktion weggelassen wurde.

Einige Interpreter verwenden statt MID¤ das Schlüsselwort SEG¤.

Zum Experimentieren mit diesen Textfunktionen kann das **Programm 8.3** verwendet werden.

Programm 8.3. Untersuchung einer Zeichenkette

```
100 REM <<<<<<<<<< ZERLEGEN EINER ZEICHENKETTE >>>>>>>>>>
110 :
120 PRINT
130 INPUT "Zeichenkette" ; T¤
140 PRINT "Erstes Zeichen = " ; LEFT¤(T¤,1)
150 PRINT "Zweites Zeichen = " ; MID¤(T¤,2,1)
160 PRINT "Alle Zeichen ab zweitem = " ; MID¤(T¤,2)
170 PRINT "Letztes Zeichen = " ; RIGHT¤(T¤,1)
180 GOTO 120
190 :
200 REM ================================================
```

8.1.3. Zusammenfügen von Texten

Textausdrücke lassen sich in BASIC nicht nur analysieren, sondern auch synthetisieren. Dazu dient ein Operator, der meist wie das Additionszeichen geschrieben wird. Er fügt die beiden angegebenen Textoperanden aneinander, klebt sie gewissermaßen lückenlos zusammen. Dabei ergibt sich wiederum ein Textausdruck vom Datentyp STRING. Mit der bereits benutzten Textvariablen S¤ = "SCHLOSSER" kann man den Textausdruck

LET S¤ = "KURT " + S¤

berechnen. Anschließend hat S¤ den Wert "KURT SCHLOSSER".

Um den praktischen Einsatz zu demonstrieren, sind mehrere Beispiele vorgesehen:

Ein einfacher Fall liegt im **Programm 8.4** vor. In der eingegebenen Zeichenkette wird ein Punkt gesucht. Ist einer vorhanden, so wird er durch ein Komma ersetzt.

Programm 8.4. Korrektur eines einzelnen Zeichens

```
100 REM <<<<<<<<<<< DEZIMALZAHLEN MIT KOMMA >>>>>>>>>>>>
110 :
120 INPUT ; "gebrochene Dezimalzahl" ; Z¤
130 LET P = INSTR(Z¤,".")
140 IF P=0 THEN GOTO 120
150 :
160 LET Z¤ = LEFT¤(Z¤,P-1) + "," + MID¤(Z¤,P+1)
170 PRINT " -> " ; Z¤
180 GOTO 120
190 :
200 REM ================================================
```

Das **Programm 8.5** ist schon komplizierter. Hier wird die Position einer auszuführenden Korrektur durch Angabe des ersten zu ändernden Zeichens adressiert. Da dieses Zeichen bis zur Korrekturstelle mehrfach auftreten könnte, benötigt das Programm auch diesen Parameter. Dann werden so viele Zeichen im alten Text korrigiert, wie neue Zeichen eingegeben wurden, wobei über das Ende hinaus geschrieben werden darf.

Programm 8.5. Korrektur einer adressierten Zeichenfolge

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                  TEXTKORREKTUR                  "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 PRINT "Zu korrigierende Textzeile:"
160 LINE INPUT T¤
170 INPUT "Korrektur ab N-tem Zeichen C¤ (N,C¤)" ; N,C¤
180 INPUT "neue Zeichenkette" ; K¤
190 GOSUB 230 ; REM Textkorrektur
200 PRINT " -> " ; T¤
210 END
220 :
230 LET P = 0
240 FOR I=1 TO N
250   LET P = INSTR(P+1,T¤,C¤)
260 NEXT I
270 LET T¤ = LEFT¤(T¤,P-1) + K¤ + MID¤(T¤,P+LEN(K¤))
280 RETURN
290 :
300 REM ================================================
```

Noch weiter geht das **Programm 8.6**. Es durchmustert die eingegebene Zeile und sucht dabei eine bestimmte Zeichenfolge, die es bei jedem Auftreten gegen eine neue austauscht. Dabei dürfen die Längen unterschiedlich sein.

Programm 8.6. Korrektur einer angegebenen Zeichenfolge

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"      AUSTAUSCH VON ZEICHENKETTE IN TEXT       "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 PRINT "zu korrigierende Textzeile:"
160 LINE INPUT T¤
170 PRINT "alte Zeichenkette, neue Zeichenkette" ;
180 INPUT A¤,N¤
190 GOSUB 250 : REM Austausch
200 PRINT " -> " ; T¤
210 END
220 :
230 REM ------------------------------------------------
240 :
250 LET A = LEN(A¤)
260 LET N = LEN(N¤)
270 LET P = 1
280 :
290 LET P = INSTR(P,T¤,A¤)
300 IF P=0 THEN RETURN
310 :
320 LET T¤ = LEFT¤(T¤,P-1) + N¤ + MID¤(T¤,P+A)
330 LET P = P + N
340 GOTO 290
350 :
360 REM ================================================
```

8.1.4. Wiederholungsfunktionen

Bei der Gestaltung von Schirmbildern oder Ausdrucken entsteht häufig die Aufgabe, dasselbe Zeichen mehrfach hintereinander auszugeben. Das läßt sich leicht durch eine Laufanweisung realisieren. Noch einfacher ist es mit der Textfunktion

STRING¤(*anzahl,zeichen*)

Der Interpreter berechnet den arithmetischen Ausdruck für die Anzahl und rundet ihn. Dann erzeugt er eine *Zeichenkette*, die das angegebene Zeichen entsprechend oft enthält. Diese Funktion wurde bereits in den Programmen 6.13 und 6.15 benutzt, wobei das Zeichen als Textkonstante, d. h. in Anführungsstrichen, angegeben wurde. Manche Interpreter lassen es auch zu, einen Textausdruck zu verwenden. Besteht dessen Wert aus mehreren Zeichen, so wird nur das erste benutzt. Eine weitere Variante wird im Abschnitt 8.2.2 behandelt. Danach wird jedes Zeichen durch eine Dezimalzahl charakterisiert. Eine entsprechende Anwendung der STRING¤-Funktion vermittelt das Programm 8.14.

Für den Fall, daß eine Kette von Leerzeichen erzeugt werden soll, könnte man in der STRING¤-Funktion das Zeichen " " oder das Äquivalent 32 angeben. Kürzer ist die Textfunktion

SPACE¤(*anzahl*)

Diese Funktion liefert eine Zeichenkette mit der angegebenen Anzahl von Leerzeichen. Sie wird z. B. im Programm 9.3 eingesetzt.

8.2. Konvertierungen

Es wurde bereits mehrfach darauf hingewiesen, daß Daten einen Typ haben. Er gibt an, welche Werte die betreffenden Daten haben können und welche Operationen sich mit ihnen ausführen lassen. Mit Zahlen beispielsweise kann man rechnen, aber auch hier unterscheidet sich u. a. die Division für ganze Zahlen (Typ INTEGER) von derjenigen für gebrochene Zahlen (Typ REAL). Zeichenketten (Typ STRING) kann man zeichenweise analysieren, zerlegen und zusammensetzen, aber nicht miteinander multiplizieren.

Trotz dieser grundsätzlichen Unterschiede spielen Umrechnungen zwischen Zahlen und Zeichenketten eine große Rolle. An welchen Stellen das der Fall ist und worin diese Konvertierungen bestehen, soll im folgenden behandelt werden.

8.2.1. Umwandeln des Datentyps von Ziffernfolgen

Gibt man eine Folge aneinandergereihter Ziffern an, beispielsweise 123, so stellt das für den Computer eine Zahl dar. Mit ihr kann er Rechenoperationen ausführen, also z. B. 567 hinzuaddieren oder die Quadratwurzel daraus ziehen. Er kann aber keine Textverarbeitungsoperationen damit ausführen, also nicht etwa die ersten beiden Ziffern von der dritten trennen. Das wäre aber möglich, wenn man ihm diese Ziffern in Anführungsstrichen übergibt, also "123" schreibt. Aber mit dieser Zeichenkette kann der Computer nicht mehr rechnen. Und schreibt man trotzdem: "123" + "567", so ist das Resultat "123567" und nicht etwa 690.

Mitunter ist es aber vorteilhaft, wenn man von einem Datentyp zum anderen übergehen kann. BASIC bietet dafür zwei Konvertierungsfunktionen, die aber nur für Zahlen gelten sowie für solche Zeichenketten, die Dezimalzahlen repräsentieren:

STR¤(*arithmetischer ausdruck*)

Diese Funktion berechnet den angegebenen arithmetischen Ausdruck, rundet ihn und liefert das Resultat (also die ermittelte Zahl) einschließlich des Vorzeichens als eine *Zeichenkette*. Dabei entsteht z. B. aus der Zahl 123 die Zeichenkette "123". Diesen Unterschied merkt man natürlich nicht, wenn man sich den Wert ausdrucken läßt, weil das Drucken eines internen Zah-

lenwertes die obengenannte Umwandlung stets einschließt. Lediglich das Leerzeichen würde fehlen, mit dem eine Zahlenausgabe abgeschlossen wird.

Ein einfaches Beispiel zeigt das **Programm 8.7**. Man erkennt an der Bildschirmausgabe, daß vor der (positiven) Nummer des Wochentags ein Leerzeichen (anstelle des Vorzeichens) steht, dahinter jedoch keines.

Programm 8.7. Einsatz der STR□-Funktion

```
100 REM <<<<<<<<<<<<<<<<<< WOCHENTAGE >>>>>>>>>>>>>>>>>>
110 :
120 LET W$ = "MO DI MI DO FR SA SO "
130 INPUT "Wochentag (MO...SO) " ; T$
140 LET P = INSTR(W$,T$)
150 IF P<>0 THEN GOTO 180
160   PRINT T$ ; " ist nicht zulaessig!"
170   GOTO 130
180 PRINT T$ ; " ist der" ; STR$(INT((P+2)/3)) ;
190 PRINT "te Tag der Woche."
200 GOTO 130
210 :
220 REM ================================================
```

Programm 8.8. Druckaufbereitung von ganzen Zahlen

```
100 REM <<<<<< DRUCKAUFBEREITUNG VON GANZEN ZAHLEN >>>>>>>
110 :
120 INPUT ; "Zahl" ; Z
130 LET Z = INT(Z)
140 LET Z$ = STR$(Z)
150 LET Z$ = MID$(Z$,2) : REM Vorzeichen abschneiden
160 LET L = LEN(Z$)
170 IF L>3 THEN LET Z$ = LEFT$(Z$,L-3) +" "+ RIGHT$(Z$,3)
180 IF Z<0 THEN LET Z$ = "- " + Z$
190 PRINT "  -> " ; Z$
200 GOTO 120
210 :
220 REM ================================================
```

Für anschließende Rechenoperationen ist diese Umwandlung von großer Bedeutung. Um an das obige Beispiel anzuschließen:

PRINT 123 + 567

liefert den Wert 670, dagegen erhält man bei

PRINT STR□(123) + STR□(567) .

das bereits erwähnte Resultat 123567. Im **Programm 8.8** wird die STR□-Funktion benutzt, um bei ganzen Zahlen zwischen der Hunderter- und der Tausenderstelle ein Leerzeichen einzuschieben. Außerdem wird bei negativen Zahlen auch zwischen dem Vorzeichen und der vordersten Ziffer ein zusätzliches Leerzeichen ausgegeben, während es bei positiven völlig entfällt.
In umgekehrter Richtung konvertiert die Funktion

VAL(*textausdruck*)

Sie durchmustert eine übergebene Zeichenkette nach *Ziffern* und sammelt sie, wobei Leerzeichen, Tabulatoren und Zeilenendezeichen unbeachtet bleiben. Sobald jedoch andere als diese drei Zeichen auftreten, wird die Untersuchung abgebrochen. Die Funktion liefert den Zahlenwert (VALue) der in der Zeichenkette gefundenen Ziffern:

"42 195 km"	42195
"96.50 M"	96.5
"M 96.50"	0
"−123.50 m"	−123

Der Leser kann die Wirkung der VAL-Funktion mit Hilfe des **Programms 8.9** untersuchen.

Programm 8.9. VAL-Funktion

```
100 REM <<<<<<<<<<<<<<<<<< VAL-FUNKTION >>>>>>>>>>>>>>>>>>
110 :
120 LINE INPUT ; "Zeichenkette? " , T$
130 PRINT "  ->  VAL =" ; VAL(T$)
140 GOTO 120
150 :
160 REM ================================================
```

8.2.2. Umwandeln des Datentyps von Bitmustern

Ein eingegebenes Zeichen wird in einem Rechenautomaten in Form eines Bitmusters abgespeichert (Abschn. 1.1.4). Jedem Zeichen entspricht ein bestimmtes solches Muster, wobei diese Zuordnung von Computer zu Computer unterschiedlich sein kann. Insbesondere für Mikrorechner hat sich aber eine bestimmte Kodierung von Textzeichen durchgesetzt, wobei jeweils ein Byte belegt wird.

Aber auch Zahlen werden als Bitmuster gespeichert. Dabei lassen sich in einem Byte die Zahlen 0 bis 255 binär verschlüsselt ausdrücken. Der Rechenautomat wendet auf beide Arten von Bitmustern – dem Datentyp entsprechend – verschiedene Operationen an. Trotz ihrer unterschiedlichen Bedeutung sehen aber die Bitmuster für Zahlen und Zeichen gleich aus.

Danach läßt sich nun jeder Zahl des angegebenen Wertebereichs ein Textzeichen zuordnen und umgekehrt. Man spricht in diesem Zusammenhang von *Textkodes*. Der allgemein verbreitete Textkode für Mikrorechner ist unter verschiedenen Namen bekannt, am gebräuchlichsten

Programm 8.10. Tafel der druckbaren ASCII-Zeichen

```
1000 REM ************************************************
1010 LPRINT
1020 LPRINT TAB(13); "TABELLE DER DRUCKBAREN ASCII-ZEICHEN"
1030 LPRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM ++++++++++++++++++ TABELLENKOPF ++++++++++++++++++
1080 :
1090 GOSUB 1420 : REM Strich
1100 GOSUB 1490 : REM Leerzeile
1110 GOSUB 1570 : REM Kopfzeile
1120 GOSUB 1490 : REM Leerzeile
1130 GOSUB 1420 : REM Strich
1140 :
1150 REM ++++++++++++++++++ TABELLE ++++++++++++++++++++
1160 :
1170 FOR I=32 TO 55
1180   LET R = I - INT(I/4)*4 : REM R = I MOD 4
1190   IF R=0 THEN GOSUB 1490 : REM Leerzeile
1200   FOR J=0 TO 3
1210     LET K = I + 24*J
1220     LPRINT "|" ;
1230     IF K=127 THEN LPRINT SPC(14) ; : GOTO 1260
1240     IF K<100 THEN LPRINT SPC(2) ; ELSE LPRINT SPC(1) ;
1250     LPRINT K ; SPC(3) ; CHR$(K) ; SPC(4) ;
1260   NEXT J
1270   LPRINT "|"
1280 NEXT I
1290 :
1300 REM ++++++++++++++++++ PROGRAMMENDE ++++++++++++++++++
1310 :
1320 GOSUB 1490 : REM Leerzeile
1330 GOSUB 1420 : REM Strich
1340 :
1350 PRINT "Tabelle ausgedruckt."
1360 END
1370 :
```

```
1380 REM +++++++++++++++ DRUCKHILFSPROGRAMME +++++++++++++++++
1390 :
1400 REM ------------- STRICH MIT ABSCHLUSS --------------
1410 :
1420 LPRINT "|" ;
1430 LPRINT STRING¤(59,"-") ;
1440 LPRINT "|"
1450 RETURN
1460 :
1470 REM ------------ LEERE ZEILE MIT SPALTEN -------------
1480 :
1490 FOR K=1 TO 4
1500   LPRINT "|" ; SPC(14) ;
1510 NEXT K
1520 LPRINT "|"
1530 RETURN
1540 :
1550 REM -------------------- KOPFZEILE ---------------------
1560 :
1570 FOR K=1 TO 4
1580   LPRINT "|" ; " Zahl Zeichen " ;
1590 NEXT K
1600 LPRINT "|"
1610 RETURN
1620 :
1630 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

TABELLE DER DRUCKBAREN ASCII-ZEICHEN

Zahl	Zeichen	Zahl	Zeichen	Zahl	Zeichen	Zahl	Zeichen
32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	¤	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Programm 8.11. Untersuchung der Bildschirmkodierung

```
100 REM <<<<<<<<<<<<<< BILDSCHIRMKODIERUNG >>>>>>>>>>>>>>
110 :
120 INPUT ; N
130 IF N>255 THEN END
140 :
150 PRINT " -> " ; CHR¤(N) ; "    " ;
160 GOTO 120
170 :
180 REM ================================================
```

Programm 8.12. Untersuchung der Tastaturkodierung

```
100 REM <<<<<<<<<<<<<< TASTATURKODIERUNG >>>>>>>>>>>>>>
110 :
120 PRINT ASC(INPUT¤(1)) ;
130 GOTO 120
140 :
150 REM ================================================
```

Programm 8.13. Einfügen von Zeichenketten in Texte

```
100 REM <<<<<<<<<<<<<< EINFUEGEN VON TEXT >>>>>>>>>>>>>>
110 :
120 LINE INPUT "zu ergaenzender Text? " ; T¤
130 PRINT
140 INPUT "Position  v o r  der Einschiebung" ; P
150 IF P=0 THEN END
160 :
170 LET E¤ = "" : REM Anfangswert der Einschiebung
180 PRINT "einzuschiebender Text? " ;
190 LET C¤ = INPUT¤(1)
200 IF ASC(C¤) = 13 THEN PRINT : GOTO 240
210   PRINT C¤ ;
220   LET E¤ = E¤ + C¤
230   GOTO 190
240 LET T¤ = LEFT¤(T¤,P) + E¤ + MID¤(T¤,P+1)
250 PRINT " -> " ; T¤
260 GOTO 130
270 :
280 REM ================================================
```

ist die Bezeichnung ASCII-Kode. Diese Zuordnung ist in der Tafel A.1 dargestellt, Teile dieser Tafel sind in den Tafeln A.2 und A.3 genauer erläutert.

Die druckbaren ASCII-Zeichen sind als Resultat des **Programms 8.10** zusammengestellt. Dieses Programm benutzt die Textfunktion

CHR¤(*arithmetischer ausdruck*)

Bei dieser Funktion berechnet der Interpreter den arithmetischen Ausdruck, rundet ihn und liefert das äquivalente *ASCII-Zeichen* (CHaRacter).

Die CHR¤-Funktion kann man zur Erfassung von Zeichen benutzen, für die auf der Tastatur keine Taste vorhanden ist: Man gibt dann einfach das dezimale Äquivalent ein. Das geschieht im **Programm 8.11**, um die Reaktion des Bildschirms und seiner Gerätebedienungsroutine auf bestimmte Zeichen – insbesondere Steuerzeichen – zu untersuchen. So löschen manche Mikrorechner bei der Ausgabe von CHR¤(12) den gesamten Bildschirm. Hat man CHR¤(8) in der Ausgabeliste, so wird die Druckposition um eine Stelle zurückgesetzt.

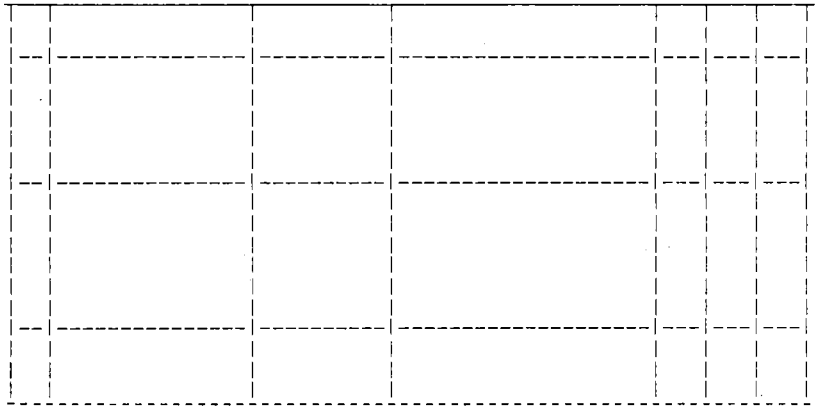
In umgekehrter Richtung konvertiert die Textfunktion

ASC(*textausdruck*)

Diese Funktion liefert das *numerische Äquivalent* des angegebenen Textzeichens entsprechend dem ASCII-Kode. Besteht der Wert des Textausdrucks aus mehreren Zeichen, so wird nur das erste benutzt.

Programm 8.14. Druck eines Tafelrahmens

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"          DRUCK EINES TABELLENRAHMENS          "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM ------------------ VORBEREITUNG ------------------
1080 :
1090 READ S : REM Anzahl der Spalten
1100 READ Z : REM Anzahl der Zeilen
1110 DIM B(S) , H(Z)
1120 FOR K=1 TO S
1130   READ B(K)
1140 NEXT K
1150 FOR I=1 TO Z
1160   READ H(I)
1170 NEXT I
1180 :
1190 REM +++++++++++++++++++ AUSDRUCK +++++++++++++++++++
1200 :
1210 FOR I=1 TO Z
1220   IF I=1 THEN LET C = 95 : GOSUB 1360 ELSE GOSUB 1450
1230   FOR J=1 TO H(I)
1240     GOSUB 1540 : REM Leerzeile
1250   NEXT J
1260 NEXT I
1270 :
1280 LET C = 126 : GOSUB 1360 : REM Strich
1290 :
1300 PRINT "Ausdruck beendet."
1310 PRINT
1320 END
1330 :
1340 REM -------------- DURCHGEHENDER STRICH --------------
1350 :
1360 LET N = 1
1370 FOR K=1 TO S
1380   LET N = N + B(K) +1
1390 NEXT K
1400 LPRINT STRING¤(N,C)
1410 RETURN
1420 :
1430 REM ------------- UNTERBROCHENER STRICH --------------
1440 :
1450 FOR K=1 TO S
1460   LPRINT "|" ;
1470   LPRINT STRING¤(B(K),45) ;
1480 NEXT K
1490 LPRINT "|"
1500 RETURN
1510 :
1520 REM ------------ UNTERBROCHENE LEERZEILE -------------
1530 :
1540 FOR K=1 TO S
1550   LPRINT "|" ; SPC(B(K)) ;
1560 NEXT K
1570 LPRINT "|"
1580 RETURN
1590 :
1600 REM ------------------ ABMESSUNGEN -------------------
1610 :
1620 DATA 7 , 4
1630 DATA 2 , 15 , 10 , 20 , 3 , 3 , 3
1640 DATA 2 , 5 , 6 , 3
1650 :
1660 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```



Eine einfache Anwendung zeigt das **Programm 8.12**. Mit seiner Hilfe kann man untersuchen, welcher Kode von der Gerätebedienungsroutine der Tastatur beim Drücken einer Taste geliefert wird. Das ist insbesondere bei individuell beschrifteten Steuertasten von Interesse. Man könnte auf diese Weise finden, daß die Zeilenendetaste eines gegebenen Computers den Kode 13 hat. Diese Kenntnis wird im **Programm 8.13** dazu benutzt, auf die Eingabe dieses Zeichens zu testen, das in diesem Fall das Ende des einzufügenden Textes signalisiert. Übrigens wurde das Programm so geschrieben, daß es mehrere Einfügungen nacheinander zuläßt; als Abbruch wird P = 0 gewertet.

In der STRING¤-Funktion ist die behandelte Konvertierung implizit enthalten; hier kann man das wiederholt auszudruckende Zeichen auch durch sein Kodeäquivalent angeben. Diese Möglichkeit wurde im **Programm 8.14** angewendet, das zum Herstellen von Formularen eingesetzt werden kann. Anzahl und Abmessungen der Zeilen und Spalten werden durch DATA-Anweisungen festgelegt.

Einige BASIC-Systeme haben *Eingabe-/Ausgabe-Anweisungen*, die zugleich eine *Textkonvertierung* enthalten und bisher noch nicht behandelt wurden. Sie lassen sich folgendermaßen mit den *hier* besprochenen Sprachelementen beschreiben.

INCHAR
läßt sich realisieren mit Hilfe von
ASC(INPUT¤(1))

Diese Funktion wartet auf die Eingabe eines Zeichens und liefert den entsprechenden ASCII-Kode.

n OUTCHAR *arithmetischer ausdruck*
läßt sich realisieren mit Hilfe von
n PRINT CHR¤ (*arithmetischer ausdruck*)

Der Wert des arithmetischen Ausdrucks wird berechnet und das entsprechende ASCII-Zeichen ausgedruckt.

8.3. Vergleiche

Eine Klasse weiterer, für Zeichenketten möglicher Operationen sind die Vergleiche. Ein Test auf Gleichheit oder Verschiedenheit ist sofort anschaulich verständlich und wurde bereits in bisherigen Programmbeispielen benutzt. Es sind aber auch alle anderen Vergleichsoperationen zulässig. Grundlage dafür ist das ASCII-Äquivalent.

Der Vergleich zweier Zeichenketten wird dabei – von *links* beginnend – *zeichenweise* durchgeführt. Zunächst werden die Kodes der beiden ersten Zeichen miteinander verglichen. Wenn sich

daraufhin noch keine Vergleichsaussage machen läßt, so folgt ein Vergleich der zweiten Zeichen. Das wird fortgesetzt, bis sich entweder die Aussage bereits vorzeitig ergibt oder das Ende der längeren der beiden Zeichenketten erreicht wird. Die kürzere wird dabei mit Leerzeichen aufgefüllt.

Programm 8.15. Sortieren eingegebener Textkonstanten

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"  SORTIEREN VON TEXTKONSTANTEN BEIM EINLESEN   "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM -------------------- ABFRAGE ---------------------
150 :
160 PRINT
170 INPUT "Maximale Anzahl von Textkonstanten" ; N
180 DIM T¤(N)
190 PRINT "Textkonstanten eingeben. Endezeichen = @"
200 :
210 REM ------------- EINGABE UND SORTIERUNG -------------
220 :
230 FOR I=1 TO N
240   INPUT T¤(I)
250   IF T¤(I)="@" THEN GOTO 330
260   FOR J=I TO 2 STEP -1
270     IF T¤(J-1)<=T¤(J) THEN GOTO 310 : REM NEXT J
280       LET Z¤ = T¤(J-1)
290       LET T¤(J-1) = T¤(J)
300       LET T¤(J) = Z¤
310   NEXT J
320 NEXT I
330 LET N = I-1
340 PRINT "Insgesamt" ; N ; "Textkonstanten eingegeben."
350 :
360 REM --------------------- AUSGABE ----------------------
370 :
380 INPUT "Name der Ausgabedatei" ; D¤
390 LPRINT "Datei "; D¤ ;":"
400 LPRINT
410 FOR I=1 TO N
420   LPRINT T¤(I) ,
430 NEXT I
440 LPRINT
450 :
460 OPEN "O" , #1 , D¤ + ".TEX"
470 PRINT #1 , N
480 FOR I=1 TO N
490   PRINT #1 , T¤(I)
500 NEXT I
510 CLOSE #1
520 PRINT "Datei "; D¤ ;" ausgegeben."
530 :
540 RESET
550 END
560 REM ==================================================
```

```
Datei NORDBEZ:

Berlin          Frankfurt       Magdeburg       Neubrandenburg
Potsdam         Rostock         Schwerin


Datei SUEDBEZ:

Cottbus         Dresden         Erfurt          Gera
Halle           Karl-Marx-Stadt                 Leipzig
Suhl
```

Programm 8.16. Konvertierung von Klein- in Großbuchstaben

```
100 REM <<<<<<<<<< KONVERTIERUNG klein -> GROSS >>>>>>>>>>
110 :
120 PRINT "Kleinbuchstabe " ;
130 LET B¤ = INPUT¤(1)
140 PRINT B¤ ;
150 IF B¤>="a" AND B¤<="z" THEN GOTO 180
160   PRINT " ???"
170   GOTO 120
180 LET B¤ = CHR¤(ASC(B¤)-32)
190 PRINT " -> Grossbuchstabe " ; B¤
200 GOTO 120
210 :
220 REM ================================================
```

Programm 8.17. Suchen in einem Textfeld

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"              SUCHEN IN TEXTFELDERN              "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 DEF FNK¤(X¤) = CHR¤(ASC(X¤)-32*(X¤>="a")*(X¤<="z"))
150 DIM M¤(12)
160 FOR I=1 TO 12
170   READ M¤(I)
180 NEXT I
190 :
200 PRINT "Monatsname: " ;
210 LET T¤ = INPUT¤(3)
220 GOSUB 370 : REM Kleinbuchstaben -> Grossbuchstaben
230 LET N¤ = T¤
240 FOR I=1 TO 12
250   LET T¤ = LEFT¤(M¤(I),3)
260   GOSUB 370 : REM Kleinbuchstaben -> Grossbuchstaben
270   IF T¤=N¤ THEN GOTO 320
280 NEXT I
290 PRINT N¤ ; " bezeichnet keinen Monatsnamen!"
300 GOTO 200
310 :
320 PRINT M¤(I) ; " ->" ; I
330 GOTO 200
340 :
350 REM ----------------- KONVERTIERUNG ------------------
360 :
370 FOR P=1 TO LEN(T¤)
380   LET T¤=LEFT¤(T¤,P-1)+FNK¤(MID¤(T¤,P,1))+MID¤(T¤,P+1)
390 NEXT P
400 RETURN
410 :
420 REM ------------------------------------------------
430 :
440 DATA Januar , Februar , Maerz , April
450 DATA Mai , Juni , Juli , August
460 DATA September , Oktober , November , Dezember
470 :
480 REM ================================================
```

Enthalten die verglichenen Zeichenketten nur Buchstaben, so ergibt sich auf diese Weise eine *lexikografische* Ordnung. Auch Ziffern sind möglich; sie haben eine geringere Wertigkeit als Buchstaben. Sonderzeichen (beispielsweise Bindestriche) stören die Sortierung.

Eine Anwendung bringt das **Programm 8.15.** Anfangs wird ein Textfeld deklariert. In dieses Feld werden die nacheinander eingegebenen Textkonstanten lexikografisch eingeordnet. Dazu vergleicht das Programm jedes neue Element mit den bisher bereits vorhandenen; erforderlichenfalls sortiert es sofort um. Am Ende wird das sortierte Feld nicht nur ausgedruckt, sondern auch als Datei auf einer Diskette abgespeichert. Die entsprechenden Anweisungen werden erst im Abschnitt 9.2 behandelt, wo die erstellte Datei weiterverarbeitet wird.

Programm 8.18. Umrechnung von römischen Zahlen in Dezimalzahlen

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"   UMWANDLUNG: ROEMISCHE ZAHL -> DEZIMALZAHL   "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 DIM Z¤(9) : REM Hilfsvektor
150 PRINT
160 INPUT ; "Roemische Zahl" ; R¤
170 LET L = LEN(R¤)
180 IF L=0 THEN PRINT : END
190 :
200 LET D = 0 : REM Anfangswert Dezimalzahl
210 LET P = 1 : REM Anfangsposition in roemischer Zahl
220 :
230 IF MID¤(R¤,P,1)<>"M" THEN GOTO 290 : REM Hunderter
240   LET D = D+1000
250   LET P = P+1
260   IF P>L THEN GOTO 480 : REM Ausdrucken
270   GOTO 230 : REM naechster Tausender
280 :
290 FOR I=3 TO 1 STEP -1
300   FOR J=1 TO 9
310     READ Z¤(J)
320   NEXT J
330   FOR J=9 TO 1 STEP -1
340     LET Z = LEN(Z¤(J))
350     IF MID¤(R¤,P,Z)=Z¤(J) THEN GOTO 390
360   NEXT J
370   GOTO 420 : REM NEXT I
380 :
390   LET D = D + J*10^(I-1)
400   LET P = P+Z
410   IF P>L THEN GOTO 480 : REM Ausdrucken
420 NEXT I
430 :
440 IF P>L THEN GOTO 480 : REM Ausdrucken
450   PRINT " hat Fehler ab Position" ; P
460   GOTO 490
470 :
480 PRINT " -> Dezimalzahl =" ; D
490 RESTORE
500 GOTO 150
510 :
520 DATA C, CC, CCC, CD, D, DC, DCC, DCCC, CM
530 DATA X, XX, XXX, XL, L, LX, LXX, LXXX, XC
540 DATA I, II, III, IV, V, VI, VII, VIII, IX
550 :
560 REM ================================================
```

Vergleiche sind auch in den beiden folgenden Beispielen enthalten. Das **Programm 8.16** fordert einen Kleinbuchstaben an und konvertiert ihn in den entsprechenden Großbuchstaben. Wird ein falsches Zeichen eingegeben, so erfolgt eine Fehlermeldung.

Das **Programm 8.17** benutzt einen ähnlichen Umwandlungsalgorithmus. Er transformiert ebenfalls Kleinbuchstaben in Großbuchstaben, läßt aber alle übrigen Zeichen unverändert. Dabei wird wieder die Eigenschaft des verwendeten BASIC-Systems ausgenutzt, die logische Aussage *wahr* durch den Zahlenwert −1 darzustellen.

Das Programm 8.17 arbeitet folgendermaßen: Es fordert einen Monatsnamen an, benötigt aber nur die ersten drei Zeichen, und zwar beliebig groß oder klein geschrieben. Dann vergleicht es diese Anfangsbuchstaben mit den Monatsnamen im Textfeld M¤, das am Anfang mit den in den DATA-Anweisungen angegebenen Textkonstanten gefüllt wird. Zu diesem Vergleich werden beide (Teil-) Zeichenketten in Großbuchstaben transformiert. Läßt sich ein entsprechendes Feldelement finden, so werden sowohl der Inhalt (Monatsname) als auch der Index (Monatsnummer) ausgeschrieben. Ansonsten druckt das Programm eine Fehlermeldung.

