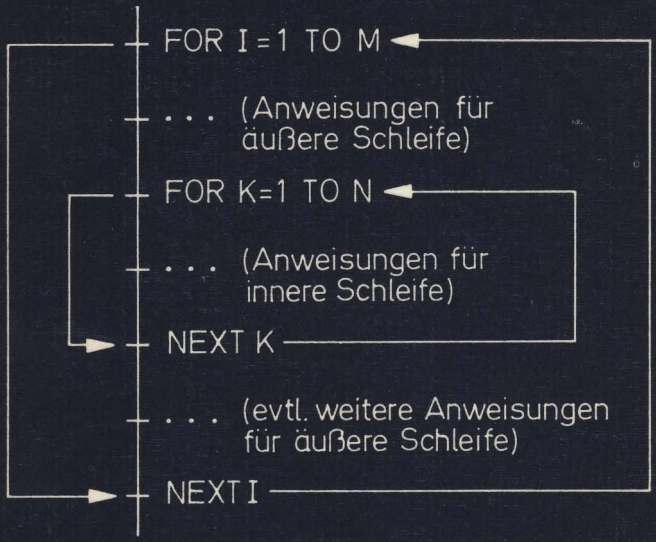
amateurreihe

electronica



Steffen Kober

234

Einführung in BASIC

electronica · Band 234

STEFFEN KOBER

Einführung in BASIC

MILITÄRVERLAG
DER DEUTSCHEN DEMOKRATISCHEN
REPUBLIK

Kober, S.:
Einführung in BASIC. – Berlin:
Militärverlag der DDR (VEB), 1987. –
96 S.: 9 Bilder – (electronica 234)

ISBN 3-327-00348-3

1. Auflage, 1987

Lizenz-Nr. 5
Printed in the German Democratic Republic
Lichtsatz: Druckerei Phönix
Druck und buchbinderische Weiterverarbeitung:
Druckerei des Ministeriums für Nationale
Verteidigung (VEB) – Berlin – 34328-6
Lektor: Rainer Erlekampf
Zeichnungen: Angelika Ulsamer
Typografie: Martina Schwarz
Redaktionsschluß: 18. Juli 1986
LSV 3539
Bestellnummer: 7469252
00190

Inhaltsverzeichnis

Vorwort

Die problemorientierte Programmiersprache BASIC wurde 1964 von *J. G. Kemeny* und *T. E. Kurtz* am Dartmouth College (USA) entwickelt. BASIC steht für **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode, was etwa «symbolischer Allzweck-Befehlscode für Anfänger» heißt, und war ursprünglich dazu gedacht, Schüler an die Programmierung von Großrechnern heranzuführen. Weltweite Verbreitung erfuhr BASIC jedoch erst durch die Entwicklung der Mikrorechentechnik. Für Mikrorechner ist BASIC die derzeit am häufigsten gebrauchte problemorientierte Programmiersprache. Sie ist leicht erlernbar und dialogorientiert.
Leider hat BASIC auch einige erhebliche Nachteile. Da es keine zwingende Programmstruktur kennt, verführt es zum «wilden» Programmieren. Dem ist nur zu begegnen, wenn man sich an übliche Grundprinzipien der Programmierung hält. Schwerer wiegt dagegen der Nachteil, daß sich aus dem ursprünglichen Dartmouth-BASIC inzwischen entgegen allen Standardisierungsbestrebungen eine Vielzahl von BASIC-Versionen entwickelt hat. Das liegt daran, daß jeder Rechnerhersteller sich für seine Rechner entsprechend deren Leistungsfähigkeit und Einsatzzweck BASIC-Varianten maßschneiderte, indem einige Sprachelemente gegen neue ausgetauscht wurden. Von einer einheitlichen Programmiersprache BASIC kann somit eigentlich gar keine Rede mehr sein. Auch in der DDR gibt es für Kleinst-, Personal-, Büro- und Arbeitsplatzcomputer viele, unterschiedlich leistungsfähige BASIC-Versionen. Den meisten ist aber ein gewisser Stamm von Elementen gemeinsam, und auf dessen ausführliche Erläuterung orientiert diese Broschüre. Auf häufige Abweichungen wird im Text hingewiesen. Dadurch ist es dem Leser möglich, sich zunächst mit Hilfe der Beschreibungen, Beispiele und Übungsaufgaben die Grundzüge von BASIC anzueignen. Anhand des Anwenderhandbuchs, welches es zu jedem Rechner gibt, kann man sich danach in sein spezielles BASIC ein-

arbeiten. Es fällt später auch nicht schwer, sich in Problemkreise hineinzufinden, die hier nicht oder nur andeutungsweise dargestellt werden konnten, weil sie bei den verschiedenen Rechnern zu unterschiedlich realisiert sind (Grafik, Dateiarbeit usw.). In der DDR werden Standards für universelle, höhere Programmiersprachen ausgearbeitet, um die Vielzahl von Sprachversionen künftig auszuschließen und Portabilität zu sichern. 1986 wurde ein Standard für FORTRAN 77 vorgestellt, Standards für PASCAL und voraussichtlich 2 Versionen von BASIC werden folgen [10]. Amateurrechner verfügen im Moment meist noch nicht über die dafür benötigte Mindestspeicherkapazität. Durch die rasche Entwicklung der Mikroelektronik werden Speicherkapazitätsfragen aber auch auf diesem Gebiet bald immer mehr in den Hintergrund treten. Die Mikrorechentechnik wird sich weitere Anwendungsbereiche erschließen und eine noch größere Verbreitung erfahren. Man ist gut beraten, sich bereits heute darauf einzustellen. Dieses Büchlein soll interessierte Laien ansprechen, auch wenn der «fortgeschrittene Leser» sich später vielleicht einer anspruchsvolleren Programmiersprache zuwenden wird.

1. Einleitung

1.1. Einige Grundbegriffe

Als **Hardware** bezeichnet man alle technischen Bestandteile eines Rechners. Das sind ganz allgemein Eingabegeräte, Zentraleinheit (bestehend aus Arbeitsspeicher, Steuerwerk und Rechenwerk), Ausgabegeräte und externe Speicher. Bei einem Mikrorechner ist das Eingabegerät in der Regel die Tastatur. Als Ausgabegerät wird ein Bildschirm sowie ggf. zusätzlich ein Drucker oder eine Schreibmaschine benutzt. Mögliche externe Speicher zur Aufbewahrung von Programmen und Daten sind z. B.:

- Spulenmagnetbandgeräte,
- Kassettenrecorder,
- spezielle Kassettenlaufwerke für Digital- oder Mikrokassetten und
- Diskettenlaufwerke (Floppy Disk Drive) für Minidisketten (5¼″ Durchmesser) oder Standarddisketten (8″ Durchmesser).

Die **Software** ist eine Folge von Anweisungen, die die Hardware veranlaßt, etwas zu tun. Zur Software gehören unter anderem Betriebssystem, Übersetzer (Assembler, Compiler oder Interpreter) und die Anwenderprogramme. Die Hardware läßt sich nur in Betrieb nehmen, wenn die Anweisungen in der **Maschinensprache** vorliegen. Die Maschinensprache besteht aus einer Folge von Binärziffern (0 und 1), die der Rechner direkt in Steuersignale umsetzen kann. Es ist aber für den Menschen recht schwer, sich in die jeweilige Maschinensprache hineinzudenken und seine Probleme direkt in einer solchen Sprache zu programmieren. Außerdem ist es sehr zeitaufwendig, und solche Programme sind extrem unübersichtlich und dadurch mit großem Fehlerrisiko behaftet. Deshalb wurden **Programmiersprachen** entwickelt und die dazugehörigen **Übersetzer** geschaffen. Die Übersetzer sind selbst Programme, die die in einer Programmier-

sprache vorliegenden Anwenderprogramme automatisch in die Maschinensprache übersetzen. Dies können sie aber nur, wenn die Anwenderprogramme syntaktisch fehlerfrei sind. Unter der **Syntax** einer Programmiersprache versteht man die Regeln dieser Sprache, nach denen die Programme aus dem vorhandenen Zeichenvorrat zusammengesetzt werden. Die Bedeutung dieser Zeichenkombinationen wird dagegen **Semantik** genannt. Programmiersprachen können maschinenorientiert und problemorientiert sein.

Maschinenorientierte Programmiersprachen sind z.B. die Assemblersprachen. Ihre Übersetzungszeiten sind recht kurz, da in einer Assemblersprache jeder Maschinenbefehl durch einen symbolischen Ausdruck (Mnemonic) ersetzt ist. Sie hängen dadurch aber vom Rechnertyp ab, und der Programmieraufwand ist außerdem immer noch recht beachtlich.

Problemorientierte Programmiersprachen sind z.B. ALGOL, FORTRAN, PL/1, COBOL, BASIC, Pascal und MODULA 2. Sie haben den Vorteil, daß nur noch ihre Übersetzer (Compiler bzw. Interpreter) rechnerabhängig sind, da problemorientierte Programmiersprachen unabhängig von der jeweiligen Maschinensprache für einen bestimmten Problemkreis zugeschnitten sind. Deshalb werden sie auch höhere Programmiersprachen genannt. Solche Programmiersprachen sind wesentlich leichter erlernbar. Außerdem wird viel weniger Zeit für das Programmieren benötigt, und die Programme sind übersichtlicher und damit auch flexibler, da sich spätere Erweiterungen und Änderungen leichter einarbeiten lassen. Leider sind sowohl die Übersetzungs- als auch die Abarbeitungszeiten solcher Programme meist länger als die eines in einer maschinenorientierten Sprache geschriebenen Programms. Compiler und Interpreter lassen sich wesentlich schwieriger entwickeln als Assembler und sind aus diesem Grund teurer.
Hat man den Rechner, ein Anwenderprogramm und den zugehörigen Übersetzer, stellen sich die Fragen:
Wie erkennt der Rechner, daß ein Programm oder Daten eingegeben werden sollen?

Wie erkennt der Rechner, daß der Übersetzer das eingegebene Programm auf Fehler prüfen und übersetzen soll?
Wie erkennt der Rechner, ob über Drucker oder Bildschirm ausgegeben werden soll?
Wie erkennt der Rechner, wo Programme und Daten gespeichert werden sollen usw.?
Dazu fehlt noch das wichtigste Bindeglied, das **Betriebssystem**. Die Bezeichnung Organisationsprogramm wäre dafür vielleicht treffender, denn das Betriebssystem ist ein Komplex von Hilfsprogrammen, die die Kommunikation zwischen dem Rechner, den Anwenderprogrammen und dem Bediener organisieren und steuern. Zu den wichtigsten Aufgaben eines Betriebssystems gehören das Steuern der Programmabarbeitung (Start, Beenden, Stop, Abbruch usw.), Ein- und Ausgabesteuerung und das Verwalten des Arbeitsspeichers. Zum Kommunizieren mit dem Betriebssystem stehen **Kommandos** zur Verfügung. Darauf wird in Abschnitt 3.1. und Abschnitt 4. eingegangen.
Bei Mikrorechnern wird der Dialog zwischen dem Rechner und dem Bediener meist über den Bildschirm abgewickelt. Eine Lichtmarke **(Cursor)** zeigt dem Bediener, wo das nächste Zeichen einzugeben ist bzw. ausgegeben wird. Der Cursor ist meist ein einzelner Unterstreichungsstrich oder ein kleines Rechteck, mitunter blinkt er auch, um die Aufmerksamkeit des Bedieners auf die aktuelle Ein- bzw. Ausgabeposition zu lenken. Viele Rechner haben auch spezielle Cursorsteuerungstasten, mit denen der Cursor an eine beliebige Stelle auf dem Bildschirm dirigiert werden kann.

1.2. BASIC-Compiler und BASIC-Interpreter

BASIC hat gegenüber vielen anderen Programmiersprachen den Vorteil, daß als Übersetzer sowohl Interpreter als auch Compiler zur Verfügung stehen. Einschränkend muß gesagt werden, daß Compiler wesentlich mehr Speicherplatz als Interpreter benötigen und daher in der Regel nur bei Mikrorechnern einsetzbar sind, die mindestens über einen 64-Kbyte-Arbeitsspeicher verfügen.

Ein **BASIC-Compiler** übersetzt das gesamte eingegebene BASIC-Programm in die Maschinensprache und speichert das übersetzte Programm im Arbeitsspeicher (oder auch auf einem externen Speicher) ab. Findet er syntaktische Fehler, wird ein Fehlerprotokoll ausgegeben. Nach jeder Fehlerkorrektur muß das gesamte BASIC-Programm neu übersetzt werden. Erst wenn es fehlerfrei ist, kann das Programm abgearbeitet werden. Es gibt auch BASIC-Compiler, die das BASIC-Programm nicht direkt in die Maschinensprache, sondern erst in einen Zwischencode übersetzen, der dann noch mit einem Linker (Binder) bearbeitet werden muß. Darauf soll hier aber nicht weiter eingegangen werden. Die Arbeitsstufen eines BASIC-Compilers werden noch einmal durch Bild 1.1 verdeutlicht.

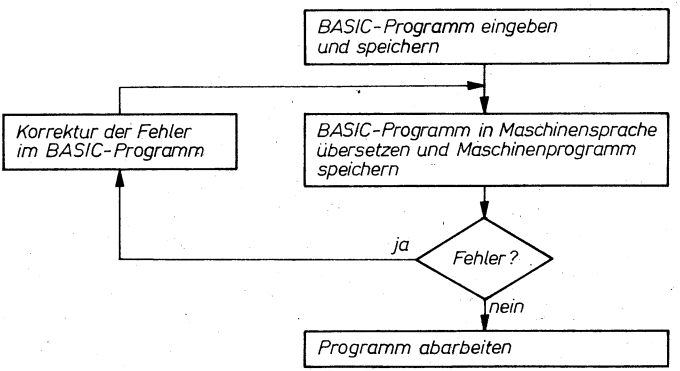


Bild 1.1: Arbeitsstufen bei der Übersetzung eines BASIC- Programms mit Hilfe eines BASIC-Compilers

Ein **BASIC-Interpreter** übersetzt eine Programmzeile in die Maschinensprache und prüft sie auf syntaktische Fehler. Ist sie fehlerfrei, wird sie auch gleich ausgeführt, andernfalls kann man die entsprechende Programmzeile nach der Fehlermeldung sofort korrigieren. Erst nach Ausführung der 1. Programmzeile wird die 2. übersetzt und ausgeführt usw., bis das Programmende

erreicht ist. Auf Grund des zeilenweisen Abarbeitens erübrigt sich das Speichern des übersetzten Programms. Die Arbeitsstufen eines BASIC-Interpreters werden im Bild 1.2 dargestellt. Compilierte Programme haben den Vorteil, daß sie nach der Übersetzung in Maschinensprache vorliegen und sich daher sehr schnell abarbeiten lassen. Sie erfordern aber mehr Speicherplatz, Fehler sind schwerer zu lokalisieren, und nach jeder Fehlerkorrektur muß das gesamte Programm neu compiliert werden.