Komplizierter liegen die Verhältnisse im **Programm 8.18**, das eine römische Zahl in eine Dezimalzahl umwandelt. Am Anfang werden die evtl. vorhandenen Tausender umgerechnet. Für die Hunderter-, die Zehner- und die Einerpositionen wird folgendermaßen vorgegangen: Für jede dieser Gruppen stehen die entsprechenden Zeichenketten in einer DATA-Anweisung bereit. Sie werden als Textfeld in den Hilfsvektor Z□ eingelesen. Dann werden alle gespeicherten Zeichenketten nacheinander daraufhin untersucht, ob sie an der aktuellen Position in der römischen Zahl vorkommen. Erforderlichenfalls wird die Dezimalzahl entsprechend erhöht. Auf diese Weise werden Hunderter, Zehner und Einer analysiert. Wie im Programm 8.17, wird auch hier in Textfeldern gesucht, jedoch sind die Längen der jeweils verglichenen Zeichenketten unterschiedlich.

9. Einsatz externer Zusatzspeicher unter BASIC

Für die Eingabe und das Testen eines Programms wird meist viel Zeit benötigt. Es ist ein verständlicher Wunsch, dieses Softwareprodukt aufheben und bei Bedarf wiederverwenden zu können. Dieselben Überlegungen bestehen bei Datenbeständen, vor allem wenn sie umfangreich sind und laufend überarbeitet werden sollen. In diesen Fällen ist es vorteilhaft, wenn ein externer Zusatzspeicher zur Verfügung steht (Abschn. 1.4). Auf ihm werden die Programme und Daten in Form von Dateien abgelegt:

- Unter einer *Datei* (einem *File*) versteht man eine strukturierte Menge zusammengehöriger Daten, die unter einem gemeinsamen Namen auf einen externen Zusatzspeicher ausgelagert wurden.

9.1. Dateiverwaltung

Die Verwaltung von Datenbeständen und Programmen auf den externen Zusatzspeichern ist eine Aufgabe des Betriebssystems (Abschn. 1.5). Es gibt sehr vielfältige Betriebssysteme, und entsprechend unterschiedlich sind Art und Weise dieser Verwaltung sowie die Schnittstelle zum Nutzer. Daher bleibt das konkrete Betriebssystem auch bei der Nutzung höherer Sprachen dem Nutzer nicht verborgen. Bei BASIC hat es sogar einen wesentlichen Einfluß auf die Bildung von Dialekten. Diese Tatsache bringt für ein Buch Probleme mit sich. Deshalb sollen hier nur einige grundsätzliche Aspekte der Organisation und der Nutzung eines Dateisystems behandelt werden. Die Beispielprogramme arbeiten dabei mit einem MBASIC-Interpreter, der unter dem Betriebssystem CP/M läuft.

In einer Datei können recht verschiedene Informationen abgelegt werden. Quelldateien (Abschn. 2.4.1) enthalten (Programm-)Texte. Maschinenkodedateien (Abschn. 2.4.2) werden für ladbare, ablaufbereite Maschinenprogramme benötigt. Weiterhin sind aber auch Verarbeitungsdaten unterschiedlicher Typen zu speichern. Trotzdem wird meist ein einheitlicher Verwaltungsapparat gewählt. Allerdings unterscheiden sich die Dienste, die den Nutzern zur Verfügung gestellt werden:
Für das Laden eines Programms steht eine andere Anweisung zur Verfügung als für das Lesen einzelner Verarbeitungsdaten.

Der Nutzer identifiziert eine Datei durch einen *Bezeichner*, der unter Einhaltung betriebssystemspezifischer Regeln zu bilden ist, bei CP/M beispielsweise folgendermaßen:

name.spezifikation

- Der *Name* muß mit einem Buchstaben beginnen und darf 1 . . . 8 Zeichen lang sein. Er ist vom Programmierer frei wählbar.
- Die *Spezifikation* (auch *Klasse* genannt) ist 3 Zeichen lang und charakterisiert den Inhalt der Datei. Dabei gelten teilweise gewisse Konventionen, die von den benutzten Systemprogrammen (z. B. vom BASIC-Interpreter oder Assembler) abhängen.

  BEI093.BAS
  BASIC.TEX
  TELEFON.DAT

Ein Datenträger auf einem externen Zusatzspeicher wird in der Regel viele Dateien enthalten.

Interessiert man sich für die Bezeichner der gespeicherten Dateien, verwendet man die Systemanweisung

FILES

Dieses Kommando veranlaßt die Ausgabe des *Dateiverzeichnisses* der im Systemlaufwerk vorhandenen Diskette. Durch eine als Parameter zusätzlich angegebene Zeichenkette kann man diese Anweisung noch modifizieren, worauf hier aber nicht eingegangen werden soll. Bei anderen BASIC-Systemen wird für dieses Kommando das Schlüsselwort CAT (CATalogue) benutzt.

Für das *Ändern* eines Dateibezeichners (Name und Spezifikation) gibt es die Sprachanweisung

n **NAME** *alter bezeichner* **AS** *neuer bezeichner*

Die beiden Parameter sind Textausdrücke. Durch diese Anweisung erhält die an erster Stelle angegebene Datei den als zweiten Parameter angegebenen Bezeichner.

```
770 NAME "BEI093.BAS" AS "P0901.BAS"
260 NAME "TELEFON.DAT" AS "TELEFON.VRZ"
```

Verschiedentlich wird auch das Schlüsselwort RENAME benutzt. Weiterhin lassen sich vorhandene Dateien auch wieder *löschen*:

n **KILL** *bezeichner*

Diese Anweisung bewirkt, daß die durch einen Textausdruck bezeichnete Datei im Verzeichnis gestrichen wird.

```
590 KILL D□+".TEX"
```

Manche Interpreter verwenden statt KILL die Schlüsselwörter ERASE oder UNSAVE.

9.2. Programmdateien

Dateien mit BASIC-Programmen sind im Prinzip immer *Quelldateien*. Trotzdem haben sich verschiedene *Aufzeichnungsformen* eingebürgert:

- Das Programm kann als Text im *ASCII-Kode* abgespeichert werden, also in derselben Form, wie es vom Bediener in den Computer eingegeben wurde.
- Weiterhin ist es möglich, den *verdichteten Kode* aus dem Programmspeicher des BASIC-Systems auf den externen Zusatzspeicher auszulagern

Verschiedene Interpreter bieten darüber hinaus noch weitere Varianten an.

9.2.1. Binäre Programmdateien

Im Regelfall wird man das Programm so auf den externen Zusatzspeicher ablegen, wie es im Programmspeicher des BASIC-Systems steht. Bekanntlich wird dort ein verdichteter Kode eingetragen, bei dem u. a. die Schlüsselwörter durch eine binäre Kodierung angegeben sind (Abschn. 3.2.4). Dadurch werden sowohl Speicherplatz auf dem externen Datenträger als auch Übertragungszeit beim Aus- und Einlagern gespart.

Zum *Abspeichern* dient die Systemanweisung

SAVE *programmname*[*.spezifikation*]

Für den Programmnamen muß ein Textausdruck verwendet werden, der den Vorschriften des Betriebssystems genügt:

```
SAVE "BEI079"
```

Bei CP/M sind nur Großbuchstaben zulässig. Falls keine Spezifikation angegeben wird, fügt das BASIC-System selbständig BAS hinzu. Ist bereits eine Datei gleichen Namens vorhanden, so geht sie ohne Warnung verloren!

Das *Einlesen* eines Programms vom externen Zusatzspeicher erfolgt durch das Kommando

LOAD *programmname*[*.spezifikation*]

Auch hier ist für den Parameter ein Textausdruck zu verwenden; bei fehlender Spezifikation wird wieder BAS angenommen:

```
LOAD "P0901"
```

Diese Systemanweisung enthält implizit das Kommando NEW; ein bisher im Speicher enthaltenes Programm und alle bisher benutzten Variablen werden gelöscht.

Von Interesse könnte es weiterhin sein, von einem laufenden Programm her ein anderes vom externen Zusatzspeicher *laden* und anschließend *starten* zu lassen. Hierfür gibt es eine Modifikation der LOAD-Anweisung mit dem zusätzlichen Parameter R (*Run*):

LOAD *programmname*[*.spezifikation*]**,R**

Beim Laden wird auch hier das alte (also das aufrufende) Programm gelöscht, allerdings werden die benutzten Dateien nicht abgeschlossen.

```
230 LOAD "P0907" , R
```

Andere Interpreter benutzen für ähnliche Sprachanweisungen die Schlüsselwörter XEQ (eXEQute) oder GET.

BASIC-Systeme, die als externe Datenträger *Magnetbandkassetten* benutzen, haben für die behandelten Aufgaben zwei andere Kommandos (C = Cassette):

CSAVE *programmname*
CLOAD *programmname*

Bei der für den Programmnamen angegebenen Zeichenkette sind wieder bestehende Konventionen zu beachten.

9.2.2. ASCII-Programmdateien

Manchmal kann es zweckmäßig sein, das Programm im ASCII-Kode abzuspeichern. Das gilt beispielsweise in solchen Fällen, wenn das Programm auch außerhalb des jeweiligen BASIC-Systems benutzt werden soll, denn der verdichtete Programmkode ist ja systemspezifisch. So könnte eine Abarbeitung mit einem anderen BASIC-Interpreter oder die Bearbeitung mit einem allgemeinen Texteditor beabsichtigt sein. Es gibt aber auch BASIC-Anweisungen, die es fordern, daß das Programm im ASCII-Kode vorliegt.

Zur *Abspeicherung* in dieser Form dient das bereits behandelte SAVE-Kommando mit dem zusätzlichen Parameter A (ASCII):

SAVE *programmname*[*.spezifikation*]**,A**

Ein Beispiel dafür ist

```
SAVE "BEI004" , A
```

Diese Datei würde unter dem Namen BEI004.BAS abgespeichert. Das Laden erfolgt auch hier mit dem LOAD-Kommando, das selbständig das Aufzeichnungsformat erkennt.

Andere BASIC-Interpreter verwenden für das Speichern im ASCII-Kode das Schlüsselwort ASAVE. In diesen Fällen steht für das Laden ein besonderes Kommando ALOAD zur Verfügung.

Beim Laden von ASCII-Programmen wird – wie bei binär gespeicherten – ein bisher im Speicher stehendes Programm gelöscht. Möchte man das vermeiden, muß man eine andere Systemanweisung benutzen:

MERGE *programmname*[*.spezifikation*]

Dieses Kommando veranlaßt das *Laden* eines im ASCII-Kode abgespeicherten Programms. Dabei werden die darin enthaltenen Anweisungen wie *zusätzliche Eingaben* betrachtet, die ein Bediener zur Aktualisierung eines bereits vorhandenen Programms macht. Nach dem Laden stehen *beide Programme* im Speicher, wobei manche Zeilen des ursprünglichen Programms durch

neu geladene ersetzt sein könnten. Man wird ein solches Kommando beispielsweise zum Nachladen von Unterprogrammen benutzen. Einige Interpreter verwenden für diese Anweisung das Schlüsselwort APPEND.

Kassettenorientierte BASIC-Systeme haben für das Abspeichern und Wiederlesen von BASIC-Programmen im ASCII-Kode häufig die Anweisungen

LIST #*geraetenummer* **,** *programmname*
LOAD #*geraetenummer* **,** *programmname*

Dabei ist eine systemspezifische Nummer für das gewünschte Gerät anzugeben.

9.3. Dateien mit Verarbeitungsdaten

Beim Speichern und Laden von Programmen braucht der Bediener nur sehr geringe Kenntnisse von der Dateiverwaltung; eigentlich genügen hier bereits die Regeln zur Bildung von Bezeichnern. Setzt er aber externe Zusatzspeicher für Verarbeitungsdaten ein, so muß er mehr Einzelheiten wissen. Trotzdem ist es nicht erforderlich, sich mit Fragen der unmittelbaren Form der (physischen) Speicherung auf dem Datenträger zu befassen. Der Programmierer sieht die Datei stets so, wie er sie selbst seinen problemspezifischen Bedingungen entsprechend in Form von (logischen) Datensätzen strukturiert hat.

9.3.1. Zugriffsmethoden

Nach der Art und Weise des *Zugriffs* zu einer Datei, das heißt des Schreibens und des Lesens, unterscheidet man folgende Klassen von Dateien:

- Eine *sequentielle Datei* wird fortlaufend geschrieben, wobei die einzelnen Datensätze verschiedene Länge haben dürfen. Wurde die Abspeicherung einmal abgeschlossen, so kann die Datei bei manchen Betriebssystemen nicht mehr erweitert werden.
  Das Lesen einer solchen Datei kann nur vom Anfang an sequentiell, Datensatz für Datensatz, erfolgen. Dieses Vorgehen wurde bereits im Zusammenhang mit der READ-Anweisung behandelt (Abschn. 4.2.2, Bild 4.2).
  Diese Vorgehensweise entspricht der fortlaufenden Aufzeichnung auf Loch- oder Magnetband. Sie ist aber auch bei Disketten möglich.
- Eine *Direktzugriffsdatei* arbeitet mit Datensätzen definierten Aufbaus. Sowohl beim Schreiben als auch beim Lesen adressiert der Programmierer den betreffenden Datensatz durch eine – im Rahmen des Betriebssystems – frei wählbare Nummer. Dabei ist beim Schreiben oder Lesen eine beliebige Reihenfolge der aufgerufenen Nummern zulässig.
  Direktzugriffsdateien kann man auch nach einem Abschluß noch erweitern. Sie lassen sich effektiv nur auf Disketten implementieren. Bei Magnetbandkassetten treten verschiedene Schwierigkeiten auf, u. a. ergeben sich recht lange Suchzeiten.

Im folgenden sollen einige Beispiele zum Einsatz sequentieller Dateien vermittelt werden. Die Behandlung der Arbeit mit Direktzugriffsdateien würde zu viele Details erfordern und der Zielstellung dieses Buches nicht entsprechen.

9.3.2. Eröffnen von Dateien

Dateien sind Datenbestände, die außerhalb eines konkreten Programms auf externen Datenträgern existieren. Es ist möglich, daß mehrere Programme dieselbe Datei verwenden, aber auch, daß ein Programm unterschiedliche Datenbestände nutzt. Um unter diesen Bedingungen hinreichend flexibel arbeiten zu können, unterscheidet man zwei Formen der *Identifikation* einer Datei:

- Auf dem *Datenträger* wird eine Datei unter einem Namen verwaltet, der dem Betriebssystem

bekannt ist. Dieser *Bezeichner* wird auch *physisch* genannt, weil er auf dem Datenträger aufgezeichnet ist, oder global (programmextern), weil er für alle Nutzer gilt. Hierzu sind auch die Programmbezeichner (Abschn. 9.2) zu zählen.

- In Ergänzung dazu wird für jede in einem konkreten *Programm* benötigte Datei noch eine lokale (programminterne) Dateinummer eingeführt. Diese *logische* (das heißt problemspezifische) *Nummer* wird bei jedem Zugriff zur Datei benutzt.

Vor dem ersten Zugriff zu einer Datei muß der Programmierer eine *Verbindung* zwischen beiden Benennungen herstellen und festlegen, ob die betreffende Datei *geschrieben* oder *gelesen* werden soll, damit das Betriebssystem die nötigen Vorbereitungen treffen kann:

- Als *Eröffnen* einer Datei bezeichnet man eine Deklaration, in der die programminterne Nummer und der programmexterne Bezeichner einer zu benutzenden Datei sowie die Zugriffsart festgelegt werden.

n **OPEN** *zugriffsart*, [#]*dateinummer*, *bezeichner*

Zu den angegebenen Parametern sind folgende Erläuterungen zu geben:

- Für die *Zugriffsart* ist ein Textausdruck einzusetzen, wobei nur die Werte

  "O" für eine *Ausgabedatei* (Output) oder
  "I" für eine *Eingabedatei* (Input)

  möglich sind.
- Als *Dateinummer* ist ein arithmetischer Ausdruck zu verwenden. Der zulässige Wertebereich ist 0 . . . 15. Das Nummernzeichen darf entfallen.
- Als *Dateibezeichner* dient ein Textausdruck, der auch die Dateispezifikation enthalten muß.

```
150 OPEN "I", #1, D□+".TEX"
320 OPEN "O", N, "TELE.DAT"
```

Wird eine Eingabedatei eröffnet, so prüft das Betriebssystem, ob sie vorhanden ist; erforderlichenfalls wird eine Fehlermeldung veranlaßt. Bei einer Ausgabedatei wird statt dessen eine entsprechende Eintragung im Dateiverzeichnis vorgenommen. Ist bereits eine Datei gleichen Namens vorhanden, so geht sie dabei verloren.

Einige Interpreter verwenden statt OPEN das Schlüsselwort FILE.

9.3.3. Abschließen von Dateien

Beendet man die Arbeit mit einer Datei, sollte man das ebenfalls dem Betriebssystem mitteilen.

- Unter dem *Abschließen* einer Datei versteht man alle organisatorischen Aktionen des Betriebssystems, die *nach* dem Nutzen einer Datei erforderlich sind.

n **CLOSE** [[#]*dateinummer*]

Diese BASIC-Anweisung veranlaßt das Abschließen der angegebenen Datei.

```
630 CLOSE #2
```

Gibt man keinen Parameter an, so werden *alle* offenen Dateien abgeschlossen.

Diese CLOSE-Anweisung ist in der END-Anweisung (im Gegensatz zu STOP) implizit enthalten.

Diskettenbetriebssysteme übernehmen häufig am Anfang das Dateiverzeichnis vom Datenträger in den Hauptspeicher, um bei Zugriffen zum externen Zusatzspeicher Zeit zu sparen. Änderungen werden meist zunächst nur in dieser Kopie eingetragen, noch nicht auf der Diskette. In der Zwischenzeit abgespeicherte Dateien stehen zwar auf dem Datenträger, sie gehen aber möglicherweise verloren, wenn man eine Diskette unvermittelt aus dem Laufwerk nimmt; denn sie stehen nicht in allen Fällen bereits im Verzeichnis. Hier hilft die BASIC-Anweisung

n **RESET**

Sie veranlaßt das Betriebssystem, das *aktuelle Dateiverzeichnis* aus dem Hauptspeicher *auf die Diskette* zu übertragen. Außerdem enthält sie die CLOSE-Anweisung. Um den Verlust von Dateien

zu vermeiden, sollte jedes Programm, das Dateien aktualisiert oder neu einrichtet, am Ende eine RESET-Anweisung enthalten.

9.3.4. Schreiben von Dateien

Der Einsatz eines externen Zusatzspeichers bringt – außer dem bereits behandelten Eröffnen und Abschließen der benutzten Dateien – keinerlei Komplikationen für den Programmierer mit sich. Das Schreiben von Daten erfolgt mit einer Modifikation der PRINT-Anweisung:

n **PRINT #** *dateinummer* , *ausgabeliste*

Diese Sprachanweisung bewirkt die *Ausgabe* der in der Liste angegebenen Datenelemente *auf den externen Datenträger* in genau gleicher Weise, wie sie auf dem Bildschirm oder (bei der LPRINT-Anweisung) auf dem Papier stehen würden. Als *Trennzeichen* sind sowohl Kommas als auch Semikolons möglich; jedoch wird bei Kommas die entsprechende Anzahl von Leerzeichen ausgegeben, die auf dem Datenträger Speicherplatz belegen.

Bei *Zeichenketten* sind einige Komplikationen zu beachten. Um ihre Ursachen zu verstehen, muß man sich die Arbeit des Interpreters beim Einlesen der abgespeicherten Datenelemente vorstellen. Während bei der Ausgabe hinter jeder Zahl ein Leerzeichen ausgeschrieben wird, ist das Ende einer Zeichenkette nicht mehr zu erkennen. In der PRINT#-Anweisung evtl. vorhandene Anführungsstriche werden ja weggelassen!

Um die einzelnen Zeichenketten in der richtigen Weise voneinander zu trennen, gibt es folgende Möglichkeiten, die sich der Leser an entsprechenden Bildschirmausgaben klarmachen kann:

- In jeder PRINT#-Anweisung wird nur je *ein* Ausgabeelement angegeben. Dann dient das Zeilenendezeichen als Trennzeichen:

  ```
  370 PRINT #1 , "TECHNISCHE"
  380 PRINT #1 , "UNIVERSITAET"
  ```

- Nach jeder Zeichenkette in der Ausgabeliste wird die Ausgabe eines *Kommas* erzwungen, das als Trennzeichen wirkt:

  ```
  370 PRINT #1 , "TECHNISCHE" ; "," ; "UNIVERSITAET"
  ```

- Schwierigkeiten ergeben sich, wenn eine ausgegebene Zeichenkette selbst ein Komma enthält. So würden durch die einzelne Ausgabe

  ```
  350 PRINT #2 , "LEHMANN, HELMUT"
  ```

  auf dem Datenträger zwei voneinander durch Komma getrennte Zeichenketten stehen! Eine Möglichkeit besteht hier darin, die Ausgabe von *Anführungszeichen* auf das Speichermedium zu erzwingen. Das muß allerdings mit der CHR□-Funktion erfolgen, weil Anführungszeichen selbst ja eliminiert werden:

  ```
  350 PRINT #2 , CHR□(34) ; "LEHMANN, HELMUT" ; CHR□(34)
  ```

Manche Interpreter umgehen die Schwierigkeiten mit Hilfe einer speziellen Ausgabeanweisung:

n **WRITE #** *dateinummer* , *ausgabeliste*

Diese Anweisung setzt zwischen alle Datenelemente auf dem externen Zusatzspeicher *Kommas* und schließt Zeichenketten in *Anführungszeichen* ein. Als Trennzeichen sind hier in der Ausgabeliste nur Kommas zulässig.

```
350 WRITE #2 , "LEHMANN, HELMUT"
370 WRITE #2 , "TECHNISCHE" , "UNIVERSITAET"
```

Die einzelnen Datenelemente dürfen bei sequentiellen Dateien – wie bereits erwähnt – unterschiedliche Längen aufweisen. In jedem Sektor einer Diskette läßt sich aber immer nur eine bestimmte Anzahl von Bytes unterbringen (Abschn. 1.4.2). Das Betriebssystem realisiert deshalb eine sog. *Blockung*. Beim Eröffnen einer Ausgabedatei wird für sie ein *Puffer* reserviert, das ist ein Bereich im Hauptspeicher vom Umfang eines Diskettensektors. In diesem Puffer werden die Werte der in der Ausgabeliste angegebenen Elemente zunächst zwischengespeichert. Ist der Puf-

Programm 9.1. Erfassen einer Textdatei

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                    TEXTERFASSUNG                "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM -------------------- ABFRAGE ---------------------
150 :
160 PRINT
170 INPUT "Dateiname" ; D¤
180 LET D¤ = LEFT¤(D¤,8)
190 LET D0¤ = D¤ + ".OLD"
200 LET D1¤ = D¤ + ".TEX"
210 PRINT "Datei "; D¤ ;" bereits vorhanden (J|-)" ;
220 INPUT A¤
230 IF A¤<>"J" THEN OPEN "O", #1, D1¤ : GOTO 430
240 :
250 REM -------------------- KOPIEREN --------------------
260 :
270 NAME D1¤ AS D0¤
280 OPEN "O", #1, D1¤
290 OPEN "I", #2, D0¤
300 :
310 LINE INPUT #2, T¤
320 PRINT #1, T¤
330 IF NOT EOF(2) THEN GOTO 310
340 :
350 CLOSE #2
360 PRINT "Datei "; D0¤ ;" nach Datei "; D1¤ ;" kopiert."
370 PRINT "Datei "; D0¤ ;" loeschen (J|-)" ;
380 INPUT A¤
390 IF A¤="J" THEN KILL D0¤
400 :
410 REM -------------------- ERFASSEN --------------------
420 :
430 PRINT "Datensaetze eingeben. Dateiende = @"
440 :
450 LINE INPUT T¤
460 IF RIGHT¤(T¤,1)="@" THEN GOTO 490
470 PRINT #1, T¤
480 GOTO 450
490 PRINT #1, LEFT¤(T¤,LEN(T¤)-1) : REM  @ abtrennen
500 :
510 CLOSE #1
520 PRINT "Datei "; D1¤ ;" abgeschlossen."
530 :
540 RESET
550 END
560 REM ==================================================
```

fer gefüllt, so wird er auf den externen Zusatzspeicher übertragen. Außerdem wird der Puffer beim *Abschließen* der Datei geleert.

Ein Beispiel für das Schreiben einer Diskettendatei wurde bereits im Programm 8.15 gebracht. Hier wurde eine Ausgabedatei mit wählbarem Namen und der Spezifikation TEX eröffnet und mit den vom Bediener eingegebenen, sortierten Zeichenketten gefüllt.

Das **Programm 9.1** kann dazu dienen, einen fortlaufenden Text in einer Ausgabedatei zu erfassen. Da hier Leerzeichen und Kommas auftreten, wurde für die Bedienereingabe die LINE-INPUT-Anweisung benutzt. Weiterhin erlaubt es das Programm, eine bereits vorhandene Datei zu ergänzen. In dem benutzten BASIC-System ist es dazu erforderlich, zunächst die alte Datei in eine neue zu kopieren. Dann erst können die zusätzlichen Zeichen eingegeben werden. Die alte Datei bleibt auf Wunsch erhalten; sie erhält im vorliegenden Fall die Spezifikation OLD. **Bild 9.1** zeigt dazu den Programmablaufplan.

BASIC-Systeme, die mit *Magnetbandkassetten* arbeiten, haben häufig keine solche Dateiverwaltung, Sie bieten mitunter nur die Möglichkeit, die Werte aller Elemente eines einzelnen *Da-*

tenfelds auszulagern:

n **CSAVE*** *programmname* ; *feldname*

Der Programmname ist eine Zeichenkette, während der Name des Datenfelds ohne Anführungszeichen anzugeben ist. Eröffnen und Abschließen der Dateien entfallen meist in diesen einfachen Systemen. Eine Blockung ist bei Magnetbändern nicht unbedingt erforderlich, falls Blöcke unterschiedlicher Länge geschrieben werden können.

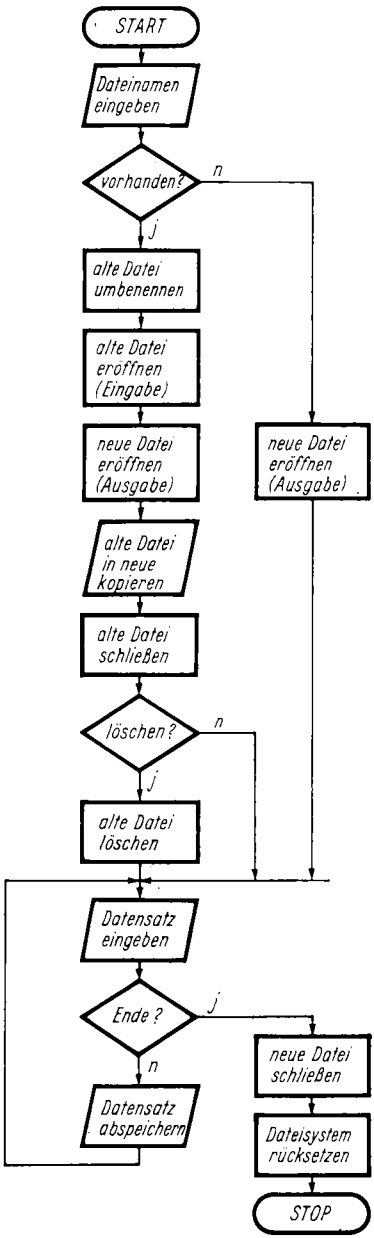


Bild 9.1. Erfassen von Datensätzen auf Diskette

9.3.5. Lesen von Dateien

Wie für das Schreiben auf externe Datenträger gibt es auch für das Lesen eine Modifikation der bekannten Standardanweisung:

n **INPUT #***dateinummer* **,** *eingabeliste*

Diese Anweisung *liest* die auf dem externen Speichermedium abgelegten Daten *fortlaufend ein* und ordnet sie der Reihe nach den Variablen zu, die in der Eingabeliste enthalten sind. Als *Trennzeichen* in der Eingabeliste sind Kommas zu verwenden.

```
220 INPUT #3 , N , T¤ , C¤
```

Am Beginn der Arbeit, nach der OPEN-Anweisung, steht der *Datenzeiger* am Anfang der Datei. Beim Einlesen *wandert* er – wie bei der READ-Anweisung (Bild 4.2) – durch den *sequentiell geordneten* Datenbestand und zeigt dabei immer auf das *nächste* noch nicht gelesene Element. Im Gegensatz zum Lesen in DATA-Dateien gibt es aber hier eine Funktion, mit der man prüfen kann, ob noch Daten vorhanden sind:

EOF(*dateinummer***)**

Programm 9.2. Ausdrucken einer Textdatei

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"          AUSDRUCKEN EINER TEXTDATEI            "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM -------------------- ABFRAGE ---------------------
150 :
160 PRINT
170 INPUT "Dateiname" ; D¤
180 OPEN "I" , #1 , D¤ + ".TEX"
190 INPUT "Zeilenanzahl je Seite" ; Z
200 INPUT "Ausgabe auf Drucker (P)" ; A¤
210 LET P = A¤="P" OR A¤="p"
220 IF P THEN LPRINT "Datei "; D¤ ;":" : LPRINT ELSE PRINT
230 LET I = 0 : REM Zeilenzaehler
240 :
250 REM -------------------- AUSGABE ---------------------
260 :
270 LINE INPUT #1 , T¤
280 IF P THEN LPRINT T¤ ELSE PRINT T¤
290 LET I = I+1
300 IF I<Z THEN GOTO 340
310   IF P THEN PRINT "Formularwechsel! - fertig" ;
320   INPUT A
330   LET I = 0
340 IF NOT EOF(1) THEN GOTO 270
350 :
360 IF NOT P THEN PRINT
370 CLOSE #1
380 PRINT "Datei "; D¤ ;" ausgedruckt."
390 :
400 END
410 REM ==================================================
```

```
Datei BASIC:

Was denken Sie ueber BASIC?¤
BASIC ist eine einfache Programmiersprache, die
man in vielen Bereichen einsetzen kann. Sie laesst
sich leicht erlernen und ermoeglicht eine schnelle
Programmentwicklung im Dialogbetrieb.¤
Versuchen Sie es doch einmal, in BASIC zu
programmieren! Sie werden ueberrascht und
zufrieden sein.
```

Diese logische Funktion liefert nur dann den Wert wahr, wenn das Ende der angegebenen Datei (*End Of File*) erreicht wurde.

```
590 IF EOF(1) THEN CLOSE #1 : GOTO 700
```

Wie bereits im Abschnitt 9.3.4 behandelt, benötigt die INPUT#-Anweisung zur einwandfreien Trennung der Daten auf dem externen Speichermedium bestimmte Begrenzungszeichen, die vom Datentyp abhängen:

- *Zahlen:* Leerzeichen, Komma, Zeilenendezeichen
- *Zeichenketten:* Komma, Zeilenendezeichen. (Führende und abschließende Leerzeichen gehen verloren.)

Für das Einlesen von Zeichenketten, die führende und abschließende Leerzeichen sowie Kom-

Programm 9.3. Ausdrucken einer Textdatei im Blocksatz

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"                  RANDAUSGLEICH                "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++ EINGABE ++++++++++++++++++++
1080 :
1090 INPUT "Dateiname" ; D¤
1100 OPEN "I" , #1 , D¤ + ".TEX"
1110 INPUT "Zeichenanzahl je Zeile" ; N
1120 INPUT "Zeilenanzahl je Seite" ; M
1130 :
1140 REM ++++++++++++++++++ VERARBEITUNG ++++++++++++++++++
1150 :
1160 LET A¤ = "¤" : REM Kennzeichen fuer Absatz
1170 LET L¤ = "" : REM Anfangswert Ausgabetext
1180 LET L = 0 : REM Laenge Ausgabetext
1190 LET B1 = 0 : REM Anfangswert fuer Test: Absatz?
1200 LET I = 0 : REM Anfangswert Zeilennummer
1210 :
1220 REM ---------- NAECHSTE ZEILE BEREITSTELLEN ----------
1230 :
1240 IF L>N+1 THEN GOTO 1420 : REM Text muss geteilt werden
1250 IF L=N+1 OR B1 OR EOF(1) THEN GOTO 1340
1260 :
1270 LINE INPUT #1 , T¤ : REM naechste Eingabezeile
1280 LET L¤ = L¤ + T¤
1290 LET B1 = RIGHT¤(L¤,1)=A¤
1300 IF NOT B1 THEN LET L¤ = L¤ +" " : REM Abschluss: SP|A¤
1310 LET L = LEN(L¤)
1320 GOTO 1240
1330 :
1340 LET N¤ = L¤ : REM auszudruckender Rest
1350 LET L¤ = ""
1360 LET L = 0
1370 LET B1 = 0
1380 GOTO 1800 : REM Ausdrucken
1390 :
1400 REM ---------- TEILEN ----------
1410 :
1420 FOR P=N TO 1 STEP -1
1430   IF MID¤(L¤,P,1)=" " THEN GOTO 1530 :REM Leerzeichen!
1440 NEXT P
1450 :
1460 LET N¤ = LEFT¤(L¤,N-1) + "-" : REM Wort abgeteilt
1470 LET L¤ = MID¤(L¤,N) : REM Rest des abgeteilten Wortes
1480 LET L = LEN(L¤)
1490 GOTO 1800 : REM Ausdrucken
1500 :
```

```
1510 REM --------------- LEERZEICHEN SUCHEN ---------------
1520 :
1530 LET Z¤ = LEFT¤(L¤,P-1) : REM P = Position Leerzeichen
1540 LET Z = LEN(Z¤)
1550 LET L¤ = MID¤(L¤,P+1) : REM Rest der Zeile
1560 LET L = LEN(L¤)
1570 LET E = N-Z: REM Anzahl der einzufuegenden Leerzeichen
1580 LET V = 0 : REM Anzahl der vorhandenen Leerzeichen
1590 :
1600 FOR J=1 TO Z
1610   IF MID¤(Z¤,J,1)=" " THEN LET V = V+1
1620 NEXT J
1630 IF V=0 THEN LET N¤ = Z¤ : GOTO 1800 : REM Ausdrucken
1640 :
1650 REM ------------ LEERZEICHEN EINSCHIEBEN -------------
1660 :
1670 LET N¤ = "" : REM Anfangswert Ausgabezeile
1680 FOR J=1 TO Z
1690   LET C¤ = MID¤(Z¤,J,1)
1700   LET N¤ = N¤ + C¤
1710   IF C¤<>" " THEN GOTO 1760 : REM NEXT J
1720     LET K = INT(E/V)
1730     LET N¤ = N¤ + SPACE¤(K)
1740     LET E = E-K
1750     LET V = V-1
1760 NEXT J
1770 :
1780 REM +++++++++++++++++++ AUSGABE +++++++++++++++++++++
1790 :
1800 LET B2 = RIGHT¤(N¤,1)=A¤
1810 IF B2 THEN LET N¤ = LEFT¤(N¤,LEN(N¤)-1)
1820 LPRINT N¤
1830 LET I = I+1 : REM naechste Zeile
1840 IF EOF(1) AND L=0 THEN PRINT : CLOSE : END
1850 :
1860 IF B2 THEN LPRINT : LET I = I+1 : REM Absatz
1870 IF I<M THEN GOTO 1240
1880   PRINT "Formularwechsel! - fertig" ;
1890   INPUT F
1900   LET I=0 : REM neue Seite
1910 GOTO 1240
1920 :
1930 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Was denken Sie ueber BASIC?

BASIC ist eine einfache Programmiersprache, die man in vielen Bereichen einsetzen kann. Sie laesst sich leicht erlernen und ermoeglicht eine schnelle Programmentwicklung im Dialogbetrieb.

Versuchen Sie es doch einmal, in BASIC zu programmieren! Sie werden ueberrascht und zufrieden sein.

Was denken Sie ueber BASIC?

BASIC ist eine einfache Programmiersprache, die man in vielen Bereichen einsetzen kann. Sie laesst sich leicht erlernen und ermoeglicht eine schnelle Programmentwicklung im Dialogbetrieb.

Versuchen Sie es doch einmal, in BASIC zu programmieren! Sie werden ueberrascht und zufrieden sein.

mas enthalten und mit einzelnen PRINT#-Anweisungen ausgegeben wurden, eignet sich besonders eine Modifikation der LINE-INPUT-Anweisung:

n LINE INPUT #*dateinummer*, *textvariablenname*

Wie bereits im Abschnitt 4.3.3 erläutert, benutzt diese Anweisung zur Trennung nur das *Zeilenendezeichen.* Daher kann man beispielsweise die durch

220 PRINT #1, "SCHUBERT, FRANZ"

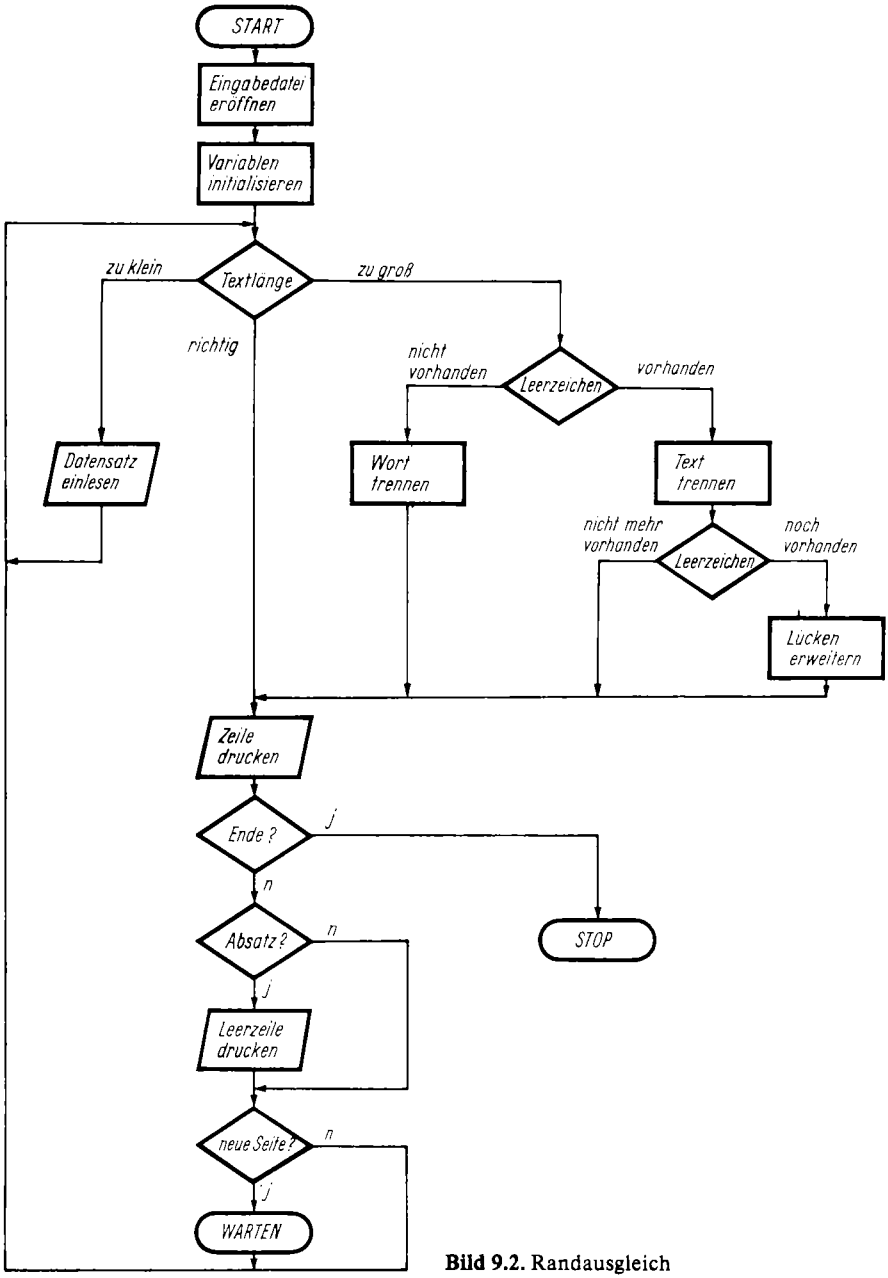


Bild 9.2. Randausgleich

ausgegebene Zeichenkette mit Hilfe von

930 LINE INPUT #2 , N¤

richtig wieder einlesen, ohne explizit oder durch die WRITE-Anweisung Anführungszeichen auf den Datenträger zu schreiben.

Programm 9.4. Mischen zweier sortierter Dateien

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"       MISCHEN ZWEIER SORTIERTER DATEIEN       "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++ EINGABE +++++++++++++++++++++
1080 :
1090 INPUT "Name der ersten Eingabedatei" ; D1¤
1100 OPEN "I" , #1 , D1¤ + ".TEX"
1110 INPUT #1 , K : REM Anzahl der Elemente
1120 DIM A¤(K)
1130 FOR I=1 TO K
1140   INPUT #1 , A¤(I)
1150 NEXT I
1160 CLOSE #1
1170 PRINT "Datei "; D1¤ ;" eingelesen."
1180 :
1190 INPUT "Name der zweiten Eingabedatei" ; D2¤
1200 OPEN "I" , #2 , D2¤ + ".TEX"
1210 INPUT #2 , L : REM Anzahl der Elemente
1220 DIM B¤(L)
1230 FOR I=1 TO L
1240   INPUT #2 , B¤(I)
1250 NEXT I
1260 CLOSE #2
1270 PRINT "Datei "; D2¤ ;" eingelesen."
1280 :
1290 REM +++++++++++++++++++ MISCHEN +++++++++++++++++++++
1300 :
1310 LET E = K+L
1320 DIM C¤(E)
1330 LET I = 1
1340 LET J = 1
1350 LET M = 0
1360 :
1370 LET M = M+1
1380 IF M>E THEN GOTO 1620 : REM -> Ausgabe
1390   IF A¤(I)<B¤(J) THEN GOTO 1520
1400 :
1410 REM ---------------- B¤(J) EINORDNEN -----------------
1420 :
1430 LET C¤(M) = B¤(J)
1440 LET J = J+1
1450 IF J<=L THEN GOTO 1370 : REM naechstes Element
1460 :
1470 LET B¤(L) = A¤(K) : REM  B¤  leer
1480 LET J = J-1
1490 GOTO 1370 : REM naechstes Element
1500 :
1510 REM ---------------- A¤(I) EINORDNEN -----------------
1520 :
1530 LET C¤(M) = A¤(I)
1540 LET I = I+1
1550 IF I<=K THEN GOTO 1370 : REM naechstes Element
1560 :
1570 LET A¤(K) = B¤(L) : REM  A¤ leer
1580 LET I = I-1
1590 GOTO 1370 : REM naechstes Element
1600 :
```

```
1610 REM ++++++++++++++++++++ AUSGABE ++++++++++++++++++++
1620 :
1630 INPUT "Name der Ausgabedatei" ; D3¤
1640 LPRINT "Datei "; D3¤ ;":"
1650 LPRINT
1660 FOR M=1 TO E
1670   LPRINT C¤(M) ,
1680 NEXT M
1690 LPRINT
1700 :
1710 OPEN "O" , #1 , D3¤ + ".TEX"
1720 PRINT #1 , E : REM Anzahl der Elemente
1730 FOR M=1 TO E
1740   PRINT #1 , C¤(M)
1750 NEXT M
1760 CLOSE #3
1770 PRINT "Datei "; D3¤ ;" ausgegeben."
1780 :
1790 PRINT
1800 RESET
1810 END
1820 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
Datei BEZIRKE:

Berlin          Cottbus         Dresden         Erfurt
Frankfurt       Gera            Halle           Karl-Marx-Stadt
Leipzig         Magdeburg       Neubrandenburg
Potsdam         Rostock         Schwerin        Suhl
```

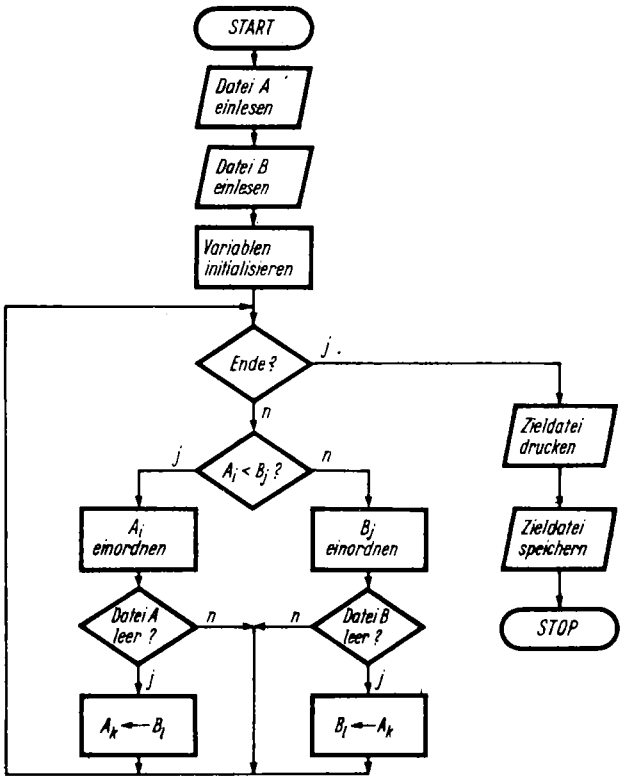


Bild 9.3. Mischen zweier sortierter Dateien

Programm 9.5. Verwaltung einer Datei von Kennsätzen
(Rahmenprogramm für die Programme 9.6 bis 9.9)

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"     VERWALTEN EINER DATEI VON KENNSAETZEN     "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 PRINT "Kennsaetze erfassen ..... 1"
160 PRINT "Kennsaetze streichen ..... 2"
170 PRINT "Kennsaetze suchen ..... 3"
180 PRINT "Datei ausdrucken ..... 4"
190 INPUT "Auswahl" ; A
200 PRINT
210 IF A=0 THEN RESET : END
220 :
230 ON A GOTO 260 , 270 , 280 , 290 : REM Nachladen
240 GOTO 190
250 :
260 LOAD "P0906" , R
270 LOAD "P0907" , R
280 LOAD "P0908" , R
290 LOAD "P0909" , R
300 :
310 REM ==================================================
```

Programm 9.6. Erfassen von Kennsätzen

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"            ERFASSEN VON KENNDATEN              "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Dateiname" ; D¤
160 LET D¤ = LEFT¤(D¤,8)
170 PRINT "DATEI "; D¤ ;" bereits vorhanden (J|N)" ;
180 INPUT A¤
190 INPUT "Maximale Anzahl neuer Kennsaetze" ; N
200 LET D = A¤="N" OR A¤="n"
210 IF D THEN LET I = 1 : DIM V¤(N) , V(N) : GOTO 390
220 LET D = A¤="J" OR A¤="j"
230 IF NOT D THEN PRINT "Fehleingabe!" : GOTO 170
240 :
250 REM -------------------- KOPIEREN --------------------
260 :
270 OPEN "I" , #1 , D¤ + ".DAT"
280 INPUT #1 , NO : REM Anzahl vorhandener Kennsaetze
290 LET N = N+NO
300 DIM V¤(N) , V(N)
310 FOR I=1 TO NO
320   INPUT #1 , V¤(I) , V(I)
330 NEXT I
340 CLOSE #1
350 PRINT "Datei "; D¤ ;" eingelesen."
360 :
370 REM ------------- ERFASSEN UND SORTIEREN -------------
380 :
390 PRINT "Neue Kennsaetze eingeben. Endezeichen = @"
400 FOR I=I TO N
410   INPUT ; V¤(I)
420   IF V¤(I)="@" THEN GOTO 480
430   INPUT "  " ; V(I)
440   FOR J=I TO 2 STEP -1
450     IF V¤(J-1)>V¤(J) OR V¤(J-1)=V¤(J) AND V(J-1)>V(J)
          THEN GOSUB 650 : REM Vertauschen
460   NEXT J
470 NEXT I
480 PRINT
490 :
```

```
500 REM ------------------ ABSPEICHERN -------------------
510 :
520 LET N = I-1 : REM neue Anzahl vorhandener Kennsaetze
530 OPEN "O" , #2 , D¤ + ".DAT"
540 PRINT #2 , N
550 FOR I=1 TO N
560   PRINT #2 , V¤(I) ; "," ; V(I)
570 NEXT I
580 CLOSE #2
590 PRINT "Datei "; D¤ ;" abgespeichert."
600 PRINT
610 LOAD "P0905" , R : REM Rahmenprogramm wieder laden
620 :
630 REM ------------------ VERTAUSCHEN -------------------
640 :
650 LET Z¤ = V¤(J-1)     : LET Z = V(J-1)
660 LET V¤(J-1) = V¤(J) : LET V(J-1) = V(J)
670 LET V¤(J) = Z¤       : LET V(J) = Z
680 RETURN
690 :
700 REM ================================================
```

Ein einfaches Beispiel ist im **Programm 9.2** dargestellt. Hier wird eine Textdatei, die mit dem Programm 9.1 abgespeichert wurde, zeilenweise wieder ausgeschrieben.

Im Gegensatz dazu führt das **Programm 9.3,** dessen Programmablaufplan im **Bild 9.2** gezeigt ist, eine Textverarbeitung durch. Die Eingabedatei enthält wieder einen fortlaufenden Text. Der Bediener kann nun angeben, wie viele Zeichen je Zeile und wie viele Zeilen je Seite gedruckt werden sollen. Das Programm trennt den für die Ausgabe bestimmten Text (L¤) zwischen zwei Wörtern (→N¤) und füllt ihn dann durch Vervielfachung der vorhandenen Leerzeichen so auf, daß er rechts- und linksbündig abschließt. Wörter, die länger als eine Zeile sind, werden willkürlich geteilt. Am Ende jeder Seite wartet der Computer, bis der Bediener das neue Formular eingespannt hat; die Fortsetzung erfolgt durch eine beliebige Eingabe.

Das **Programm 9.4** behandelt das Mischen zweier sortierter Dateien **(Bild 9.3),** die am Anfang in zwei Textvektoren eingelesen werden. Dann wird die neue Datei schrittweise aufgebaut, indem immer die beiden im Augenblick kleinsten Elemente der beiden Vektoren miteinander ver-

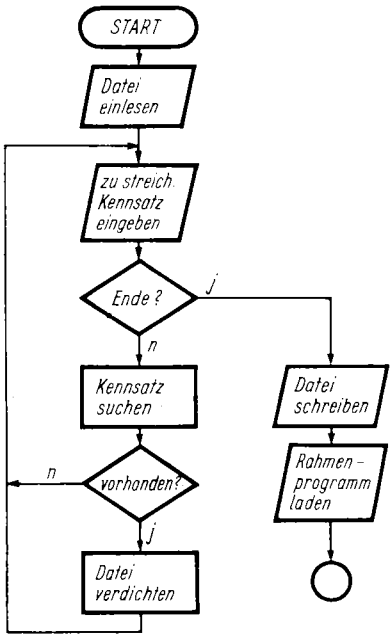


Bild 9.4. Streichen von Kennsätzen in einer Datei

glichen werden und das kleinere von beiden in den Zielvektor übernommen wird. Ist einer von beiden Vektoren bereits aufgebraucht, so wird pro forma das letzte Element des anderen Vektors auch bei ihm als letztes Element eingetragen (Zeile 1470 bzw. 1570). Dadurch läuft zunächst der Vergleichsalgorithmus einwandfrei weiter. Am Ende fällt dieses zusätzlich aufgenommene Element wieder weg, weil die Mischung durch Abzählen beendet wird (Zeile 1380).

In dem folgenden Programmsystem wird das Modell einer kleinen Datenbank am Beispiel eines Telefonbuchs demonstriert. Das **Programm 9.5** fragt nach den Wünschen des Bedieners. Zunächst ist es möglich, neue Paare *Name–Nummer* in die Datei aufzunehmen **(Programm 9.6)** oder solche Kennsätze zu streichen **(Programm 9.7, Bild 9.4)**. Dann kann man Anfragen an den

Programm 9.7. Streichen von Kennsätzen

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"            STREICHEN VON KENNSAETZEN            "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Dateiname " ; D¤
160 :
170 REM -------------------- KOPIEREN --------------------
180 :
190 OPEN "I", #1, D¤ + ".DAT"
200 INPUT #1, N : REM Anzahl vorhandener Kennsaetze
210 DIM V¤(N) , V(N)
220 FOR I=1 TO N
230   INPUT #1, V¤(I), V(I)
240 NEXT I
250 CLOSE #1
260 PRINT "Datei "; D¤ ;" eingelesen."
270 :
280 REM -------------------- STREICHEN --------------------
290 :
300 INPUT ;"Zu streichender Kennsatz (Name,Nummer)"; S¤, S
310 IF S¤="@" THEN PRINT : GOTO 520
320 :
330 LET A = 1 : REM Anfang des Intervalls
340 LET E = N+1 : REM Ende des Intervalls
350 :
360 LET M = INT((A+E)/2) : REM Mitte des Intervalls
370 LET B = V¤(M)=S¤ : REM gesuchter Name gefunden
380 IF B AND V(M)=S THEN GOTO 430 : REM Streichen
390   IF A=M THEN PRINT " ist nicht vorhanden!" : GOTO 300
400     IF V¤(M)<S¤ OR (B AND V(M)<S) THEN A=M ELSE E=M
410     GOTO 360 : REM naechster Schritt
420 :
430 FOR I=M+1 TO N
440   LET V¤(I-1) = V¤(I) : LET V(I-1) = V(I)
450 NEXT I
460 LET N = N-1
470 PRINT " wurde gestrichen."
480 GOTO 300
490 :
500 REM ------------------ ABSPEICHERN ------------------
510 :
520 PRINT "Ende der Korrektur."
530 OPEN "O", #2, D¤ + ".DAT"
540 PRINT #2, N
550 FOR I=1 TO N
560   PRINT #2, V¤(I) , "," , V(I)
570 NEXT I
580 CLOSE #2
590 PRINT "Korrigierte Datei "; D¤ ;" gespeichert."
600 PRINT
610 LOAD "P0905" , R : REM Rahmenprogramm wieder laden
620 :
630 REM ==================================================
```

Programm 9.8. Suchen von Kennsätzen

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"              SUCHEN VON KENNSAETZEN             "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Dateiname" ; D¤
160 :
170 REM -------------------- KOPIEREN --------------------
180 :
190 OPEN "I", #1, D¤ + ".DAT"
200 INPUT #1, N : REM Anzahl vorhandener Kennsaetze
210 DIM V¤(N) , V(N)
220 FOR I=1 TO N
230   INPUT #1, V¤(I) , V(I)
240 NEXT I
250 CLOSE #1
260 PRINT "Datei "; D¤ " eingelesen."
270 ;
280 REM --------------------- SUCHEN ---------------------
290 :
300 INPUT ; "Zu suchender Name" ; S¤
310 IF S¤<>"@" THEN GOTO 350
320   PRINT : PRINT
330   LOAD "P0905" , R : REM Rahmenprogramm wieder laden
340 :
350 LET P = -1
360 FOR I=1 TO N
370   IF V¤(I)<>S¤ THEN GOTO 400 : REM NEXT I
380     IF P THEN PRINT "  -> Nummer: " ; : LET P = 0
390     PRINT V(I) ;
400 NEXT I
410 IF P THEN PRINT " ist nicht vorhanden!" ELSE PRINT
420 GOTO 300
430 :
440 REM ==================================================
```

Computer richten, um zu einem gegebenen Namen die zugehörige Nummer zu suchen (**Programm 9.8**). Schließlich läßt sich auch die gesamte Datei ausdrucken (**Programm 9.9**). Das für den gewünschten Service erforderliche Programm wird vom Programm 9.5 geladen und gestartet; am Ende wird das Programm 9.5 wieder in den Hauptspeicher gebracht und fragt erneut nach den Wünschen des Bedieners.

Die Programme 9.4 bis 9.9 arbeiten mit Datenfeldern, in die die Dateien vor der Verarbeitung eingelesen werden. Dadurch wird zwar die Laufzeit verkürzt, aber der Speicherbedarf ist – entsprechend dem Umfang der Dateien – recht hoch. Falls der vorhandene Arbeitsspeicher dazu nicht ausreicht, muß zu einer sequentiellen Verarbeitung übergegangen werden. Dabei wird Datensatz für Datensatz vom externen Speicher gelesen, bearbeitet und – mit einer geeigneten Modifikation des Bezeichners – sofort wieder auf den externen Speicher zurückgeschrieben.

Kassettenorientierte BASIC-Systeme bieten häufig nur die Möglichkeit, die ausgelagerten Werte für ein konkretes Datenfeld wieder einzulesen:

n **CLOAD*** *programmname* ; *feldname*

Der Programmname ist als Zeichenkette anzugeben, der Feldname wie eine Variable.

Programm 9.9. Ausdrucken der Kennsatzdatei

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"          AUSDRUCKEN VON KENNSAETZEN           "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Dateiname" ; D¤
160 :
170 REM -------------------- KOPIEREN --------------------
180 :
190 OPEN "I", #1, D¤ + ".DAT"
200 INPUT #1, N : REM Anzahl vorhandener Kennsaetze
210 DIM V¤(N) , V(N)
220 FOR I=1 TO N
230   INPUT #1, V¤(I) , V(I)
240 NEXT I
250 CLOSE #1
260 PRINT "Datei "; D¤ ;" eingelesen."
270 :
280 REM ------------------ AUSDRUCKEN ------------------
290 :
300 INPUT "Zeilenanzahl je Seite" ; Z
310 LPRINT "Datei "; D¤ ;":"
320 LPRINT
330 LPRINT "Name" ; TAB(21) ; "Nummer" ;
340 LPRINT TAB(30) ; "|" ; TAB(35) ;
350 LPRINT "Name" ; TAB(55) ; "Nummer"
360 LPRINT TAB(30) ; "|"
370 LET M¤ = "\          \ #####"
380 LET U = INT(N/2)<>N/2 : REM ungerade Anzahl
390 LET K = 0 : REM Anfangswert fuer Zeilennummer
400 FOR I=1 TO INT(N/2)
410   GOSUB 550 : REM linke Haelfte drucken
420   LET J = I + INT(N/2) - U
430   LPRINT USING M¤ ; V¤(J) , V(J)
440   LET K = K+1
450   IF K<Z THEN GOTO 500 : REM NEXT I
460     LET K = 0
470     PRINT
480     PRINT "Formularwechsel! - fertig"
490     INPUT S
500 NEXT I
510 IF U THEN GOSUB 550 : REM linke Haelfte drucken
520 LPRINT
530 LOAD "P0905" , R : REM Rahmenprogramm wieder laden
540 :
550 LPRINT USING M¤ ; V¤(I) , V(I) ;
560 LPRINT TAB(30) ; "|" ; TAB(35) ;
570 RETURN
580 :
590 REM ==================================================
```

```
Datei TELEFON:

Name                Nummer  |   Name                Nummer
                            |
Boehme               7581   |   Richter              4991
Fischer              2421   |   Schmidt             17274
Koehler              6036   |   Schroeter            5468
Lehmann             16358   |   Schulze              8888
Muelder              3877   |   Wolf                18381
Mueller              4788   |
```

10. Hardwareorientierte BASIC-Elemente

Bereits einleitend wurde erwähnt, daß BASIC seine weltweite Verbreitung den Mikrocomputern verdankt. Durch diese Klasse von Rechenautomaten wurde aber auch der Inhalt der Sprachdialekte beeinflußt. So entstanden einige charakteristische Elemente, die einen direkten Zugriff zur Hardware gestatten:

- Lesen und Beschreiben von *numerisch* adressierten Bytes des Hauptspeichers
- Einsatz von Unterprogrammen, die im *Maschinenkode* vorliegen
- Eingaben und Ausgaben über *numerisch* adressierte Kanäle (*Ports*).

Durch diese Ergänzungen rückte BASIC in die Reihe der *Systemprogrammiersprachen* auf. Es wurde möglich, diese Sprache für Aufgaben der *Prozeßsteuerung* einzusetzen. Dabei liegt ihre Stärke eindeutig darin, daß die interpretative Abarbeitung ein einfaches, experimentelles Arbeiten gestattet. Die Echtzeitanforderungen dagegen dürfen nicht allzu kritisch sein, weil die Abarbeitungsdauer jeder einzelnen Anweisung im Bereich von Millisekunden liegt.

10.1. Informationsdarstellung

Zum Verständnis der folgenden Abschnitte ist es erforderlich, noch einige Einzelheiten zur Darstellung von Informationen in Rechenautomaten zu behandeln. Im Abschnitt 1.1.4 wurde bereits von Bit und Byte gesprochen. Jetzt sollen Ergänzungen zur binären Kodierung von Zahlen und Adressen folgen.

10.1.1. Zahlen

Natürliche Zahlen werden in einem binären Stellenwertsystem ausgedrückt **(Bild 10.1)**. Die Werte der *Bits* $b_7 \ldots b_0$ können dabei nur 0 oder 1 sein; der Wert eines *Bytes* beträgt

$$b_7 \cdot 2^7 + b_6 \cdot 2^6 + b_5 \cdot 2^5 + b_4 \cdot 2^4 + b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

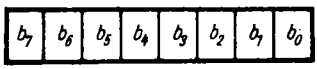


Bild 10.1. Binäre Beschreibung des Wertes eines Bytes

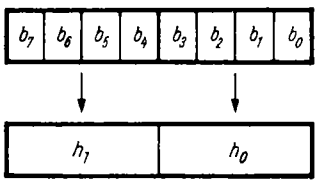


Bild 10.2. Hexadezimale Beschreibung des Wertes eines Bytes

Er liegt zwischen 0 und 255. Die Angabe des Wertes einer Binärzahl durch die Bit $b_7 \ldots b_0$ ist aber umständlich. Bei Mikrorechnern hat es sich daher eingebürgert, jeweils vier Bit zu einer *Tetrade* (*nibble*) zusammenzufassen **(Bild 10.2)**. Hier umfaßt der Wertebereich 0 ... 15; in jede Tetrade paßt gerade eine *Hexadezimalziffer*. Den Inhalt eines Bytes kann man demgemäß durch eine Hexadezimalzahl ausdrücken, die aus zwei Hexadezimalziffern h_1 und h_0 besteht. Um sie

von Dezimalziffern bzw. -zahlen zu unterscheiden, werden in BASIC die Zeichen & (als Hinweis auf eine besondere Darstellungsform für Zahlen) und H (für Hexadezimalzahl) vorgesetzt:

$\&H h_1 h_0 = h_1 \cdot 16^1 + h_0 \cdot 16^0$

Um jede Hexadezimalziffer durch genau ein Zeichen ausdrücken zu können, werden für die Ziffern 10 bis 15 die Buchstabenzeichen A bis F benutzt **(Tafel 10.1)**:

Tafel 10.1. Hexadezimalziffern

Hexadezimalziffer	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Dezimalwert	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

&HC9 = 201
&HF800 = 63488

Eine Umrechnung zwischen beiden Darstellungsformen für natürliche Zahlen wird durch das **Programm 10.1** realisiert.

Dezimale und hexadezimale Zahlen sind nur verschiedene Schreibweisen für dieselbe Zahl. Dezimalzahlen sind vom täglichen Leben her geläufig. Hexadezimalzahlen sind aber der Darstellung im Rechenautomaten besser angepaßt, sie spiegeln die interne Struktur der Binärzahlen wider.

Bisher wurde nur über die Repräsentation von natürlichen Zahlen im Computer gesprochen. Darüber hinaus gibt es noch zahlreiche weitere Zahlentypen. In diesem Buch sollen nur noch die *vorzeichenbehafteten* ganzen Zahlen von *zwei* Byte Länge behandelt werden. Bei ihnen wird für *negative* Werte die sog. *Zweierkomplementdarstellung* verwendet.
Man erhält sie auf folgendem Wege **(Bild 10.3)**:

- Zunächst wird der *Betrag* der Zahl binär kodiert (Bild 10.3a).
- Dann wird das Bitmuster *negiert* (Bild 10.3b).
- Abschließend wird *eine Eins addiert* (Bild 10.3c).

Dieses Verfahren hat zur Folge, daß bei negativen Zahlen stets das höchstwertige Bit gesetzt ist.

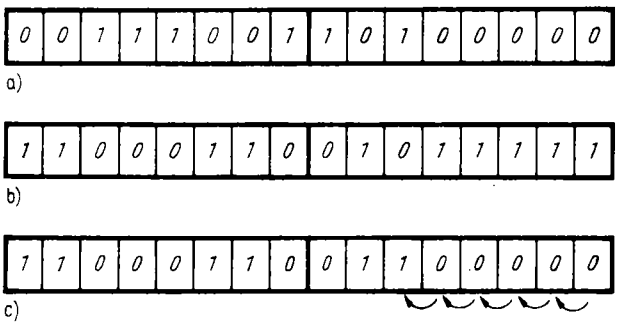


Bild 10.3. Zweierkomplementdarstellung der Zahl −14752
a) binäre Kodierung des Betrags
b) Negation
c) Addition von 1 (Überträge beachten!)