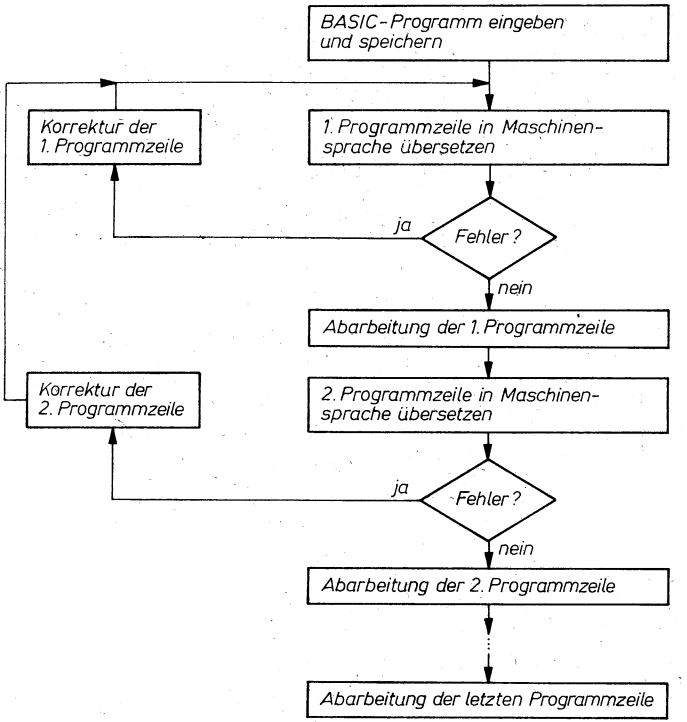


Bild 1.2: Arbeitsstufen bei der Übersetzung eines BASIC-Programms mit Hilfe eines BASIC-Interpreters

Bei interpretierender Arbeitsweise dauert das Abarbeiten eines Programms durchschnittlich 10- bis 15mal so lange wie das eines compilierten Programms, weil bei Interpretern Übersetzen und Abarbeiten nicht getrennt ablaufen. Die schrittweise Ausführung ermöglicht aber einen komfortablen Dialog zwischen Bediener und Rechner, Fehler kann man leichter erkennen und sofort beseitigen. Der Programmtest ist einfacher, da der Bediener das Programm an jeder beliebigen Stelle unterbrechen und dann weiterarbeiten kann. (Dies wird im Abschnitt 4. noch etwas erläutert.) Alles in allem gelangt man also beim Einsatz eines BASIC-Interpreters sehr viel einfacher und schneller zu einem fehlerfreien Programm.
Hier wird offensichtlich, welch enormer Vorteil sich bietet, wenn man sowohl über einen BASIC-Interpreter als auch über einen BASIC-Compiler verfügt. Durch Kombination dieser beiden Übersetzertypen ist erst eine wirklich effektive Arbeit möglich. Das BASIC-Programm wird mit dem Interpreter so lange im Dialogbereich bearbeitet und getestet, bis es sowohl syntaktisch als auch semantisch fehlerfrei ist. Erst dann wird es mit dem Compiler übersetzt, damit man bei der anschließenden Nutzung des Programms möglichst kurze Abarbeitungszeiten erreicht.
Da Amateuren für ihre Mikrorechner meist keine BASIC-Compiler zur Verfügung stehen werden, bezieht sich alles Folgende ausschließlich auf BASIC-Interpreter.

2. Das Programmieren

2.1. Problemaufbereitung

Der Rechner soll den Nutzer bei der Lösung von Problemen unterstützen. Wie soll man aber dem Rechner ein Problem mitteilen?
Dazu muß zunächst einmal festgestellt werden, ob sich das Problem überhaupt zur Bearbeitung durch einen Rechner eignet. Als erstes wird deshalb eine sogenannte Problemanalyse durchgeführt. Sie umfaßt folgende Schritte:

1. Exaktes und vollständiges Formulieren des Problems.
2. Eindeutiges Beschreiben der Ausgangsdaten des Problems.
3. Eindeutiges Beschreiben der gewünschten Ergebnisse, die die Problemlösung liefern soll (Leistungsumfang des Programms).
   In welcher Form sollen sie ausgegeben werden?
4. Welche Randbedingungen sind zu beachten?
5. Welche Sonderfälle können auftreten?
6. Gibt es einen Algorithmus (Lösungsweg), der mit den zur Verfügung stehenden Ausgangsdaten zu den gewünschten Ergebnissen führt?

Kann ein solcher Lösungsweg nicht gefunden werden, weil das Problem nicht eindeutig beschrieben worden ist und sich deshalb keine logischen Zusammenhänge herstellen lassen oder weil die vorliegenden Ausgangsinformationen nicht ausreichen oder weil zu viele Sonderfälle und Randbedingungen zu beachten sind, so ist das Problem auch mit Hilfe eines Rechners nicht lösbar. Mitunter läßt sich dann doch noch eine Lösung finden, wenn die Schritte 1 bis 5 neu durchdacht werden.
Die Analyse sollte auch bei kleineren Problemen stets gründlich und vollständig durchgeführt werden. Oft ist die spätere Enttäuschung von einem fertigen Programm darauf zurückzuführen, daß man sich nicht die Zeit nahm, das Problem umfassend zu for-

mulieren und zu analysieren. Man darf von einem Programm nicht mehr erwarten, als man bei der Beschreibung der zu lösenden Aufgabe vorgesehen hat! Eine oberflächlich durchgeführte Problemanalyse rächt sich oft mit einem beträchtlichen zeitlichen Mehraufwand für nachträgliche Programmerweiterungen. Diese machen zudem das Programm unübersichtlich, womit dann weitere Änderungen noch schwieriger und zeitaufwendiger, wenn nicht gar unmöglich werden.

Nicht selten gibt es für eine Aufgabe auch mehrere Lösungsvarianten. Dann gehört es auch zur Problemanalyse, davon den für eine Bearbeitung mit dem Rechner günstigsten Lösungsweg auszuwählen.

Ist der prinzipielle Lösungsweg klar, muß man ihn noch detailliert ausarbeiten und in eine rechnergerechte (programmierbare) Form bringen.

Für Mikrorechner hat sich dabei bewährt, das Problem kontinuierlich von oben nach unten durch schrittweises Verfeinern zu bearbeiten (engl.: top-down approach by stepwise refinement). Der grobe Lösungsweg wird nicht sofort bis ins kleinste Detail ausgearbeitet, sondern nach und nach so verfeinert, bis die Stufe erreicht ist, daß man die Verfeinerung direkt in eine Anweisung einer Programmiersprache umsetzen kann. Diese Methode hat den Vorteil, daß man den Überblick behält und der Lösungsweg gut strukturiert wird. Welche grafische Darstellungsmethode man benutzt (Programmablaufpläne, Struktogramme usw.), ist nicht entscheidend.

2.2. Grobstruktur eines BASIC-Programms

Nach der Problemaufbereitung kann mit dem Programmieren begonnen werden. Das Programm wird zunächst vollständig auf einem Blatt Papier entworfen und danach über die Tastatur in den Rechner eingegeben. Der BASIC-Interpreter übersetzt es Zeile für Zeile in den Maschinencode.
Wie sieht nun solch ein BASIC-Programm aus? Ein BASIC-Programm ist eine Folge von Anweisungen. Bei den meisten BASIC-Interpretern muß jede Anweisung auf einer Zeile stehen. Nur einige größere BASIC-Interpreter lassen mehrere Anweisungen je Zeile, meist durch Doppelpunkt getrennt, zu. Dies wird hier aber nicht weiter betrachtet. Jede Programmzeile beginnt mit einer Zeilennummer. Sie legt die Abarbeitungsreihenfolge fest. Programmzeilen müssen nicht unbedingt entsprechend ihrer logischen Reihenfolge eingegeben werden, da der Interpreter sie automatisch entsprechend den Zeilennummern ordnet. Das ist z. B. vorteilhaft, wenn eine vergessene Programmzeile nachträglich eingefügt werden soll. Deshalb ist es auch üblich, die Zeilen in 10er-Abständen zu numerieren, um Platz für Einfügungen zu haben. Jede Zeilennummer darf aber nur einmal vorkommen. Die Zeilennummer ist meist eine maximal 4stellige oder 5stellige natürliche Zahl.
Anweisungen, die etwas vereinbaren (z. B. den Speicherplatz für Variablenfelder) werden auch Vereinbarungen genannt, und es empfiehlt sich, diese in einem Vereinbarungsteil zusammenzufassen und an den Anfang des Programms zu stellen. Damit ist gewährleistet, daß alle Vereinbarungen getroffen wurden, bevor sie im Programmablauf benötigt werden, und man überblickt sofort alle Vereinbarungen.
Erst dann folgt der eigentliche Anweisungsteil, der die Umsetzung des ausgearbeiteten Lösungsweges in BASIC darstellt. Schließlich wird das Programm durch die Programmbeendungsanweisung END abgeschlossen. Bei den meisten Interpretern kann diese Anweisung auch weggelassen werden, da der Interpreter sie automatisch nach der letzten Programmzeile annimmt.
Die Grobstruktur eines Programms sieht wie folgt aus:

```
10      Vereinbarung 1          Vereinbarungsteil
20      Vereinbarung 2
.
.
...     Vereinbarung n
...     Anweisung 1             Anweisungsteil
...     Anweisung 2
.
.
.
...     Anweisung n
...     END
```

Jede Anweisung beginnt mit einem Schlüsselwort, welches angibt, was für eine Anweisung auszuführen ist. Solche Schlüsselwörter sind z. B. PRINT («drucke») oder INPUT («Eingabe»). Darauf folgen meist die Parameter der Anweisung, die angeben, mit welchen Werten und wie die Anweisung ausgeführt wird.
Die Länge einer Anweisung ist begrenzt. Die maximale Zeichenanzahl, die eine Anweisung haben darf, ist bei den einzelnen BASIC-Interpretern unterschiedlich und muß dem Anwenderhandbuch entnommen werden. Sie richtet sich meist nach der Anzahl der Zeichen, die in eine Bildschirmzeile passen (z. B. 40, 64 oder 80), darf oft aber auch über mehrere Bildschirmzeilen gehen (üblich sind z. B. 255 Zeichen). Sind auszugebende Texte länger, als es die maximale Zeichenanzahl für eine Anweisung erlaubt, so muß der Text auf mehrere Anweisungen verteilt werden.

Die Programmentwicklung umfaßt insgesamt folgende Schritte:

1. Problemaufbereitung;
2. Programmieren;
3. Programmeingabe, Programmtest und Programmkorrektur (siehe Abschnitt 3.1. und Abschnitt 4.);
4. Programmdokumentation für späteres wiederholtes Nutzen des Programms.

Erst dann steht dem Nutzer ein wirklich fertiges Programm zur Problembearbeitung zur Verfügung.

Zum Beschreiben der Syntax der Anweisungen wird in diesem Buch eine stark vereinfachte Symbolik benutzt:
Großbuchstaben repräsentieren Schlüsselwörter, die unverändert in die jeweilige Anweisung zu übernehmen sind. Anstelle von mit **Kleinbuchstaben** geschriebenen Teilen werden deren Bedeutung gemäß die entsprechenden Konstanten, Variablen, Ausdrücke usw. in die Anweisung eingesetzt. **Trennungszeichen**, wie Komma und Semikolon, und **runde Klammern** ( ) muß man wie angegeben in die Anweisung übernehmen. **Eckige Klammern** [ ] bedeuten lediglich, daß der darin stehende Teil der Anweisung auch fehlen kann (weggelassen werden darf). **Geschweifte Klammern** { } schließen Anweisungsteile ein, die auch mehrfach hintereinander in einer Anweisung auftreten dürfen.
Im Zweifelsfall wird der richtige Gebrauch der Anweisungsteile durch die angeführten Beispiele deutlich.

3. Systematische Darstellung der Sprachelemente von BASIC

3.1. Eingabe, Speichern und Start eines BASIC-Programms

Damit der Leser die im folgenden beschriebenen Anweisungen und Beispiele sofort an seinem Rechner ausprobieren kann, muß vorher etwas über Programmeingabe und -start gesagt werden. Nachdem der BASIC-Interpreter seine Bereitschaft gemeldet hat, kann das Programm Zeile für Zeile über die Tastatur eingegeben werden. Nach jeder Zeile muß die ENTER-Taste betätigt werden. Bei einigen Rechnern ist diese Taste auch mit ET1, RETURN oder ENTRY bezeichnet. Ist das Programm vollständig eingegeben, startet man es durch das Kommando

RUN (ENTER)

Das Programm wird übersetzt und ausgeführt. Ein fehlerfreies Programm liefert die gewünschten Ergebnisse, und der BASIC-Interpreter meldet dem Bediener auf dem Bildschirm durch die Ausschrift OK, READY oder ähnliches, daß er das Programm abgearbeitet hat.

Soll das BASIC-Programm auch nach Abschalten des Rechners erhalten bleiben, weil man es später erneut verwenden will, so muß es vom Arbeitsspeicher des Rechners auf einen externen Speicher gerettet werden. Da unterschiedliche Speichermedien möglich sind (Kassetten, Disketten usw.), kann das notwendige Speicherkommando hier nicht exakt angegeben werden. Man findet es im jeweiligen Anwenderhandbuch. Das Speicherkommando lautet im allgemeinen SAVE (rette, bewahre), bei Kassetten auch CSAVE. Danach ist der Programmname anzugeben. Er muß mit einem Buchstaben beginnen und darf meist keine oder nur bestimmte Sonderzeichen enthalten.

Falls der verwendete Interpreter die Angabe von Geräteadressen vorschreibt, muß man diese ebenfalls der Anwenderdokumenta-

tion entnehmen. Der Programmname wird vom Nutzer selbst festgelegt und darf aus bis zu 8 (mitunter bis zu 16) Zeichen bestehen.
Benötigt man ein auf einem externen Speicher abgelegtes Programm erneut, so muß es umgekehrt wieder vom externen Speicher in den Arbeitsspeicher des Rechners geladen werden. Dieses Ladekommando heißt also LOAD (lade) oder GET (nimm), bei Kassetten auch CLOAD. Danach sind ebenfalls Geräteadresse und Programmname anzugeben, damit der Rechner weiß, woher er was in den Arbeitsspeicher laden soll. Beim Laden eines Programms aus einem externen Speicher wird der vorherige Arbeitsspeicherinhalt automatisch gelöscht.
Möchte man über die Tastatur ein neues Programm eingeben, kann der alte Arbeitsspeicherinhalt auch von Hand durch das Kommando
NEW (ENTER)
gelöscht werden.

■ Beispiel:
Es soll ein kleines Programm eingeben werden, welches Zahlen summiert.

```
10  S=0
20  INPUT "Anzahl der Summanden: ",N
30  FOR I=1 TO N
40  INPUT "Summand: ",A
50  S=S+A
60  NEXT I
70  PRINT "Summe: ";S
```

Dieses Programm erhält den Namen «SUMME» und wird auf Kassette gespeichert. Anschließend soll der Arbeitsspeicher gelöscht werden, um ein anderes Programm eingeben zu können, und später soll das Programm «SUMME» wieder geladen und damit gerechnet werden. Die dazu notwendige Kommandofolge lautet:

CSAVE"SUMME"	Speichern von «SUMME» auf Kassette
NEW	Löschen des Arbeitsspeichers
.	
.	Eingabe eines anderen Programms
.	
CLOAD"SUMME"	Laden von «SUMME» von der Kassette zurück in den Arbeitsspeicher
RUN	Start des Programms

Will man sich ein im Arbeitsspeicher befindliches Programm noch einmal auf dem Bildschirm anschauen, benutzt man dazu das Kommando

LIST (ENTER)

Durch Eingabe von Zeilennummern nach LIST kann ein Programm auch ausschnittsweise ab oder bis zu einer bestimmten Zeilennummer sichtbar gemacht werden. Weitere Kommandos sind im Abschnitt 4. behandelt.