Tafel 10.2. Lage der negativen Dezimalzahlen im Zahlenraum des Computers

Beschreibungsform	Vorderes Bit = 0	Vorderes Bit = 1
Binäre Zahlen	00000000 ... 01111111	10000000 ... 11111111
Hexadezimale Zahlen	0000 ... 7FFF	8000 ... FFFF
Natürliche Zahlen	0 ... 32767	32768 ... 65535
Dezimalzahlen (mit Vorzeichen)	0 ... 32767	−32768 ... −1

Programm 10.1. Konvertierung zwischen der dezimalen und der hexadezimalen Zahlendarstellung

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"    KONVERTIERUNG: DEZIMAL <-> HEXADEZIMAL     "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM ----------------------- ABFRAGE ---------------------
150 :
160 DEF FNK¤(X¤) = CHR¤(ASC(X¤) - 32*(X¤>="a")*(X¤<="z"))
170 PRINT
180 PRINT "dezimal -> hexadezimal ..... D"
190 PRINT "hexadezimal -> dezimal ..... H"
200 PRINT "?"
210 LET A¤ = INPUT¤(1)
220 IF A¤=CHR¤(13) THEN PRINT : END
230 :
240 IF A¤="D" OR A¤="d" THEN GOSUB 300
250   IF A¤="H" OR A¤="h" THEN GOSUB 450
260 GOTO 200 : REM Wiederholung
270 :
280 REM ------------ DEZIMAL -> HEXADEZIMAL -------------
290 :
300 INPUT "Dezimalzahl (1...65535)" ; D
310 LET N = INT(ABS(D))
320 :
330 LET H¤ = "" : REM Anfangswert Hexadezimalzahl
340 LET I = INT(N/16) : REM ganzzahliger Anteil
350 LET R = N - 16*I  : REM Rest
360 IF R<=9 THEN LET C = 48 ELSE LET C = 55
370 LET H¤ = CHR¤(R+C) + H¤
380 IF I=0 THEN PRINT "Hexadezimalzahl = " ; H¤ : RETURN
390 :
400 LET N = I
410 GOTO 340 : REM naechste Hexadezimalstelle
420 :
430 REM ------------ HEXADEZIMAL -> DEZIMAL -------------
440 :
450 INPUT "Hexadezimalzahl (1...FFFF)" ; H¤
460 FOR P=1 TO LEN(H¤)
470   LET H¤=LEFT¤(H¤,P-1)+FNK¤(MID¤(H¤,P,1))+MID¤(H¤,P+1)
480 NEXT P
490 LET L = LEN(H¤)
500 LET N = 0 : REM Anfangswert Dezimalzahl
510 LET P = 0 : REM Position Hexaziffer
520 :
530 LET P = P+1
540 LET Z = ASC(MID¤(H¤,P,1))
550 IF Z>=48 AND Z<=57 THEN LET C = 48 : GOTO 600
560   IF Z>=65 AND Z<=70 THEN LET C = 55 : GOTO 600
570     PRINT "keine Hexadezimalzahl!"
580     GOTO 450 : REM -> Eingabe
590 :
600 LET N = N+Z-C : REM aufgelaufene Dezimalzahl
610 IF P>=L THEN PRINT "Dezimalzahl =" ; N : RETURN
620 LET N = 16*N
630 GOTO 530 : REM naechste Hexaziffer
640 :
650 REM ================================================
```

Der Wertebereich der positiven Zahlen umfaßt

0 ... 32767
&H0000 ... &H7FFF

Die negativen Zahlen betragen

-32768 ... -1
&H8000 ... &HFFFF

Tafel 10.2 zeigt die Lage der negativen Zahlen im Wertevorrat der Binärzahlen.

Auf die interne Darstellung weiterer Zahlentypen soll in diesem Buch nicht eingegangen werden. Der interessierte Leser wird daher auf entsprechende Spezialliteratur verwiesen [10.2].

10.1.2. Adressen

Die Zellen des *Hauptspeichers* eines Mikrorechners haben häufig die Größe eines Bytes. Alle diese Zellen sind fortlaufend *numeriert*, die entsprechende natürliche Zahl heißt *Speicheradresse*. Der mögliche Maximalumfang des Hauptspeichers wird durch die Länge dieser Adresse bestimmt. Hat die Adresse eine Größe von zwei Byte, so lassen sich 2^{16} Speicherzellen unmittelbar erreichen, der Adreßumfang ist

0 ... 65535
&H0000 ... &HFFFF

Viele Mikrorechner gestatten es aber nicht, positive ganze Dezimalzahlen über 32767 anzugeben. Hier muß man die Zweierkomplementdarstellung negativer Zahlen beachten; Adressen über 32767 sind für den Computer *negativ*. Die Umrechnung für solche Werte erfolgt ganz einfach mit Hilfe der Beziehung

anzugebende Zahl = *gewünschte Adresse* − 65536

die man anschaulich aus der Tafel 10.2 ableiten kann.

Auch die *Eingabe/Ausgabe-Kanäle* (Abschn. 10.4) haben *numerische Adressen*. Bei dem benutzten Mikroprozessor haben sie eine Länge von einem Byte, die Werte liegen also im Bereich

0 ... 255
&H00 ... &HFF

10.2. Bitverarbeitung

Über die logischen Aussagen *wahr* (TRUE) und *falsch* (FALSE) und ihre Verbindung durch logische Operationen, wie AND, OR und XOR, wurde bereits im Abschnitt 5.3 diskutiert. Zur Darstellung des Wertes einer Aussage wurde dort eine *Zahl* verwendet. Falls der Zahlentyp INTEGER im BASIC-Interpreter implementiert ist, wären dafür also mindestens zwei Byte erforderlich, anderenfalls noch mehr. Das ist aber eigentlich viel zuviel Speicherplatz! Für die Repräsentation einer Wahrheitsaussage bietet sich ja geradezu das *Bit* an, das ebenfalls genau zwei mögliche Werte hat:

Aussage = TRUE → Bit = 1
Aussage = FALSE → Bit = 0

Im folgenden soll nun über die Verarbeitung von Bitmustern gesprochen werden, bei denen *jede einzelne* Position eine selbständige logische Aussage repräsentiert.

10.2.1. Logische Bitmuster

Wie bereits erwähnt, wurde bisher zur Darstellung einer logischen Aussage eine Zahl mit (mindestens) 16 Bit verwendet. Verschiedene Interpreter benutzen dabei nur das niederwertigste Bit und kennzeichnen den Wert TRUE durch den Zahlenwert +1, FALSE durch 0. Andere Interpreter verwenden dagegen *alle* Bits. Besteht eine solche interne Binärzahl nur aus Einsen, so ist ihr Dezimalwert unter Beachtung der Zweierkomplementdarstellung gleich −1. Diese Eigenschaft wurde im vorliegenden Buch bereits häufig benutzt. Enthält die Zahl andererseits nur Nullen, so ist ihr Wert 0.

Diejenigen Interpreter, die genau diese Darstellung für logische Werte benutzen, realisieren die logischen Operationen in der Regel *bitweise*. Das bedeutet, daß dabei jeweils alle *gleichstelligen* Bits der beiden Operanden entsprechend dem angegebenen Operator miteinander verknüpft werden. Dieser Zusammenhang ist im **Bild 10.4** dargestellt. In analoger Weise wird bei der einstelligen Operation NOT verfahren; das heißt, es wird jedes einzelne Bit negiert.

Die Vorteile dieser Arbeitsweise kommen natürlich nicht zum Tragen, wenn die Operanden aus Vergleichen entstanden sind, weil in diesen Fällen stets alle einzelnen Bits denselben Wert haben. Anders liegen die Verhältnisse, wenn allgemeine Bitmuster verarbeitet werden sollen, die z. B. beim Rechnereinsatz für Regelungsaufgaben vorhanden sein und einen komplexen Zustand widerspiegeln könnten. Hier lassen sich mit BASIC Operationen ausführen, für die sonst die Maschinensprache mit ihren entsprechenden Befehlen zur Bitmanipulation eingesetzt werden müßte.

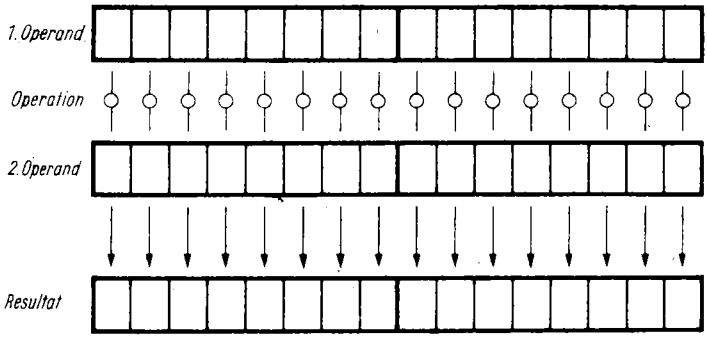


Bild 10.4. Bitweise Ausführung zweistelliger logischer Operationen

10.2.2. Logische Operationen

Bei der Bitverarbeitung werden sog. *Masken* benutzt, das sind Bitmuster, bei denen bestimmte, im konkreten Zusammenhang interessierende Bits auf den Wert 1 gesetzt sind. **Bild 10.5** zeigt ein Beispiel; der Einfachheit halber wurde nur ein einziges Bit gesetzt. Außerdem werden im folgenden aus demselben Grund stets nur Bitmuster von *einem* Byte Länge betrachtet.

7	6	5	4	3	2	1	0
0	0	1	0	0	0	0	0

Bild 10.5. Beispiele für eine Testmaske zur Bitverarbeitung

Verknüpft man eine solche Maske M unter Benutzung des logischen Operators **AND** mit einem zu untersuchenden Bitmuster B, so *löschen* die Nullen der Maske alle entsprechenden Stellen des Musters; nur die in der Maske gesetzten Positionen bleiben unverändert erhalten. Man *schneidet* also die markierten Stellen *heraus* und kann sie weiter untersuchen. So soll folgendes Beispiel die Maske M nach Bild 10.5 benutzen:

```
280 LET M = &H20
290 IF B AND M THEN GOTO 500
```

Der Sprung wird nur ausgeführt, falls im Bitmuster B das Bit 5 gesetzt ist. Dabei ist es ausreichend, wenn mindestens *eine* der Bitpositionen sowohl in B als auch in M gesetzt ist. Dagegen ergibt der Vergleich

```
(B AND M) = M
```

nur dann den Wert *wahr*, wenn alle in M gesetzten Bits auch in B den Wert 1 aufweisen.

Der Operator AND wird häufig eingesetzt, um bestimmte Bits zu löschen. So liefern manche Eingabegeräte (z. B. Lochbandleser) Textzeichen, bei denen das siebente Bit für Prüfzwecke eingesetzt ist. Die Anweisung

```
420 LET C¤ = STR¤(B AND &H7F)
```

löscht dieses Bit und erzeugt ein „sauberes" ASCII-Zeichen. Im Gegensatz dazu dient der logische Operator **OR** dazu, in einem Muster Bits auf 1 zu *setzen*. Dabei wird die Eigenschaft von OR verwendet, daß im Resultat jedes Bit gesetzt ist, das in mindestens einem der beiden Ope-

Programm 10.2. Logische Operationen

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"              LOGISCHE OPERATIONEN                "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++++ ABFRAGE +++++++++++++++++++++
1080 :
1090 DEF FNK¤(X¤) = CHR¤(ASC(X¤) - 32*(X¤>="a")*(X¤<="z"))
1100 INPUT "gewuenschte Operation" ; O¤
1110 FOR P=1 TO LEN(O¤)
1120   LET O¤=LEFT¤(O¤,P-1)+FNK¤(MID¤(O¤,P,1))+MID¤(O¤,P+1)
1130 NEXT P
1140 PRINT
1150 IF O¤="AND" THEN GOTO 1280
1160   IF O¤="NOT" THEN GOTO 1350
1170     IF O¤="OR" THEN GOTO 1440
1180       IF O¤="XOR" THEN GOTO 1490
1190         IF O¤="SHIFT" THEN GOTO 1580
1200           IF O¤="" THEN PRINT : END
1210             PRINT "AND, NOT, OR , SHIFT oder XOR!"
1220             GOTO 1030
1230 :
1240 REM +++++++++++++++++ OPERATIONEN +++++++++++++++++++
1250 :
1260 REM ---------------------- AND -----------------------
1270 :
1280 GOSUB 1720 : REM Eingabe Operanden X,Y
1290 LET R = X AND Y
1300 GOSUB 1980 : REM Ausgabe Resultat R
1310 GOTO 1030
1320 :
1330 REM ---------------------- NOT -----------------------
1340 :
1350 PRINT "Binaeroperand? " ;
1360 GOSUB 1830 : REM Eingabe Operand B
1370 LET R = NOT B
1380 PRINT
1390 GOSUB 2110 : REM Ausgabe Resultat R
1400 GOTO 1030
1410 :
1420 REM ---------------------- OR ------------------------
1430 :
1440 GOSUB 1720 : REM Eingabe Operanden X,Y
1450 LET R = X OR Y
1460 GOSUB 1980 : REM Ausgabe Resultat R
1470 GOTO 1030
1480 :
1490 REM ---------------------- XOR -----------------------
1500 :
1510 GOSUB 1720 : REM Eingabe Operanden X,Y
1520 LET R = X XOR Y
1530 GOSUB 1980 : REM Ausgabe Resultat R
1540 GOTO 1030
1550 :
1560 REM --------------------- SHIFT ----------------------
1570 :
1580 PRINT "Binaeroperand? " ;
1590 GOSUB 1830 : REM Eingabe Operand B
1600 INPUT "Richtung der Verschiebung (R | L)" ; A¤
1610 INPUT "Anzahl der Bitstellen" ; N
1620 IF A¤="R" OR A¤="r" THEN LET N = -N
1630 LET R = INT(B*2^N) AND &HFF : REM Verschiebung
1640 PRINT
1650 GOSUB 2110 : REM Ausgabe Resultat R
1660 GOTO 1030
1670 :
```

```
1680 REM ++++++++ EINGABE-/AUSGABE-UNTERPROGRAMME +++++++++
1690 :
1700 REM ---------- EINGABE ZWEIER BINAERZAHLEN -----------
1710 :
1720 PRINT "1. Binaeroperand? " ;
1730 GOSUB 1830 : REM Eingabe B
1740 LET X = B
1750 PRINT "2. Binaeroperand? " ;
1760 GOSUB 1830 : REM Eingabe B
1770 LET Y = B
1780 PRINT
1790 RETURN
1800 :
1810 REM ------------ EINGABE EINER BINAERZAHL ------------
1820 :
1830 LET B¤ = "" : REM Anfangswert Binaerzahl
1840 FOR N=0 TO 7
1850   LET C¤ = INPUT¤(1)
1860   IF C¤<>"0" AND C¤<>"1" THEN GOTO 1900
1870   PRINT C¤ ;
1880   LET B¤ = B¤ + C¤
1890 NEXT N
1900 PRINT
1910 LET B = 0 : REM Anfangswert Dezimalzahl
1920 FOR P=1 TO N
1930   LET B = 2*B + VAL(MID¤(B¤,P,1))
1940 NEXT P
1950 RETURN
1960 :
1970 REM ----------- AUSGABE BEI ZWEI OPERANDEN -----------
1980 LET B = X
1990 GOSUB 2240 : REM Ausgabe B
2000 PRINT " " ; O¤ ; " " ;
2010 LET B = Y
2020 GOSUB 2240 : REM Ausgabe B
2030 PRINT "  =  " ;
2040 LET B = R
2050 GOSUB 2240 : REM Ausgabe B
2060 PRINT
2070 RETURN
2080 :
2090 REM ---------- AUSGABE BEI EINEM OPERANDEN ----------
2100 :
2110 PRINT O¤ ; " " ;
2120 GOSUB 2240 : REM Ausgabe B
2130 IF O¤<>"SHIFT" THEN GOTO 2160
2140   IF N<0 THEN PRINT " RIGHT" ; ELSE PRINT " LEFT" ;
2150   PRINT STR¤(ABS(N)) ;
2160 PRINT " = " ;
2170 LET B = R
2180 GOSUB 2240 : REM Ausgabe B
2190 PRINT
2200 RETURN
2210 :
2220 REM ------------ AUSGABE EINER BINAERZAHL ------------
2230 :
2240 LET B¤ = "" : REM Anfangswert Binaerzahl
2250 LET M = 1 : REM Maske
2260 FOR I=1 TO 8
2270   LET B¤ = RIGHT¤(STR¤(-((B AND M)<>0)),1) + B¤
2280   LET M = 2*M
2290 NEXT I
2300 PRINT B¤ ;
2310 RETURN
2320 :
2330 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

randen den Wert 1 hat. Möchte man also in einem Muster bestimmte Bits setzen, so benutzt man eine Maske, in der genau diese Stellen den Wert 1 haben.

```
370 LET B = B OR &H80
```

setzt z. B. das Bit 7.

Das exklusive Oder **XOR** hingegen ergibt nur dann das Resultat 1, wenn die entsprechenden Bits der beiden Operanden unterschiedliche Werte aufweisen. Das kann dazu dienen, ganz bestimmte, durch eine Maske ausgewählte Bits eines Musters zu *negieren*. Dazu kann wieder eine Maske nach Bild 10.5 betrachtet werden:

```
610 LET B = B XOR M
```

- Hat eine Bitstelle der Maske den Wert 0, so ergibt sich im Resultat
  - der Wert 1, wenn im Muster das entsprechende Bit gesetzt ist (weil sich die Stellen in Maske und Muster unterscheiden!)
  - der Wert 0, wenn im Muster das Bit gelöscht ist (weil Maske und Muster dann denselben Wert haben!)

  In diesem Fall wird also der Wert des Musters *reproduziert*.
- Hat eine Bitstelle der Maske dagegen den Wert 1, so erhält man als Resultat
  - den Wert 0, wenn das betreffende Bit im Muster gesetzt ist (gleiche Werte!)
  - den Wert 1, wenn das Bit im Muster gelöscht ist (unterschiedliche Werte!)

  Hier wird demnach der Bitwert des Musters *negiert*.

Während also die Anweisung

```
940 LET B = NOT B
```

alle Bits negiert, bleiben beispielsweise bei

```
940 LET B = B XOR &H80
```

die Werte der Stellen 0 bis 6 erhalten, nur das siebente Bit wird negiert.

Um sich die Wirkung dieser bitweisen logischen Operationen anschaulich klar zu machen, wurde das **Programm 10.2** geschaffen. Der Leser kann beliebige Bitmuster eingeben und sie mit Hilfe der beschriebenen Operationen verknüpfen. Dabei sind die eingegebenen Werte der Bitstellen Zeichenketten aus den Textzeichen 0 und 1, die zunächst in die interne, binäre Darstellung verwandelt werden müssen, wobei jedes Zeichen den Wert des zugeordneten Bits bestimmt. Entsprechend erfolgt am Ende die Umwandlung aus einem internen Byte in eine Zeichenkette aus acht Zeichen 0 bzw. 1.

10.2.3. Verschiebungen

Im Programm 10.2 wird noch eine weitere Möglichkeit zur Bitverarbeitung demonstriert, die logische Verschiebung von Bitmustern nach links oder rechts. Dabei gehen die herausfallenden Bits verloren; für sie werden von der anderen Seite her Nullen nachgeschoben **(Bild 10.6)**. Da BASIC dafür keine Operationen bereitstellt, wird die Verschiebung durch eine Multiplikation bzw. Division mit entsprechenden Potenzen von 2 erreicht.

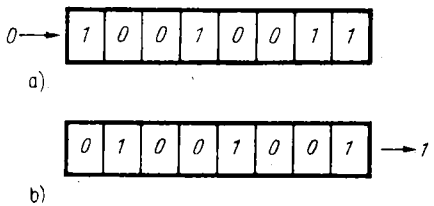


Bild 10.6. Logische Verschiebung eines Bitmusters um eine Stelle nach rechts

10.3. Lesen und Schreiben im Hauptspeicher

Eine höhere Programmiersprache verbirgt die konkrete Nutzung des Hauptspeichers vor dem Programmierer. Dieser kennt nur die *symbolischen Namen* von Speicherzellen, in denen die Werte von verwendeten Variablen abgelegt sind, nicht aber deren *numerische Adressen*. Andererseits erfordern typische Einsatzfälle von Mikrorechnern häufig gerade den Zugriff zu bestimmten numerisch adressierten Zellen. Solche Möglichkeiten bieten jedoch nur maschinenorientierte Sprachen, wie Assemblersprachen und Systemprogrammiersprachen. Moderne BASIC-Dialekte haben für diese Aufgaben mehrere Anweisungen und Funktionen.

10.3.1. Lesen im Speicher

PEEK(*adresse*)

Diese Funktion liefert den *Inhalt* derjenigen Speicherzelle (von einem Byte Größe), die durch den angegebenen arithmetischen Ausdruck adressiert ist, der vom Interpreter erforderlichenfalls gerundet wird. Das betreffende Bitmuster wird dabei als natürliche Zahl aufgefaßt. Wie jede andere arithmetische Funktion kann auch die PEEK-Funktion in arithmetischen Ausdrücken verwendet werden.

Programm 10.3. Ausgabe eines Hauptspeicherbereichs

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"        AUSGABE EINES SPEICHERBEREICHS          "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM ------------------- ABFRAGE ---------------------
150 :
160 PRINT
170 INPUT "Anfangsadresse" ; A
180 INPUT "Endadresse" ; E
190 INPUT "Ausgabe dezimal (D) oder als Text (T)" ; A¤
200 PRINT
210 PRINT USING "#####   " ; A ;
220 IF A¤="T" OR A¤="t" THEN GOTO 380
230 :
240 REM ---------------- DEZIMALE AUSGABE ----------------
250 :
260 LET N = A
270 PRINT USING "###  " ; PEEK(N) ;
280 LET N = N+1
290 IF N>E THEN GOTO 480
300 :
310 IF INT(N/10)<>N/10 THEN GOTO 270
320   PRINT
330   PRINT USING "#####   " ; N ;
340   GOTO 270
350 :
360 REM ------------------ TEXTAUSGABE -------------------
370 :
380 LET N = A
390 LET Z = PEEK(N)
400 IF Z<32 OR Z>126 THEN PRINT "_" ; ELSE PRINT CHR¤(Z) ;
410 LET N = N+1
420 IF N>E THEN GOTO 480
430 :
440 IF INT(N/50)<>N/50 THEN GOTO 390
450   PRINT
460   PRINT USING "#####   " ; N ;
470   GOTO 390
480 PRINT
490 :
500 END
510 REM ==================================================
```

In der Anweisung

110 LET Z = PEEK (&HF800)

wird der Variablen Z der Zahlenwert (0 . . . 255) desjenigen Bitmusters zugewiesen, das in der Speicherzelle mit der hexadezimalen Adresse F800 gespeichert ist.

Eine Anwendung zeigt das **Programm 10.3.** Es gibt den Inhalt eines Hauptspeicherbereichs auf den Bildschirm aus, und zwar wahlweise als Dezimalzahlen oder im ASCII-Kode. Nicht druckbare Zeichen werden dabei durch Unterstreichungszeichen ersetzt.

Manche BASIC-Systeme enthalten zusätzlich

DEEK(*adresse*)

Diese Funktion liest im Unterschied zu PEEK *zwei* Byte aus dem Hauptspeicher ein (Double pEEK), wobei die angegebene und die nächsthöhere Adresse benutzt werden. Sie läßt sich dann günstig einsetzen, wenn eine ganzzahlige Variable von zwei Byte Länge gelesen werden soll, z. B. eine numerische Adresse.

10.3.2. Schreiben in den Speicher

Weiterhin bietet BASIC die Sprachanweisung

n **POKE** *adresse* , *wert*

Der Interpreter *schreibt* hier den angegebenen Wert in die Speicherzelle mit der spezifizierten Adresse. Dabei wird nur der Inhalt des niederwertigen Bytes übertragen. Der Wert muß also unter 255 liegen, wenn ein Fehler vermieden werden soll. Sowohl für die Adresse als auch für den Wert sind arithmetische Ausdrücke einzusetzen, die Resultate werden erforderlichenfalls vom Interpreter gerundet.

Einige BASIC-Dialekte bieten darüber hinaus die Anweisung

n **DOKE** *adresse* , *wert*

Im Unterschied zur POKE-Anweisung überträgt der Interpreter hier jeweils *zwei* Byte (Double pOKE). Dabei werden wie bei der DEEK-Funktion die angegebene und die darauf folgende Speicheradresse benutzt.

Die POKE- und die DOKE-Anweisung können beispielsweise dazu eingesetzt werden, unter Umgehung der entsprechenden Gerätebedienungsroutine Direktausgaben auf dem Bildschirm zu realisieren. Dabei wird die bekannte Tatsache benutzt, daß ein Schirmbild ständig wiederholt geschrieben werden muß (Abschn. 1.3.2). Der *Bildwiederholspeicher* ist bei den meisten Mikrorechnern ein Teil des Hauptspeichers, läßt sich also mit POKE-Anweisungen direkt erreichen. Er liegt meist am oberen Ende des Speichers, seine genaue Lage ist der Bedienungsanleitung zu

Programm 10.4. Direktzugriff zum Bildwiederholspeicher

```
100 REM <<<< DIREKTZUGRIFF ZUM BILDWIEDERHOLSPEICHER >>>>>
110 :
120 REM Bildschirm: 24 Zeilen * 80 Spalten
130 REM Speicheranfangsadresse = 63488 = -2048
140 :
150 INPUT "Anfangsposition: Zeile,Spalte" ; Z , S
160 LET A = -2129 + 80*Z + S
170 PRINT "Text fortlaufend eingeben. " ;
180 PRINT "Abschluss durch Zeilenendetaste."
190 PRINT CHR$(12) : REM Bildschirm loeschen
200 :
210 LET Z = ASC(INPUT$(1))
220 IF Z=13 THEN PRINT CHR$(12) : END
230 :
240 POKE A , Z
250 LET A = A+1
260 GOTO 210
270 :
280 REM ==========================================
```

entnehmen. Sie läßt sich auch durch Probieren ermitteln; allerdings riskiert man dabei, Informationen des Betriebssystems oder des Interpreters zu zerstören.

Im Bildwiederholspeicher stehen die auszugebenden Zeichen zeilenweise geordnet. Das wird vom **Programm 10.4** benutzt, um eine durch die Angabe von Zeile und Spalte spezifizierte Position des Bildschirms zu berechnen (Zählung ab Zeile 1/Spalte 1). Von dort an wird ein fortlaufend eingegebener Text, der durch die Zeilenendetaste (ASCII-Kode 13) abzuschließen ist, in den Hauptspeicher übertragen.

Eine weitere Anwendung vermittelt das **Programm 10.5.** Hier wird die adressierte Bildschirmposition durch den Pseudozufallszahlengenerator des BASIC-Systems ausgewählt. Beim jeweils ersten Zugriff auf eine Position schreibt das Programm einen *Punkt*.

Trifft es zum zweitenmal auf dieselbe Stelle, so wird der Punkt durch das *Währungssymbol* ersetzt. Bei weiteren Zugriffen wird es gegen einen *Stern* ausgetauscht, dieser gegen das *Nummernzeichen*. Auf diese Weise kann man die Gleichmäßigkeit der Bildung von Pseudozufallszahlen durch die RND-Funktion anschaulich grafisch verfolgen.

Programm 10.5. Test der RND-Funktion

```
100 REM <<<<<<<<<<<< TEST DER RND-FUNKTION >>>>>>>>>>>>>
110 :
120 PRINT CHR$(12) : REM Bildschirm auf 00 loeschen
130 LET A = &HF800 + INT(1920*RND)
140 LET V = PEEK(A)
150 IF V = 0 THEN POKE A,46 : GOTO 130 : REM "."
160 IF V = 46 THEN POKE A,36 : GOTO 130 : REM "¤"
170 IF V = 36 THEN POKE A,42 : GOTO 130 : REM "*"
180 POKE A,35 : GOTO 130 : REM "#"
190 :
200 REM ================================================
```

10.4. Eingeben und Ausgeben über Ports

Die Realisierung von Eingaben und Ausgaben gehört zu den Aufgaben eines Betriebssystems. Höhere Programmiersprachen greifen i. allg. auf diese Funktionen zurück. Lediglich Systemprogrammiersprachen, die ja auch zur Implementierung von Eingabe/Ausgabe-Programmen benutzt werden, gestatten den unmittelbaren Zugriff zu den peripheren Geräten.

Beim Einsatz von Mikrorechnern taucht nun häufig das Problem auf, individuelle Peripherie zu bedienen, für die keine Standardprogramme des Betriebssystems vorliegen. Ein Beispiel hierfür sind Aufgaben der Laborautomatisierung. Um auch in solchen Fällen mit BASIC arbeiten zu können, bieten moderne Dialekte Funktionen und Anweisungen zum unmittelbaren Zugriff auf sog. *Ports*. Ports sind *Eingabe/Ausgabe-Tore*, die als *Puffer* zwischen dem Mikrorechner und den peripheren Geräten angeordnet sind. Vom Mikrorechner aus sind sie über eine *Nummer*, die *Eingabe/Ausgabe-Adresse*, zu erreichen, die bei den Mikroprozessoren U880 und Z80 jeweils im Bereich von 0 bis 255 liegt.

Die Bedeutung dieser Ports ist gerätespezifisch. Häufig werden sie mit Hilfe von hochintegrierten, programmierbaren *Eingabe/Ausgabe-Schaltkreisen* realisiert. Der daran interessierte Leser wird auf die entsprechende Fachliteratur verwiesen, wenn er tiefer in den Aufbau und die Funktionsweise der in den Programmen benutzten Schaltkreise eindringen will [10.1]. Diese Beispiele wurden in das vorliegende Buch aufgenommen, um die breiten Einsatzmöglichkeiten von BASIC zu demonstrieren.

10.4.1. Ausgeben über Ports

Zur Ausgabe dient die Sprachanweisung

n **OUT** *adresse* , *wert*

Stößt der Interpreter auf diese Anweisung, so berechnet er zunächst die arithmetischen Aus-

drücke für die Portadresse und den auszugebenden Wert und rundet die Resultate erforderlichenfalls. Werte über 255 führen zu Fehlermeldungen. Dann legt der Interpreter die Adresse und das niederwertige Byte des ermittelten Wertes auf das Sammelleitungsbündel (*Bus*) zu den Eingabe/Ausgabe-Toren. Die Daten müssen nun von dem adressierten Port übernommen und an das angeschlossene periphere Gerät weitergeleitet werden.

Als Beispiel wird im **Programm 10.6** ein Schaltkreis zur *bitparallelen Eingabe/Ausgabe* (PIO – *Parallel Input/Output*) zuerst auf Ausgabe initialisiert. Anschließend können über ihn Bytes zu einem Gerät übermittelt werden.

Programm 10.6. Direktausgabe über einen PIO-Schaltkreis

```
100 REM <<<<<<<<<<<<<<<<<< PIO-AUSGABE >>>>>>>>>>>>>>>>>>
110 :
120 OUT 41 , &HFF : REM Modus 3
130 OUT 41 , &H0 : REM Ausgabe
140 :
150 INPUT "Auszugebendes Bitmuster" ; M
160 OUT 40 , M
170 GOTO 150
180 :
190 REM ================================================
```

Programm 10.7. Erzeugen eines Sekundentakts mit einem CTC-Schaltkreis

```
100 REM <<<<<<<<<<<<<<<<<< SEKUNDENTAKT >>>>>>>>>>>>>>>>>>
110 :
120 OUT 32,&H37 : REM CTC-Kanal 0 , Modus
130 OUT 32,217  : REM Zeitkonstante 22 ms
140 OUT 33,&H57 : REM CTC-Kanal 1 , Modus
150 OUT 33,45   : REM Zeitkonstante 1 s
160 :
170 REM ================================================
```

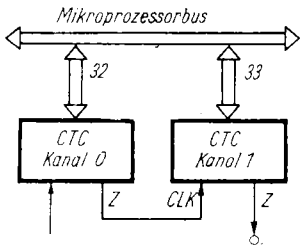


Bild 10.7. Erzeugung eines Sekundentakts mit einem CTC-Schaltkreis

Im **Programm 10.7** wird ein *Zähler/Zeitgeber-Schaltkreis* (CTC – *Counter/Timer Circuit*) eingesetzt (**Bild 10.7**). Der Kanal 0 arbeitet als Zeitgeber. Die Nulldurchgänge erfolgen bei der gewählten Initialisierung alle 22 Millisekunden. Die entsprechenden Signale werden dem Kanal 1 zugeführt und dort gezählt. Genau alle Sekunden ergibt sich dann an dessen Ausgang Z ein Signal.

10.4.2. Eingeben über Ports

Zur Eingabe bietet BASIC die Funktion

INP(*adresse*)

Hier berechnet der Interpreter zunächst den für die Portadresse angegebenen arithmetischen Ausdruck und rundet ihn erforderlichenfalls. Adressen über 255 verursachen Fehler. Dann wird diese Adresse auf den Mikrorechnerbus gelegt und versucht, von dem angesprochenen Port ein Datenbyte zu erhalten. Dieses Bitmuster wird dann in Form einer natürlichen Zahl zur Verfü-

gung gestellt; es kann einer Zahlenvariablen zugewiesen werden:

390 LET Z = INP (24)

Auf diese Weise werden im **Programm 10.8** Bitmuster über einen PIO-Schaltkreis eingelesen, der auf Eingabe programmiert ist.

Programm 10.8. Direkteingabe über einen PIO-Schaltkreis

```
100 REM <<<<<<<<<<<<<<<<<< PIO-EINGABE >>>>>>>>>>>>>>>>>>
110 :
120 OUT 41 , &HFF : REM Modus 3
130 OUT 41 , &HFF : REM Eingabe
140 :
150 PRINT "Bitmuster eingeben! - " ;
160 INPUT "fertig" : A
170 PRINT "Eingegebener Wert =" ;
180 PRINT INP(40)
190 GOTO 150
200 :
210 REM ================================================
```

10.4.3. Warten auf eine bestimmte Eingabe von einem Port

Bei Eingaben tritt häufig das Problem auf, daß auf ein bestimmtes Ereignis gewartet werden soll. Dieses Ereignis könnte darin bestehen, daß ein bestimmtes Bit eines Ports gesetzt oder gelöscht wird. Man könnte hier nun so vorgehen, daß der Wert des betreffenden Ports laufend eingelesen und analysiert wird. BASIC bietet für solche Zwecke die Sprachanweisung

n **WAIT** *adresse* , *testmaske* [, *negierungsmaske*]

Alle Parameter sind arithmetische Ausdrücke, die evtl. gerundet werden; Werte über 255 ergeben Fehlermeldungen. Der Interpreter realisiert folgende Aktionen:

- *Einlesen* eines Bitmusters vom adressierten Port entsprechend dem folgenden Beispiel:

  500 LET B = INP(*adresse*)

- *Negieren* derjenigen Bits des eingelesenen Musters, die in der Negierungsmaske angegeben sind:

  501 LET B = B XOR *negierungsmaske*

- *Herausschneiden* der durch die Testmaske angegebenen Bits:

  502 LET B = B AND *testmaske*

- *Prüfen*, ob mindestens *eines* der interessierenden Bits den Wert 1 aufweist:

  503 IF B THEN GOTO 510

Anderenfalls wird *erneut* das im Port vorliegende Bitmuster eingelesen:

504 GOTO 500

Alle diese Aktionen werden durch die WAIT-Anweisung zusammengefaßt. Dabei darf die Angabe der Negierungsmaske entfallen. Der Interpreter nimmt dann dafür den Wert 0 an, negiert also *nichts*.

Das **Programm 10.9** demonstriert eine Anwendung. Es initialisiert zunächst einen PIO-Schaltkreis für Eingaben. Dann wartet es darauf, daß gerade das Bit 7 gesetzt wird. Die Testmaske &H80 sorgt dafür, daß alle anderen Bit ohne Einfluß sind. Anschließend wartet das Programm, daß dieses Bit wieder gelöscht wird. In der Negierungsmaske brauchten übrigens nicht alle Bits gesetzt zu sein; es würde Bit 7 allein genügen (&H80).

Die WAIT-Anweisung kann auch dazu benutzt werden, um auf *eine* Änderung des gerade anliegenden Wertes zu warten. Als Negierungsmaske ist dann genau dasjenige Bitmuster zu benutzen, auf dessen Veränderung gewartet werden soll, während in der Testmaske diejenigen Bits gesetzt werden müssen, die zu überwachen sind. Ein entsprechendes Beispiel zeigt das **Programm 10.10.**

Programm 10.9. Warten auf eine Direkteingabe über einen PIO-Schaltkreis

```
100 REM <<<<<<<<<<<< WARTEN AUF PIO-EINGABE >>>>>>>>>>>>
110 :
120 OUT 41 , &HFF : REM Modus 3
130 OUT 41 , &HFF : REM Eingabe
140 :
150 PRINT "linkes Bit setzen!"
160 WAIT 40 , &H80
170 :
180 PRINT "linkes Bit loeschen!"
190 WAIT 40 , &H80 , &HFF
200 PRINT "Ende"
210 :
220 END
230 REM ================================================
```

Programm 10.10. Warten auf eine Änderung des in einem Port anliegenden Bitmusters

```
100 REM <<<<<<<<<<<<< WARTEN AUF AENDERUNG >>>>>>>>>>>>>
110 :
120 OUT 41 , &HFF : REM PIO-Modus 3
130 OUT 41 , &HFF : REM Eingabe
140 :
150 PRINT "Bitmuster eingeben!"
160 INPUT "fertig" ; A
170 LET M = INP(40)
180 PRINT "Beliebige Aenderung vornehmen!"
190 WAIT 40 , &HFF , M
200 GOTO 150
210 :
220 REM ================================================
```

Bei der Anwendung der WAIT-Anweisung ist sorgfältig darauf zu achten, daß bei der Wahl der Masken kein Fehler unterläuft. Der Wartezyklus kann nämlich nur durch eine Eingabe beendet werden, die den festgelegten Masken entspricht. Ist das logisch oder technisch unmöglich, so tritt eine *Verklemmung* auf. Da die CTRL-C-Taste nur zwischen zwei Anweisungen wirksam ist, kann man die Warteschleife nicht in gewohnter Weise abbrechen, sondern muß den Computer ausschalten; falls keine RESET-Taste vorhanden ist.

10.5. Einbinden von Maschinenprogrammen

In den vorstehenden Abschnitten wurde gezeigt, daß BASIC über eine Reihe leistungsfähiger Sprachmittel verfügt, die einen direkten Zugriff zur Hardware gestatten. Trotzdem kann eine im Prinzip maschinenunabhängige höhere Sprache nicht alle Besonderheiten eines Computers ausnutzen. Das bleibt der Maschinensprache vorbehalten. Moderne BASIC-Dialekte bieten deshalb auch die Möglichkeit, innerhalb von BASIC-Programmen Unterprogramme einzusetzen, die in der Maschinensprache kodiert sind. Leider werden diese Unterprogramm- und Funktionsaufrufe in den einzelnen Interpretern recht unterschiedlich gehandhabt. Sie sind nicht nur vom Betriebssystem abhängig, wie die Verwaltung von Programmen und Daten auf externen Zusatzspeichern. Zusätzlich werden auch die Randbedingungen der jeweiligen Programmiersysteme beachtet, mit deren Hilfe die Maschinenprogramme implementiert werden. Es ist daher unbedingt erforderlich, sich anhand des Bedienungshandbuchs des jeweiligen Mikrorechnersystems mit den konkreten Verhältnissen vertraut zu machen. Um das prinzipielle Vorgehen aber anhand von Beispielen zeigen zu können, wird im folgenden eine einfache, relativ maschinennahe Variante besprochen.

10.5.1. Bereitstellen der Maschinenprogramme

Die einzusetzenden Maschinenprogramme können u. U. im Rechner bereits zur Verfügung stehen, beispielsweise als Unterprogramme des Betriebssystems. Anderenfalls sind sie vom Programmierer zu entwerfen und zu *implementieren*. Dabei bestehen verschiedene Möglichkeiten:

- Das Programm wird *unmittelbar* in der Maschinensprache kodiert, z. B. in hexadezimaler Form. Dieser Weg ist umständlich und fehleranfällig. Man benötigt aber dafür keine technischen Hilfsmittel.
- Es wird ein Quellprogramm in der *Assemblersprache* geschrieben und mit Hilfe eines Assemblers in den Maschinenkode übersetzt (Abschn. 2.4.2).

Das implementierte Maschinenprogramm muß mit einem Unterprogramm-Rücksprungbefehl (*RET*) abgeschlossen werden, um eine Rückkehr in das aufrufende BASIC-Programm zu erreichen.

Für das Programm ist im Hauptspeicher ein hinreichend großer Bereich zu *reservieren:*

- Eventuell werden in dem benutzten Computer bestimmte Speicherbereiche bei der Arbeit mit BASIC nicht benötigt.
- Anderenfalls ist dem BASIC-System mitzuteilen, daß es sich mit einem kleineren Arbeitsspeicher begnügen muß, z. B. mit Hilfe der CLEAR-Anweisung (Abschn. 3.5.1).

Dann ist das Maschinenprogramm in diesen Hauptspeicherbereich zu bringen:

- Wenn der Programmierer zur Kodierung die interne Maschinensprache benutzt hat, so kann

Programm 10.11. Eingeben eines Maschinenprogramms in den Hauptspeicher

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"        EINGABE EINES MASCHINENPROGRAMMS        "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 DEF FNK¤(X¤) = CHR¤(ASC(X¤) - 32*(X¤>="a")*(X¤<="z"))
150 PRINT
160 INPUT "Anfangsadresse" ; A
170 PRINT "Maschinenprogramm byteweise hexadezimal" ;
180 PRINT " eingeben!"
190 PRINT "Endezeichen = @"
200 :
210 PRINT
220 PRINT A ;
230 FOR I=1 TO 20
240   INPUT ; "-" , H¤
250   IF H¤="@" THEN GOTO 440 : REM Ende
260   :
270   FOR P=1 TO 2
280     LET H¤=LEFT¤(H¤,P-1)+FNK¤(MID¤(H¤,P,1))+MID¤(H¤,P+1)
290   NEXT P
300   LET D = 0 : REM Anfangswert
310   FOR P=1 TO 2
320     LET Z = ASC(MID¤(H¤,P,1))
330     IF Z>=48 AND Z<=57 THEN LET C = 48 : GOTO 370
340       IF Z>=65 AND Z<=70 THEN LET C = 55 : GOTO 370
350         PRINT "???" ;
360         GOTO 410
370     LET D = 16*D + Z - C
380   NEXT P
390   POKE A , D
400   LET A = A+1
410 NEXT I
420 GOTO 210
430 :
440 PRINT : PRINT
450 PRINT "Endadresse:" ; A-1
460 :
470 END
480 REM ==================================================
```

er das Programm unmittelbar in den Speicher *eingeben*, z. B. mit Hilfe des BASIC-**Programms 10.11.**

- Wurde das Programm in der Assemblersprache geschrieben und mit Hilfe eines Assemblers übersetzt, so ist es mit dem im Betriebssystem verfügbaren *Lader* in den Speicher zu bringen.

Programm 10.12. Abspeichern eines Maschinenprogramms auf einer Diskette

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"      ABSPEICHERN EINES MASCHINENPROGRAMMS      "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Anfangsadresse" ; A
160 INPUT "Endadresse" ; E
170 INPUT "Programmname" ; P¤
180 LET P¤ = LEFT¤(P¤,8)
190 OPEN "O" , #1 , P¤ + ".MC "
200 :
210 LET N = A
220 PRINT #1 , PEEK(N)
230 LET N = N+1
240 IF N<=E THEN GOTO 220
250 :
260 CLOSE #1
270 PRINT "Maschinenprogramm unter dem Namen " ; P¤ ;
280 PRINT " abgespeichert."
290 :
300 RESET
310 END
320 REM ==================================================
```

Programm 10.13. Laden eines Maschinenprogramms von einer Diskette

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"         LADEN EINES MASCHINENPROGRAMMS         "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Programmname" ; P¤
160 OPEN "I" , #1 , P¤ + ".MC "
170 INPUT "Ladeadresse" ; A
180 :
190 LET N = A
200 INPUT #1 , V
210 POKE N , V
220 LET N = N+1
230 IF NOT EOF(1) THEN GOTO 200
240 :
250 CLOSE #1
260 PRINT "Maschinenprogramm "; P¤ ;" eingelesen."
270 :
280 END
290 REM ==================================================
```

Maschinenprogramme lassen sich mit dem **Programm 10.12** aus dem Hauptspeicher auf eine Diskette auslagern und mit dem **Programm 10.13** von diesem externen Datenträger wieder einlesen.

10.5.2. Aufruf von Unterprogrammen

Für den Aufruf eines im *Maschinenkode* vorliegenden Unterprogramms bieten BASIC-Dialekte die Sprachanweisung

n **CALL** *adresse*

Bei der Abarbeitung dieser Anweisung ermittelt der Interpreter zunächst den aktuellen Wert der Startadresse, für die meist eine Zahl oder eine einfache Variable anzugeben ist. Dann führt er einen *Unterprogrammsprung* (in der Assemblersprache ebenfalls CALL genannt) zu dem Maschinenprogramm aus.

Manche Interpreter gestatten darüber hinaus die Angabe von Parametern; die konkrete Form ist allerdings recht unterschiedlich. Der Leser muß daher auf die jeweilige Sprachbeschreibung verwiesen werden.

10.5.3. Definition und Aufruf von Funktionen

Es ist auch möglich, anwendereigene Funktionen im *Maschinenkode* bereitzustellen. Wie bei den nutzerspezifischen BASIC-Funktionen (Abschnitt 6.3.2) ist es auch hier erforderlich, solche Funktionen durch eine *Definition* einzuführen:

DEF USR[*nummer*] = *adresse*

Durch diese Anweisung wird dem Interpreter die *Startadresse* des zugehörigen Maschinenprogramms mitgeteilt. Der Name der Funktion ist USR. Manchmal lassen sich mehrere solcher Funktionen definieren; sie sind dann durch eine zusätzliche Nummer zu unterscheiden:

```
140 DEF USR5 = &HD000
```

Andere Interpreter kennen diese Definition nicht. Bei ihnen haben die Startadressen in bestimmten Speicherzellen zu stehen; der Programmierer muß sie mit POKE- oder DOKE-Anweisungen dorthin bringen.

Der *Aufruf* erfolgt – wie bei BASIC-Funktionen üblich – durch die Angabe des Funktionsnamens innerhalb eines arithmetischen Ausdrucks

USR[*nummer*](*argument*)

Findet der Interpreter einen Funktionsaufruf, so veranlaßt er einen Unterprogrammsprung zur definierten Adresse. Dabei wird die Adresse (und der Typ) des Arguments in systemspezifischer Weise übergeben.

```
720 LET Z = USR5(X) + 9
```

10.5.4. Parametervermittlung

Für die Übergabe von Parametern vom BASIC-Programm zum Maschinenprogramm und zurück lassen sich Zellen des Hauptspeichers verwenden, deren numerische Adressen beiden Programmen bekannt sind. Im BASIC-Programm kann man zu diesem Zweck die POKE-Anweisung und die PEEK-Funktion einsetzen. Im folgenden Beispiel wird das Byte *vor* der Startadresse eines Maschinenkodeunterprogramms zur Parameterübergabe in beiden Richtungen benutzt.

```
610 POKE -12288,Z : REM Parameter
620 CALL -12287 : REM Maschinenkodeunterprogramm
630 LET X = PEEK (-12288) : REM Resultat
```

Die DOKE-Anweisung und die DEEK-Funktion bieten hier den Vorteil, auch ganze Zahlen von zwei Byte Länge übermitteln zu können.

In diesem Zusammenhang unterstützen manche BASIC-Dialekte die Möglichkeit, dem Maschinenkodeprogramm statt des Wertes einer Variablen ihre Speicheradresse zu übergeben:

VARPTR(*variablenname*)

Diese Funktion liefert die *Anfangsadresse* desjenigen Speicherbereichs, in dem die betreffende Variable vom Interpreter in interner Form abgespeichert ist. Diese Form der Parametervermittlung ist insbesondere für reelle Zahlenvariablen und Textvariablen sowie für Datenfelder von Interesse. Aber auch hier muß der Leser auf die Beschreibung des jeweiligen Interpreters verwiesen werden.

Programm 10.14. Testrahmen für ein Maschinenprogramm

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"      TESTRAHMEN FUER EIN SORTIERPROGRAMM      "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM -------------------- ABFRAGE ---------------------
150 :
160 PRINT
170 INPUT "Anfangsadresse fuer Programm" ; P
180 INPUT "Freier Speicher fuer Daten" ; F
190 INPUT "Anzahl der Daten" ; N
200 PRINT
210 PRINT "Ansteigende Zahlenfolge ..... 1"
220 PRINT "Konstante Zahlenfolge ....... 2"
230 PRINT "Abfallende Zahlenfolge ....... 3"
240 INPUT "Auswahl" ; A
250 ON A GOSUB 550 , 600 , 650
260 PRINT
270 PRINT "Beginn der Sortierung"
280 :
290 REM ----------- MASCHINENPROGRAMM AUFRUFEN -----------
300 :
310 POKE P , F AND &HFF     : REM Adresse / niederwert.Byte
320 POKE P+1 , INT(F/&H100) AND &HFF : REM hoeherwert.Byte
330 POKE P+2 , N AND &HFF   : REM Anzahl / niederwert.Byte
340 POKE P+3 , INT(N/&H100) AND &HFF : REM hoeherwert.Byte
350 LET P = P+4
360 CALL P                  : REM Start des Programms
370 :
380 REM ------------------ DATEN AUSGEBEN ----------------
390 :
400 PRINT
410 PRINT "Sortiertes Datenfeld:"
420 PRINT
430 FOR I=0 TO N-1
440   PRINT PEEK(F+I) ;
450 NEXT I
460 PRINT
470 :
480 PRINT
490 INPUT "Wiederholen (J|-)" ; A¤
500 IF A¤="J" OR A¤="j" THEN GOTO 200
510 END
520 :
530 REM --------------- DATEN EINSCHREIBEN ---------------
540 :
550 FOR I=0 TO N-1
560   POKE F+I , INT(I/N*&H100) AND &HFF
570 NEXT I
580 RETURN
590 :
600 FOR I=0 TO N-1
610   POKE F+I , 85
620 NEXT I
630 RETURN
640 :
650 FOR I=0 TO N-1
660   POKE F+I , INT((N-1-I)/N*&H100) AND &HFF
670 NEXT I
680 RETURN
690 :
700 REM ==================================================
```

An gleicher Stelle findet man Angaben darüber, wie das *Argument* der USR-Funktion und evtl. *Parameter* der CALL-Anweisung übermittelt werden, außerdem die Form der Übergabe der berechneten *Funktionswerte.* Üblich ist es, dazu Register des Mikroprozessors oder den von ihm verwalteten Kellerspeicher (*stack*) zu benutzen; mitunter aber auch bestimmte Speicherbereiche.

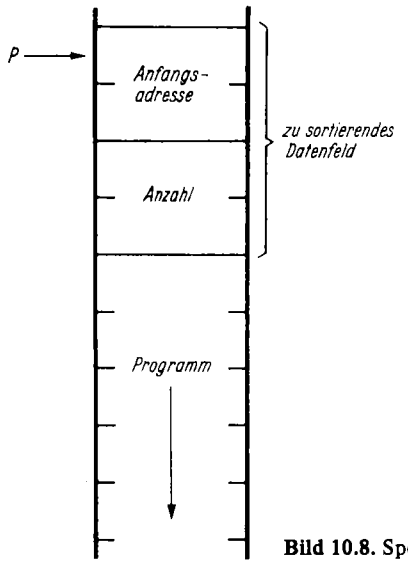


Bild 10.8. Speicherbelegung beim Programm 10.14

Den Einsatz eines Unterprogramms demonstriert das **Programm 10.14**. Hier wird das im Maschinenkode geschriebene Sortierprogramm 2.1 durch ein BASIC-Programm aufgerufen. Die Parameter werden über Hauptspeicherzellen vor der Startadresse übermittelt. **Bild 10.8** zeigt die Speicherbelegung. Dabei mußte das Maschinenprogramm noch durch einige Befehle ergänzt werden, die die Parameter aus diesen Zellen in die erforderlichen Register des Mikroprozessors bringen.

11. BASIC-Programme

Im folgenden sollen einige umfangreichere Programme zusammengestellt werden. Ziel ist es dabei, Beispiele für charakteristische Einsatzgebiete zu vermitteln und möglichst viele der dargestellten Mittel und Methoden einzusetzen. Dem Leser sollen Anregungen für die eigene Arbeit gegeben und mögliche Realisierungswege gezeigt werden. Dagegen ist es nicht beabsichtigt, ihm eine Programmbibliothek zur Verfügung zu stellen. Es ist ja der entscheidende Vorteil von BASIC, daß man sich für kleinere Verarbeitungsaufgaben mit geringem Aufwand eigene, problemspezifisch zugeschnittene Lösungen schaffen kann!

11.1. Numerik

11.1.1. Untersuchung von Funktionen (Programm 11.1)

Ziel des Programmsystems ist es, eine gegebene Funktion auf verschiedene Eigenschaften hin zu analysieren. Dazu ist die betreffende Funktion zunächst in Form einer *Definition* einzugeben. Dann wird der Bediener nach der gewünschten Untersuchung gefragt; die Verzweigung nach dem gewählten Teilprogramm erfolgt durch einen Unterprogrammrufverteiler.

Beim Bestimmen des absoluten *Maximums* und des absoluten *Minimums* der Funktion hat der Bediener den Untersuchungsbereich und eine angemessene Schrittweite anzugeben, in der die Funktionswerte berechnet werden sollen. Am Ende werden die beiden Extremwerte ausgedruckt.

Programm 11.1. Analyse einer gegebenen Funktion

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"          UNTERSUCHUNG VON FUNKTIONEN          "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++ ABFRAGE ++++++++++++++++++++
1080 :
1090 PRINT "Zu untersuchende Funktion bereits definiert?"
1100 PRINT "1140 DEF FNY(X) = ..."
1110 PRINT "Wenn ja, dann mit CONT fortsetzen;"
1120 PRINT "sonst noch definieren und erneut starten!"
1130 STOP
1140 REM Zeile reserviert fuer Funktionsdefinition
1150 :
1160 PRINT
1170 PRINT "Extremwerte bestimmen ..... 1"
1180 PRINT "Nullstellen suchen ........ 2"
1190 PRINT "Nullstellen berechnen ..... 3"
1200 PRINT "Numerische Differentiation ... 4"
1210 PRINT "Numerische Integration ..... 5"
1220 :
1230 INPUT "Auswahl" ; A
1240 PRINT
1250 IF A=0 THEN END
```

```
1260 ON A GOSUB 1310 , 1500 , 1790 , 1940 , 2110
1270 GOTO 1160
1280 :
1290 REM ++++++++++++ EXTREMWERTE BESTIMMEN ++++++++++++++
1300 :
1310 INPUT "Kleinstes Argument" ; K
1320 INPUT "Groesstes Argument" ; G
1330 INPUT "Schrittweite      " ; S
1340 :
1350 LET MI = FNY(K) : REM Anfangswert Minimum
1360 LET MA = MI     : REM Anfangswert Maximum
1370 FOR X=K+S TO G+S/2 STEP S
1380   LET M = FNY(X)
1390   IF M<MI THEN LET MI = M : GOTO 1410
1400   IF M>MA THEN LET MA = M
1410 NEXT X
1420 :
1430 PRINT" -> Minimum =" ; MI
1440 PRINT"    Maximum =" ; MA
1450 :
1460 RETURN
1470 :
1480 REM +++++++++++++++ NULLSTELLEN SUCHEN +++++++++++++++
1490 :
1500 INPUT "Kleinstes Argument" ; K
1510 INPUT "Groesstes Argument" ; G
1520 INPUT "Schrittweite" ; S
1530 INPUT "Maximale Anzahl von Nullstellen" ; N
1540 DIM X1(N), X2(N) : REM Argumente vor/hinter Nullstelle
1550 :
1560 LET N = 0 : REM laufende Nummer der Nullstelle
1570 FOR X=K TO G-S/2 STEP S
1580   IF SGN(FNY(X)) = SGN(FNY(X+S)) THEN GOTO 1620
1590     LET N = N+1
1600     LET X1(N) = X
1610     LET X2(N) = X+S
1620 NEXT X
1630 :
1640 IF N>0 THEN GOTO 1680
1650   PRINT "Keine Nullstellen vorhanden!"
1660   RETURN
1670 :
1680 PRINT "Nullstellen liegen zwischen den Argumenten:"
1690 LET M¤ = "#.######^^^^ & #.######^^^^"
1700 FOR I=1 TO N
1710   PRINT USING M¤ ; X1(I) , "..." , X2(I)
1720 NEXT I
1730 :
1740 ERASE X1 , X2
1750 RETURN
1760 :
1770 REM ++++++++++++ NULLSTELLEN BERECHNEN ++++++++++++++
1780 :
1790 INPUT "Zwei Naeherungswerte fuer Nullstellen" ; X1, X2
1800 LET Y1 = FNY(X1)
1810 LET Y2 = FNY(X2)
1820 :
1830 LET X = (X2+X1)/2
1840 LET Y = FNY(X)
1850 IF SGN(Y)=SGN(Y1) THEN LET X1 = X ELSE LET X2 = X
1860 IF ABS(X1-X2)>.000001 THEN GOTO 1830
1870 :
1880 PRINT " -> Exakter Wert der Nullstelle  X =" ; X
1890 :
1900 RETURN
1910 :
1920 REM ++++++++++++ NUMERISCHE DIFFERENTIATION ++++++++++++
1930 :
1940 INPUT ; "X = " , X
1950 PRINT TAB(18) ; "Y =" ; FNY(X) ;
1960 LET Y1 = 1 : REM Anfangswert Ableitung
```

```
1970 LET H = .1 : REM Anfangswert Hoehe (halbe Differenz)
1980 :
1990 LET Y2 = (FNY(X+H) - FNY(X-H))/2/H
2000 LET F = ABS((Y2-Y1)/Y1)
2010 LET Y1 = Y2 : REM verbesserte Ableitung
2020 LET H = H/2
2030 IF F>.000001 THEN GOTO 1990
2040 :
2050 PRINT TAB(35) ; "Y' =" ; Y1
2060 :
2070 RETURN
2080 :
2090 REM ++++++++++++ NUMERISCHE INTEGRATION ++++++++++++
2100 :
2110 INPUT "Untere Grenze" ; X1
2120 INPUT "Obere Grenze " ; X2
2130 :
2140 LET H = (X2-X1)/2 : REM Hoehe (Stuetzstellenabstand)
2150 LET N = 1 : REM halbe Anzahl der Intervalle
2160 LET R = FNY(X1) + FNY(X2) : REM Beitrag der Randpunkte
2170 LET G = 0 : REM Beitrag Stutzstellen gerader Nummer
2180 LET U = FNY(X1+H) : REM Stuetzstellen ungerader Nummer
2190 LET S = (R + 4*U)*H/3 : REM Anfangswert des Integrals
2200 :
2210 LET H = H/2
2220 LET N = 2*N
2230 LET G = G+U : REM ungerade -> gerade Stuetzstellen
2240 :
2250 LET U = 0 : REM Anfangswert ungerade Stuetzstellen
2260 LET X = X1+H : REM erste, ungerade Stutzstelle
2270 FOR I=1 TO N
2280   LET U = U + FNY(X)
2290   LET X = X + 2*H
2300 NEXT I
2310 :
2320 LET S1 = (R + 2*G + 4*U)*H/3
2330 LET F = ABS((S-S1)/S1)
2340 LET S = S1
2350 IF E>.000001 THEN GOTO 2210
2360 :
2370 PRINT "Wert des Integrals =" ; S
2380 :
2390 RETURN
2400 :
2410 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Ein ähnliches Vorgehen wird beim Aufsuchen von *Nullstellen* eingeschlagen. Hier werden in äquidistanten Schritten Funktionswerte ermittelt und diejenigen Argumente abgespeichert und abschließend ausgedruckt, zwischen denen sich das Vorzeichen der Funktionswerte ändert. Dabei werden allerdings **nur** einfache Nullstellen erfaßt.

Die gewonnenen Näherungswerte können als Eingabewerte für das nächste Teilprogramm verwendet werden, das in Anlehnung an das Newtonsche Eingabelungsverfahren den genauen Wert einer Nullstelle berechnet. Dieser iterative Algorithmus arbeitet nach dem folgenden Prinzip: Es sind zwei Punkte (x_1, y_1) und (x_2, y_2) der Funktion gegeben, die beiderseits der Nullstelle liegen **(Bild 11.1)**. Aus den beiden Abszissen x_1 und x_2 wird nun der Mittelwert gebildet und als neue, bessere Näherung für die Nullstelle der Funktion benutzt. Für die weitere Arbeit wird er mit demjenigen der beiden alten Näherungswerte kombiniert, der auf der anderen Seite der Nullstelle liegt, dessen Funktionswert also das umgekehrte Vorzeichen hat (x_2 im Bild 11.1). Diese schrittweise Verbesserung wird so lange wiederholt, bis die relative Änderung nicht mehr über 10^{-6} liegt.

Ein weiteres Teilprogramm bildet die *Ableitung* der zu untersuchenden Funktion bei einem vom Bediener bestimmten Wert x des Arguments. Dazu wird der Differenzenquotient für die Argumente x + h und x − h ermittelt und die Differenz h so lange halbiert, bis die dadurch bedingte relative Änderung des Differenzenquotienten nicht mehr größer als 10^{-6} ist.

Schließlich ist es auch noch möglich, das *bestimmte Integral* der Funktion zwischen zwei vom Bediener gegebenen Grenzen zu berechnen. Dazu wird die Simpsonsche Regel benutzt, bei der der Integrationsbereich in (Doppel-)Intervalle der Länge 2 h unterteilt wird (**Bild 11.2**). Für Anfang (a), Mitte (m) und Ende (e) dieser Intervalle sind die entsprechenden Funktionswerte zu berechnen und zu addieren:

$f(x_a) + 4\,f(x_m) + f(x_e)$.

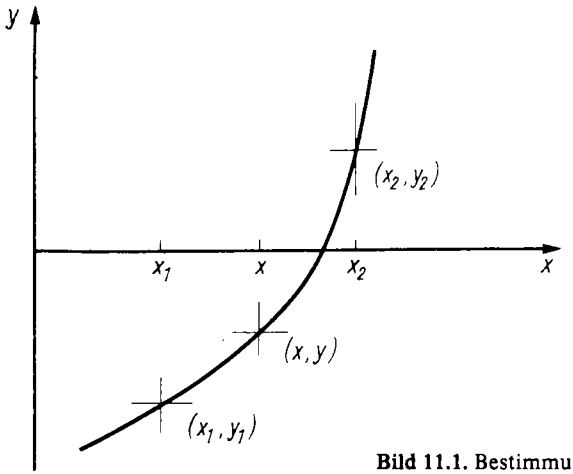


Bild 11.1. Bestimmung der Nullstelle einer Funktion durch Eingabeln

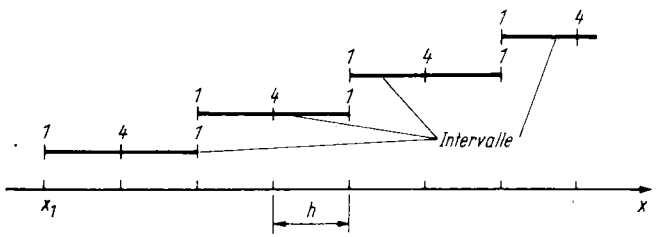


Bild 11.2. Erläuterung zur Simpsonschen Regel

Dann sind die Beiträge aller dieser Intervalle zusammenzuzählen. Dabei ist für die Funktionswerte an den beiden Grenzen x_1 und x_2 der Faktor 1, für die geradzahligen Stützstellen dazwischen der Faktor 2 (weil sie sowohl Ende des einen als auch Anfang des nächsten Intervalls sind) sowie für die ungeradzahligen Stützstellen der Faktor 4 zu verwenden. Nun wird laufend der Abstand h der Stützstellen halbiert und die gewichtete Summe der Funktionswerte berechnet, bis deren relative Änderung nicht mehr größer als 10^{-6} ist.

11.1.2. Berechnung eines Polynoms nach dem Hornerschen Schema (Programm 11.2)

Es ist das Ziel des Programms, die Funktionswerte von Polynomen und deren erste Ableitungen zu berechnen. Dazu werden vom Bediener die Ordnung n und die Koeffizienten a_i dieses Polynoms, die Grenzen des Argumentbereichs und die gewünschte Schrittweite angefordert. Daraus berechnet das Programm die Werte der Funktion y und der ersten Ableitung y' nach dem bekannten Hornerschen Schema:

$$y = (\ldots (a_n x + a_{n-1})x + \ldots)\,x + a_1 x + a_0$$

$$y' = (\ldots (n a_n x + (n-1)a_{n-1})x + \ldots)\,x + a_1.$$

Programm 11.2. Berechnung eines Polynoms nach dem Hornerschen Schema

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"              HORNERSCHES SCHEMA                "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM -------------------- EINGABE ---------------------
150 :
160 PRINT
170 INPUT "Ordnung des Polynoms" ; N
180 DIM A(N)
190 :
200 PRINT "Eingabe der Koeffizienten:"
210 FOR I=N TO 0 STEP -1
220   PRINT I ;
230   INPUT A(I)
240 NEXT I
250 :
260 PRINT "Wertebereich des Arguments X:"
270 INPUT "untere Grenze" ; U
280 INPUT "obere Grenze " ; O
290 INPUT "Schrittweite " ; S
300 :
310 REM -------------------- AUSGABE ---------------------
320 :
330 LPRINT
340 LPRINT TAB(12) ; "Berechnung des Polynoms"
350 LPRINT
360 LPRINT TAB(12) ; "Y = " ;
370 FOR I=N TO 0 STEP -1
380   IF A(I)=0 THEN GOTO 420
390   IF A(I)>0 THEN LPRINT "+" ; A(I) ; : GOTO 410
400     LPRINT "-" ; -A(I) ;
410   LPRINT "* X^" ; CHR$(8) ; I ;
420 NEXT I
430 LPRINT : LPRINT
440 LPRINT TAB(7) ; "X" ; TAB(23) ; "Y" ; TAB(39) ; "Y1"
450 LPRINT
460 FOR X=U TO O+S/2 STEP S
470   LET Y = 0
480   FOR I=N TO 0 STEP -1
490     LET Y = Y*X + A(I)
500   NEXT I
510   LET Y1 = 0
520   FOR I=N TO 1 STEP -1
530     LET Y1 = Y1*X + I*A(I)
540   NEXT I
550   LPRINT USING "##.#####^^^^     " ; X , Y , Y1
560 NEXT X
570 :
580 LPRINT
590 PRINT "Tabelle ausgedruckt."
600 PRINT
610 END
620 REM ==================================================
```

```
          Berechnung des Polynoms

          Y = + 2 * X^0

     X                Y                Y1

0.00000E+00      2.00000E+00      0.00000E+00
1.00000E+00      2.00000E+00      0.00000E+00
2.00000E+00      2.00000E+00      0.00000E+00
3.00000E+00      2.00000E+00      0.00000E+00
4.00000E+00      2.00000E+00      0.00000E+00
5.00000E+00      2.00000E+00      0.00000E+00
```

```
          Berechnung des Polynoms

          Y = + 3 * X^1 - 2 * X^0

     X                Y                Y1

 0.00000E+00     -2.00000E+00      3.00000E+00
 2.00000E+00      4.00000E+00      3.00000E+00
 4.00000E+00      1.00000E+01      3.00000E+00
 6.00000E+00      1.60000E+01      3.00000E+00
 8.00000E+00      2.20000E+01      3.00000E+00
 1.00000E+01      2.80000E+01      3.00000E+00


          Berechnung des Polynoms

          Y = - 2 * X^2 + 2 * X^0

     X                Y                Y1

-1.00000E+00      0.00000E+00      4.00000E+00
-8.00000E-01      7.20000E-01      3.20000E+00
-6.00000E-01      1.28000E+00      2.40000E+00
-4.00000E-01      1.68000E+00      1.60000E+00
-2.00000E-01      1.92000E+00      8.00000E-01
-2.98023E-08      2.00000E+00      1.19209E-07
 2.00000E-01      1.92000E+00     -8.00000E-01
 4.00000E-01      1.68000E+00     -1.60000E+00
 6.00000E-01      1.28000E+00     -2.40000E+00
 8.00000E-01      7.20000E-01     -3.20000E+00
 1.00000E+00      2.38419E-07     -4.00000E+00
```

Das Druckbild wird durch den Einsatz von Formatschablonen übersichtlich gestaltet. Das letzte der angegebenen Beispiele zeigt den Einfluß von Rundungsfehlern durch die begrenzte Darstellungsgenauigkeit im Computer.

11.1.3. Integration einer linearen Differentialgleichung erster Ordnung nach dem Verfahren von *Runge* und *Kutta* (Programm 11.3)

Das Verfahren von *Runge* und *Kutta* ermittelt eine spezielle Lösung der linearen Differentialgleichung erster Ordnung

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

durch schrittweise Integration mit der Schrittweite h von einem gegebenen Punkt $(x_0, y(x_0))$ aus. Dabei wird folgendermaßen vorgegangen:

Bekannt sei ein Näherungswert y_i der gesuchten Funktion beim Argument x_i. Dann werden folgende Ausdrücke berechnet:

$$v_1 = h\, f(x_i , y_i)$$

$$v_2 = h\, f(x_i + \frac{h}{2} , y_i + \frac{v_1}{2})$$

$$v_3 = h\, f(x_i + \frac{h}{2} , y_i + \frac{v_2}{2})$$

$$v_4 = h\, f(x_i + h , y_i + v_3)$$

Damit ergibt sich ein Näherungswert y_{i+1} für das Argument $x_{i+1} = x_i + h$:

$$y_{i+1} = y_i + \frac{1}{6} (v_1 + 2\, v_2 + 2\, v_3 + v_4).$$

Programm 11.3. Integration einer linearen Differentialgleichung erster Ordnung nach dem Verfahren von *Runge* und *Kutta*

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT" INTEGRATION EINER DIFF.-GL. NACH RUNGE/KUTTA  "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 DEF FNF(X) = 1/X          : REM  Y' = F(X,Y)
150 DEF FNY(X) = LOG(X)       : REM exakte Loesung
160 LET M¤ = "##.#####^^^^    "
170 :
180 PRINT
190 INPUT ; "Anfangspunkt: X0 = " , X0
200 INPUT "  Y0 = " , Y0
210 INPUT "Endabszisse:  X1 = " , X1
220 INPUT "Schrittweite: H  = " , H
230 INPUT "Abszissenabstand der Ausgaben" ; D
240 :
250 LPRINT TAB(7) ;"X"; TAB(22) ;"Y"; TAB(33) ;"FNY(X)+C"
260 LPRINT
270 LET C = Y0 - FNY(X0)
280 LET X2 = X0
290 LET J = 0 : REM Anfangswert des Zaehlers
300 LET Z = INT(D/H) : REM Endwert des Zaehlers
310 :
320 FOR I=1 TO INT((X1-X0)/H +.5)
330   IF J=0 THEN GOSUB 460
340   LET V1 = H*FNF(X0)      : REM F(X0,Y0)
350   LET V2 = H*FNF(X0+H/2)  : REM F(X0+H/2,Y0+V1/2)
360   LET V3 = H*FNF(X0+H/2)  : REM F(X0+H/2,Y0+V2/2)
370   LET V4 = H*FNF(X0+H)    : REM F(X0+H,Y0+V3)
380   LET Y0 = Y0 + (V1 + 2*V2 + 2*V3 + V4)/6
390   LET X0 = X2 + I*H
400   LET J = J+1
410   IF J=Z THEN LET J = 0
420 NEXT I
430 GOSUB 460
440 END
450 :
460 LPRINT USING M¤ ; X0 , Y0 , FNY(X0)+C : RETURN
470 :
480 REM ================================================
```

```
      X              Y           FNY(X)+C

 1.00000E+00    1.00000E+00    1.00000E+00
 2.00000E+00    1.69315E+00    1.69315E+00
 3.00000E+00    2.09861E+00    2.09861E+00
 4.00000E+00    2.38630E+00    2.38629E+00
 5.00000E+00    2.60944E+00    2.60944E+00
 6.00000E+00    2.79176E+00    2.79176E+00
 7.00000E+00    2.94591E+00    2.94591E+00
 8.00000E+00    3.07944E+00    3.07944E+00
 9.00000E+00    3.19722E+00    3.19722E+00
 1.00000E+01    3.30259E+00    3.30259E+00
```

Dieser Algorithmus wurde in BASIC umgesetzt und auf die spezielle Differentialgleichung

$$y' = \frac{1}{x}$$

angewandt, deren Lösung die logarithmische Funktion ist. Der Bediener hat den Anfangspunkt, die Endabszisse und die Schrittweite anzugeben sowie den gewünschten Abszissenabstand der Zwischenresultate. Da auch die exakte Lösung $\ln(x)+C$ mit ausgegeben wird, läßt sich daran die Genauigkeit der Integration bei der gewählten Schrittweite erkennen.