3.2. Grundelemente von BASIC

3.2.1. Zeichenvorrat und Namen

In BASIC sind wie auch bei vielen anderen höheren Programmiersprachen 3 Arten von Zeichen zulässig:

a) Buchstaben: A, B, C, ..., Z
   (bei größeren BASIC-Interpretern sind zusätzlich auch die Kleinbuchstaben a, b, c, ..., z möglich);
b) Ziffern: 0, 1, 2, ..., 9;
c) Sonderzeichen: ! ? " ¤ % ( ) * + − / ^ ; : < > = # & ' _ − [ ] { } , . | \
   (dies ist lediglich eine Auswahl der gebräuchlichsten Sonderzeichen).

Der Name einer Variablen (was eine Variable ist, wird im Abschnitt 3.2.3. noch genauer erklärt) beginnt grundsätzlich mit einem Buchstaben. Die meisten den Amateuren derzeit zur Verfügung stehenden Interpreter ermöglichen nach dem Buchstaben lediglich noch eine Zahl. Variablennamen sind in diesem Sinne:

A, B, ..., Z
A0, B0, ..., Z0
. . .
. . .
. . .
A9, B9 ..., Z9

Es sei jedoch dazu noch angemerkt, daß bei leistungsstarken BASIC-Interpretern die Variablennamen aus bis zu 40 Zeichen bestehen können, z. B. QUADRANT, KB12 usw., wodurch die Programme erheblich übersichtlicher werden. Die Variablennamen dürfen dabei aber keine Sonderzeichen und keine Schlüsselwörter (reservierte Namen von BASIC-Anweisungen, Kommandos und Operatoren) enthalten.
Der Variablenname stellt für den Rechner die Adresse dar, unter der während des Programmablaufes der jeweils aktuelle Wert der Variablen abgespeichert wird.

3.2.2. Konstanten

Konstanten werden einmalig fest im Programm verankert und ändern ihren Wert während der Programmabarbeitung nicht. Sie benötigen deshalb auch keinen Namen. Es wird zwischen numerischen Konstanten (Zahlen) und Zeichenketten oder Stringkonstanten (Text) unterschieden. Konstanten sind insbesondere dann vorteilhaft, wenn in einem Programm Folgen von Zahlen oder Texte (z. B. Überschriften) wiederholt auftreten (siehe Abschnitt 3.3.4.).

■ Beispiele für numerische Konstanten sind:
3, 4.718, −0.09, 1.7E8, −2E−5
Da das Komma als Trennungszeichen zwischen Konstanten verwendet wird, muß als Markierung zwischen ganzem und gebrochenem Teil einer Zahl der Dezimalpunkt benutzt werden. Schriebe man statt 4.718 versehentlich 4,718, würde der BASIC-Interpreter dies als 2 Konstanten auffassen, nämlich die Zahlen 4 und 718. Die Konstanten 1.7E8 und −2E−5 sind Zahlen in Exponentialschreibweise, es bedeutet

1.7E8 $1{,}7 \cdot 10^8 = 170000000$
−2E−5 $-2 \cdot 10^{-5} = -0.00002$

■ Beispiele für Zeichenkettenkonstanten sind:
«NAME», «TELEFON», " PUNKT 1: ","*****"," ","".
Das Komma fungiert als Trennungszeichen zwischen Zeichenkettenkonstanten, darf aber selbst auch in einer Zeichenkettenkonstanten enthalten sein. Das Zeichen " markiert Anfang und Ende einer Zeichenkettenkonstanten, es kann daher nicht selbst Bestandteil einer Zeichenkettenkonstanten sein.
Der Zeichenvorrat des Rechners ist jedoch größer als der Zeichenvorrat des BASIC-Interpreters (siehe Anhang, Tabelle ASCII-Code). Jedes Zeichen aus dem Zeichenvorrat des Rechners wird im Rechner intern durch eine Binärzahl dargestellt. Der 7-Bit-ASCII-Code ist der am häufigsten gebrauchte Code, mit ihm lassen sich $2^7 = 128$ Zeichen darstellen. Zur leichteren Handhabung werden statt der Binärzahlen in BASIC deren Dezimaläquivalente benutzt.

■ Beispiel:
Die Umrechnung der Binärzahl 1101011 entspricht:
$\mathbf{1} \cdot 2^6 + \mathbf{1} \cdot 2^5 + \mathbf{0} \cdot 2^4 + \mathbf{1} \cdot 2^3 + \mathbf{0} \cdot 2^2 + \mathbf{1} \cdot 2^1 + \mathbf{1} \cdot 2^0$
$= 64 + 32 + 8 + 2 + 1 = 107$. Das heißt die Binärzahl 1101011 hat das Dezimaläquivalent 107 (das ist der dezimale ASCII-Code des Kleinbuchstaben k).
Zeichen, die nicht zum Zeichenvorrat des Interpreters gehören, können in Zeichenkettenkonstanten benutzt werden, indem ihr dezimaler ASCII-Code angegeben wird. Die Codezahl wird dabei mitunter in < > eingeschlossen, meist aber mit Hilfe der CHR $-Funktion angegeben (siehe Abschnitt 3.2.5.). 34 ist z. B. die dezimale ASCII-Codezahl des Zeichens ", falls dieses Zeichen doch einmal für eine Zeichenkettenkonstante benötigt wird.

3.2.3. Variablen

Der Begriff Variable ist vielen Lesern sicher als Zahlensymbol aus der Mathematik bekannt. Solche Zahlensymbole werden benutzt, um Aussagen oder Rechenvorschriften zu formulieren, die für mehrere Zahlen gelten. Die Vertauschbarkeit der Summanden bei der Addition (Kommutationsgesetz) kann z.B. durch die Formel

$a + b = b + a$

ausgedrückt werden. a und b sind hier Variable, die beliebig viele Zahlenwerte annehmen können, da das Kommutationsgesetz für alle reellen Zahlen gilt. a + b = b + a steht also stellvertretend für $1 + 2 = 2 + 1, 3 + 8 = 8 + 3, 8 + ½ = ½ + 8$ usw.
In BASIC können Variablen nicht nur Zahlenwerte annehmen, sondern auch Buchstaben und Sonderzeichen enthalten.
Es wird deshalb zwischen numerischen Variablen (Variablen, die nur Zahlenwerte annehmen können) und Zeichenketten- oder Stringvariablen (Variablen, die Texte enthalten) unterschieden. Numerische Variablen wiederum können reelle (real) oder ganzzahlige (integer) Variablen sein. Man bezeichnet dies auch als Typ der Variablen. Reelle Variablen sind der «Normalfall», sie werden nicht besonders gekennzeichnet. Ganzzahlige Variablen werden durch das Zeichen % am Ende ihres Namens markiert, z.B. A1%, J%, K%.

■ **Beispiele:**
Werte ganzzahliger Variabler: 13, −9, 0, 20000
Werte reeller Variabler: 1,5, 0, −25.118, 12, 1.7E10
Es ist vorteilhaft, ganzzahlige Variablen zu benutzen, da für jeden Namen einer ganzzahligen Variablen vom Rechner nur 2 byte Speicherplatz reserviert werden. Im Gegensatz dazu benötigt jede reelle Variable 4 byte, also doppelt soviel, bzw. bei doppelter Genauigkeit sogar 8 byte Speicherplatz. Außerdem werden im allgemeinen Operationen mit ganzzahligen Variablen vom Rechner schneller ausgeführt als solche mit reellen Variablen. Allerdings können ganzzahlige Variablen meist nur ganz-

zahlige Werte zwischen −32768 und +32767 annehmen. Für größere oder kleinere Werte müssen reelle Variablen bereitgestellt werden, da solche Werte nur in Exponentialform verarbeitbar sind. Statt 38000 wird z. B. 3.8E4 (≙ $3{,}8 \cdot 10^4$) verwendet. Selbstverständlich können auch reelle Variablen nicht beliebig kleine oder große Werte annehmen.
Obere und untere Grenzen dafür sind aber von Interpreter zu Interpreter recht verschieden und müssen dem jeweiligen Anwenderhandbuch entnommen werden.
Namen von Zeichenkettenvariablen muß man mit ¤ (bei manchen Rechnern auch $) kennzeichnen, z. B. A¤, B1¤. Die meisten Interpreter reservieren für Zeichenkettenvariable automatisch 10 byte Speicherplatz, d. h., eine solche Zeichenkettenvariable kann bis zu 10 Zeichen Text aufnehmen. Sind jedoch alle Werte, die eine Zeichenkettenvariable annehmen kann, kürzer als 10 Zeichen, so kann man Speicherplatz sparen, indem man diese Zeichenkettenvariable entsprechend der nötigen Länge dimensioniert. Zeichenkettenvariable, die Zeichenketten aufnehmen sollen, welche länger als 10 Zeichen sind, müssen sogar dimensioniert werden. Zur Dimensionierung von Variablen wird jedoch auf Abschnitt 3.3.1. verwiesen.
Sowohl numerische als auch Zeichenkettenvariablen können auch einfach oder mehrfach indiziert auftreten. Man spricht dann von Variablenfeldern. Bei kleinen oder mittleren Interpretern sind meist maximal 2 Indizes zugelassen, d. h., es gibt nur 1dimensionale und 2dimensionale Felder.
Sei Z1¤ der Name eines 1dimensionalen Zeichenkettenfeldes mit 4 Komponenten, dann werden seine 4 Komponenten (auch Feldelemente genannt) mit Z1¤ (1), Z1¤ (2), Z1¤ (3) und Z1¤ (4) bezeichnet. Jedes dieser Elemente enthält eine Zeichenkette. Z1¤ enthält also 4 Zeichenketten.
Z1¤ = [Z1¤ (1) Z1¤ (2) Z1¤ (3) Z1¤ (4)]
Die in den runden Klammern stehende Zahl ist der sogenannte Index. Er gibt die Stellung des Elementes innerhalb des Feldes an, Z1¤ (3) ist also das 3. Element des Feldes Z1¤.
Sei M% der Name eines 2dimensionalen ganzzahligen Feldes mit 6 Komponenten, bestehend aus 2 Zeilen und 3 Spalten:

$$M\% = \begin{bmatrix} M\%(1,1) & M\%(1,2) & M\%(1,3) \\ M\%(2,1) & M\%(2,2) & M\%(2,3) \end{bmatrix}$$

Der 1. Index ist dann der sogenannte Zeilenindex, der 2. Index ist der Spaltenindex. M%(1,2) bezeichnet das Element, das in der 1. Zeile und in der 2. Spalte steht. Alle Feldelemente müssen immer vom selben Typ wie der Name der Feldvariablen sein (im Beispiel M% ganzzahlige Variablen).
Analog zu den Zeichenkettenvariablen wird für Feldvariablen von den meisten Interpretern automatisch Speicherplatz reserviert, und zwar für je 10 Werte je Index. Ein 1dimensionales Feld hat dann also maximal 10 Komponenten, ein 2dimensionales Feld 10×10=100. Kleinere Felder sollten, größere müssen dimensioniert werden. Dazu findet der Leser jedoch mehr in Abschnitt 3.3.1.

3.2.4. Operatoren

In BASIC unterscheidet man 4 Typen: arithmetische Operatoren, Vergleichsoperatoren, logische Operatoren und Zeichenkettenoperatoren. Tabelle 3.1. enthält eine Übersicht. Es werden jedoch nur die gebräuchlichsten Operatoren vorgestellt.
Alle **arithmetischen Operatoren** können zwischen 2 Operanden stehen. Als Operanden sind grundsätzlich nur arithmetische Ausdrücke zugelassen. Was Ausdrücke sind, wird im Abschnitt 3.2.6. noch genauer erklärt. Die Operatoren + und − können auch als Vorzeichen gebraucht werden.

■ Beispiele:
A+B, 9^4, (A−2*B)^3, (−C)^3
Vergleichsoperatoren stehen immer zwischen 2 Operanden desselben Typs, wobei als Operanden ebenfalls Ausdrücke auftreten. Hier dürfen es jedoch nicht nur arithmetische Ausdrücke, sondern auch Zeichenkettenausdrücke sein.
Das Ergebnis eines Vergleiches hat entweder den Wert WAHR (W) oder FALSCH (F), das sind die sogenannten logischen Wahrheitswerte. Ergebnisse von Vergleichen können deshalb mit logischen Operatoren weiterverknüpft werden.

Tabelle 3.1. Operatoren

Operator	Bedeutung	Bemerkungen
arithmetische Operatoren		
+	positives Vorzeichen; Addition	
–	negatives Vorzeichen; Subtraktion	
*	Multiplikation	
/	Division	
^	Potenzieren	
MIN	Minimieren	auch ↑ oder ** gebräuchlich
MAX	Maximieren	nur bei einigen Interpretern
MOD	Modulo	
\	ganzzahlige Division	
Vergleichsoperatoren		
<	kleiner	
>	größer	
=	gleich	
<>	ungleich	mitunter auch #
<=	kleiner gleich	
>=	größer gleich	
logische Operatoren		
NOT	log. Negation	auch ¬, ~ oder NEG
AND	log. Und	auch &
OR	log. Oder	auch !
XOR	log. Entweder-Oder	nur bei einigen Interpretern
EQV	Äquivalenz	
IMP	Implikation	
Zeichenkettenoperator		
+	Verkettungsoperator	auch &

■ **Beispiele:**

A+B>=C

(A−2*B)^3>0

A^2=B+C

▼ *Übung 1:*

Geben Sie die Wahrheitswerte der im obigen Beispiel genannten Vergleiche an, wenn die Variablen folgende Werte haben:

A=3, B=4, C=5.

Zeichenketten werden vom BASIC-Interpreter verglichen,

indem die ASCII-Codierungen (siehe Tabelle ASCII-Code im Anhang) jedes Zeichens der Zeichenketten, von links beginnend, miteinander verglichen werden. Der Vergleich "A"<"B" liefert den Wert W, da der dezimale ASCII-Code von A 65, der von B dagegen 66 und somit größer ist. WAHR ist ebenfalls der Vergleich "A3"<"AB", da die Ziffern kleinere ASCII-Codes als die Buchstaben haben (3 hat den ASCII-Code 51) und hier beide Operanden mit A beginnen, so daß der Vergleich von 3 mit B den Ausschlag gibt. Insbesondere hat auch "A"<"AB" den Wert W, weil hier nach den beiden A die leere Zeichenkette mit B verglichen wird.

▼ *Übung 2:*
Ermitteln Sie die Wahrheitswerte der folgenden Zeichenkettenvergleiche mit Hilfe der ASCII-Code-Tabelle im Anhang!
"MUELLER"<="MEIER"
"3.6.1986"<>"3.6.86"
"ABSTAND">"AB STAND"
"(ET1)">=" [ET1] "
Der logische Operator NOT («nicht», Negation) wird auf den nachfolgenden Operanden angewendet, alle anderen logischen Operatoren stehen immer zwischen 2 Operanden. Als Operanden können logische Ausdrücke und Vergleichsausdrücke auftreten. Ergebnis einer logischen Operation sind die Werte W oder F. Da es in BASIC aber keinen Booleschen (logischen) Variablentyp gibt, werden die logischen Werte W und F von den Interpretern durch numerische Werte ersetzt. Dies ist aber bei den verschiedenen BASIC-Interpretern sehr unterschiedlich realisiert, weshalb in dieser Broschüre der Einfachheit halber weiter W und F verwendet wird, um Mißverständnissen vorzubeugen. Allgemein üblich ist z.B. die Zuordnung 1 für W und 0 für F, es gibt aber noch einige andere Zuordnungsvarianten. Deshalb muß wieder auf das jeweilige Anwenderhandbuch verwiesen werden. Die Ergebnisse der gebräuchlichsten logischen Operationen sind aus Tabelle 3.2. ersichtlich.

Tabelle 3.2. Ergebnisse logischer Operationen

logischer Operator	Operanden		Ergebnis
NOT	X		NOT X
	W		F
	F		W
AND	X	Y	X AND Y
	W	W	W
	W	F	F
	F	W	F
	F	F	F
OR	X	Y	X OR Y
	W	W	W
	W	F	W
	F	W	W
	F	F	F

■ **Beispiele:**
(3>5) OR (NOT(4<>2^2))
(4*3>13) AND (4=2^2)

▼ *Übung 3:*
Ermitteln Sie die Ergebnisse der obigen logischen Operationen! Zeichenketten können mit dem Verkettungsoperator verknüpft werden.

■ **Beispiel:**

```
10  A¤="LEITER"
20  B¤="HALB"
30  PRINT B¤+A¤,A¤+"SPROSSE"
```

Im Ergebnis dieses Miniprogramms werden die durch Verkettung entstandenen Begriffe HALBLEITER und LEITERSPROSSE ausgegeben.

3.2.5. Standardfunktionen

Man unterscheidet zwischen numerischen Standardfunktionen und Zeichenkettenstandardfunktionen. Tabelle 3.3. und Tabelle 3.4. geben eine Übersicht über die häufigsten Standardfunktionen:

Tabelle 3.3. Numerische Standardfunktionen

numerische Standard-funktionen	mathematische Schreibweise	Bedeutung
SQR(X)	$\sqrt{x}$	Quadratwurzel, x>=0
EXP(X)	e^x	Exponentialfunktion
LOG(X)	ln x	natürlicher Logarithmus, x>0
SIN(X)	sin x	Sinus, x im Bogenmaß
COS(X)	cos x	Cosinus, x im Bogenmaß
TAN(X)	tan x	Tangens, x im Bogenmaß
ATN(X)	arctan x	Arcustangens, "x im Bogenmaß
ABS(X)	|x|	Absolutbetrag von x
INT(X)	[x]	größte ganze Zahl <=x
SGN(X)	sgn x	Signum (Vorzeichen) von x $\text{sgn } x = \begin{cases} 1 \text{ für } x>0 \\ 0 \text{ für } x=0 \\ -1 \text{ für } x<0 \end{cases}$
RND[(X)]		Pseudozufallszahl z, 0<=z<1
FIX(X)		ganzzahliger Anteil von x[1])
ASN(X)	arcsin x	Arcussinus[1])
ACS(X)	arccos x	Arcuscosinus[1])
LGT(X)	lg x	dekadischer Logarithmus, x>0[1])

[1]) nur bei einigen Interpretern

Tabelle 3.4. Zeichenkettenstandardfunktion

Zeichenketten-standardfunktionen	Bedeutung
LEN(X$)	Länge von X$
LEFT$(X$,N%)	die ersten N% Zeichen von X$
RIGHT$(X$,N%)	die letzten N% Zeichen von X$ (mitunter auch die letzten Zeichen von X$ ab dem N%-ten Zeichen)
MID$(X$,N%[,M%])	[M% Zeichen von] X$ ab dem N%-ten Zeichen (mitunter statt dessen SEG X$,N%,M%), N%-tes bis M%-tes Zeichen von X$)
STR$(X)	Umwandlung von X in eine Zeichenkette
VAL(X$)	Ermittlung des numerischen Wertes der Zeichenkette X$
ASC(X$)	dezimaler ASCII-Code des ersten Zeichens von X$
CHR$(N%)	dem dezimalen ASCII-Code N% entsprechendes Zeichen
Beispiele für Funktionen, die nur bei einigen Interpretern vorhanden sind:	
INSTR([N%,]X$,Y$)	Position, ab der die Zeichenkette Y$ in der X$
(od. POS(X$,Y$))	[ab Position N%] enthalten ist (liefert den Wert 0, falls Y$ nicht in X$ enthalten ist)
STRING$(N%,X$)	Zeichenkette, bestehend aus N%-mal dem ersten Zeichen von X$
STRING$(N%,M%)	Zeichenkette, bestehend aus N%-mal dem Zeichen mit dem dezimalen ASCII-Code M%
INPUT$(N%)	Eingabe einer Zeichenkette von genau N% Zeichen Länge ohne Betätigung der ENTER-Taste
INKEY$	Eingabe von Zeichen über die Tastatur, ohne die Eingabe mit der ENTER-Taste abzuschließen (auch als Funktion GET üblich)
SPACE$(N%)	Zeichenkette aus N% Leerzeichen
HEX$(X)	Zeichenkette, die den hexadezimalen Wert von X angibt

■ Beispiel:

```
10  X=2.4

20  Y=ABS(2*X)

30  Z=ABS(-X)
```

Y erhält den Wert 4.8 und Z 2.4.
Von den Standardfunktionen sollen hier nur diejenigen noch etwas näher erläutert werden, deren Verwendungsmöglichkeiten nicht ohne weiteres aus Tabelle 3.3. und Tabelle 3.4. ersichtlich sind, insbesondere die Standardfunktionen zur Zeichenkettenverarbeitung.
Bei den Winkelfunktionen ist noch zu beachten, daß das Argument x im Bogenmaß anzugeben ist ($2\pi \triangleq 360°$). Winkelangaben in Grad müssen erst mit der nachstehenden Beziehung umgerechnet werden.
Bogenmaß $\approx 0.017453 \cdot$ Gradzahl.

RND [(X)]
Die RND-Funktion erzeugt eine Reihe von Zufallszahlen Z im Bereich $0 \leqq Z < 1$. Leider ist die Realisierung dieser Standardfunktion bei den verschiedenen BASIC-Interpretern recht unterschiedlich.
Bei manchen Interpretern muß noch eine Zahl X angegeben werden. RND(X) erzeugt dann Zufallszahlen Z im Bereich $0<Z<X$, oder die Angabe von X hat Einfluß auf den Start der Zufallszahlenreihe. Bei anderen Interpretern erzeugt RND eine feststehende Zufallszahlenreihe, und durch die zusätzliche Anweisung RANDOMIZE kann der Startwert der Zufallszahlenreihe verändert werden.
Über die genaue Handhabung der RND-Funktion informiere man sich deshalb im Anwenderhandbuch seines Rechners. Trotz aller Unterschiede ist die RND-Funktion in jedem Falle eine wichtige Funktion bei der Programmierung von Spielen. Man kann den Computer nämlich auch «würfeln» lassen. Wenn RND Zufallszahlen Z mit $0 \leqq Z < 1$ erzeugt, so gewinnt man durch INT (6*RND+1) Würfelergebnisse. Einige Beispiele zeigt Tabelle 3.5.
Verfügt man über eine RND-Funktion RND(X), die Zufallszahlen Z im Bereich $0 \leqq Z < X$ erzeugt, so «würfelt» man mit INT (RND(6)+1). Die durch RND(X) erzeugten Zufallszahlen heißen Pseudozufallszahlen, weil es sich dabei um keine echten Zufallszahlen handelt, sondern um eine gleichverteilte Zahlen-

Tabelle 3.5. Beispiele für mit der RND-Funktion erzeugte Würfelergebnisse

Durch RND erzeugte Zufallszahl	6*RND+1	Würfelergebnis INT (6*RND+1)
0	1	1
0,9	6,4	6
0,11	1,66	1
0,32	2,86	2
0,7	5,2	5

folge, die nach einem bestimmten Algorithmus berechnet wird. Diese Zahlen können jedoch als Ersatz für Zufallszahlen benutzt werden, da die Perioden, in denen sich diese Zahlen wiederholen, recht groß sind.

LEN(X¤)
ermittelt die aktuelle Länge (engl.: length) einer Zeichenkette, d. h., LEN(X¤) gibt die Anzahl der Zeichen von X¤ an. In X¤ enthaltene Leerzeichen werden dabei mitgezählt!

■ Beispiel:

```
10  M%=LEN("BAECKER")
20  N%=LEN("JUERGEN MEIER")
30  A¤="TEL.3981"
40  P%=LEN(A¤)
```

Zeile 10: M% erhält den Wert 7,
Zeile 20: N% erhält den Wert 13,
Zeile 40: P% erhält den Wert 8 (Länge von A¤).

LEFT (X¤, N%)
ermittelt die linken (engl.: left) N% Zeichen von X¤. Enthält X¤ weniger als N% Zeichen, so bleibt X¤ unverändert (bzw. manche Interpreter geben dann auch eine Fehlermeldung aus).

■ Beispiel:

```
10  A¤=LEFT¤("AUSGANG",3)
20  B¤="1140 BERLIN"
30  C¤=LEFT¤(B¤,4)
40  N%=5
50  D¤=LEFT¤(B¤,N%-1)
```

Zeile 10: A¤ erhält den Wert "AUS".
Zeile 30: C¤ erhält den Wert "1140".
Zeile 50: D¤ erhält ebenfalls den Wert "1140", da N% den Wert 4 hat.

RIGHT¤ (X¤, N%)
ermittelt die rechten (engl.: right) N% Zeichen von X¤. Es gilt analog das für LEFT¤ (X¤, N%) Gesagte.

■ Beispiel:

```
10  A¤=RIGHT¤("AUSGANG",4)
20  B¤="1140 BERLIN"
30  C¤=RIGHT¤(B¤,6)
40  N%=5
50  D¤=RIGHT¤(B¤,N%+1)
```

Zeile 10: A¤ erhält den Wert "GANG",
Zeile 30: C¤ erhält den Wert "BERLIN",
Zeile 50: D¤ erhält ebenfalls den Wert "BERLIN", da N%+1 den Wert 6 hat.

MID¤ (X¤, N% [,M%])

ermittelt M% Zeichen von X¤, angefangen beim N%-ten Zeichen. Fehlt das Argument M%, so werden ab dem N%-ten Zeichen noch alle weiteren (rechts vom N%-ten stehenden) Zeichen ermittelt.

■ Beispiel:

```
10  A¤=MID¤("1115 BERLIN-BUCH",6,6)
20  B¤=MID¤("1115 BERLIN-BUCH",6)
30  C¤="4090 HALLE-NEUSTADT"
40  D¤=MID¤(C¤,6,5)
50  N%=6
60  E¤=MID¤(C¤,N%,N%-1)
```

Zeile 10: A¤ erhält den Wert "BERLIN",
Zeile 20: B¤ erhält den Wert "BERLIN-BUCH",
Zeile 40: D¤ erhält den Wert "HALLE",
Zeile 60: E¤ erhält ebenfalls den Wert "HALLE", da N% den Wert 6 und N%−1 den Wert 5 hat.

STR¤ (X)
wandelt einen numerischen Wert in eine Zeichenkette (engl.: string) um. Das Argument X darf auch ein Ausdruck sein. STR¤ (X) kann z. B. in Kombinaten mit der LEN-Funktion zum Feststellen der Länge von Zahlen verwendet werden.

■ Beispiel:

```
10  A=5
20  B=-A/4
30  Z¤=STR¤(A)
40  K¤=STR¤(B-A)
50  L%=LEN(STR¤(B))
```

Es erhält Z¤ den Zeichenkettenwert "5", K¤ den Zeichenkettenwert "−6.25" und L% den numerischen Wert 5 (Länge der Zeichenkette STR¤ (B), welche den Wert "−1.25" hat).

VAL(X¤)
ermittelt den numerischen Wert (engl.: value) einer Zeichenkette X¤. Dies ist nur möglich, wenn die Zeichenkette X¤ mit Zeichen beginnt, denen ein numerischer Wert zugeordnet werden kann (Ziffern, positives oder negatives Vorzeichen, Dezimalpunkt). Nachfolgende Buchstaben oder Sonderzeichen werden dann ignoriert. Beginnt X¤ mit Buchstaben oder Sonderzeichen außer +, − und Dezimalpunkt, so ermittelt VAL(X¤) den Wert 0 (bzw. manche Interpreter geben in diesem Falle eine Fehlermeldung aus).

■ **Beispiel:**

```
10  P¤="-1.5"
20  Q¤="2.ABSCHNITT"
30  W=VAL(P¤)
40  Y=VAL(Q¤)
50  Z=VAL(P¤+"43")
```

Es erhalten W, Y und Z die numerischen Werte −1.5; 2 bzw. −1.543.

ASC(X¤)
ermittelt den dezimalen ASCII-Code des 1. Zeichens von X (siehe Tabelle ASCII-Code im Anhang).

■ **Beispiel:**

```
10  A¤="HALLE-NEUSTADT"
20  B%=ASC(A¤)
30  C%=ASC(RIGHT¤(A¤,8))
```

B% erhält den Wert 72 (dezimaler ASCII-Code von H),
C% erhält den Wert 78 (dezimaler ASCII-Code von N).

CHR¤ (X)
ermittelt das der dezimalen ASCII-Codezahl X entsprechende Zeichen (engl.: character). Der Wert von X muß im Bereich zwischen 0 und 255 liegen. Ist der Wert von X nicht ganzzahlig, wird nur der ganzzahlige Anteil berücksichtigt. Bei komfortablen BASIC-Interpretern ermöglicht CHR¤ (X) in Verbindung mit der Ausgabeanweisung PRINT auch oft die Ausgabe nicht auf dem Bildschirm darstellbarer Zeichen.

■ Beispiel:

```
10  A¤=CHR¤(72)
20  B¤=CHR¤(78)
```

A¤ erhält den Wert "H"
B¤ erhält den Wert "N"

▼ *Übung 4:*
1) Schreiben Sie mit Hilfe von Standardfunktionen
   ln 5
   e^{3a}
   sin 30°
   $e^{-x} - e^{y}$
2) Wie lassen sich mit Hilfe von INT(X) Zahlen X auf ganze Zahlen runden?
3) Welche Werte haben die folgenden Ausdrücke?
   LEN("")
   LEN(" ")
   VAL(STR¤ (138)+CHR¤ (46)+"35 PFENNIGE")
   ASC("%")
   MID¤ (LEFT¤ ("PAUSENZEICHEN",5),2,3)

3.2.6. Ausdrücke

Es wurde schon mehrfach der Begriff «Ausdruck» gebraucht, ohne ihn näher zu erläutern. Den meisten Lesern dürfte aus dem Mathematikunterricht noch geläufig sein, was unter einem arith-

metischen Ausdruck zu verstehen ist. Ganz allgemein besteht in BASIC ein Ausdruck aus Variablen und Konstanten, auf die Funktionen angewendet sein dürfen und die durch Operatoren verknüpft sind.
Im Gegensatz zur mathematischen Schreibweise müssen in BASIC jedoch alle Operatoren aufgeschrieben werden, um dem Interpreter die Abarbeitung des Ausdrucks eindeutig vorzuschreiben.
Der mathematische Ausdruck $a^2+2ab+b^2$ lautet in BASIC A^2+2*A*B+B^2.