11.2. Statistik

11.2.1. Statistische Analyse von Stichproben (Programm 11.4)

Ziel des Programms ist die Auswertung von Stichproben, z. B. bei wiederholten Messungen derselben physikalischen Größe. Dazu wird entsprechend einer Eingabe des Bedieners ein Datenfeld reserviert. Dort werden die nacheinander angeforderten Werte eingetragen und zugleich deren Summe gebildet. Daraus wird nach dem Ende der Eingaben zunächst der *Mittelwert* M berechnet. Dann werden die Quadrate der Abweichungen der einzelnen Werte vom Mittelwert ermittelt, deren Summe durch N−1 geteilt und daraus die Wurzel gezogen, um die *Standardabweichung* S der Stichproben zu erhalten. Am Ende der Berechnungen werden die einzelnen Werte sowie S, M und der *Vertrauensbereich* des Mittelwertes ausgedruckt.

Programm 11.4. Statistische Stichprobenanalyse

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"     STATISTISCHE ANALYSE VON STICHPROBEN      "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 PRINT
150 INPUT "Anzahl" ; N
160 DIM X(N)
170 PRINT "Stichproben eingeben!"
180 :
190 LET S = 0 : REM Anfangswert fuer Summe
200 FOR I=1 TO N
210   INPUT ;"  "; X(I)
220   LET S = S + X(I)
230 NEXT I
240 PRINT
250 LET M = S/N : REM Mittelwert
260 :
270 LET S = 0 : REM Anfangswert fuer Summe
280 FOR I=1 TO N
290   S = S + (X(I)-M)^2
300 NEXT I
310 LET S = SQR(S/(N-1)) : REM Standardabweichung
320 :
330 LPRINT "Stichproben:"
340 LPRINT
350 FOR I=1 TO N
360   LPRINT X(I) ,
370 NEXT I
380 LPRINT : LPRINT
390 LPRINT "Standardabweichung = " ; S
400 LPRINT "Mittelwert         = " ; M ; "+|-" ; S/SQR(N)
410 PRINT "Resultate ausgedruckt."
420 :
430 END
440 REM ================================================
```

```
Stichproben:

 103            102            97             95
 101            104            99             105
 96             98

Standardabweichung =  3.49603
Mittelwert         =  100 +|- 1.10554
```

11.2.2. Berechnung einer Regressionsgeraden (Programm 11.5)

Ziel des Programms ist die Auswertung von Stichproben, denen ein linearer funktionaler Zusammenhang zugrunde liegt. Ein Beispiel dafür sind Messungen einer physikalischen Größe Y,

die linear von einem im Experiment variierten Parameter X abhängt. Anfangs wird der Bediener nach der Anzahl von Stichproben gefragt, und es werden zwei entsprechende Felder X und Y reserviert. Dann können Wertepaare (X, Y) eingegeben werden. Als Endezeichen dient der X-Wert @.

Es werden zunächst die Mittelwerte X1 und Y1 für alle X- bzw. Y-Werte berechnet, daraus dann die Summen

$$Q = \sum_{I=1}^{N} (X(I) - X1)^2 \quad \text{und} \quad K = \sum_{I=1}^{N} (X(I) - X1)\,(Y(I) - Y1).$$

Programm 11.5. Berechnung einer Regressionsgeraden

```
100 REM ++++++++++++++++++++++++++++++++++++++++++++++++++++
110 PRINT"                 LINEARE REGRESSION                "
120 REM ++++++++++++++++++++++++++++++++++++++++++++++++++++
130 :
140 REM -------------------- EINGABE ---------------------
150 :
160 PRINT
170 INPUT "maximale Anzahl der Tupel X,Y" ; N
180 DIM X(N) , Y(N)
190 :
200 PRINT "Werte der Tupel X,Y eingeben! Endezeichen = @"
210 PRINT
220 FOR I=1 TO N
230   PRINT I ;
240   INPUT " X" ; X$
250   IF X$="@" THEN LET N = I-1 : GOTO 290
260     LET X(I) = VAL(X$)
270     INPUT "  Y" ; Y(I)
280 NEXT I
290 PRINT
300 :
310 REM ------------------ MITTELWERTE -------------------
320 :
330 LET X0 = 0 : REM Anfangswert Summe X-Werte
340 LET Y0 = 0 : REM Anfangswert Summe Y-Werte
350 :
360 FOR I=1 TO N
370   LET X0 = X0 + X(I)
380   LET Y0 = Y0 + Y(I)
390 NEXT I
400 :
410 LET X1 = X0/N : REM Mittelwert der X-Werte
420 LET Y1 = Y0/N : REM Mittelwert der Y-Werte
430 :
440 REM --------- REGRESSIONSGERADE  Y = A*X + B ---------
450 :
460 LET Q = 0 : REM Anfangswert Quadratsumme
470 LET K = 0 : REM Anfangswert Korrelationssumme
480 :
490 FOR I=1 TO N
500   LET X2 = X(I) - X1 : REM X-Abweichung
510   LET Y2 = Y(I) - Y1 : REM Y-Abweichung
520   LET Q = Q + X2*X2
530   LET K = K + X2*Y2
540 NEXT I
550 :
560 LET A = K/Q : REM Steigung
570 LET B = Y1 - A*X1 : REM Absolutglied
580 PRINT "Ausgleichsgerade:  Y =" ; A ; "X +" ; B
590 :
600 END
610 REM ================+==============================
```

Mit ihrer Hilfe erhält man Steigung A und Absolutglied B der *Ausgleichsgeraden*

$$Y = A\,X + B,$$

die den funktionalen Zusammenhang zwischen X und Y widerspiegelt.

11.3. Lineare Gleichungssysteme und Matrizen

11.3.1. Matrizenrechnung (Programm 11.6)

Das Programmsystem soll eine Reihe von Matrizenoperationen demonstrieren. Dabei wurde versucht, im Rahmen der von BASIC gebotenen Möglichkeiten eine modulare Struktur zu erreichen.

Programm 11.6. Matrizenrechnung

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"                MATRIZENRECHNUNG                "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++ ABFRAGE ++++++++++++++++++++
1080 :
1090 CLEAR
1100 PRINT "Gewuenschte Operation:"
1110 PRINT "  Addition                  A+B  (1)"
1120 PRINT "  Subtraktion               A-B  (2)"
1130 PRINT "  Multiplikation mit Skalar S*A  (3)"
1140 PRINT "  Multiplikation            A*B  (4)"
1150 PRINT "  transponierte Matrix      A'   (5)"
1160 PRINT "  symmetrische Matrix       A+A' (6)"
1170 PRINT "  antimetrische Matrix      A-A' (7)"
1180 INPUT "Auswahl" ; R
1190 :
1200 ON R GOSUB 1280,1280,1530,1670,1860,1990,1990
1210      REM Ausfuehren der Operation
1220 PRINT
1230 INPUT "Fortsetzen (J|-)" ; A$
1240 IF A$="J" OR A$="j" THEN GOTO 1030
1250 :
1260 PRINT
1270 END
1280 :
1290 REM -------------- ADDITION/SUBTRAKTION --------------
1300 :
1310 GOSUB 2140 : REM Matrix A eingeben
1320 GOSUB 2320 : REM Matrix B eingeben
1330 IF MA=MB THEN GOTO 1370
1340   PRINT TAB(5) ; "Zeilenanzahl unterschiedlich!"
1350   ERASE A,B
1360   GOTO 1310
1370 IF NA=NB THEN GOTO 1420
1380   PRINT TAB(5) ; "Spaltenanzahl unterschiedlich!"
1390   ERASE A,B
1400   GOTO 1310
1410 :
1420 LET L = MA
1430 LET M = NA
1440 ON R GOSUB 2700,2790 : REM Ausfuehren der Operation
1450 :
1460 LET MC = MA
1470 LET NC = NA
1480 GOSUB 2550 : REM Resultat ausgeben
1490 RETURN
1500 :
```

```
1510 REM -------- MULTIPLIKATION MIT SKALARFAKTOR ---------
1520 :
1530 GOSUB 2140 : REM Matrix A eingeben
1540 GOSUB 2500 : REM Skalarfaktor S eingeben
1550 :
1560 LET L = MA
1570 LET M = NA
1580 GOSUB 2880 : REM Operation ausfuehren
1590 :
1600 LET MC = MA
1610 LET NC = NA
1620 GOSUB 2550 : REM Resultat ausgeben
1630 RETURN
1640 :
1650 REM ----------------- MULTIPLIKATION -----------------
1660 :
1670 GOSUB 2140 : REM Matrix A eingeben
1680 GOSUB 2320 : REM Matrix B eingeben
1690 IF NA=MB THEN GOTO 1740
1700   PRINT TAB(5) ; "Spaltenanzahl A <> Zeilenanzahl B"
1710   ERASE A,B
1720   GOTO 1670
1730 :
1740 LET L = MA
1750 LET M = NA
1760 LET N = NB
1770 GOSUB 2970 : REM Operation ausfuehren
1780 :
1790 LET MC = MA
1800 LET NC = NB
1810 GOSUB 2550 : REM Resultat ausgeben
1820 RETURN
1830 :
1840 REM -------------- TRANSPONIERTE MATRIX --------------
1850 :
1860 GOSUB 2140 : REM Matrix A eingeben
1870 :
1880 LET L = MA
1890 LET M = NA
1900 GOSUB 3090 : REM Operation ausfuehren
1910 :
1920 LET MC = NA
1930 LET NC = MA
1940 GOSUB 2550 : REM Resultat ausgeben
1950 RETURN
1960 :
1970 REM ----- SYMMETRISCHE UND ANTIMETRISCHE MATRIX ------
1980 :
1990 GOSUB 2140 : REM Matrix A eingeben
2000 IF MA<>NA THEN PRINT TAB(5) ; "M<>N" : GOTO 1990
2010 :
2020 LET L = MA
2030 ON R-5 GOSUB 3180,3270 : REM Operation ausfuehren
2040 :
2050 LET MC = NA
2060 LET NC = MA
2070 GOSUB 2550 : REM Resultat ausgeben
2080 RETURN
2090 :
2100 REM ++++++++++++ EIN-/AUSGABE-PROGRAMME ++++++++++++
2110 :
2120 REM ---------------- EINGABE MATRIX A ----------------
2130 :
2140 PRINT
2150 PRINT "Matrix A"
2160 INPUT "  Anzahl der Zeilen, Spalten" ; MA , NA
2170 IF MA=0 OR NA=0 THEN RETURN
2180 :
2190 DIM A(MA,NA)
2200 PRINT "  Elemente eingeben!"
```

```
2210 FOR I=1 TO MA
2220   PRINT "  Zeile" ; I ;
2230   FOR J=1 TO NA
2240     INPUT ; "  " ; A(I,J)
2250   NEXT J
2260 PRINT
2270 NEXT I
2280 RETURN
2290 :
2300 REM ---------------- EINGABE MATRIX B ----------------
2310 :
2320 PRINT
2330 PRINT "Matrix B"
2340 INPUT "  Anzahl der Zeilen, Spalten" ; MB , NB
2350 IF MB=0 OR NB=0 THEN RETURN
2360 :
2370 DIM B(MB,NB)
2380 PRINT "  Elemente eingeben!"
2390 FOR I=1 TO MB
2400   PRINT "  Zeile" ; I ;
2410   FOR J=1 TO NB
2420     INPUT ; "  " ; B(I,J)
2430   NEXT J
2440   PRINT
2450 NEXT I
2460 RETURN
2470 :
2480 REM ----------------- SKALARFAKTOR S -----------------
2490 :
2500 INPUT "Skalarfaktor S" ; S
2510 RETURN
2520 :
2530 REM ---------------- AUSGABE MATRIX C ----------------
2540 :
2550 PRINT
2560 PRINT "Resultat:"
2570 FOR I=1 TO MC
2580   PRINT
2590   FOR J=1 TO NC
2600     PRINT C(I,J) ,
2610   NEXT J
2620 NEXT I
2630 PRINT
2640 RETURN
2650 :
2660 REM +++++++++++++++ MATRIXOPERATIONEN +++++++++++++++
2670 :
2680 REM ------ ADDITION: A(L,M) + B(L,M) => C(L,M) -------
2690 :
2700 FOR I=1 TO L
2710   FOR J=1 TO M
2720     LET C(I,J) = A(I,J) + B(I,J)
2730   NEXT J
2740 NEXT I
2750 RETURN
2760 :
2770 REM ---- SUBTRAKTION: A(L,M) - B(L,M) => C(L,M) ------
2780 :
2790 FOR I=1 TO L
2800   FOR J=1 TO M
2810     LET C(I,J) = A(I,J) - B(I,J)
2820   NEXT J
2830 NEXT I
2840 RETURN
2850 :
2860 REM ------- MULTIPLIKATION: S*A(L,M) => C(L,M) -------
2870 :
2880 FOR I=1 TO L
2890   FOR J=1 TO M
2900     LET C(I,J) = S*A(I,J)
2910   NEXT J
```

```
2920 NEXT I
2930 RETURN
2940 :
2950 REM --- MULTIPLIKATION: A(L,M) * B(M,N) => C(L,N) ----
2960 :
2970 FOR I=1 TO L
2980   FOR K=1 TO N
2990     LET C(I,K) = 0
3000     FOR J=1 TO M
3010       LET C(I,K) = C(I,K) + A(I,J)*B(J,K)
3020     NEXT J
3030   NEXT K
3040 NEXT I
3050 RETURN
3060 :
3070 REM ----- TRANSPONIERTE MATRIX A'(M.L) => C(M,L) -----
3080 :
3090 FOR I=1 TO L
3100   FOR J=1 TO M
3110     LET C(J,I) = A(I,J)
3120   NEXT J
3130 NEXT I
3140 RETURN
3150 :
3160 REM   SYMMETRISCHE MATRIX (A(L,L)+A'(L,L))/2 => C(L,L)
3170 :
3180 FOR I=1 TO L
3190   FOR J=1 TO L
3200     LET C(I,J) = (A(I,J)+A(J,I))/2
3210   NEXT J
3220 NEXT I
3230 RETURN
3240 :
3250 REM  ANTIMETRISCHE MATRIX (A(L,L)-A'(L,L))/2 => C(L,L)
3260 :
3270 FOR I=1 TO L
3280   FOR J=1 TO L
3290     LET C(I,J) = (A(I,J)-A(J,I))/2
3300   NEXT J
3310 NEXT I
3320 RETURN
3330 :
3340 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Der erste Komplex von Moduln beschäftigt sich mit der Entgegennahme und Realisierung der Wünsche des Bedieners. Die *Abfrageschleife* arbeitet mit einem Unterprogrammrufverteiler, der den gewählten Matrizenoperationen spezifische *Ablaufsteuermoduln* zuordnet. Diese sorgen dafür, daß die erforderlichen Matrizen eingelesen werden, lassen die angewiesene Operation ausführen und veranlassen schließlich die Ausgabe der Resultate. Dabei greifen sie auf Moduln der beiden anderen Komplexe zurück.

Der zweite Komplex enthält die *Eingabe/Ausgabe-Moduln* für die verschiedenen Matrizen, während die Moduln des dritten Komplexes alle *Matrizenoperationen* realisieren.

Man erkennt deutlich den Aufwand, der vor den Unterprogrammaufrufen getrieben werden muß, um die aktuellen Eingabeparameter (MA, NA, MB, NB) an die in den Unterprogrammen benutzten globalen Variablen (L, M, N, MC, NC) anzupassen. Die Definition von nutzerspezifischen Funktionen, die mehrere Parameter haben und über mehrere Zeilen hinweggehen, würde hier einen übersichtlicheren Programmtext ergeben.

Die verschiedentlich auftretenden Anweisungen CLEAR bzw. ERASE A,B sind erforderlich, um eine doppelte Dimensionierung von Feldern zu vermeiden. Ansonsten enthalten die Moduln die Kodierung der bekannten Algorithmen für die Verknüpfung von Matrizenelementen bei der Ausführung von Matrizenoperationen.

11.3.2. Lösung eines linearen Gleichungssystems (Programm 11.7)

Das Programm benutzt das Gaußsche Eliminationsverfahren zur Lösung eines linearen Gleichungssystems:

$$a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = y_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = y_2$$
$$\ldots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = y_n$$

Ziel ist es dabei, die Koeffizientenmatrix in eine *Dreiecksform* überzuführen, wobei die Elemente auf der Hauptdiagonalen möglichst groß sein sollen, auf jeden Fall aber von null verschieden sein müssen:

$$\begin{array}{ccccccc} a'_{11} & a'_{12} & \ldots & a'_{1j} & \ldots & a'_{1n} \\ 0 & a'_{22} & & a'_{2j} & \ldots & a'_{2n} \\ \ldots & & & & & \\ 0 & 0 & & a'_{jj} & & a'_{jn} \\ \ldots & & & & & \\ 0 & 0 & \ldots & 0 & \ldots & a'_{nn} \end{array}$$

Diese Umformung erreicht man dadurch, daß man für jeden Wert j von 1 bis n−1 jeweils folgende Reihe von Linearkombinationen der Zeilen bildet:

$$\text{(Zeile j+1)} - \text{(Zeile j)} * \frac{a_{j+1,j}}{a_{jj}} \rightarrow \text{(Zeile j+1)}$$

$$\text{(Zeile j+2)} - \text{(Zeile j)} * \frac{a_{j+2,j}}{a_{jj}} \rightarrow \text{(Zeile j+2)}$$

$$\text{(Zeile n)} - \text{(Zeile j)} * \frac{a_{n,j}}{a_{jj}} \rightarrow \text{(Zeile n).}$$

Dabei müssen auch die rechten Seiten des Gleichungssystems mit umgeformt werden! Verschwindet bei dieser Umformung ein Diagonalelement ($<10^{-6}$), so ist die Matrix *singulär* und das Gleichungssystem nicht eindeutig lösbar.

Programm 11.7. Lösung eines linearen Gleichungssystems nach dem Eliminationsverfahren von *Gauß*

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"       GAUSZSCHES ELIMINATIONSVERFAHREN        "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM ++++++++++++++++++++ EINGABE ++++++++++++++++++++
1080 :
1090 INPUT "Dimension" ; N
1100 DIM A(N,N) , B(N) , X(N)
1110 :
1120 PRINT
1130 PRINT "Koeffizientenschema eingeben!"
1140 FOR I=1 TO N
1150   PRINT "Zeile" ; I ;
1160   FOR J=1 TO N
1170     INPUT ; "  " ; A(I,J)
1180   NEXT J
1190   PRINT
```

```
1200 NEXT I
1210 PRINT
1220 :
1230 PRINT "Absolutglieder eingeben!"
1240 PRINT "Zeile" ;
1250 FOR I=1 TO N
1260   PRINT "  " ; I ;
1270   INPUT ; B(I)
1280 NEXT I
1290 PRINT
1300 LPRINT "Eingegebenes Gleichungssystem:"
1310 GOSUB 2010
1320 :
1330 REM +++++++++++++++++++ UMFORMEN +++++++++++++++++++
1340 :
1350 FOR J=1 TO N-1
1360 :
1370 REM ------- GROESSTES SPALTENELEMENT A1 SUCHEN -------
1380 :
1390   LET A1 = 0
1400   FOR I=J TO N
1410     IF ABS(A(I,J))<=A1 THEN GOTO 1440 : REM NEXT I
1420       LET A1 = ABS(A(I,J))
1430       LET I1 = I : REM Zeile mit groesstem Element
1440   NEXT I
1450 :
1460 REM ---------- ZEILEN J UND I1 VERTAUSCHEN ----------
1470 :
1480   FOR K=J TO N
1490     LET Z = A(J,K)
1500     LET A(J,K) = A(I1,K)
1510     LET A(I1,K) = Z
1520   NEXT K
1530   LET Z = B(J)
1540   LET B(J) = B(I1)
1550   LET B(I1) = Z
1560   IF ABS(A(J,J)/A(1,1))<.000001 THEN GOTO 1930
1570 :
1580 REM ----------------- ELIMINIEREN -----------------
1590 :
1600   FOR I=J+1 TO N
1610     LET F = A(I,J)/A(J,J)
1620     FOR K=1 TO N
1630       LET A(I,K) = A(I,K) - F*A(J,K)
1640     NEXT K
1650     LET B(I) = B(I) - F*B(J)
1660   NEXT I
1670 NEXT J
1680 IF ABS(A(N,N)/A(1,1))<.000001 THEN GOTO 1930
1690 LPRINT "Gleichungssystem mit dreieckigem " ;
1700 LPRINT "Koeffizientenschema:"
1710 GOSUB 2010
1720 :
1730 REM +++++++++++++++ LOESUNG BERECHNEN +++++++++++++++
1740 :
1750 LET X(N) = B(N)/A(N,N)
1760 FOR I=N-1 TO 1 STEP -1
1770   LET S = 0
1780   FOR J=I+1 TO N
1790     LET S = S + A(I,J)*X(J)
1800   NEXT J
1810   LET X(I) = (B(I)-S)/A(I,I)
1820 NEXT I
1830 :
1840 REM +++++++++++++++++++ AUSGABE +++++++++++++++++++++
1850 :
1860 LPRINT "Loesungsvektor des Gleichungssystems:"
1870 LPRINT
1880 FOR I=1 TO N
1890   LPRINT USING "X(##)  = ##.#####^^^^   " ; I , X(I)
1900 NEXT I
```

```
1910 GOTO 1940
1920 :
1930 LPRINT "Koeffizientenschema ist singulaer!"
1940 PRINT
1950 PRINT "Resultate ausgedruckt."
1960 PRINT
1970 END
1980 :
1990 REM -------- AUSDRUCKEN DES GLEICHUNGSSYSTEMS --------
2000 :
2010 LPRINT
2020 FOR L=1 TO N
2030   FOR M=1 TO N
2040     LPRINT USING "##.#####^^^^    " ; A(L,M) ;
2050   NEXT M
2060   IF LPOS(I)>46 THEN LPRINT : LPRINT TAB(46) ;
2070   LPRINT USING "= ##.#####^^^^    " ; B(L)
2080 NEXT L
2090 LPRINT
2100 RETURN
2110 :
2120 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```
Eingegebenes Gleichungssystem:

 1.00000E+00    2.00000E+00   -1.00000E+00   =  4.00000E+00
 1.00000E+00   -3.00000E+00    1.00000E+00   =  3.00000E+00
 2.00000E+00   -4.00000E+00   -1.00000E+00   =  2.00000E+00

Gleichungssystem mit dreieckigem Koeffizientenschema:

 2.00000E+00   -4.00000E+00   -1.00000E+00   =  2.00000E+00
 0.00000E+00    4.00000E+00   -5.00000E-01   =  3.00000E+00
 0.00000E+00    0.00000E+00    1.37500E+00   =  2.75000E+00

Loesungsvektor des Gleichungssystems:

X( 1)  =  4.00000E+00
X( 2)  =  1.00000E+00
X( 3)  =  2.00000E+00


Eingegebenes Gleichungssystem:

-1.00000E+00    4.00000E+00   =  2.00000E+00
 2.00000E+00   -1.00000E+00   =  3.00000E+00

Gleichungssystem mit dreieckigem Koeffizientenschema:

 2.00000E+00   -1.00000E+00   =  3.00000E+00
 0.00000E+00    3.50000E+00   =  3.50000E+00

Loesungsvektor des Gleichungssystems:

X( 1)  =  2.00000E+00
X( 2)  =  1.00000E+00
```

Am Ende des Umformungsprozesses lassen sich die *Unbekannten* – beginnend bei x_n – nacheinander aus den jeweils bisher bekannten x_i berechnen.

11.4. Sortieren (Programm 11.8)

In den vorstehenden Abschnitten wurde bereits häufig ein Sortierverfahren als Beispiel benutzt. Jetzt sollen zusätzlich weitere Verfahren vorgestellt werden.

Programm 11.8. Vergleich verschiedener Sortierverfahren

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"                SORTIERVERFAHREN                "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++++ ABFRAGEN ++++++++++++++++++++
1080 :
1090 REM ------------- ZU SORTIERENDE FELDER --------------
1100 :
1110 INPUT "Anzahl der zu sortierenden Werte" ; N
1120 DIM A(N)
1130 :
1140 PRINT
1150 PRINT "Aufsteigend geordnete Zahlen ..... 1"
1160 PRINT "Konstante Zahlen .................2"
1170 PRINT "Abfallend geordnete Zahlen ....... 3"
1180 PRINT "Zufaellig verteilte Zahlen ....... 4"
1190 PRINT "Direkt eingegebene Zahlen ........ 5"
1200 INPUT "Auswahl" ; A
1210 PRINT
1220 IF A=0 THEN END
1230 :
1240 PRINT "Eingegebenes Feld:"
1250 PRINT
1260 ON A GOSUB 1500 , 1580 , 1660 , 1750 , 1830
1270 PRINT : PRINT
1280 :
1290 REM ---------------- SORTIERVERFAHREN ----------------
1300 :
1310 INPUT "Nummer des Sortierverfahrens" ; S
1320 IF S<1 OR S>7 THEN GOTO 1310
1330 ON S GOSUB 1960,2100,2240,2310,2400,2550,2670
1340 :
1350 REM -------------------- AUSGABE ---------------------
1360 :
1370 PRINT
1380 PRINT "Sortiertes Feld:"
1390 PRINT
1400 FOR I=1 TO N
1410   PRINT A(I) ,
1420 NEXT I
1430 PRINT : PRINT
1440 GOTO 1150
1450 :
1460 REM ++++++++++++ HILFSPROGRAMME ZUR EINGABE ++++++++++++
1470 :
1480 REM ---------- AUFSTEIGEND GEORDNETE ZAHLEN ----------
1490 :
1500 FOR I=1 TO N
1510   LET A(I) = I
1520   PRINT A(I) ,
1530 NEXT I
1540 RETURN
1550 :
1560 REM ---------------- KONSTANTE ZAHLEN ----------------
1570 :
1580 FOR I=1 TO N
1590   LET A(I) = 85
1600   PRINT A(I) ,
1610 NEXT I
1620 RETURN
1630 :
1640 REM ----------- ABFALLEND GEORDNETE ZAHLEN -----------
1650 :
1660 FOR I=1 TO N
1670   LET A(I) = N+1-I
1680   PRINT A(I) ,
```

```
1690 NEXT I
1700 RETURN
1710 :
1720 REM ----------- ZUFAELLIG VERTEILTE ZAHLEN -----------
1730 :
1740 RANDOMIZE
1750 FOR I=1 TO N
1760   LET A(I) = RND
1770   PRINT A(I) ,
1780 NEXT I
1790 RETURN
1800 :
1810 REM ----------- DIREKT EINGEGEBENE ZAHLEN ------------
1820 :
1830 FOR I=1 TO N
1840   INPUT ; "  " ; A(I)
1850 NEXT I
1860 PRINT
1870 RETURN
1880 :
1890 REM +++++++++++++++++ SORTIERPROGRAMME +++++++++++++++++
1900 :
1910 REM Parameter: A() = zu sortierendes Feld
1920 REM             N  = Anzahl der Elemente
1930 :
1940 REM ---------------------- SORT 1 ----------------------
1950 :
1960 FOR I=1 TO N-1
1970   FOR J=I+1 TO N
1980    IF A(I)>A(J) THEN GOSUB 2030 : REM Vertauschen
1990   NEXT J
2000 NEXT I
2010 RETURN
2020 :
2030 LET Z = A(I)
2040 LET A(I) = A(J)
2050 LET A(J) = Z
2060 RETURN
2070 :
2080 REM ---------------------- SORT 2 ----------------------
2090 :
2100 FOR K=1 TO N-1
2110   FOR I=1 TO N-1
2120     IF A(I)>A(I+1) THEN GOSUB 2170 : REM Vertauschen
2130   NEXT I
2140 NEXT K
2150 RETURN
2160 :
2170 LET Z = A(I)
2180 LET A(I) = A(I+1)
2190 LET A(I+1) = Z
2200 RETURN
2210 :
2220 REM ---------------------- SORT 3 ----------------------
2230 :
2240 FOR I=1 TO N-1
2250   IF A(I)>A(I+1) THEN GOSUB 2170 : GOTO 2240
2260 NEXT I
2270 RETURN
2280 :
2290 REM ---------------------- SORT 4 ----------------------
2300 :
2310 LET S = -1 : REM sortiert = TRUE
2320 FOR I=1 TO N-1
2330   IF A(I)>A(I+1) THEN LET S = 0 : GOSUB 2170
2340 NEXT I
2350 IF S THEN RETURN
2360 GOTO 2310
2370 :
2380 REM ---------------------- SORT 5 ----------------------
2390 :
```

```
2400 LET S = N : REM Schrittweite
2410 :
2420 LET S = INT(S/2)
2430 IF S=0 THEN RETURN
2440 :
2450 FOR K=1 TO N-S
2460   FOR I=K TO 1 STEP -S
2470     LET J = S+I
2480     IF A(I)>A(J) THEN GOSUB 2030 : REM Vertauschen
2490   NEXT I
2500 NEXT K
2510 GOTO 2420
2520 :
2530 REM ---------------------- SORT 6 ----------------------
2540 :
2550 FOR I=2 TO N
2560   LET Z = A(I)
2570   FOR J=I TO 2 STEP -1
2580     IF A(J-1)<=Z THEN GOTO 2610
2590     LET A(J) = A(J-1)
2600   NEXT J
2610   LET A(J) = Z
2620 NEXT I
2630 RETURN
2640 :
2650 REM ---------------------- SORT 7 ----------------------
2660 :
2670 LET S = N : REM Schrittweite
2680 :
2690 LET S = INT(S/2)
2700 IF S=0 THEN RETURN
2710 :
2720 FOR I=S+1 TO N
2730   LET Z = A(I) : REM einzufuegendes Element
2740   FOR J=I TO S+1 STEP -S
2750     IF A(J-S)<=Z THEN GOTO 2780
2760     LET A(J) = A(J-S)
2770   NEXT J
2780   LET A(J) = Z
2790 NEXT I
2800 GOTO 2690
2810 RETURN
2820 :
2830 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Das Programmsystem 11.8 bietet zunächst verschiedene *Ausgangsvektoren,* die vom Bediener ausgewählt werden können:

- aufsteigend geordnete Werte (das heißt richtig sortiert)
- konstante Werte
- absteigend geordnete Werte
- zufällig erzeugte Werte
- vom Bediener eingegebene Werte.

Diese Vektoren können mit Hilfe verschiedener, ebenfalls wählbarer Verfahren sortiert werden. Dabei gibt es zunächst Verfahren, die jeweils *zwei* Elemente miteinander *vergleichen* und erforderlichenfalls *austauschen*:

- Das Verfahren SORT 1 stimmt im Prinzip mit dem Programm 6.6 überein. Bei ihm wird der Vektor *vom Anfang* her sortiert: Zunächst wird das erste Element mit allen übrigen verglichen und evtl. vertauscht **(Bild 11.3)**. Am Ende dieses Durchlaufs steht an der Position des ersten Elements der kleinste Wert. Dann wird das zweite Element mit allen folgenden in Beziehung gebracht usf.
- Beim Verfahren SORT 2 werden dagegen jeweils die Werte zweier *benachbarter* Elemente des Datenfelds verglichen und erforderlichenfalls vertauscht **(Bild 11.4)**. Dazu wird der Vektor so

oft von Anfang bis Ende durchgemustert, bis man sicher sein kann, daß jeder Wert den richtigen Platz erreicht hat; bei N Elementen also (N−1)-mal.

- Das Vorgehen bei SORT 1 und SORT 2 ist dann uneffektiv, wenn der Vektor bereits fast richtig sortiert ist. Das Verfahren SORT 3 – eine Modifikation von SORT 2 – geht daher anders vor. Es bricht das Durchmustern ab, sobald ein falsch sortiertes Paar von Werten vertauscht wurde, und *beginnt von vorn*. Gelangt man bei einer Inspektion bis zum Ende, so ist das Datenfeld folglich sortiert.
- Eine weitere Variante von SORT 2 zeigt das Verfahren SORT 4. Hier wird zunächst angenommen, daß der Vektor sortiert und die Variable S auf TRUE gesetzt ist. Macht sich während des Durchmusterns eine Vertauschung zweier Werte erforderlich, so war diese Annahme offensichtlich falsch, und S erhält den Wert FALSE. Das Inspizieren des gesamten Datenfelds wird nun so lange fortgesetzt, bis S bei einem solchen Durchlauf den Wert TRUE behalten hat, das Feld also richtig sortiert ist.
- Das Verfahren SORT 5 arbeitet im Prinzip ähnlich wie SORT 2. Es vergleicht aber am Anfang nicht die unmittelbar benachbarten Elemente, sondern das erste, das mittlere und das letzte miteinander. Dann wird der Abstand (Schrittweite S) der in Beziehung gebrachten Elemente fortlaufend halbiert, bis am Ende (S = 1) bei einem Durchlauf alle Nachbarn verglichen werden. Dadurch ergeben sich insgesamt weniger Vertauschungen.