Merke:
Exponentialoperator und Multiplikationszeichen dürfen in BASIC nicht weggelassen werden!

Wie in der Mathematik haben auch in BASIC alle Operatoren eine gewisse Priorität, die bestimmt, in welcher Reihenfolge die Operationen vom Rechner durchgeführt werden. Auch in BASIC gilt die Grundregel «Punktrechnung geht vor Strichrechnung». Operatoren gleicher Priorität werden grundsätzlich von links nach rechts abgearbeitet. Wird eine andere Abarbeitungsreihenfolge gewünscht als die durch die Priorität und Reihenfolge der Operatoren vorgeschriebene, so müssen auch in BASIC Klammern gesetzt werden. Das bedeutet, daß Klammern eine höhere Priorität als allen Operatoren eingeräumt wird. Statt des oben erwähnten Ausdrucks könnte man auch $(a+b)^2$ schreiben. In BASIC lautet dieser Ausdruck (A+B)^2. Das Setzen der Klammern bewirkt, daß die Addition von A und B vor der Berechnung des Quadrats ausgeführt wird.
Entsprechend der verwendeten Operatoren können vier Typen von Ausdrücken unterschieden werden: arithmetische Ausdrücke, Zeichenkettenausdrücke, Vergleichsausdrücke und logische Ausdrücke.

Arithmetische Ausdrücke

Die erwähnten Ausdrücke A^2+2*A*B+B^2 und (A+B)^2 sind arithmetische Ausdrücke. Sie liefern als Ergebnis immer einen Zahlenwert. Bestandteile eines arithmetischen Ausdrucks sind numerische Variablen und Konstanten, verknüpft durch

Funktionen, Klammern und arithmetische Operatoren. Funktionen haben dabei eine noch höhere Priorität als Klammern. Es wird zuerst immer das Argument der Funktion berechnet, wobei das Argument selbst wieder ein Ausdruck sein kann.

■ **Beispiel:**
3*(SQR(A^2+C)+B)
Reihenfolge der Operationen:
1) A^2
2) A^2+C
3) SQR(A^2+C)
4) SQR(A^2+C)+B
5) 3*(SQR(A^2+C)+B)
Eine weitere Regel ist, daß 2 arithmetische Operatoren nie unmittelbar aufeinander folgen dürfen. Vorzeichenbehaftete Variablen und Konstanten muß man deshalb in Klammern setzen!

■ **Beispiel:**
A*(−B)
C−(−D)

▼ *Übung 5:*
Wie sehen die folgenden mathematischen Ausdrücke in BASIC aus?
1) (a+b) (a−b)
2) 3(sin a^2+5)
3) $\frac{a^2-b^2}{n+1}$
In welcher Reihenfolge werden die Operationen in den folgenden arithmetischen Ausdrücken ausgeführt?
4) 2*A/B^3
5) 2−(3*SIN((A+B)/2))/4
6) A*B/C−2*ABS(D+2)
7) ABS(1/(SQR(A)+3)−5)

Zeichenkettenausdrücke
bestehen aus Zeichenkettenvariablen und Zeichenkettenkonstanten, verknüpft durch Funktionen und Verkettungsoperator. Zeichenkettenausdrücke haben als Ergebnis einen Zeichenkettenwert. Zeichenkettenfunktionen, die einen numerischen Wert ergeben, können deshalb nicht Bestandteil eines Zeichenkettenausdrucks sein, wohl aber in arithmetischen Ausdrücken auftreten (z.B.: VAL, ASC, LEN).
Arithmetische Ausdrücke sind in Zeichenkettenausdrücken möglich, wenn ihr Ergebnis durch STR¤ in eine Zeichenkette umgewandelt wird.

■ Beispiel:
```
A¤+"NAME"
LEFT¤ (A¤,3)+B¤
LEFT¤ (STR¤ (N+1),2)+A¤
```
Umgekehrt können auch Zeichenkettenausdrücke in arithmetischen Ausdrücken enthalten sein, nämlich als Argument von Funktionen wie VAL, ASC oder LEN.

■ Beispiel:
```
1+LEN(AX+BX)
2*VAL(LEFT¤ (A¤,2)+C¤))
```

Vergleichsausdrücke
entstehen durch Gebrauch von Vergleichsoperatoren zwischen Konstanten, Variablen und Ausdrücken.
Ergebnisse von Vergleichsausdrücken sind die logischen Wahrheitswerte WAHR (W) oder FALSCH (F).
Ein ganz einfacher Vergleichsausdruck wäre z.B. A=1. Hier wird der Wert der Variablen A mit der Konstante 1 verglichen. Man kann einen solchen Vergleichsausdruck als Frage auffassen: «Ist A gleich 1?». Je nachdem, ob diese Frage mit ja oder nein beantwortet wird, liefert der Vergleichsausdruck den Wert WAHR oder FALSCH.
Auch Zeichenketten können miteinander verglichen werden. Wie, wurde bereits im Abschnitt 3.2.4. dargelegt. Treten arith-

metische oder Zeichenkettenausdrücke auf, so werden diese zunächst ausgewertet und erst dann der Vergleich durchgeführt, d.h., Vergleichsoperatoren haben eine niedrigere Priorität als die arithmetischen Operatoren und der Verkettungsoperator. Vergleichsoperatoren sind untereinander alle gleichwertig.

■ Beispiel:

1) A+B<1
   Es wird zuerst der Wert von A+B ermittelt und dann überprüft, ob dieser kleiner als 1 ist oder nicht. Sei z.B. A=0 und B=2, dann ist A+B=2 und 2<1 ist FALSCH.
2) "MONTAG" = A¤+"TAG"
   Dieser Vergleichsausdruck hat genau dann den Wert WAHR, wenn A¤ die Zeichenkette "MON" enthält. Für alle anderen Werte von A¤ liefert der Vergleich den Wert FALSCH.

Logische Ausdrücke

entstehen durch Verknüpfung von Konstanten, Variablen und Ausdrücken mit logischen Operatoren. Den Konstanten, Variablen und Ausdrücken muß dabei ein logischer Wahrheitswert zugeordnet werden können. Wie bereits im Abschnitt 3.2.4. erwähnt, ist die Zuordnung numerischer Werte zu den logischen Wahrheitswerten unterschiedlich realisiert, häufig wird jedoch W die 1 und F die 0 zugeordnet. Der logische Ausdruck NOT A ergibt demnach W, wenn A gleich 0 ist, und F, wenn A gleich 1 ist. Logische Ausdrücke entstehen häufig durch Verknüpfung mehrerer Vergleichsausdrücke, z.B. dann, wenn eine Programmverzweigung von mehreren Bedingungen abhängig gemacht werden soll. Der logische Ausdruck B>=1 AND B<=5 ist für alle Werte von B WAHR, die zwischen 1 und 5 liegen.
Von den logischen Operatoren hat NOT die höchste Priorität, dann folgt AND und schließlich OR. 2 logische Operatoren dürfen nur in Form von AND NOT oder OR NOT ohne Klammern direkt hintereinander stehen. Unmittelbar vor NOT darf kein Ausdruck stehen, da NOT sich nur auf einen, und zwar den nachfolgenden, Operanden bezieht.

■ Beispiel:
A AND NOT B
A AND B OR NOT C
(A OR B) AND C
A<B AND B<C
A*B+5<=10 OR C

▼ *Übung 7:*
Habe A den Wert W, B den Wert F und C den Wert W. Welchen Ergebniswert haben dann die folgenden logischen Ausdrücke?
1) (A OR B) AND C
2) A OR B AND C
3) NOT(A OR B) AND NOT C
4) (13<15 AND B) OR (NOT A OR "AB" <> "AC)")

Zum Schluß dieses Abschnittes wird durch Tabelle 3.6. noch ein Überblick über die Prioritäten, die bei der Abarbeitung von Ausdrücken zu beachten sind, gegeben. Operatoren gleicher Priorität sind nebeneinander aufgeführt. Mitunter haben AND und OR entgegen dieser Darstellung auch die gleiche Priorität.

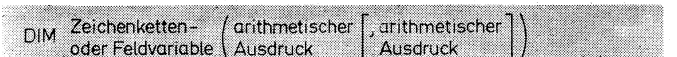
Tabelle 3.6. Prioritäten bei der Abarbeitung von Ausdrücken

höchste Priorität	Funktionen
	Klammern
	Vorzeichen +, –
	Potenzieren ^
	Multiplikation *, Division /
	Addition/Verkettung +, Subtraktion –
	Vergleichsoperatoren <, <=, =, >=, >, <>
	NOT
	AND
niedrigste Priorität	OR

3.3. BASIC-Anweisungen

3.3.1. Dimensionierung von Variablen

Felder, die nicht dimensioniert wurden, werden von den meisten Interpretern automatisch mit maximal 10 Werten je Index angenommen, d.h., für ein 2 dimensionales Feld wird z.B. für $10 \times 10 = 100$ Elemente Speicherplatz reserviert. Man spricht von einer impliziten Feldvereinbarung. In diesem Fall wird der Speicherplatz reserviert, sobald der Name der Feldvariablen das erste Mal im Programm auftaucht. Es ist jedoch nicht vorteilhaft, sich auf diese Art der Feldvereinbarung zu verlassen. Zum einen ist der Speicherplatz der meisten Amateurrechner z.Z. noch recht begrenzt, und man spart Speicherplatz, wenn bei kleineren Feldern die wirklich benötigte Maximalgröße vereinbart wird. Zum anderen muß man Felder, deren Indizes mehr als 10 Werte annehmen können, ohnehin explizit vereinbaren. Es ist daher wegen der besseren Übersichtlichkeit über die benötigten Speicherplätze üblich, grundsätzlich alle Zeichenketten- und Feldvariablen am Anfang des Programmes explizit zu dimensionieren, obwohl dies die Programmiersprache BASIC im Unterschied zu anderen höheren Programmiersprachen nicht zwingend vorschreibt. In BASIC kann die Dimensionsanweisung auch mitten im Programm stehen. Sie muß lediglich vor der 1. Benutzung der zu dimensionierenden Variablen im Programm stehen.
Die Dimensionsanweisung lautet (für maximal 2dimensionale Felder):



■ Beispiel:
10 DIM A¤ (20),B%(15),C(4,2)
vereinbart eine Zeichenkettenvariable A¤ für maximal 20 Zei-

chen, ein 1dimensionales Feld B% mit den 15 Elementen B%(1) bis B%(15) und ein aus 4 Zeilen und 2 Spalten bestehendes 2dimensionales Feld C mit den Elementen C(1,1) bis C(4,2).
Bemerkung:
Bei einigen Interpretern beginnt die Indexzählung nicht bei 1, sondern bei 0. Dann vereinbart die erwähnte DIM-Anweisung für das Feld B% die 15 Elemente B%(0) bis B%(14) und für das Feld C die Elemente C(0,0) bis C(3,1)!

```
20  J%=15
30  DIM A¤ (J%+5,3*J%)
```

vereinbart ein Zeichenkettenfeld, bestehend aus 20 Zeichenketten mit je maximal 45 Zeichen.
Wird das Maximum, das ein Index annehmen kann, wie hier in Form eines arithmetischen Ausdrucks angegeben, so muß den darin enthaltenen Variablen vorher ein Wert zugewiesen worden sein. Im obigen Beispiel durch Zeile 20.
Bemerkung:
Bei einigen kleineren BASIC-Interpretern sind nur Konstanten zur Angabe des Maximalwertes eines Index zugelassen!

▼ *Übung 7:*

1) Dimensionieren Sie folgendes 2dimensionales Feld A:

$$A=\begin{bmatrix} 4 & 10 & 2 & 0 \\ 3 & -2 & 1 & 1 \\ 0 & 0 & -3 & -2 \end{bmatrix}$$

2) Dimensionieren Sie ein Zeichenkettenfeld B, das folgende Zeichenketten aufnehmen soll:
   MUELLER
   LEHMANN
   SCHULZE
   MEIER
   ACKERMANN
   BAUM
   MEIER-MOTZEN
   ELLERT

In der Regel darf jede Variable nur einmal dimensioniert werden. Einige größere BASIC-Interpreter ermöglichen ein soge-

nanntes Redimensionieren (wiederholtes Dimensionieren), nachdem die Variable zuvor durch eine Anweisung (z.B. ERASE) «gelöscht» wurde.

3.3.2. Die Kommentaranweisung

Da längere Programme oft unübersichtlich sind, ist es zweckmäßig, Kommentare (engl.: remark) einzufügen, um die Struktur des Programms zu verdeutlichen. Die Kommentaranweisung

```
REM beliebiger Text
```

hat keinen Einfluß auf den Programmablauf. Die Kommentare sollten aber möglichst knapp gehalten werden, da sonst zuviel Speicherplatz verlorengeht. Beispielsweise könnten mit der REM-Anweisung Beginn und Ende von Unterprogrammen, Programmschleifen usw. hervorgehoben oder die Bedeutung wichtiger Variablen erläutert werden.

■ **Beispiel:**

```
220 REM *** BEGINN UNTERPROGRAMM 1  ***
```

3.3.3. Die Ergibtanweisung

Durch die Ergibtanweisung wird einer Variablen ein Wert zugewiesen. Die Ergibtanweisung hat die Form

```
[LET] Variable=Ausdruck
```

Das =-Zeichen hat hier nicht die Funktion eines Gleichheitszeichens, sondern die eines Zuweisungszeichens, d.h., der Wert des auf der rechten Seite stehenden Ausdrucks wird durch = der Variablen zugewiesen. Beispielsweise wird der Variablen A% durch die Ergibtanweisungen

```
10 LET A%=7
20 LET A%=A%+2
```

zuerst der Wert 7 zugewiesen und anschließend um 2 erhöht.

Durch Zeile 20 wird demnach A% der Wert 9 zugewiesen. Bei den meisten BASIC-Interpretern kann das Schlüsselwort LET bei der Ergibtanweisung weggelassen werden, d. h., statt der obigen Zeilen schreibt man einfacher

```
10 A%=7
20 A%=A%+2.
```

Wichtig ist es, bei der Ergibtanweisung auf den Typ der links von = stehenden Variablen zu achten. Ist es z. B. eine Zeichenkettenvariable, so muß der Ausdruck ein Zeichenkettenausdruck sein, andernfalls gibt der BASIC-Interpreter eine Fehlermeldung über Typunverträglichkeit aus.

▼ *Übung 8:*
Welches sind keine korrekten Ergibtanweisungen?

```
10  LET A¤="BEISPIEL"
20  C+2=7
30  B¤="11.11.86"
40  B¤="DATUM "+B¤
50  LET B¤=SIN(C*1.7)
60  LET D=SQR(D+3*SQR(D))
70  F="ORT"
80  B¤=B¤+17+"UHR"
```

Die meisten BASIC-Interpreter «initialisieren» alle Variablen, d. h., die Variablen werden beim ersten Auftauchen im Programm zunächst mit einem Anfangswert belegt. Dieser Anfangswert ist für numerische Variablen 0 und für Zeichenkettenvariablen ""(die leere Zeichenkette). In diesem Fall kann ein Programm auch mit einer Zeile wie

```
10 A%=A%+1
```

beginnen. A% erhält dann den Wert 0+1, also 1. Es gibt jedoch einige kleinere BASIC-Interpreter, bei denen dies nicht der Fall ist. Dann müssen allen Variablen durch das Programm Anfangswerte zugewiesen werden, sonst ist ihr Zustand undefiniert. (Sie

enthalten dann einen beliebigen unbekannten Anfangswert.) Bei Feldern sollte man dies gleich nach der Dimensionierung tun, und zwar in Form einer Programmschleife, wie sie noch im Abschnitt 3.3.8. beschrieben wird. Das sieht dann so aus:

```
10  DIM B(15)
20  FOR I%=1 TO 15
30  B(I%)=0
40  NEXT I%
```

Hier wird den 15 Elementen B(1) bis B(15) des Feldes B der Wert 0 zugewiesen.

3.3.4. Das Einlesen von Konstanten

Treten bei einem zu programmierenden Problem immer wieder dieselben Konstanten auf (das kann auch ein sich häufig wiederholender Text sein!), so können diese Konstanten mit der DATA-Anweisung im Programm verankert werden. Diese Konstanten brauchen dann an den entsprechenden Stellen nur noch durch eine READ-Anweisung eingelesen zu werden.
Die DATA-Anweisung hat die Form

```
DATA Konstante {[,Konstante]}
```

■ Beispiel:

```
100  DATA 2,4,8,16,32
110  DATA "NAME","VORNAME","TEL.-NR."
120  DATA 1;"BERUF",0.05,"TAETIGKEIT"
```

DATA-Anweisungen können an beliebiger Stelle im Programm stehen. Das Einlesen beginnt bei der DATA-Anweisung mit der niedrigsten Zeilennummer.
Durch eine READ-Anweisung der Form

```
READ Variable {[,Variable]}
```

werden die Konstanten eingelesen und ihre Werte den Variablen der READ-Anweisung zugewiesen. Feldvariablen weist die READ-Anweisung dabei so viele Werte zu, wie es deren Dimensionierung verlangt.

■ **Beispiel:**

```
 10  DIM A(4)
 20  DATA 1,3,0.05,-7.4,"BEISPIEL"
      .
      .
      .
100  READ A,B¤
      .
      .
      .
```

Die Variable A steht als erste nach READ. Da es sich um eine Feldvariable mit 4 Elementen handelt, werden ihr die ersten 4 Werte der DATA-Anweisung zugewiesen. Danach erhält die Variable B¤ den 5. Wert der DATA-Anweisung. Dies könnte natürlich auch mit der Ergibtanweisung programmiert werden und würde dann folgendermaßen aussehen:

```
 10  DIM A(4)
      .
      .
      .
100  A(1)=1
110  A(2)=3
120  A(3)=0.05
130  A(4)=-7.4
140  B¤="BEISPIEL"
      .
      .
      .
```

Um die vereinbarten Konstanten mehrmals einlesen zu können, wird eine weitere Anweisung benötigt, die RESTORE-Anweisung:

Die RESTORE-Anweisung positioniert den DATA-Zeiger (engl.: pointer). Steht vor einer READ-Anweisung keine RESTORE-Anweisung oder fehlt bei der RESTORE-Anweisung nach dem Schlüsselwort RESTORE die Angabe der Zeilennummer, so wird der Zeiger auf die 1. Konstante der DATA-Anweisung mit der niedrigsten Zeilennummer eingestellt. Nach dem Einlesen der 1. Konstante rückt der Zeiger auf die 2. Konstante weiter usw., bis alle Konstanten einer DATA-Anweisung eingelesen sind. Dann wird der Zeiger auf die 1. Konstante der nächsten DATA-Anweisung gesetzt usw., bis alle READ-Anweisungen abgearbeitet wurden. Sind für eine READ-Anweisung keine Konstanten mehr vorhanden, gibt der Interpreter eine Fehlermeldung aus.
Durch Angabe einer Zeilennummer kann mit der RESTORE-Anweisung der Zeiger auf eine beliebige DATA-Anweisung vor- oder auch zurückgesetzt werden.

■ Beispiel:

```
 10  DATA 1,10,100
 20  DATA -1,-10,-100
 30  READ A,B,C
      .
      .
      .
150  RESTORE
160  READ X,Y,Z,A,B,C
      .
      .
      .
270  RESTORE 20
280  READ X,Y,Z
      .
      .
      .
```

In Zeile 30 werden den Variablen A, B und C die Werte der 1. Data-Anweisung zugewiesen. Der DATA-Zeiger zeigt danach auf die 1. Konstante der 2. DATA-Anweisung. Durch Zeile 150 wird der DATA-Zeiger auf die 1. Konstante der 1. DATA-Anweisung zurückgesetzt. In Zeile 160 erhalten die Variablen X, Y und Z die Werte der 1. DATA-Anweisung und die Variablen A, B, C die Werte der 2. DATA-Anweisung. Der DATA-Zeiger steht danach am Ende der 2. DATA-Anweisung, und ein weiteres Einlesen von Konstanten wäre ohne RESTORE nicht möglich. Zeile 270 setzt den DATA-Zeiger auf die 1. Konstante der 2. DATA-Anweisung zurück. Die Werte der 2. DATA-Anweisung werden dann durch Zeile 280 den Variablen X, Y, Z zugewiesen.

▼ *Übung 9:*

Welche Werte haben die Variablen A und B nach Ausführung des folgenden Programms?

```
 10  DIM A(3)
 20  DATA 1,1,3,18,4
 30  READ A
 40  RESTORE 20
 50  READ B,A(3)
 60  B=B+A(2)
 70  RESTORE 80
 80  DATA 4,5
 90  READ A
100  B=B+A(2)
110  DATA 1
```

3.3.5. Die Eingabeanweisung

Für Variablen, deren Wert sich ständig ändert oder denen erst während der Programmabarbeitung vom Bediener ein Wert zugewiesen werden soll, sind die Ergibtanweisung und die Konstantenzuweisung nicht geeignet. Dafür benutzt man die Eingabeanweisung, die im allgemeinen die Form

INPUT [Zeichenkettenkonstante,] Variable {[,Variable]}

hat. Einige BASIC-Interpreter lassen jedoch nur eine Variable je INPUT-Anweisung zu, d.h., man muß dann so viele INPUT-Anweisungen programmieren, wie Variablenwerte eingegeben werden sollen. Bei manchen Interpretern gibt es auch eine gesonderte Eingabeanweisung für Zeichenkettenvariablen. Diese hat meist die gleiche Form, jedoch heißt das Schlüsselwort dann LINE INPUT (oder abgekürzt LINPUT). Im folgenden wird aber nur die INPUT-Anweisung verwendet.
Beim Abarbeiten einer INPUT-Anweisung erscheint auf dem Bildschirm ein Fragezeichen. Damit fordert das Programm zu einer Eingabe über die Tastatur auf. Jede Eingabe wird durch Betätigen der Taste ENTER (entspricht bei manchen Tastaturen ET1, RETURN oder ENTRY) abgeschlossen. Nachdem die Eingabe beendet ist, arbeitet das Programm weiter, und zwar beginnend mit der der INPUT-Anweisung folgenden Programmzeile.

■ **Beispiel:**

```
10  DIM B¤(20)
    .
    .
    .
50  INPUT A,B¤
60  A=A+20
    .
    .
    .
```

Für A soll 17.5 eingegeben werden, für B□ der Text "DAS IST EIN BEISPIEL". Nachdem das 1. Fragezeichen auf dem Bildschirm erscheint, gibt der Bediener über die Tastatur 17.5 und anschließend den Befehl ENTER ein:
?17.5 (ENTER)
Nach dem Betätigen der ENTER-Taste erscheint nun wieder ein Fragezeichen auf dem Bildschirm und fordert zur 2. Eingabe auf. Nach
?DAS IST EIN BEISPIEL (ENTER)
arbeitet das Programm schließlich ab Zeile 60 weiter.
Bei Eingabe mit der INPUT-Anweisung brauchen Zeichenketten also nicht durch "-Zeichen begrenzt zu werden.
Die Ausgabe des Fragezeichens kann durch Voranstellen einer Zeichenkettenkonstante vor die Variablenliste unterbunden werden. Anstelle des Fragezeichens erscheint dann diese Zeichenkettenkonstante als Eingabeaufforderung auf dem Bildschirm.

■ Beispiel:
50 INPUT "EINGABE VON A: ",A

3.3.6. Einfache Ausgabe

Für die Ausgabe steht in BASIC die PRINT-Anweisung zur Verfügung. Wie der Name verrät (PRINT, dt.: drucke), war diese Anweisung ursprünglich zur Ausgabe über Drucker gedacht. Bei den heutigen Mikrorechnern wird die PRINT-Anweisung jedoch in erster Linie zur Ausgabe auf dem Bildschirm verwendet, und für die Ausgabe über Drucker werden oft zusätzliche Ausgabeanweisungen benötigt, z. B. L PRINT (LINE PRINT). Da Amateuren nur selten ein Drucker zur Verfügung stehen wird und die Druckerunterstützung durch die verschiedenen BASIC-Versionen recht unterschiedlich ist, beschränken sich die folgenden Ausführungen auf die Bildschirmausgabe. Die einfache Ausgabeanweisung lautet:

PRINT [Ausdruck] {[;oder,[Ausdruck]]}

Die Ausdrucke dürfen von verschiedenen Typen sein, d.h., arithmetische und Zeichenkettenausdrücke können in ein und derselben PRINT-Anweisung stehen.
Nach jeder Zahl erscheint ein Leerzeichen, bei positiven Zahlen wird anstelle des Vorzeichens + vor der Zahl ebenfalls ein Leerzeichen ausgegeben. Zeichenketten werden ohne voran- oder nachgestellte Leerzeichen ausgegeben.
Verwendet man als Trennungszeichen das Semikolon, so werden die Werte der Ausdrücke hintereinander gedruckt.

■ Beispiel:

```
10  A%=-5
20  B%=6
30  C¤="ENDE"
40  PRINT "A=";A%;"B=";B%;C¤
```

Auf dem Bildschirm erscheint:
A=−5 B= 6 ENDE

Wird als Trennungszeichen das Komma gewählt, liegt die Ausgabe im sogenannten Standardspaltenformat vor. Das Standardspaltenformat ist jedoch vom vorhandenen Bildschirm abhängig. Üblich sind unter anderem Bildschirme mit 40, 64 oder 80 Zeichen je Zeile. Entsprechend teilt das Standardspaltenformat den Bildschirm z.B. in 4 oder 5 Spalten zu je 10, 12, 14 oder 15 Zeichen ein. Zahlen werden wieder mit vorangestelltem Leerzeichen bzw. negativem Vorzeichen ausgegeben. Die Werte durch Komma getrennter Ausdrücke erscheinen so, daß jeder Wert an einem Spaltenanfang beginnt.

■ Beispiel:

```
10  A%=-3
20  B=7.91846
30  C¤="EIN BEISPIEL"
40  PRINT A%,B,C¤,B,A%
```

Auf einem Bildschirm, der in 4 Spalten zu je 10 Zeichen eingeteilt wird, würde folgender Bildschirminhalt erzeugt:

```
-3            7.91846   EIN BEISPIEL
 7.91846      -3
```

In einer PRINT-Anweisung dürfen auch beide Trennungszeichen vorkommen. Mit veränderter Zeile 40 des obigen Beispiels 40 PRINT A%;B;C□ ,B;A% erscheint auf dem Bildschirm:

```
-3  7.91846  EIN BEISPIEL     7.91846
-3
```

Weitere Möglichkeiten der Formatierung der Ausgabe werden im Abschnitt 3.3.11. behandelt.
Eine PRINT-Anweisung ohne Argument bewirkt einen Zeilenvorschub.

■ **Beispiel:**

```
10  PRINT "ANFANG"

20  PRINT

30  PRINT "ENDE"
```

Zeile 20 erzeugt auf dem Bildschirm zwischen der Ausgabe der Worte ANFANG und ENDE eine Leerzeile.
Da die PRINT-Anweisung sehr oft benötigt wird, gibt es bei vielen Interpretern ein Kürzel für das Schlüsselwort PRINT. Meist ist es das Fragezeichen, mitunter auch der Doppelpunkt oder ein anderes Sonderzeichen. Obiges Beispiel sieht dann etwa so aus:

```
10  ? "ANFANG"
20  ?
30  ? "ENDE"
```

Wird eine PRINT-Anweisung durch Komma oder Semikolon abgeschlossen, so beginnt die nächste PRINT-Anweisung die Ausgabe auf derselben Zeile, andernfalls wird nach jeder PRINT-Anweisung ein Zeilenvorschub ausgeführt, d.h., die nächste PRINT-Anweisung beginnt die Ausgabe am Anfang der folgenden Zeile. Paßt ein auszugebender Wert nicht mehr in die alte Zeile, wird die Ausgabe automatisch am Anfang der nächsten Zeile fortgesetzt.

■ Beispiel:

```
10  A¤="MUSTER"
20  B¤="BEISPIEL"
30  PRINT A¤
40  PRINT B¤
50  PRINT A¤;
60  PRINT B¤;"---";A¤+B¤;
70  PRINT "---";A¤+B¤
```

Bei einem Bildschirm mit 40 Zeichen je Zeile erzeugt dieses Programm folgenden Bildschirminhalt:
MUSTER
BEISPIEL
MUSTERBEISPIEL---MUSTERBEISPIEL---
MUSTERBEISPIEL

3.3.7. Bedingte und unbedingte Sprünge, Sprungverzweigungen

Soll unabhängig von einer Bedingung zu einer bestimmten Zeilennummer gesprungen werden, so kann dies durch einen unbedingten Sprung der Form

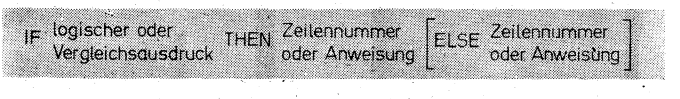
```
GOTO Zeilennummer
```

realisiert werden.

■ **Beispiel:**

```
 10  REM *** PROGRAMM ZUR BERECHNUNG DES VOLUMENS
 20  REM      GERADER KREISZYLINDER ***
 30  P=3.141593
 40  PRINT
 50  INPUT "HOEHE DES ZYLINDERS H=",H
 70  INPUT "RADIUS DER GRUNDFLAECHE R=",R
 90  V=P*R*R*H
100  PRINT "VOLUMEN DES ZYLINDERS V=";V
110  GOTO 40
```

Dieses Programm berechnet nach der Formel $V=\pi r^2 h$ das Volumen von geraden Kreiszylindern. Der unbedingte Sprung in Zeile 110 bewirkt, daß sofort nach der Berechnung und Ausgabe des Volumens eines Zylinders erneut zur Eingabe der Höhe eines weiteren Zylinders aufgefordert wird.
Meist soll jedoch nur dann ein Sprung ausgeführt werden, wenn eine bestimmte Bedingung erfüllt ist – im obigen Programm wäre das ein Rücksprung, wenn man wirklich noch das Volumen eines weiteren Zylinders berechnen will. In jedem anderen Fall kann das Programm beendet werden. Dies läßt sich durch einen bedingten Sprung verwirklichen:



Hat der Ausdruck den Wert WAHR, so wird zu der nach THEN stehenden Zeilennummer gesprungen bzw. die nach THEN stehende Anweisung ausgeführt. Hat der Ausdruck den Wert FALSCH und es ist ein ELSE-Zweig angegeben, so wird zu der nach ELSE stehenden Zeilennummer gesprungen bzw. die nach ELSE stehende Anweisung ausgeführt. Hat der logische Ausdruck den Wert FALSCH und es ist kein ELSE-Zweig angegeben, so wird gar kein Sprung ausgeführt, sondern die Programm-

abarbeitung mit der nächsten Anweisung fortgesetzt. Bild 3.1 und Bild 3.2 verdeutlichen die verbalen Ausführungen.