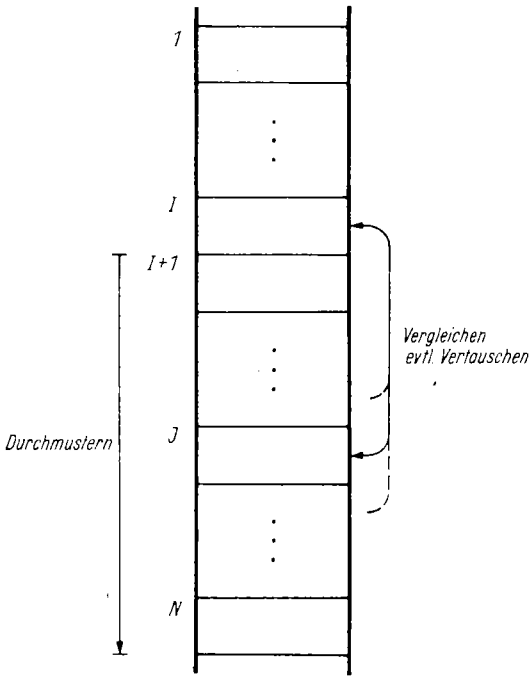


Bild 11.3. Sortierverfahren SORT 1

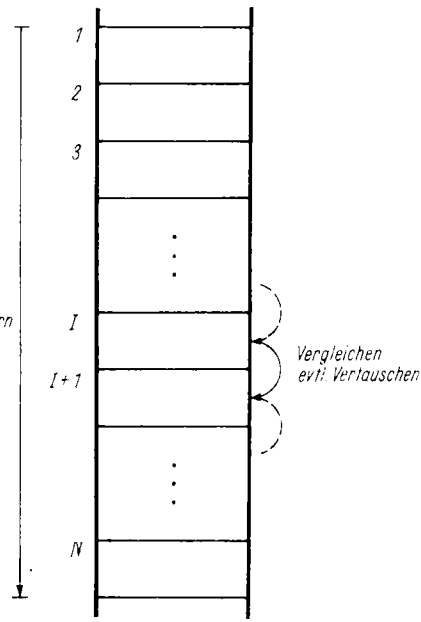


Bild 11.4. Sortierverfahren SORT 2

Eine weitere Klasse von Verfahren vergleicht ebenfalls jeweils die Werte zweier mehr oder weniger voneinander entfernter Elemente. Sind sie falsch sortiert, so erfolgt aber nicht wie bisher eine paarweise Vertauschung, sondern ein *Ringtausch* unter Einschluß aller dazwischenliegenden Elemente:

- Die Vorgehensweise bei diesen Sortierverfahren soll anhand von SORT 6 erläutert werden. Dazu wird angenommen, daß alle Elemente mit den Indizes 1 ... I – 1 bereits richtig sortiert sind. Nun wird das Element A(I) hinzugenommen. Gesucht ist diejenige Position des sortier-

ten Teilvektors, an der A(I) einzuordnen ist. Dazu wandert eine für die Aufnahme des (ursprünglichen) Wertes Z von A(I) bestimmte *Lücke* durch diesen Teilvektor und läßt auf ihrem Wege die bisher vorhandenen Elemente – jeweils um eine Position verschoben – hinter sich zurück. **Bild 11.5** zeigt die Situation, wenn die Stelle J erreicht ist. Diese Bewegung ist abgeschlossen, wenn das nächstkleinere Element A(J−1) des sortierten Teilvektors kleiner als oder gleich Z ist. Dann wird Z in diese Lücke eingetragen.

- Das Sortierverfahren SORT 7 (QUICK-SORT) geht im Prinzip wie SORT 6 vor, allerdings laufen die Indizes nicht in Einerschritten durch den zu sortierenden Vektor. Zunächst werden wenige, weit entfernte Elemente in den Vergleich und eine eventuelle Rotation einbezogen. Dann wird dieser Abstand S laufend halbiert, so daß weitere Elemente erfaßt und richtig eingeordnet werden. Das Verfahren endet mit S = 1. Auf diese Weise wird die Anzahl der Umspeicherungsoperationen bei den Rotationen beträchtlich verringert.

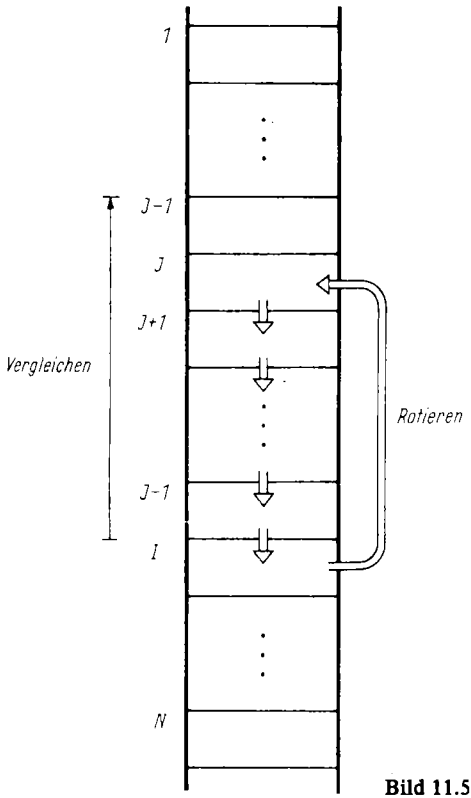


Bild 11.5. Sortierverfahren SORT 6

Tafel 11.1. Vergleich der Laufzeiten der Sortierprogramme SORT 1 . . . 7 bei 100 zu sortierenden Werten

Ausgangswerte	Richtig geordnet	Fallend geordnet	Zufällig verteilt
SORT 1	53	127	≈ 88
SORT 2	116	205	≈ 160
SORT 3	1	(> 1000)	(> 1000)
SORT 4	1	208	≈ 157
SORT 5	117	121	≈ 123
SORT 6	3	87	≈ 45
SORT 7	16	20	≈ 22

Das Programmsystem 11.8 gestattet es, die Laufzeiten der verschiedenen Sortierverfahren miteinander zu vergleichen. Die **Tafel 11.1** zeigt einige Resultate, die mit einem MBASIC-Interpreter auf einem Bürocomputer A5120 gewonnen wurden. Man kann aus diesem Vergleich folgende Schlüsse ziehen:

- Das Sortieren von Anfang an (SORT 1) ist besser als das Vertauschen benachbarter Elemente (SORT 2, vgl. aber auch SORT 4).
- Der Algorithmus SORT 3 ist ungeeignet.
- Solche Algorithmen, die für fast sortierte Vektoren günstig sind, benötigen bei ungeordneten Vektoren relativ viel Laufzeit (SORT 4 und SORT 6).
- Dagegen sind diejenigen Algorithmen, die ungeordnete Vektoren schnell sortieren, bei bereits fast sortierten recht langsam (SORT 5 und SORT 7).

11.5. Steuern und Regeln

11.5.1. Regeln einer Lichtsignalanlage (Programm 11.9)

Ziel dieses Programms ist es, die Eignung von BASIC zur Programmierung einer mikrorechnergesteuerten Regeleinrichtung zu demonstrieren. Ausgangspunkt ist das im **Bild 11.6** dargestellte Schema einer Straßenkreuzung.

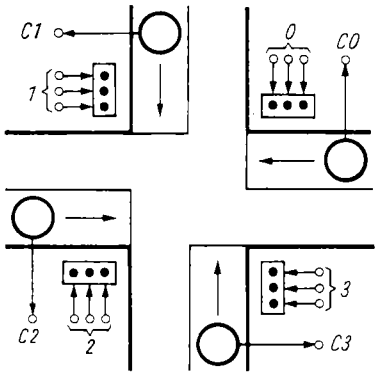


Bild 11.6. Schema einer Straßenkreuzung

Programm 11.9. Regelung einer Lichtsignalanlage

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"                LICHTSIGNALANLAGE                "
1030 PRINT
1040 REM ************************************************
1050 :
1060 DIM F(23) : REM Farbzeichen
1070 DIM Z(11) : REM Schaltzeiten
1080 DIM N(3)  : REM Fahrzeuganzahl
1090 :
1100 GOSUB 1520 : REM Initialisierung
1110 GOSUB 2110 : REM Abfrage Betriebsart A
1120 ON A GOSUB 1340 , 1190 , 1440 : REM Aufruf Signalfolge
1130 GOTO 1110
1140 :
1150 REM ++++++++++++++++++ SIGNALFOLGEN ++++++++++++++++++
1160 :
1170 REM ------------------ NORMALZYKLUS ------------------
1180 :
1190 GOSUB 2250 : REM Ermitteln der Fahrzeuganzahl
1200 GOSUB 1280 : REM Berechnen der Schaltzeiten
1210 FOR K=0 TO 7
1220   GOSUB 1980 : REM naechste Schaltphase
1230 NEXT K
1240 RETURN
1250 :
1260 REM ------------ BERECHNEN SCHALTZEITEN --------------
1270 :
1280 LET Z(2) = -((N(1)>N(3))*N(1) + (N(1)<=N(3))*N(3)) + 5
1290 LET Z(6) = -((N(0)>N(2))*N(0) + (N(0)<=N(2))*N(2)) + 5
1300 RETURN
1310 :
1320 REM ---------------------- ALARM ----------------------
1330 :
1340 LET K = 8
1350 GOSUB 1980 : REM -> GELB
1360 LET K = 9
1370 GOSUB 1980 : REM -> ROT
```

```
1380 LET K = 8
1390 GOSUB 1980 : REM -> GELB
1400 RETURN
1410 :
1420 REM ----------------- NACHTSCHALTUNG -----------------
1430 :
1440 LET K = 10
1450 GOSUB 1980 : REM -> GELB
1460 LET K = 11
1470 GOSUB 1980 : REM -> NICHTS
1480 RETURN
1490 :
1500 REM ++++++++++++++++ INITIALISIERUNG ++++++++++++++++
1510 :
1520 LET S = -1 : REM Anfangswert: Nachtschalter = TRUE
1530 GOSUB 1600 : REM PIO
1540 GOSUB 1680 : REM CTC
1550 GOSUB 1760 : REM Tabellen
1560 RETURN
1570 :
1580 REM -------------- INITIALISIERUNG PIO ---------------
1590 :
1600 OUT 41 , &HFF : REM Port A - Modus 3
1610 OUT 41 , &H0  : REM Port A - E/A-Maske
1620 OUT 43 , &HFF : REM Port B - Modus 3
1630 OUT 43 , &HF0 : REM Port B - E/A-Maske
1640 RETURN
1650 :
1660 REM -------------- INITIALISIERUNG CTC ---------------
1670 :
1680 FOR I=0 TO 3
1690   OUT 32+I , &H47 : REM Kanal I - Modus
1700   OUT 32+I , 255  : REM Kanal I - Anfangswert
1710 NEXT I
1720 RETURN
1730 :
1740 REM ------------ INITIALISIERUNG TABELLEN ------------
1750 :
1760 FOR K=0 TO 11
1770   READ F(2*K) , F(2*K+1) , Z(K)
1780 NEXT K
1790 RETURN
1800 :
1810 DATA &H50 , &H0A , 2
1820 DATA &HF0 , &H0A , 3
1830 DATA &H0A , &H05 , 20
1840 DATA &HAA , &H05 , 3
1850 DATA &HA0 , &H05 , 2
1860 DATA &HF0 , &H05 , 3
1870 DATA &H05 , &H0A , 20
1880 DATA &H55 , &H0A , 3
1890 DATA &HF0 , &H00 , 5
1900 DATA &H00 , &H0F , 10
1910 DATA &HF0 , &HF0 , 1
1920 DATA &H00 , &H00 , 1
1930 :
1940 REM ++++++++++++ EINGABE-/AUSGABE-PROGRAMME ++++++++++++
1950 :
1960 REM ------------------ SCHALTPHASEN ------------------
1970 :
1980 GOSUB 2190 : REM Ausgabe Farbzeichen
1990 GOSUB 2040 : REM Warten
2000 RETURN
2010 :
2020 REM --------------------- WARTEN ---------------------
2030 :
2040 FOR I=1 TO Z(K)
2050   FOR J=1 TO 400 : NEXT
2060 NEXT I
2070 RETURN
2080 :
```

```
2090 REM --------------- ABFRAGE BETRIEBSART ---------------
2100 :
2110 LET E = INP(42) XOR &HC0 : REM Signale low-aktiv !
2120 IF (E AND &H80)<>0 THEN LET A = 1 : RETURN
2130 IF (E AND &H40)<>0 THEN LET S = NOT S
2140 IF S THEN LET A = 3 ELSE LET A = 2
2150 RETURN
2160 :
2170 REM --------------- AUSGABE FARBZEICHEN ---------------
2180 :
2190 OUT 40 , F(2*K)
2200 OUT 42 , F(2*K+1)
2210 RETURN
2220 :
2230 REM ------------ ERMITTELN FAHRZEUGANZAHL ------------
2240 :
2250 FOR I=0 TO 3
2260   LET N(I) = 255 - INP(32+I)
2270 NEXT I
2280 GOSUB 1680 : REM Ruecksetzen CTC
2290 RETURN
2300 :
2310 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Bild 11.7 zeigt den Einsatz der benutzten peripheren Schaltkreise: Der *Zähler/Zeitgeber-Schaltkreis* CTC zählt die Fahrzeuge, die in den vier Zufahrten der Kreuzung die Induktionsschleifen passieren. Der *parallele Eingabe/Ausgabe-Schaltkreis* PIO gibt die entsprechenden Signale an die Ampeln aus. Außerdem dienen zwei Eingabeleitungen zur Wahl der *Betriebsart*:

- Nach dem Start des Programms blinken zunächst die Ampeln gelb. Das Umschalten zur zyklischen Arbeit der Lichtsignalanlage und wieder zurück zum Nachtbetrieb erfolgt durch das Signal „Tag/Nacht".
- Beim Vorliegen des Signals „Alarm" werden alle Ampeln über gelb nach rot geschaltet. Nach einer festgelegten Zeit geht der Rechner wieder zur unterbrochenen Arbeit über.

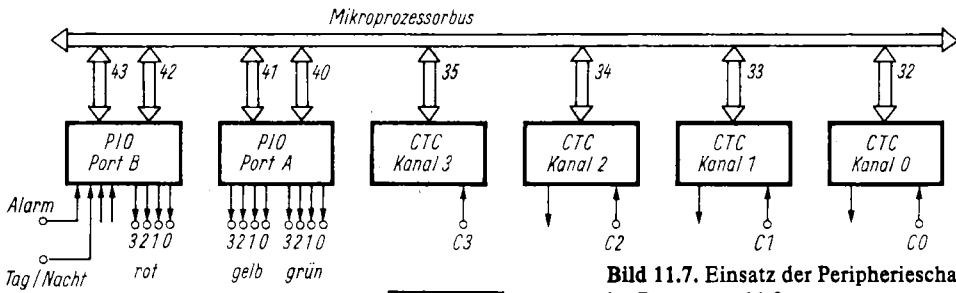


Bild 11.7. Einsatz der Peripherieschaltkreise im Programm 11.9

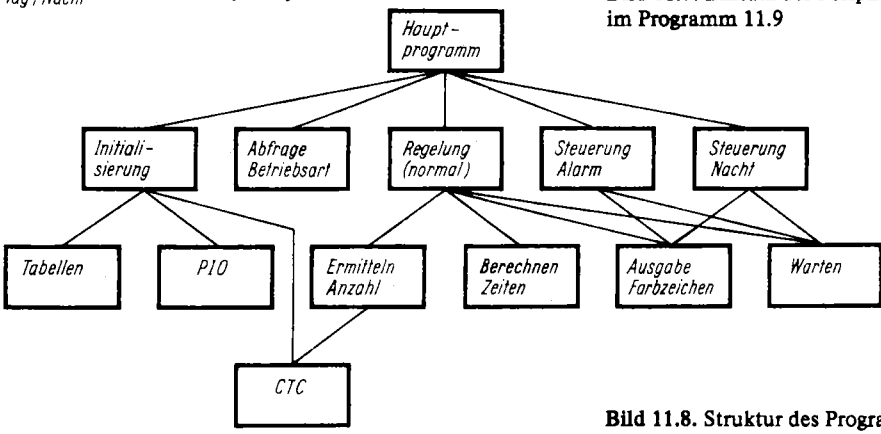


Bild 11.8. Struktur des Programmsystems 11.9

Die *Struktur* des Programmsystems ist im **Bild 11.8** dargestellt. Einige Besonderheiten sollen noch erläutert werden:

- Die verschiedenen möglichen Kombinationen von Farbzeichen sind mit der zugehörigen Zeitdauer in DATA-Anweisungen kodiert. Sie werden bei der Initialisierung der Tabellen in die Vektoren F (Farbzeichen) und Z (Zeiten) eingelesen.
- Die Dauer Z(K) der Grünphasen (K=2 für den Straßenzug 1/3, K=6 für den Straßenzug 0/2) wird aus der Anzahl N(I) der Fahrzeuge ermittelt, die auf den Kreuzungszufahrten (I = 0 . . . 3) seit der letzten Abfrage eingetroffen sind. Dabei wird ein einfacher Algorithmus eingesetzt, der für jeden Straßenzug nur die Zufahrt mit der höchsten Fahrzeuganzahl berücksichtigt.

11.5.2. Steuern einer Fernschreibmaschine (Programm 11.10)

Mit diesem Programm soll die prinzipielle Eignung von BASIC für die Steuerung einer 5-Kanal-Fernschreibmaschine als peripheres Gerät eines Mikrorechners gezeigt werden. Dazu wird zunächst eine Aufforderung zur Eingabe ausgeschrieben. Die anschließend mit Hilfe der Schreibmaschinentastatur fortlaufend eingegebenen Zeichen werden über einen *seriellen Eingabe/Ausgabe-Schaltkreis* (SIO – Serial Input/Output) in den Rechner übernommen und sofort über denselben Schaltkreis wieder an das Schreibwerk der Fernschreibmaschine ausgegeben. Außerdem erfolgt eine Kontrollausgabe auf dem Bildschirm.

Programm 11.10. Ansteuerung eines 5-Kanal-Fernschreibers

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"          STEUERUNG 5-KANAL-FERNSCHREIBER        "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 DIM B(31) , Z(31) : REM Fernschreibzeichen -> ASCII
1080 GOSUB 1270 : REM Initialisierung CTC
1090 GOSUB 1330 : REM Initialisierung SIO-Port A / Eingabe
1100 GOSUB 1430 : REM Initialisierung SIO-Port B / Ausgabe
1110 GOSUB 1530 : REM Initialisierung Konvertierungstabelle
1120 :
1130 READ A0 : REM Ausschrift
1140 GOSUB 1800 : REM Ausgabe A0
1150 IF A0<>7 THEN GOTO 1130 : REM Endezeichen BEL
1160 :
1170 GOSUB 1710 : REM Variable A1 eingeben
1180 PRINT CHR$(A1) : REM Kontrollausgabe auf Bildschirm
1190 IF A1<>7 THEN GOTO 1170 : REM Endezeichen BEL
1200 PRINT
1210 END
1220 :
1230 REM +++++++++++++++++ INITIALISIERUNG +++++++++++++++++
1240 :
1250 REM ------------------ CTC - KANAL 0 ------------------
1260 :
1270 OUT 32 , &H7   : REM Modus
1280 OUT 32 , 49    : REM Zeitkonstante
1290 RETURN
1300 :
1310 REM ------------- SIO - PORT A / EINGABE -------------
1320 :
1330 OUT 37 , 3     : REM Schreibregister 3
1340 OUT 37 , &H1   : REM Empfaenger EIN
1350 OUT 37 , 4     : REM Schreibregister 4
1360 OUT 37 , &HCC  : REM Modus
```

```
1370 OUT 37 , 5    : REM Schreibregister 5
1380 OUT 37 , &H0  : REM Sender AUS
1390 RETURN
1400 :
1410 REM ------------ SIO - PORT B / AUSGABE ------------
1420 :
1430 OUT 39 , 3    : REM Schreibregister 3
1440 OUT 39 , &H0  : REM Empfaenger AUS
1450 OUT 39 , 4    : REM Schreibregister 4
1460 OUT 39 , &HCC : REM Modus
1470 OUT 39 , 5    : REM Schreibregister 5
1480 OUT 39 , &H8  : REM Sender EIN
1490 RETURN
1500 :
1510 REM ------------ KONVERTIERUNGSTABELLEN ------------
1520 :
1530 FOR I=0 TO 31
1540   READ B(I) : REM Buchstaben
1550 NEXT I
1560 FOR I=0 TO 31
1570   READ Z(I) : REM Zahlen/Sonderzeichen
1580 NEXT I
1590 LET S = 0 : REM Modus: Buchstaben
1600 RETURN
1610 :
1620 DATA 00,69,10,65,32,83,73,35,13,68,82,74,78,70,67,75
1630 DATA 84,90,76,87,72,89,80,81,79,66,71,129,77,88,86,128
1640 DATA 00,51,10,45,32,39,56,55,13,05,52,07,44,36,58,40
1650 DATA 53,43,41,50,36,54,48,49,57,63,36,129,46,47,61,128
1660 :
1670 REM +++++++++++++++++ ASCII-ZEICHEN +++++++++++++++++
1680 :
1690 REM ------------------- EINGABE A1 -------------------
1700 :
1710 GOSUB 2120 : REM Fernschreibzeichen T einlesen
1720 GOSUB 2190 : REM Zeichen T als Echo wieder ausgeben
1730 LET A1 = -((NOT S)*B(T)) -(S*Z(T))
1740 IF A1=128 THEN LET S = 0 : GOTO 1710 : REM -> Buchst.
1750 IF A1=129 THEN LET S =-1 : GOTO 1710 : REM -> Ziffern
1760 RETURN
1770 :
1780 REM ------------------- AUSGABE A0 -------------------
1790 :
1800 IF S THEN GOTO 1890
1810   GOSUB 1980 : REM Buchstabe?
1820   IF T>0 THEN GOSUB 2190 : RETURN : REM Buchstabe!
1830     LET S = -1
1840     LET T = 27
1850     GOSUB 2190 : REM Umschaltung auf Ziffern
1860     GOSUB 2030 : REM Ziffer?
1870     GOSUB 2190 : REM unbedingte Ausgabe
1880     RETURN
1890 GOSUB 2030 : REM Ziffer?
1900 IF T>0 THEN GOSUB 2190 : RETURN : REM Ziffer!
1910   LET S = 0
1920   LET T = 31
1930   GOSUB 2190 : REM Umschaltung auf Buchstaben
1940   GOSUB 1980 : REM Buchstabe?
1950   GOSUB 2190 : REM unbedingte Ausgabe
1960   RETURN
1970 :
1980 FOR T=31 TO 1 STEP -1
1990   IF A0=B(T) THEN RETURN
2000 NEXT T
2010 RETURN
2020 :
2030 FOR T=31 TO 1 STEP -1
2040   IF A0=Z(T) THEN RETURN
2050 NEXT T
2060 RETURN
2070 :
```

```
2080 REM +++++++++++++++ FERNSCHREIBZEICHEN +++++++++++++++
2090 :
2100 REM -------------------- EINGABE ---------------------
2110 :
2120 OUT 37, 0 : REM SIO-Port A / Leseregister 0
2130 IF (INP(37) AND &H1)=0 THEN GOTO 2070:REM Puffer voll?
2140 LET T = INP(36) AND &H1F : REM Zeichen uebernehmen
2150 RETURN
2160 :
2170 REM -------------------- AUSGABE ---------------------
2180 :
2190 OUT 38, T AND &H1F : REM Zeichen uebergeben
2200 OUT 39, 0 : REM SIO-Port B / Leseregister 0
2210 IF (INP(39) AND &H4)=0 THEN GOTO 2170:REM Puffer leer?
2220 RETURN
2230 :
2240 REM ++++++++++++ AUFFORDERUNG ZUR EINGABE ++++++++++++
2250 :
2260 DATA 66,73,84,84,69,32,90,69,73,67,72,69,78
2270 DATA 32,69,73,78,71,69,66,69,78,58,10,13,7
2280 :
2290 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Die Programmstruktur ist einfach. Erläuterungen sollen nur zur *Konvertierung* gegeben werden:

- Der Fernschreibkode kennt zwei Zustände (Modi): Buchstaben und Ziffern/Sonderzeichen. Für jeden Zustand gibt es $2^5 = 32$ Bitmuster, von denen 31 für definierte Zeichen (einschließlich der Befehle für das Umschalten auf Buchstaben bzw. Ziffern/Sonderzeichen) belegt sind. Die entsprechenden ASCII-Äquivalente sind in DATA-Anweisungen gespeichert und werden bei der Initialisierung in die Vektoren B (Buchstaben) und Z (Ziffern/Sonderzeichen) eingelesen.
- Bei Eingaben läßt sich die Konvertierung unmittelbar durch einen indizierten Zugriff auf diese Vektoren durchführen. Die Kodes 128 und 129 dienen zur Kennzeichnung der Umschaltung auf Buchstaben bzw. Ziffern/Sonderzeichen.
- Bei Ausgaben muß in den Tabellen gesucht werden. Wird das Zeichen im aktuellen Zustand nicht gefunden, wird in den jeweils anderen umgeschaltet. Ist das Zeichen auch jetzt nicht zu finden – was natürlich auftreten könnte, wenn solche ASCII-Zeichen ausgegeben werden sollen, für die kein Fernschreibäquivalent vorhanden ist –, so wird das (nicht definierte) Fernschreibzeichen 32 ausgeschrieben, also *nichts*!

Der Einsatz der Schaltkreise SIO und CTC ist im **Bild 11.9** dargestellt.

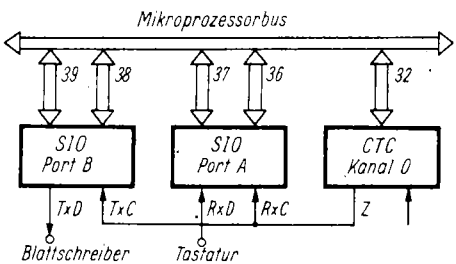


Bild 11.9. Einsatz der Peripherieschaltkreise im Programm 11.10

11.6. Spiele

11.6.1. „Mein Gegner verliert immer" (Programm 11.11)

Das Spielprogramm bringt nach dem Start eine Erläuterung der Regeln. Der Algorithmus (Zeile 1540) geht davon aus, daß der Mitspieler zuerst zieht. Der Computer bringt dann stets einen sol-

chen Komplementzug, der einen Gewinn des Mitspielers unmöglich macht. Es handelt sich hier also um kein faires Spiel, dafür ist aber der Entscheidungsalgorithmus einfach.

Programm 11.11. Spiel: Mein Gegner verliert immer

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"          MEIN GEGNER VERLIERT IMMER           "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM +++++++++++++++++ SPIELANLEITUNG +++++++++++++++++
1080 :
1090 PRINT "Wir beide spielen mit Zahlen."
1100 PRINT "Jeder hat 8 Karten mit den Zahlen 2 bis 9."
1110 PRINT "Er kann sie in beliebiger Folge aufdecken."
1120 PRINT "Es wird die laufende Summe gebildet."
1130 PRINT "Wer diese Summe auf  u e b e r  8 0  bringt,"
1140 PRINT "der hat das Spiel  v e r l o r e n ."
1150 :
1160 REM ++++++++++++++++ INITIALISIERUNG +++++++++++++++++
1170 :
1180 DIM K(9) : REM  Kartenstapel des Spielers
1190 FOR I=2 TO 9
1200   LET K(I) = 1 : REM verdeckte Karten des Spielers
1210 NEXT I
1220 LET S = 0 : REM laufende Summe
1230 LET Z = 0 : REM Zaehler fuer Spielerzahlen 2, 3 und 4
1240 :
1250 REM +++++++++++++++++ NAECHSTE RUNDE +++++++++++++++++
1260 :
1270 REM --------------- SPIELER DECKT AUF ----------------
1280 :
1290 PRINT
1300 PRINT "Welche Zahl deckst du auf? " ;
1310 LET I = VAL(INPUT$(1))
1320 PRINT I ;
1330 IF I>=2 AND I<= 9 THEN GOTO 1390
1340   PRINT
1350   PRINT TAB(5) ; "Du schwindelst! Eine" ; I ;
1360   PRINT "kannst du nicht haben."
1370   GOTO 1300 : REM Wiederholung
1380 :
1390 IF K(I) = 1 THEN GOTO 1450
1400   PRINT
1410   PRINT TAB(5) ; "Du schlaefst! Die" ; I ;
1420   PRINT "hast du schon aufgedeckt."
1430   GOTO 1300 : REM Wiederholung
1440 :
1450 LET K(I) = 0
1460 LET S = S+I
1470 PRINT "--> Summe =" ; S
1480 IF S<=80 THEN GOTO 1550
1490 PRINT
1500   PRINT TAB(6) ;"D u  h a s t  v e r l o r e n !!!"
1510   GOTO 1640 : REM -> Ende
1520 :
1530 REM --------------- RECHNER DECKT AUF ----------------
1540 :
1550 IF I>4 THEN LET J = 11-I ELSE LET Z = Z+1:LET J = 10-Z
1560 PRINT "Ich decke die Zahl" ; J ; "auf." ;
1570 LET S = S+J
1580 PRINT "      --> Summe =" ; S
1590 IF S<=80 THEN GOTO 1290 : REM naechste Runde
1600 PRINT "Du hast gewonnen."
1610 :
1620 REM +++++++++++++++ ABSCHLUSS DES SPIELS +++++++++++++++
1630 :
```

```
1640 PRINT
1650 PRINT "Willst du weiterspielen (J|-)" ;
1660 LET A¤ = INPUT¤(1)
1670 PRINT
1680 IF A¤="J" OR A¤="j" THEN GOTO 1190
1690 :
1700 END
1710 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

11.6.2. „Theseus im Labyrinth" (Programm 11.12)

Das Programm realisiert ein *Zufallsspiel.* Nach Aufforderung hat der Bediener den Zufallszahlengenerator zu initialisieren und den gewünschten Aufbau des Labyrinths einzugeben; die Aus-

Programm 11.12. Spiel: *Theseus* im Labyrinth

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"              THESEUS IM LABYRINTH               "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 REM ++++++++++++++++ INITIALISIERUNG ++++++++++++++++
1080 :
1090 REM -------------------- ABFRAGE ---------------------
1100 :
1110 PRINT "Abmessungen des Labyrinths:"
1120 INPUT "Breite" ; B
1130 INPUT "Hoehe" ; H
1140 DIM L(H,B)
1150 PRINT
1160 PRINT "Gib Aufbau des Labyrinths ein!"
1170 PRINT "Waende durch ooooooooooo andeuten."
1180 PRINT "Freie Stellen mit ........ fuellen."
1190 PRINT "Ausgang in oberster Zeile vorsehen."
1200 PRINT "Startpunkt:  *"
1210 :
1220 REM ----------------- ZUFALLSZAHLEN ------------------
1230 :
1240 PRINT "Wenn Du beginnen willst, druecke Leertaste!"
1250 LET X = -32767
1260 IF X=32767 THEN GOTO 1250
1270 LET X = X+1
1280 IF INKEY¤="" THEN GOTO 1260
1290 RANDOMIZE(X)
1300 :
1310 REM -------------- BILDSCHIRM LOESCHEN ---------------
1320 :
1330 PRINT CHR¤(12)
1340 :
1350 REM -------------- LABYRINTH EINGEBEN ----------------
1360 :
1370 FOR Z=0 TO H-1
1380   FOR P=0 TO B-1
1390     LET C¤ = INPUT¤(1)
1400     GOSUB 2450 : REM Ausschreiben
1410     IF C¤ = "." THEN LET L(Z,P) = 0 : GOTO 1460
1420       IF C¤ <> "*" THEN LET L(Z,P) = -1 : GOTO 1460
1430         LET L(Z,P) = 1 : REM Startpunkt merken
1440         LET Z1 = Z
1450         LET P1 = P
1460   NEXT P
1470 NEXT Z
1480 :
1490 LET Z = Z1 : REM Startpunkt zurueckschreiben
1500 LET P = P1
1510 GOSUB 2260 : REM Tastendruck abwarten
1520 :
```

```
1530 REM +++++++++++++++ THESEUS ENTFLIEHT +++++++++++++++++
1540 :
1550 REM ---------------- RICHTUNG SUCHEN -----------------
1560 :
1570 LET R = INT(8*RND) + 1
1580 LET I = 0
1590 GOSUB 2130 : REM neue Position Z1,P1 berechnen
1600 :
1610 LET R0 = L(Z1,P1)
1620 IF R0<0 THEN GOTO 1570 : REM Wand
1630   IF R0>0 THEN GOTO 1860 : REM bereits betreten
1640 :
1650 LET L(Z1,P1) = R
1660      REM Richtung merken, in der Z1,P1 erreicht wurde
1670 :
1680 LET Z = Z1 : REM neue Position eintragen
1690 LET P = P1
1700 LET C$ = "*"
1710 GOSUB 2450 : REM Schreiben
1720 GOSUB 2380 : REM Warten
1730 :
1740 IF Z<>0 THEN GOTO 1800 : REM Ausgang erreicht ?
1750   GOSUB 2260 : REM Tastendruck abwarten
1760   PRINT CHR$(12)
1770      REM Bildschirm loeschen, Schreibmarke setzen
1780   END
1790 :
1800 LET I = I+1
1810 IF I < 10 THEN GOTO 1590 : REM geradeaus weiter
1820 GOTO 1570 : REM neue Richtung probieren
1830 :
1840 REM ---------- SCHLEIFE ! -> IRRWEG ZURUECK ----------
1850 :
1860 LET Z2 = Z1 : REM Anfangspunkt der Schleife merken
1870 LET P2 = P1
1880 :
1890 LET R = L(Z,P) : REM Richtung im letzten Punkt
1900 LET L(Z,P) = 0 : REM letzten Punkt loeschen
1910 LET C$ = " "
1920 GOSUB 2450 : REM Schreiben
1930 GOSUB 2380 : REM Warten
1940 :
1950 IF R>4 THEN LET R = R-4 ELSE LET R = R+4
1960 GOSUB 2130 : REM auf vorhergehenden Punkt zurueck
1970 :
1980 LET Z = Z1
1990 LET P = P1
2000 :
2010 IF Z1=Z2 AND P1=P2 THEN GOTO 1570 : REM Anfang erreicht
2020   LET R = L(Z1,P1) : REM Richtung im vorhergeh. Punkt
2030   LET L(Z1,P1) = 0 : REM vorhergehenden Punkt loeschen
2040   LET C$ = " "
2050   GOSUB 2450 : REM Schreiben
2060   GOSUB 2380 : REM Warten
2070 GOTO 1950 : REM Schleife weiter zurueck
2080 :
2090 REM +++++++++++++++++ HILFSPROGRAMME +++++++++++++++++
2100 :
2110 REM -------- BERECHNUNG DES NAECHSTEN PUNKTS ---------
2120 :
2130 ON R GOTO 2150,2160,2170,2180,2190,2200,2210,2220
2140 :
2150 LET Z1 = Z   : LET P1 = P+1 : RETURN
2160 LET Z1 = Z-1 : LET P1 = P+1 : RETURN
2170 LET Z1 = Z-1 : LET P1 = P   : RETURN
2180 LET Z1 = Z-1 : LET P1 = P-1 : RETURN
2190 LET Z1 = Z   : LET P1 = P-1 : RETURN
2200 LET Z1 = Z+1 : LET P1 = P-1 : RETURN
2210 LET Z1 = Z+1 : LET P1 = P   : RETURN
2220 LET Z1 = Z+1 : LET P1 = P+1 : RETURN
2230 :
```

```
2240 REM -------------------- BLINKEN ---------------------
2250 :
2260 LET C¤ = " "
2270 GOSUB 2450 : REM Schreiben
2280 GOSUB 2380 : REM Warten
2290 LET C¤ = "*"
2300 GOSUB 2450 : REM Schreiben
2310 GOSUB 2380 : REM Warten
2320 LET S¤ = INKEY¤
2330 IF S¤="" THEN GOTO 2260
2340 RETURN
2350 :
2360 REM --------------------- WARTEN ---------------------
2370 :
2380 FOR I=1 TO 100 : NEXT
2390 RETURN
2400 :
2410 REM ------------- AUSGABE AUF BILDSCHIRM -------------
2420 :
2430 REM Z=Zeile, P=Position, C¤=Zeichen
2440 :
2450 POKE &HF800 + 80*Z + P, ASC(C¤)
2460 RETURN
2470 :
2480 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

gänge müssen dabei in der obersten Zeile sein. Außerdem ist der Anfangsstandpunkt von *Theseus* zu markieren.