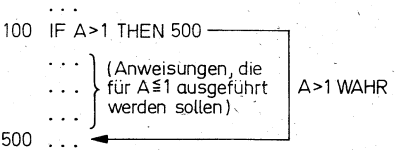


Bild 3.1:
Abarbeitung eines einfachen bedingten Sprungs

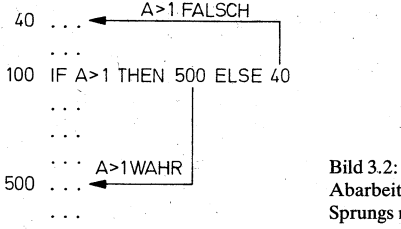


Bild 3.2:
Abarbeitung eines bedingten Sprungs mit ELSE-Zweig

■ Beispiele:

```
10  M=3
20  IF B=1 THEN M=M+1
30  PRINT M
```

Hat B den Wert 1, so ist B=1 WAHR und es wird die nach THEN stehende Anweisung M=M+1 ausgeführt. Für M wird der Wert 4 ausgegeben.

```
10  A=5
20  IF N>=1 AND N<=4 THEN 40
30  A=0
40  PRINT A
```

Hat B einen anderen Wert als 1, so liefert der Vergleich B=1 den Wert FALSCH und wird für M der Wert 3 ausgegeben.

Hat N einen Wert, der zwischen 1 und 4 liegt, so ist N> =1 AND N< =4 WAHR, und es wird zu Zeile 40 gesprungen. Für A erscheint der Wert 5.
Hat N einen Wert, der kleiner als 1 oder größer als 4 ist, so liefert der Ausdruck N =1 AND N =4 den Wert FALSCH. Es wird mit Zeile 30 fortgesetzt und für A der Wert 0 ausgegeben.

```
110  M=M+1
120  IF M<>N THEN 110 ELSE 600
```

Haben die Variablen M und N verschiedene Werte (d.h., ist M<>N WAHR), so wird zu Zeile 110 zurückgesprungen, andernfalls wird mit Zeile 600 fortgesetzt.
Einige kleinere BASIC-Interpreter verfügen gar nicht über den ELSE-Zweig, andere betrachten ELSE als gesonderte Anweisung. Auch benutzen einige Interpreter beim Sprung zu einer Zeilennummer anstelle von THEN das Schlüsselwort GOTO. Darüber hinaus können bei größeren BASIC-Interpretern nach THEN und ELSE ganze Anweisungsblöcke stehen. Das ist möglich, weil diese Interpreter entweder mehrere Anweisungen in einer Programmzeile zulassen oder über «Klammern» zum Kennzeichen von Blockanfang und Blockende (z.B. DO und DOEND) verfügen. Darauf soll hier aber nicht weiter eingegangen werden.
Das Programmbeispiel kann nun ab Zeile 110 neu formuliert werden:

```
 10  REM *** PROGRAMM ZUR BERECHNUNG DES VOLUMENS
 20  REM      GERADER KREISZYLINDER ***
 30  P=3.141593
 40  PRINT
 50  INPUT "HOEHE DES ZYLINDERS H=",H
 70  INPUT "RADIUS DER GRUNDFLAECHE R=",R
 90  V=P*R*R*H
100  PRINT "VOLUMEN DES ZYLINDERS V=";V
```

```
110  PRINT
120  PRINT
130  INPUT "WEITERE ZYLINDER ?",A¤
140  IF A¤="JA" THEN 40
150  PRINT "DANN EBEN NICHT!  ENDE!"
```

Die Zylinderberechnung wird genau dann fortgesetzt, wenn man auf die Frage "WEITERE ZYLINDER?" JA antwortet.
Lautet die Antwort JAWOHL oder VIELLEICHT, dann wird das Programm ebenso beendet, als ob NEIN geantwortet worden wäre.

▼ *Übung 10:*
Formulieren Sie das Programmbeispiel neu, und zwar so, daß das Programm nur bei der Antwort NEIN beendet wird. Lautet die Antwort auf die Frage "WEITERE ZYLINDER?" weder JA noch NEIN, so soll das Programm eine Fehlermeldung ausgeben. Die erforderlichen Anweisungen sind zwischen Zeile 140 und Zeile 150 einzufügen.
Außer den beiden bisher beschriebenen Sprungmöglichkeiten gibt es bei den meisten BASIC-Versionen noch sogenannte Sprungverzweigungen oder Verteilersprünge der Form

ON arithmetischer Ausdruck GOTO Zeilennummer {[, Zeilennummer]}

Der arithmetische Ausdruck wird ausgewertet und das Ergebnis auf die nächste ganze Zahl n aufgerundet. Anschließend springt das Programm zu der Zeilennummer, die als n-te nach GOTO steht. Befindet sich an dieser Stelle keine Zeilennummer, so wird kein Sprung ausgeführt, sondern das Programm mit der nächsten Anweisung fortgesetzt.
Bild 3.3 zeigt die Abarbeitung einer solchen Sprungverzweigung.

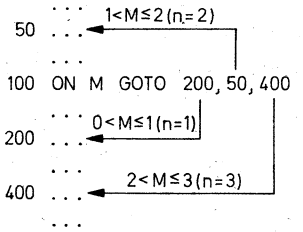


Bild 3.3:
Abarbeitung einer Sprungverzweigung

■ **Beispiel:**

```
 10  M=0
 20  ON M/3 GOTO 100,170,220
     ...
200  M=M+2
210  GOTO 20
220  END
```

Zeile 20 wird 5mal durchlaufen, und zwar:

1. Durchlauf:	M=0	M/3=0	n=0	kein Sprung
2. Durchlauf:	M=2	M/3=2/3	n=1	Sprung nach Zeile 100
3. Durchlauf:	M=4	M/3=4/3	n=2	Sprung nach Zeile 170
4. Durchlauf:	M=6	M/3=2	n=2	Sprung nach Zeile 170
5. Durchlauf:	M=8	M/3=8/3	n=3	Sprung nach Zeile 220

▼ *Übung 11:*
Formulieren Sie das Programmbeispiel ab Zeile 130 neu, indem Sie als Antworten 1 (anstelle von JA) und 0 (anstelle von NEIN) benutzen und zur Auswertung der Antwort einen Verteilersprung verwenden!

3.3.8. Programmschleifen

Programmschleifen benötigt man immer dann, wenn eine Folge von Anweisungen mehrfach durchlaufen werden soll. Ein Bei-

spiel dafür ist die Zylindervolumenberechnung aus Abschnitt 3.3.7., die Werte für h und r werden immer neu eingegeben, die Berechnungsvorschrift bleibt dieselbe. Das wiederholte Durchlaufen der Eingabe-, Berechnungs- und Ausgabeanweisungen wurde nacheinander durch die verschiedenen möglichen Sprunganweisungen (unbedingter Sprung, bedingter Sprung, Sprungverzweigung) realisiert. Ein häufiger Anwendungsfall für Programmschleifen sind Tabellen und Übersichten. Beispielsweise wenn man für eine konstante Höhe h=10cm die Zylindervolumina für verschiedene Radien r=1, 1.5, 2, 2.5, ..., 5cm tabellarisch ausgeben will. Der Radius r soll also den Anfangswert 1 haben und um die Schrittweite 0,5 erhöht werden, bis der Endwert 5 erreicht ist. Diese Programmschleife kann man auch durch den bereits bekannten bedingten Sprung erzeugen:

```
 10  P=3.141593
 20  H=10
 30  PRINT "RADIUS","VOLUMEN"
 40  PRINT "------","-------"
 50  R=1
 60  V=P*R*R*H
 70  PRINT R,V
 80  R=R+0.5
 90  IF R<=5 THEN 60
100  PRINT "______","_______"
```

Dieses Programm liefert folgende Tabelle:

RADIUS	VOLUMEN
1	31.4159
1.5	70.6859
2	125.664
2.5	196.35

3	282.743
3.5	384.845
4	502.655
4.5	636.173
5	785.398

Schleifen dieser Art lassen sich aber mit der Laufanweisung einfacher darstellen. Die Laufanweisung lautet:

```
FOR numerische Variable = arithmetischer Ausdruck TO arithmetischer Ausdruck [STEP arithmetischer Ausdruck]
```

Dazu gehört noch die NEXT-Anweisung:

```
NEXT numerische Variable
```

Sie bezeichnet das Ende der Programmschleife; die numerische Variable muß dabei mit der numerischen Variablen nach FOR übereinstimmen. Diese numerische Variable heißt Laufvariable, den arithmetischen Ausdruck nach STEP nennt man Schrittweite. Ist die Schrittweite 1, so kann diese Angabe weggelassen werden. FOR I=1 TO 4 ist also gleichbedeutend mit FOR I=1 TO 4 STEP 1. Die Variable I erhält als Anfangswert 1, der dann schrittweise um 1 erhöht wird, bis der Endwert 4 erreicht ist.

■ Beispiele für Laufanweisungen:

`FOR I=1 TO 4`	`i=1, 2, 3, 4`
`FOR J=3 TO 12 STEP 3`	`j=3, 6, 9, 12`
`FOR Y=1 TO 0 STEP -0.3`	`y=1, 0.7, 0.4, 0.1`
`FOR Z=10 TO 5 STEP 2`	`z=10`
`FOR K=A+1 TO A*2 STEP A/5`	`z.B. sei a=10, dann ist k=11, 13, 15, 17, 19`

Das Programmbeispiel kann nun mit Hilfe der Laufanweisung neu formuliert werden:

```
10  P=3.141593
20  H=10
30  PRINT "RADIUS","VOLUMEN"
40  PRINT "------","-------"
50  FOR R=1 TO 5 STEP 0.5
60  V=P*R*R*H
70  PRINT R,V
80  NEXT R
90  PRINT "______","_______"
```

▼ *Übung 12:*

1) Gegeben seien die Programmzeilen:

```
200  N=12
210  M=0.01
220  FOR I=N/4 TO N/3+1 STEP 50*M
     ...
300  NEXT I
```

Welche Werte durchläuft die Laufvariable i?

2) Tabellieren Sie mit Hilfe der Laufanweisung die Quadratzahlen x^2 und die Kubikzahlen x^3 für die Werte x=1, 2, ..., 10!

Programmschleifen können auch mehrfach auftreten, d.h., sie lassen sich ineinander schachteln, dürfen sich dabei aber nicht kreuzen (die zuletzt eröffnete Schleife muß stets als erste wieder durch eine NEXT-Anweisung geschlossen werden). Doppelschleifen (Bild 3.4) verwendet man häufig, wenn Feldern Werte zugewiesen und Tabellen erzeugt werden sollen.

Durch das folgende Programm werden die natürlichen Zahlen 1, 2, ... , 10 sowie deren Quadrate und 3. und 4. Potenzen in einem 2dimensionalen Feld A untergebracht.

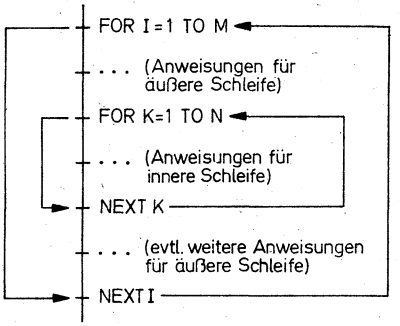


Bild 3.4:
Struktur einer Doppelschleife

■ Beispiel:

```
10  DIM A(10,4)
20  FOR I=1 TO 10
30  FOR K=1 TO 4
40  A(I,K)=I^K
50  NEXT K
60  NEXT I
```

Jedes Feldelement A(I, K) enthält die i-te Zahl in der k-ten Potenz, A(4,2) also $4^2 = 16$.
Tabelle 3.7. zeigt den Inhalt des Feldes A:

Tabelle 3.7. Inhalt der Feldvariablen

K I	1	2	3	4
1	1	1	1	1
2	2	4	8	16
3	3	9	27	81
4	4	16	64	256
5	5	25	125	625
6	6	36	216	1296
7	7	49	343	2401
8	8	64	512	4096
9	9	81	729	6561
10	10	100	1000	10 000

Auch im Programm zum Berechnen des Zylindervolumens kann man eine Doppelschleife verwenden. Beispielsweise wenn eine Tabelle erstellt werden soll, die die Zylindervolumina für h = 5, 6, 7 und r = 1, 1.5, ..., 5 enthält.
Die äußere Schleife wird durch den Radius der Zylinder gebildet, die innere Schleife durch die Höhe h:

```
 10  P=3.141593
 20  PRINT "R","H=5","H=6","H=7"
 30  PRINT "================================================"
 40  PRINT
 50  FOR R=1 TO 5 STEP 0.5
 60  PRINT R,
 70  FOR H=5 TO 7
 80  V=P*R^2*H
 90  PRINT V,
100  NEXT H
110  PRINT
120  NEXT R
```

Das Programm erzeugt folgende Tabelle:

```
R              H=5            H=6            H=7
=====================================================

 1              15.708         18.8496        21.9912
 1.5            35.3429        42.4115        49.4801
 2              62.8319        75.3983        87.9646
 2.5            98.1748        117.81         137.445
 3              141.372        169.646        197.92
 3.5            192.423        230.907        269.392
 4              251.328        301.593        351.858
 4.5            318.086        381.704        445.321
 5              392.699        471.239        549.779
```

Zeile 110 bewirkt dabei, daß für jeden Radius eine neue Zeile angefangen wird.

Bei der Laufanweisung ist es nicht erlaubt, in eine Schleife hineinzuspringen. In diesem Fall stößt der Interpreter auf eine NEXT-Anweisung, ohne daß eine Schleife eröffnet wurde, und gibt eine Fehlermeldung aus (z. B. "NEXT WITHOUT FOR").

■ **Beispiel:**

```
150  GOTO 350
  .
  .
  .
200  FOR J=1 TO 6
  .
  .
  .
350  PRINT V
  .
  .
  .
400  NEXT J
```

Der unbedingte Sprung in Zeile 150 überspringt die Schleifeneröffnung in Zeile 200.
Die Anweisungen ab Zeile 350 werden ausgeführt. In Zeile 400 soll schließlich NEXTJ («das nächste J») berechnet werden. Nun ist der Interpreter in Verlegenheit, da ihm wichtige Informationen durch Überspringen von Zeile 200 vorenthalten wurden. Er kennt ja weder den Startwert von J ("J=1"), noch weiß er, wie er das nächste J berechnen soll («erhöhe um Schrittweite 1»). Deshalb bricht er an dieser Stelle die Programmabarbeitung ab und gibt eine Fehlermeldung aus.