Nach dem Start durch den Bediener (Druck auf eine beliebige Taste) versucht *Theseus* zu entfliehen. Er geht dazu in eine zufällig ausgewählte Richtung R.

Zur *Steuerung* dient das zweidimensionale Feld L (Labyrinth). Die Werte der Elemente haben folgende Bedeutung:

L(Z,P) = − 1 : Wand
L(Z,P) = 0 : frei, noch nicht betreten
L(Z,P) = 1 . . . 8 : Richtung, in der (Z,P) bereits zuvor erreicht wurde.

- Stößt *Theseus* in der angegebenen Richtung auf kein Hindernis, so geht er einen Schritt und schreibt am neuen Punkt (Z1,P1) die Richtung auf, in der er diese Stelle erreichte. Anschließend geht er bis zu zehn Schritte geradeaus weiter, wobei er ebenfalls die Richtung in L(Z1,P1) einträgt.
- Trifft er auf eine Wand oder ist er bereits zehn Schritte ohne Widerstand gelaufen, so wählt er eine neue Richtung.
- Stößt er auf seine eigene Spur, so läuft er von dieser Stelle (Z2,P2) den zuletzt gegangenen Weg zurück, bis er wieder an diesen Punkt kommt. Dann wählt er eine andere Richtung.
  Um den Rückweg finden zu können, benutzt er die ursprünglich notierten Richtungen und löscht sie. (Dadurch geht er allerdings evtl. erneut Irrwege!)

Hat *Theseus* den Ausgang gefunden, so kann das Spiel durch einen Tastendruck abgebrochen werden.

Damit sollen die Programmbeispiele abgeschlossen werden.

Viel Freude und Erfolg bei der eigenen Arbeit mit BASIC!

Anhang

Programm A.1. Druck einer Konvertierungstafel

```
1000 REM ************************************************
1010 PRINT
1020 PRINT"                DRUCK DER TAFEL A.1                "
1030 PRINT
1040 REM ************************************************
1050 :
1060 :
1070 DIM A¤(33)
1080 FOR K=0 TO 33
1090   READ A¤(K)
1100 NEXT K
1110 FOR K=0 TO 3
1120   PRINT USING "Formular fuer Seite # einspannen!"; K+1
1130   INPUT "Zeilenendetaste druecken!" , A
1140   GOSUB 1230 : REM Kopf
1150   GOSUB 1320 : REM Rumpf
1160 NEXT K
1170 PRINT "Ausgabe beendet."
1180 PRINT
1190 END
1200 :
1210 REM ------------------ TABELLENKOPF ------------------
1220 :
1230 GOSUB 1540 : REM Strich
1240 GOSUB 1610 : REM Leerzeile
1250 GOSUB 1680 : REM Kopfzeile
1260 GOSUB 1610 : REM Leerzeile
1270 GOSUB 1540 : REM Strich
1280 RETURN
1290 :
1300 REM -----------------  TABELLENRUMPF ------------------
1310 :
1320 FOR J=64*K TO 64*K+31
1330   IF INT(J/8)=J/8 THEN GOSUB 1610 : REM Leerzeile
1340   GOSUB 1440 : REM Tabellenzeile
1350 NEXT J
1360 GOSUB 1610 : REM Leerzeile
1370 GOSUB 1540 : REM Strich
1380 RETURN
1390 :
1400 REM +++++++++++++++ AUSGABEPROGRAMME +++++++++++++++
1410 :
1420 REM ------------------ TABELLENZEILE ------------------
1430 :
1440 LET Z¤ = "| " : REM Anfangswert der Zeile
1450 LET D = J
1460 GOSUB 1730 : REM Anfuegen des vorderen Teils
1470 LET D = J+32
1480 GOSUB 1730 : REM Anfuegen des hinteren Teils
1490 LPRINT Z¤
1500 RETURN
1510 :
1520 REM ---------------------- STRICH ---------------------
1530 :
1540 LET M¤ = "|"+ STRING¤(10,"-") +"|"+ STRING¤(4,"-")
1550 LET M¤ = M¤ +"|"+ STRING¤(5,"-") +"|"+ STRING¤(5,"-")
1560 LPRINT M¤ ; M¤ ; "|"
1570 RETURN
1580 :
```

```
1590 REM ------------------- LEERZEILE --------------------
1600 :
1610 LET M¤ = "|"+ SPACE¤(10) +"|"+ SPACE¤(4)
1620 LET M¤ = M¤ +"|"+ SPACE¤(5) +"|"+ SPACE¤(5)
1630 LPRINT M¤ ; M¤ ; "|"
1640 RETURN
1650 :
1660 REM -------------- SPALTENUEBERSCHRIFT ---------------
1670 :
1680 LET M¤ = "|  BINAER  | HX | DEZ |ASCII"
1690 LPRINT M¤ + M¤ + "|"
1700 RETURN
1710 REM ------- ANFUEGEN EINER HALBEN TABELLENZEILE ------
1720 :
1730 GOSUB 1840 : REM  D -> B¤
1740 GOSUB 1940 : REM  D -> H¤
1750 GOSUB 2060 : REM  D -> D¤
1760 GOSUB 2120 : REM  D -> C¤
1770 LET Z¤ = Z¤ + B¤+" | "+H¤+" |" +D¤+" | "+C¤+" | "
1780 RETURN
1790 :
1800 REM ++++++++++++ KONVERTIERUNGSPROGRAMME ++++++++++++
1810 :
1820 REM ------------ DEZIMALZAHL -> BINAERTEXT -----------
1830 :
1840 LET B¤ = "" : REM Anfangswert Binaertext
1850 LET M = 1   : REM Maske
1860 FOR I=1 TO 8
1870   LET B¤ = MID¤(STR¤(-((D AND M)=M)),2) + B¤
1880   LET M = 2*M
1890 NEXT I
1900 RETURN
1910 :
1920 REM --------- DEZIMALZAHL -> HEXADEZIMALTEXT ---------
1930 :
1940 LET N = D
1950 LET H¤ = ""       : REM Anfangswert Hexadezimaltext
1960 LET I = INT(N/16) : REM ganzzahliger Anteil
1970 LET R = N - 16*I  : REM Rest
1980 IF R<=9 THEN LET C = 48 ELSE LET C = 55
1990 LET H¤ = CHR¤(R+C) + H¤
2000 IF I<>0 THEN LET N = I : GOTO 1960
2010 LET H¤ = SPACE¤(2-LEN(H¤)) + H¤
2020 RETURN
2030 :
2040 REM ----------- DEZIMALZAHL -> DEZIMALTEXT -----------
2050 :
2060 LET D¤ = STR¤(D)
2070 LET D¤ = SPACE¤(4-LEN(D¤)) + D¤
2080 RETURN
2090 :
2100 REM ----------- DEZIMALZAHL -> ASCII-TEXT ------------
2110 :
2120 IF D<33 OR D=127 THEN GOTO 2160
2130   IF D<128 THEN LET C¤ = CHR¤(D) ELSE C¤ = " "
2140   LET C¤ = " " + C¤ + " "
2150   RETURN
2160 IF D=127 THEN LET D = 33
2170 LET C¤ = A¤(D)
2180 RETURN
2190 :
2200 REM -------------- ASCII-STEUERZEICHEN ---------------
2210 :
2220 DATA  NUL , SOH , STX , ETX , EOT , ENQ , ACK , BEL
2230 DATA "BS ","HT ","LF ","VT ","FF ","CR ","SO ","SI "
2240 DATA  DLE , DC1 , DC2 , DC3 , DC4 , NAK , SYN , ETB
2250 DATA  CAN ,"EM ", SUB , ESC ,"FS ","GS ","RS ","US "
2260 DATA "SP ", DEL
2270 :
2280 REM XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Tafel A.1. Konvertierungstafel

BINAER	HX	DEZ	ASCII	BINAER	HX	DEZ	ASCII
00000000	0	0	NUL	00100000	20	32	SP
00000001	1	1	SOH	00100001	21	33	!
00000010	2	2	STX	00100010	22	34	"
00000011	3	3	ETX	00100011	23	35	#
00000100	4	4	EOT	00100100	24	36	¤
00000101	5	5	ENQ	00100101	25	37	%
00000110	6	6	ACK	00100110	26	38	&
00000111	7	7	BEL	00100111	27	39	'
00001000	8	8	BS	00101000	28	40	(
00001001	9	9	HT	00101001	29	41	)
00001010	A	10	LF	00101010	2A	42	*
00001011	B	11	VT	00101011	2B	43	+
00001100	C	12	FF	00101100	2C	44	,
00001101	D	13	CR	00101101	2D	45	-
00001110	E	14	SO	00101110	2E	46	.
00001111	F	15	SI	00101111	2F	47	/
00010000	10	16	DLE	00110000	30	48	0
00010001	11	17	DC1	00110001	31	49	1
00010010	12	18	DC2	00110010	32	50	2
00010011	13	19	DC3	00110011	33	51	3
00010100	14	20	DC4	00110100	34	52	4
00010101	15	21	NAK	00110101	35	53	5
00010110	16	22	SYN	00110110	36	54	6
00010111	17	23	ETB	00110111	37	55	7
00011000	18	24	CAN	00111000	38	56	8
00011001	19	25	EM	00111001	39	57	9
00011010	1A	26	SUB	00111010	3A	58	:
00011011	1B	27	ESC	00111011	3B	59	;
00011100	1C	28	FS	00111100	3C	60	<
00011101	1D	29	GS	00111101	3D	61	=
00011110	1E	30	RS	00111110	3E	62	>
00011111	1F	31	US	00111111	3F	63	?

Tafel A.1. (Fortsetzung)

BINAER	HX	DEZ	ASCII	BINAER	HX	DEZ	ASCII
01000000	40	64	@	01100000	60	96	
01000001	41	65	A	01100001	61	97	a
01000010	42	66	B	01100010	62	98	b
01000011	43	67	C	01100011	63	99	c
01000100	44	68	D	01100100	64	100	d
01000101	45	69	E	01100101	65	101	e
01000110	46	70	F	01100110	66	102	f
01000111	47	71	G	01100111	67	103	g
01001000	48	72	H	01101000	68	104	h
01001001	49	73	I	01101001	69	105	i
01001010	4A	74	J	01101010	6A	106	j
01001011	4B	75	K	01101011	6B	107	k
01001100	4C	76	L	01101100	6C	108	l
01001101	4D	77	M	01101101	6D	109	m
01001110	4E	78	N	01101110	6E	110	n
01001111	4F	79	O	01101111	6F	111	o
01010000	50	80	P	01110000	70	112	p
01010001	51	81	Q	01110001	71	113	q
01010010	52	82	R	01110010	72	114	r
01010011	53	83	S	01110011	73	115	s
01010100	54	84	T	01110100	74	116	t
01010101	55	85	U	01110101	75	117	u
01010110	56	86	V	01110110	76	118	v
01010111	57	87	W	01110111	77	119	w
01011000	58	88	X	01111000	78	120	x
01011001	59	89	Y	01111001	79	121	y
01011010	5A	90	Z	01111010	7A	122	z
01011011	5B	91	[	01111011	7B	123	{
01011100	5C	92	\	01111100	7C	124	|
01011101	5D	93	]	01111101	7D	125	}
01011110	5E	94	^	01111110	7E	126	~
01011111	5F	95	_	01111111	7F	127	DEL

Tafel A.1. (Fortsetzung)

BINAER	HX	DEZ	ASCII	BINAER	HX	DEZ	ASCII
10000000	80	128		10100000	A0	160	
10000001	81	129		10100001	A1	161	
10000010	82	130		10100010	A2	162	
10000011	83	131		10100011	A3	163	
10000100	84	132		10100100	A4	164	
10000101	85	133		10100101	A5	165	
10000110	86	134		10100110	A6	166	
10000111	87	135		10100111	A7	167	
10001000	88	136		10101000	A8	168	
10001001	89	137		10101001	A9	169	
10001010	8A	138		10101010	AA	170	
10001011	8B	139		10101011	AB	171	
10001100	8C	140		10101100	AC	172	
10001101	8D	141		10101101	AD	173	
10001110	8E	142		10101110	AE	174	
10001111	8F	143		10101111	AF	175	
10010000	90	144		10110000	B0	176	
10010001	91	145		10110001	B1	177	
10010010	92	146		10110010	B2	178	
10010011	93	147		10110011	B3	179	
10010100	94	148		10110100	B4	180	
10010101	95	149		10110101	B5	181	
10010110	96	150		10110110	B6	182	
10010111	97	151		10110111	B7	183	
10011000	98	152		10111000	B8	184	
10011001	99	153		10111001	B9	185	
10011010	9A	154		10111010	BA	186	
10011011	9B	155		10111011	BB	187	
10011100	9C	156		10111100	BC	188	
10011101	9D	157		10111101	BD	189	
10011110	9E	158		10111110	BE	190	
10011111	9F	159		10111111	BF	191	

Tafel A.1. (Fortsetzung)

BINAER	HX	DEZ	ASCII	BINAER	HX	DEZ	ASCII
11000000	C0	192		11100000	E0	224	
11000001	C1	193		11100001	E1	225	
11000010	C2	194		11100010	E2	226	
11000011	C3	195		11100011	E3	227	
11000100	C4	196		11100100	E4	228	
11000101	C5	197		11100101	E5	229	
11000110	C6	198		11100110	E6	230	
11000111	C7	199		11100111	E7	231	
11001000	C8	200		11101000	E8	232	
11001001	C9	201		11101001	E9	233	
11001010	CA	202		11101010	EA	234	
11001011	CB	203		11101011	EB	235	
11001100	CC	204		11101100	EC	236	
11001101	CD	205		11101101	ED	237	
11001110	CE	206		11101110	EE	238	
11001111	CF	207		11101111	EF	239	
11010000	D0	208		11110000	F0	240	
11010001	D1	209		11110001	F1	241	
11010010	D2	210		11110010	F2	242	
11010011	D3	211		11110011	F3	243	
11010100	D4	212		11110100	F4	244	
11010101	D5	213		11110101	F5	245	
11010110	D6	214		11110110	F6	246	
11010111	D7	215		11110111	F7	247	
11011000	D8	216		11111000	F8	248	
11011001	D9	217		11111001	F9	249	
11011010	DA	218		11111010	FA	250	
11011011	DB	219		11111011	FB	251	
11011100	DC	220		11111100	FC	252	
11011101	DD	221		11111101	FD	253	
11011110	DE	222		11111110	FE	254	
11011111	DF	223		11111111	FF	255	

Tafel A.2. ASCII-Steuerzeichen

Kode	Tasten	Kurzzeichen	Benennung	Bedeutung
0		NUL	NULL	Füllzeichen ohne Bedeutung
1	CTRL-A	SOH	Start Of Heading	Anfang des Titels einer Mitteilung
2	CTRL-B	STX	Start of TeXt	Ende des Titels, Anfang des Textes
3	CTRL-C	ETX	End of TeXt	Ende des Textes einer Mitteilung
4	CTRL-D	EOT	End Of Transmission	Ende der Übertragung von Mitteilungen
5	CTRL-E	ENQ	ENQuiry	Anforderung einer Antwort
6	CTRL-F	ACK	ACKnowledge	positive Rückmeldung
7	CTRL-G	BEL	BELL	akustisches Signal
8	CTRL-H	BS	Back-Space	Rückwärtsschritt
9	CTRL-I	HT	Horizontal Tabulation	Horizontaltabulator
10	CTRL-J	LF	Line Feed	Zeilenvorschub
11	CTRL-K	VT	Vertical Tabulation	Vertikaltabulator
12	CTRL-L	FF	Form Feed	Seitenvorschub
13	CTRL-M	CR	Carriage Return	Zeilenrücklauf
14	CTRL-N	SO	Shift Off	Dauerumschaltung
15	CTRL-O	SI	Shift In	Rückschaltung
16	CTRL-P	DLE	Data Link Escape	Datenübertragungsumschaltung
17	CTRL-Q	DC1	Device Control 1	Gerätesteuerung (Grundzustand)
18	CTRL-R	DC2	Device Control 2	Gerätesteuerung (Sonderzustand)
19	CTRL-S	DC3	Device Control 3	Gerätesteuerung (Stop)
20	CTRL-T	DC4	Device Control 4	Gerätesteuerung (Stop)
21	CTRL-U	NAK	Negative AcKnowledge	negative Rückantwort
22	CTRL-V	SYN	SYNchroneous idle	Synchronisierungszeichen
23	CTRL-W	ETB	End of Transmission Block	Ende des Datenübertragungsblocks
24	CTRL-X	CAN	CANcel	ungültig
25	CTRL-Y	EM	End of Medium	Ende der Aufzeichnung (des Datenträgers)
26	CTRL-Z	SUB	SUBstitution	Ersatzzeichen für falsche Zeichen
27		ESC	ESCape	Kodeumschaltung
28		FS	File Separator	Dateitrennzeichen
29		GS	Group Separator	Datengruppentrennzeichen
30		RS	Record Separator	Datensatztrennzeichen
31		US	Unit Separator	Datenelementtrennzeichen
32		SP	SPace	Leerschritt
127		DEL	DELete	Löschen

Tafel A.3. ASCII-Sonderzeichen

Textzeichen	Bezeichnung (Bedeutung)	Abschnitt
!	Ausrufungszeichen	4.1.5
"	Anführungszeichen	3.3.4
#	Nummernzeichen	4.1.5
$ oder ¤	Zeichen für Geldeinheit	3.3.5
%	Prozentzeichen	–
&	kommerzielles Und-Zeichen	4.1.5
'	Apostroph	–
(	öffnende runde Klammer	5.1.2
)	schließende runde Klammer	5.1.2
*	Stern = Multiplikationszeichen	5.1.1
+	Pluszeichen = Additionszeichen	3.3.3, 5.1.1
,	Komma	4.1.3
-	Minuszeichen = Subtraktionszeichen	3.3.3, 5.1.1
.	Punkt = Dezimalpunkt	3.3.3
/	Schrägstrich = Divisionszeichen	5.1.1
:	Doppelpunkt	3.2.2
;	Semikolon	4.1.3
<	Kleiner-Zeichen	5.3.1
=	Gleichheitszeichen	4.2.1, 5.3.1
>	Größer-Zeichen	5.3.1
?	Fragezeichen	4.1.1
@ oder §	kommerzielles A-Zeichen bzw. Paragraphenzeichen	–
[oder Ä	öffnende eckige Klammer (bzw. Ä)	
\ oder Ö	inverser Schrägstrich = Ganzzahl-Divisionszeichen (bzw. Ö)	4.1.5, 5.1.1
] oder Ü	schließende eckige Klammer (bzw. Ü)	
^ oder ↑	Zirkumflex bzw. Aufwärtspfeil = Potenzierungszeichen	5.1.1
_ oder ←	Unterstreichungszeichen bzw. Rückwärtspfeil	
`	Gravis	
{ oder ä	öffnende geschweifte Klammer (bzw. ä)	
| oder ö	senkrechter Strich (bzw. ö)	
} oder ü	schließende geschweifte Klammer (bzw. ü)	
~ oder ~ oder ß	Überstreichungszeichen oder Tilde (bzw. ß)	

Tafel A.4. BASIC-Steuerzeichen

ASCII	Tastatur	Bedeutung	Abschnitt
	CTRL-A	**Again:** Übergang zum Editiermodus für die zuletzt geschriebene Zeile	3.4.4
	CTRL-C	**Control:** Abbruch des laufenden Programms und Übergabe der Steuerung an den Bediener	3.4.1, 3.4.2, 3.5.2
BEL	CTRL-G	**BELL:** Glocke/Summer (soweit vorhanden)	
BS	CTRL-H	**Back-Space:** Löschen des zuletzt eingegebenen Zeichens im Puffer und auf dem Bildschirm	3.2.1
HT	CTRL-I	**Horizontal Tabulation:** Sprung der Schreibmarke zur nächsten Tabulatorposition	
	CTRL-O	**Output:** Aussetzen der Programmausgaben bis zur erneuten Eingabe von CTRL-O	
	CTRL-Q	**Continue:** Fortsetzen eines mit CTRL-S gestoppten Programms	
	CTRL-R	**Repeat:** erneute Ausgabe der zuletzt bearbeiteten Zeile	
	CTRL-S	**Stop:** operatives Anhalten des laufenden Programms ohne Übergang in den Kommandomodus	
NAK	CTRL-U	**Negative AcKnowledge:** Löschen der zuletzt eingegebenen Zeile	3.2.1
CAN	CTRL-X	**CANcel:** wie NAK	3.2.1
ESC		**ESCape:** Verlassen des Editiermodus	3.4.4
DEL		**DELete:** Löschen des zuletzt eingegebenen Zeichens im Puffer	3.2.1

Tafel A.5. BASIC-Systemanweisungen

Editierung		Programmlauf		Dateiverwaltung	
AUTO	3.4.1	CLEAR	3.5.1	FILES	9.1
DELETE	3.4.3	CONT	3.5.2	KILL	9.1
EDIT	3.4.4	RUN	3.5.1	LOAD	9.2
LIST	3.4.2	SYSTEM	3.2.1	MERGE	9.2
LLIST	3.4.2	TRON	3.6.2	NAME	9.1
NEW	3.4.1	TROFF	3.6.2		
RENUM	3.4.3				

Tafel A.6. Schlüsselwörter der BASIC-Sprachanweisungen

Programmstrukturen Datenstrukturen		Eingaben Ausgaben		Sonstige Anweisungen	
DEF FN	6.3.2	CLOSE	9.3.3	CALL	10.5.2
DIM	7.1.1	DATA	4.2.2	ERROR	3.6.3
ELSE	6.1.1, 6.1.2	INPUT	4.3.1, 4.3.2, 9.3.5	ON ERROR GOTO	3.6.3
END	3.2.4	LET	4.2.1	OUT	10.4.1
FOR	6.2.3	LINE INPUT	4.3.3, 9.3.5	POKE	10.3.2
GOSUB	6.3.1	LOAD	9.2	RANDOMIZE	5.2.5
GOTO	6	OPEN	9.3.2	REM	3.2.4
IF	6.1.1, 6.1.2	PRINT	4.1.1–4.1.4, 9.3.4	RESUME	3.6.3
NEXT	6.2.3	PRINT USING	4.1.5	STOP	3.5.2
ON	6.1.3, 6.3.1	READ	4.2.2	WAIT	10.4.3
RETURN	6.3.1	RESET	9.3.3		
STEP	6.2.3	WIDTH	4.1.6		
THEN	6.1.1, 6.1.2	WIDTH LPRINT	4.1.6		
TO	6.2.3	WRITE	9.3.4		

Tafel A.7. BASIC-Standardfunktionen

Arithmetische Funktionen		Textfunktionen		Sonstige Funktionen	
ABS	5.2.1	ASC	8.2.2	ERL	3.6.3
ATN	5.2.4	CHR□	8.2.2	ERR	3.6.3
COS	5.2.4	INSTR	8.1.1	FRE	3.4.2
EXP	5.2.3	LEFT□	8.1.2	INKEY□	4.3.5
INT	5.2.1	LEN	8.1.1	INP	10.4.2
LOG	5.2.3	MID□	8.1.2	INPUT□	4.3.4
RND	5.2.5	RIGHT□	8.1.2	PEEK	10.3.1
SGN	5.2.1	SPACE□	8.1.4	POS	4.1.4
SIN	5.2.4	STR□	8.2.1	SPC	4.1.4
SQR	5.2.2	STRING□	8.1.4	TAB	4.1.4
TAN	5.2.4	VAL	8.2.1		

Tafel A.8. Operatoren in BASIC-Ausdrücken

Rang	Symbol	Bedeutung	Abschnitt
12	( )	Klammerung	5.1.2
11	−	negatives Vorzeichen	3.3.3
10	^ oder ↑ oder **	Potenzierung	5.1.1
9	*	Multiplikation	5.1.1
	/	Division	5.1.1
8	\	ganzzahlige Division	5.1.1
7	MOD	Modulo-Rechnung	5.1.1
6	+	Addition	5.1.1
		Textverkettung	8.1.3
	−	Subtraktion	5.1.1
5	= < > <> >= <=	Vergleiche	5.3.1
4	NOT	Negation	5.3.2
3	AND	Konjunktion	5.3.2
2	OR	Disjunktion	5.3.2
1	XOR	Antivalenz	5.3.2

Tafel A.9. Beispiele für BASIC-Fehlerkodes

Nummer	Kode	Ausschrift	Bedeutung
1	NF	NEXT without FOR	NEXT ohne FOR
2	SN	Syntax error	syntaktischer Fehler
3	RG	RETURN without GOSUB	RETURN ohne GOSUB
4	OD	Out of DATA	Ende der DATA-Datei überschritten
5	FC	Illegal function call	unzulässiger Funktionsaufruf
6	OV	Overflow	Überlauf des Zahlenbereichs (Resultat zu groß)
7	OM	Out of memory	Überlauf des Arbeitsspeichers
8	UL	Undefined line	nicht definierte Zeilennummer
9	BS	Bad subscript *auch:* Subscript out of range	unzulässiger Index (Wert oder Anzahl falsch)
10	DD	Doubly dimensioned array *auch:* Duplicate definition	doppelt dimensioniertes (definiertes) Feld
11	/0	Division by zero	Division durch Null
12	ID	Illegal direct	unzulässiger Kommandomodus
13	TM	Type mismatch	falscher Typ
14	OS	Out of string space	Überlauf des Zeichenkettenspeichers
15	LS	String too long	zu lange Zeichenkette
16	ST	String formula too complex	zu komplexer Textausdruck
17	CN	Can't continue	Fortsetzung nicht möglich
18	UF	Undefined user function	nicht definierte USR-Funktion
19	NR	No RESUME	RESUME fehlt
20	RW	RESUME without error	RESUME ohne Auftreten eines Fehlers
21	UE	Unprintable error	nicht näher differenzierbarer Fehler
22	MO	Missing operand	fehlender Operand in einem Ausdruck
23	LO	Line buffer overflow	Überlauf des Eingabepuffers
26	FN	FOR without NEXT	FOR ohne NEXT
–	–	? REDO from Start	falsche Eingabe (vollständig wiederholen!)

Literaturverzeichnis

Abschnitt 1

[1.1] *Franke, K.:* Einführung in die Mikrorechentechnik. Berlin: VEB Verlag Technik 1984

[1.2] *Kieser, H.; Meder, M.:* Mikroprozessortechnik. Aufbau und Anwendung des Mikroprozessorsystems U880. 3. Aufl. Berlin: VEB Verlag Technik 1985

[1.3] *Krauß, M.; Kutschbach, E.; Woschni, E.-G.:* Handbuch der Datenerfassung. 2. Aufl. Berlin: VEB Verlag Technik 1985

[1.4] *Matschke, J.:* Von der einfachen Logikschaltung zum Mikrorechner. 3. Aufl. Berlin: VEB Verlag Technik 1986

[1.5] *Schwarz, W.; Meyer, G.; Eckardt, D.:* Mikrorechner. Wirkungsweise – Programmierung – Applikation. 3. Aufl. Berlin: VEB Verlag Technik 1984

Abschnitt 2

[2.1] *Bachmann, P., u. a.:* Programmsysteme. Anwendung – Entwicklung – Fundierung. Berlin: Akademie-Verlag 1983

[2.2] *Engelien, M.; Stahn, H.:* Software-Engineering. ARS-Technologie. Berlin: Akademie-Verlag 1984

[2.3] *Horn, E.; Baumbach, H.-D.; Straach, C.:* Software-Technologie für Mikrorechner. Berlin: Verlag Die Wirtschaft 1982

[2.4] *Myers, G. J.:* Methodisches Testen von Programmen. München, Wien: R. Oldenbourg Verlag 1982

[2.5] *Schumann, J.; Gerisch, M.:* Software-Entwurf. Prinzipien – Methoden – Arbeitsschritte – Rechnerunterstützung. Berlin: VEB Verlag Technik 1984

[2.6] *Werner, D.:* Programmierung von Mikrorechnern. Programmsysteme – Parallele Prozesse – Echtzeitbetriebssysteme. 2. Aufl. Berlin: VEB Verlag Technik 1986

Abschnitt 3

[3.1] American National Standard for the Programming Language Minimal BASIC. ANSI X3.60–1978. New York: ANSI 1978

[3.2] *Brain, D. A.; Oviate, P. R.; Paquin, P. J. A.; Stone jr., C. D.:* BASIC-Dialekte im Vergleich. Wie man Apple-, Commodore- und TRS-80-Programme untereinander konvertiert. Haar bei München: Verlag Markt & Technik 1984

[3.3] *Coll, J.:* BASIC interpreter or compiler. microprocessors and microsystems 2 (1978) H.5, S. 289–291

[3.4] *Guntheroth, K.:* The new ANSI BASIC standard. SIGPLAN Notices 18 (1983) H.7, S. 50–59

[3.5] *Harle, J.:* The proposed standard for BASIC. SIGPLAN Notices 18 (1983) H.5, S. 25–40

[3.6] *Kemeny, J. G.; Kurtz, T. E.:* BASIC Programming. New York: John Wiley & Sons 1972

[3.7] *Kurtz, T. E.:* On the way to Standard BASIC. A survey of what's in the proposed ANSI standard and why it's there. BYTE 7 (1982) H.6, S. 182–218

Abschnitt 10

[10.1] *Kieser, H.; Meder, M.:* Mikroprozessortechnik. Aufbau und Anwendung des Mikroprozessorsystems U880. 3. Aufl. Berlin: VEB Verlag Technik 1985

[10.2] *Schwarz, W.; Meyer, G.; Eckardt, D.:* Mikrorechner. Wirkungsweise – Programmierung – Applikation. 3. Aufl. Berlin: VEB Verlag Technik 1984

Abschnitt 11

[11.1] *Baumann, R.:* Computerspiele und Knobeleien programmiert in BASIC. 3. Aufl. Würzburg: Vogel-Buchverlag 1983

[11.2] *Baumann, R.:* Strukturiertes Programmieren mit BASIC. Stuttgart: Ernst Klett Verlag 1983

[11.3] *Boon, K. L.:* BASIC für Tischcomputer. München: Pflaum Verlag 1983

[11.4] *Brauch, W.:* Programmierung mit BASIC. Stuttgart: B. G. Teubner 1982

[11.5] *Busch, Rudolf:* BASIC für Einsteiger. 2. Aufl. München: Franzis-Verlag GmbH 1984

[11.6] *Feichtinger, H.:* BASIC für Mikrocomputer. München: Franzis-Verlag GmbH 1980

[11.7] *Menzel, K.:* BASIC in 100 Beispielen. Stuttgart: B. G. Teubner 1981

[11.8] *Schneider, W.:* Einführung in BASIC. 2. Aufl. Braunschweig, Wiesbaden: Vieweg Verlag 1980

[11.9] *Schwill, W.-D.; Weibezahn, R.:* Einführung in die Programmiersprache BASIC. Braunschweig: Vieweg Verlag 1976

[11.10] *Stief, S.:* BASIC. München, Wien: R. Oldenbourg Verlag 1980

[11.11] *Strelocke, K.; Hoffmann, P.:* Die Dialogprogrammiersprache BASIC. Berlin: Verlag Die Wirtschaft 1981

[11.12] *Wittig, S.:* BASIC-Brevier. 3. Aufl. Hannover: Verlag Heinz Heise GmbH 1982

Sachwörterverzeichnis

ISBN 3-341-00437-8