3.3.9. Unterprogrammtechnik

In größeren Programmen kommt es oft vor, daß sich bestimmte Berechnungen häufig wiederholen. Man könnte sie natürlich so oft in das Programm einfügen, wie sie benötigt werden. Dadurch wird das Programm aber sehr lang und unübersichtlich, es entsteht ein sogenanntes «Spaghettiprogramm». Deshalb ist es

zweckmäßig, solche Programmteile als Unterprogramme zu formulieren. Die Unterprogramme werden dann an den entsprechenden Stellen im Hauptprogramm aufgerufen und führen die benötigten Berechnungen so oft durch, wie man sie aufruft. Die Unterprogramme selbst stehen dabei nur einmal da, so daß Speicherplatz und Eingabeaufwand gespart werden. Dazu gibt es in BASIC die Anweisungen GOSUB (engl.: go to subroutine, d.h. springe zum Unterprogramm) und RETURN (d.h. kehre zurück, gemeint ist die Rückkehr ins Hauptprogramm). Die Anweisung GOSUB steht im Hauptprogramm an den Stellen, wo das Unterprogramm aufgerufen werden soll. Das Unterprogramm ruft man durch

```
GOSUB Zeilennummer des Unterprogramms
```

auf.
Die Anweisung RETURN kennzeichnet das Ende des Unterprogramms und realisiert den Rücksprung in das Hauptprogramm. Jedes Unterprogramm muß mit RETURN abgeschlossen werden. Die Anweisung lautet einfach

```
RETURN
```

und bewirkt, daß in das Hauptprogramm zu der dem letzten Aufruf dieses Unterprogramms folgenden Programmzeile zurückgesprungen wird. Der Rechner arbeitet dann das Hauptprogramm ab dieser Zeile weiter ab.
Es ist üblich, die Unterprogramme nach dem Hauptprogramm anzuordnen. Dadurch entsteht jedoch das Problem, daß die zuletzt abzuarbeitende Anweisung des Hauptprogramms nicht mehr die letzte Zeile des gesamten Programms ist. Deshalb benötigt man noch eine Anweisung zum Kennzeichnen des Hauptprogrammendes. Dies ist die Anweisung

```
END
```

Sie wird auch ganz allgemein dann verwendet, wenn das logische

Ende eines Programms sich nicht in der letzten Programmzeile befindet. Das eben Gesagte wird durch Bild 3.5 verdeutlicht.

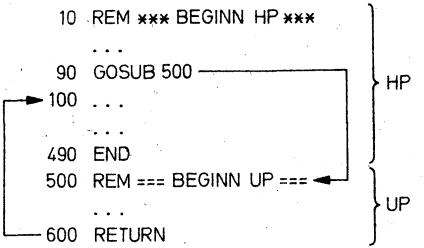


Bild 3.5:
Abarbeitung eines Unterprogramms

Häufig möchte man auszugebende Tabellen oder Ergebnisse mit einer Überschrift versehen. Das Unterstreichen und den Zeilenvorschub könnte z. B. ein Unterprogramm übernehmen und so dem Bediener ein wenig Arbeit abnehmen:

■ **Beispiel:**

```
  10  DIM U¤(40)
      ...   (Berechnungen)
 150  GOSUB 1000
      ...   (Ausgabe Tabelle 1)
 200  GOSUB 1000
      ...   (Ausgabe Tabelle 2)
 300  GOSUB 1000
      ...   (Ausgabe Tabelle 3)
 990  END
1000  REM === UP UEBERSCHRIFT ===
1010  INPUT "",U¤
1020  FOR I=1 TO LEN(U¤)
1030  PRINT "*";
1040  NEXT I
```

```
1050  PRINT
1060  PRINT
1070  RETURN
```

Durch Zeile 1010 wird der Bediener zur Eingabe der Überschrift aufgefordert ("" in der INPUT-Anweisung unterdrückt die Ausgabe des Fragezeichens, denn dies würde bei einer Überschrift stören). Die anschließende Laufanweisung gibt so viele Sternchen zur Unterstreichung der Überschrift aus, wie die Überschrift Zeichen lang ist. Das wird durch die Angabe von LEN(U$) als Endwert der Laufvariablen I erreicht. Die PRINT-Anweisungen erzeugen eine Leerzeile zwischen der Überschrift und der nachfolgenden Ergebnisausgabe.

Mit dem abschließenden Beispiel zur Unterprogrammtechnik kann man die Malfolge üben. Durch das benutzte Unterprogramm wird folgendes realisiert:

- Stellen der Aufgabe.
- Kontrollieren, ob das eingegebene Ergebnis richtig ist, und Zählen der gemachten Fehler.

Das Hauptprogramm benutzt dieses Unterprogramm für 3 verschiedene Zwecke:

- zum Üben der Malfolgen (das Unterprogramm wird in einer Schleife zehnmal aufgerufen);
- zum Üben von Multiplikationsaufgaben aus einer Folge;
- zum Üben von Multiplikationsaufgaben aus allen Folgen durcheinander.

Die im Unterprogramm zu multiplizierenden Werte werden durch die Variablen I% und N% vom Hauptprogramm an das Unterprogramm übergeben. Andererseits wird vom Unterprogramm durch die Variable F% die Fehleranzahl an das Hauptprogramm zurückgegeben. Das bedeutet, daß in BASIC Hauptprogramm und Unterprogramme dieselben Variablen aufweisen. Das Unterprogramm des Programmbeispiels kann nur die Werte von I% und N% miteinander multiplizieren. Will man aber beispielsweise das Unterprogramm auch dazu benutzen, etwa den Wert von 2 Variablen namens A% und B% miteinander zu multiplizieren und die Fehleranzahl in G% abzuspeichern, so müß-

ten dem Unterprogrammaufruf Ergibtanweisungen voran- und nachgestellt werden, etwa
... I%=A%
... N%=B%
... GOSUB 620
... G%=F%.
Um die Betrachtungen zu beenden, sei noch bemerkt, daß diese umständliche Verfahrensweise eine der entscheidenden Schwachstellen von BASIC ist. Andere höhere Programmiersprachen arbeiten wesentlich effektiver mit Unterprogrammen. Dort sind die Ein- und Ausgabevariablen des Unterprogramms sogenannte formale Parameter, die beim Aufruf des Unterprogramms im Hauptprogramm durch die aktuellen Parameter des Hauptprogramms ersetzt werden.
Doch nun zu unserem Programmbeispiel:

```
 10 DIM M¤(2),Z%(9),M%(3)
 20 PRINT "PROGRAMM ZUM UEBEN DER MALFOLGEN"
 30 PRINT "------------------------------"
 40 PRINT
 50 INPUT "BITTE GANZE ZAHL ZWISCHEN 2 UND 10 EINGEBEN: ",N%
 60 REM ...Z% REGISTRIERT, WELCHE FOLGE BEREITS GEUEBT WURDE
 70 REM ...WENN S% 9 IST, WURDEN ALLE GEUEBT
 80 Z%(N%-1)=1
 90 S%=0
100 FOR I%=2 TO 10
110 S%=S%+Z%(I%-1)
120 NEXT I%
130 PRINT "GUT, UEBEN WIR DIE ";M¤;"-ER FOLGE!"
140 F%=0
150 FOR I%=1 TO 10
160 GOSUB 570
```

```
170 NEXT I%
180 IF F%>1 THEN 360
190 GOSUB 650
200 IF F%=0 THEN PRINT "SEHR GUT, ALLES RICHTIG!"
210 IF F%=1 THEN PRINT "EIN FEHLER KANN SCHON MAL PASSIEREN."
220 F%=0
230 PRINT "JETZT 3 AUFGABEN DURCHEINANDER:"
240 FOR J%=1 TO 3
250 I%=INT(10*RND+1)
260 REM ...M% VERMEIDET WIEDERHOLUNGEN
270 IF M%(1)=I% OR M%(2)=I% THEN 300
280 M%(J%)=I%
290 GOSUB 570
300 NEXT J%
310 GOSUB 650
320 IF F%>0 THEN 360
330 IF S%=9 THEN 390
340 PRINT "PRIMA! NUN DIE NAECHSTE FOLGE!"
350 GOTO 40
360 PRINT "DU HAST";F%;"FEHLER GEMACHT!"
370 PRINT "WIR UEBEN DIE ";N%;"-ER FOLGE NOCHMAL!"
380 GOTO 140
390 PRINT "BIS JETZT HAT ALLES GEKLAPPT,"
400 PRINT "NUN GEHT'S ABER 5 MAL KREUZ UND QUER!"
410 F%=0
420 FOR J%=1 TO 5
430 I%=INT(10*RND+1)
440 N%=INT(10*RND+1)
```

```
450 GOSUB 570
460 NEXT J%
470 GOSUB 650
480 IF F%=0 THEN 520
490 PRINT "DU HAST";F%;"FEHLER GEMACHT!"
500 PRINT "WIR UEBEN LIEBER NOCH EINE RUNDE!"
510 GOTO 410
520 PRINT "GRATULIERE! ALLES RICHTIG!"
530 INPUT "NOCH EINE RUNDE ? (J/N): ",M¤
540 IF M¤="J" THEN 410
550 PRINT "NA DANN TSCHUESS, DU RECHENKUENSTLER!"
560 END
570 REM ===== UP MULTIPLIKATION UND FEHLERZAEHLUNG =====
580 PRINT I%;"*";N%;
590 INPUT "= ",E%
600 IF E%=I%*N% THEN 640
610 PRINT "FALSCH!"
620 REM ...F% ZAEHLT DIE FEHLER
630 F%=F%+1
640 RETURN
650 REM ===== UP BILDSCHIRM LOESCHEN =====
660 FOR I%=1 TO 16
670 PRINT
680 NEXT I%
690 RETURN
```

▼ *Übung 13:*

1) Welche Zeilen müssen in das obige Programm eingefügt werden, wenn der verwendete BASIC-Interpreter die Variablen nicht «initialisiert» (mit Anfangswerten belegt)? Lesen Sie dazu ggf. noch mal im Abschnitt 3.3.3. nach.
2) Was muß in dem ab Zeile 700 stehenden Unterprogramm geändert werden, wenn mit einem 24-Zeilen-Bildschirm gearbeitet wird?
3) Schreiben Sie ein Programm zum Berechnen des Volumens von Zylinderwellen (wie in Bild 3.6 dargestellt).

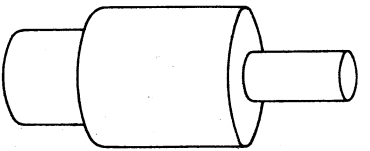


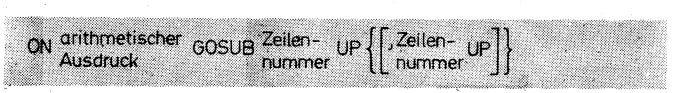
Bild 3.6:
Beispiel für eine Zylinderwelle

Benutzen Sie dabei das Programm zur Berechnung des Zylindervolumens als Unterprogramm, und verwenden Sie als Eingabeparameter N% für die Anzahl der Zylinderstücke, aus denen sich die Welle zusammensetzt, H für die Länge der Teilstücke und R für den Radius der Teilstücke.

Unterprogramme dürfen auch verschachtelt werden, d.h., es kann in einem Unterprogramm ein weiteres Unterprogramm aufgerufen werden usw. Wie oft man Unterprogramme ineinander verschachteln kann, hängt von der Leistungsfähigkeit des verwendeten BASIC-Interpreters ab.

Bild 3.7 zeigt die Abarbeitung zweier ineinander verschachtelter Unterprogramme.

Außerdem gibt es bei den meisten Interpretern analog zur Sprungverzweigung (siehe Abschnitt 3.3.7.) auch eine Mehrfachverzweigung zu Unterprogrammen der Form



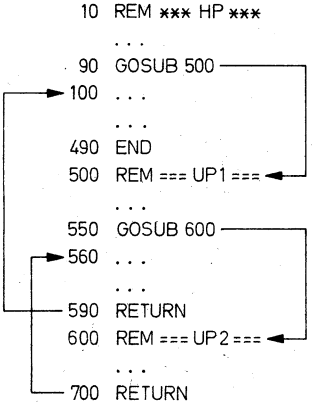


Bild 3.7:
Abarbeitung zweier ineinander verschachtelter Unterprogramme

Die Abarbeitung entspricht der der Sprungverzweigung, nur daß zu einem der nach GOSUB angegebenen Unterprogrammen gesprungen wird. Nach dem Abarbeiten des Unterprogramms gelangt man zu der der Verzweigung folgenden Zeile ins Hauptprogramm zurück.
Die Mehrfachverzweigung zu Unterprogrammen wird oft zum Realisieren der sogenannten Menütechnik eingesetzt. Dabei bekommt der Bediener vom Programm über den Bildschirm ein «Menü» angeboten. Aus diesem «Menü» kann er dann auswählen, mit welchen Teil- oder Unterprogrammen er arbeiten möchte. Menütechnik und Mehrfachverzweigung zu Unterprogrammen werden durch das nächste Programmbeispiel anschaulich dargestellt.

```
10 REM PROGRAMM ZUR VOLUMENBERECHNUNG
20 REM EINIGER GEOMETRISCHER KOERPER
30 REM ------------------------------
40 REM IM MENUE STEHT ABKUERZEND
50 REM ZYLINDER FUER GERADER KREISZYLINDER
```

```
 60 REM KEGEL FUER GERADER KREISKEGEL
 70 REM PYRAMIDE FUER PYRAMIDE MIT QUADRATISCHER GRUNDFLAECHE
 80 REM ...VEREINBARUNGSTEIL
 90 DIM K¤(1),N¤(13)
100 P=3.141593
110 REM ...MENUE
120 PRINT "    VOLUMENBERECHNUNG FUER"
130 PRINT "    ----------------------"
140 PRINT "    QUADER ........1"
150 PRINT "    PYRAMIDE ......2"
160 PRINT "    ZYLINDER ......3"
170 PRINT "    KEGEL .........4"
180 PRINT "    KUGEL .........5"
190 PRINT
200 INPUT "    BITTE KENNZIFFER EINGEBEN: ",K%
210 PRINT
220 REM ...BEHANDLUNG VON EINGABEFEHLERN
230 IF K%<1 OR K%>5 THEN 190
240 REM ...MEHRFACHVERZWEIGUNG ZU DEN BERECHNUNGS-UP
250 ON K% GOSUB 360,460,540,620,700
260 REM ...ERGEBNISAUSGABE
270 PRINT
280 PRINT "DAS VOLUMEN ";N¤;" BETRAEGT ";V
290 PRINT
300 REM ...ABBRUCHTEST
310 INPUT "WUENSCHEN SIE WEITERE BERECHNUNGEN ? (J/N): ",K¤
320 IF K¤="J" THEN 120
330 PRINT
```

```
340 PRINT "AUF WIEDERSEHEN!"
350 END
360 REM ===== UP QUADER =====
370 N¤="DES QUADERS"
380 PRINT "HOEHE ";N¤;
390 INPUT " H=",H
400 PRINT "BREITE ";N¤;
410 INPUT " B=",B
420 PRINT "LAENGE ";N¤;
430 INPUT " L=",L
440 V=H*B*L
450 RETURN
460 REM ===== UP PYRAMIDE =====
470 N¤="DER PYRAMIDE"
480 PRINT "HOEHE ";N¤;
490 INPUT " H=",H
500 PRINT "KANTENLAENGE DER GRUNDFLAECHE";
510 INPUT " L=",L
520 V=L^2*H/3
530 RETURN
540 REM ===== UP ZYLINDER =====
550 N¤="DES ZYLINDERS"
560 PRINT "HOEHE ";N¤;
570 INPUT " H=",H
580 PRINT "RADIUS ";N¤;
590 INPUT " R=",R
600 V=P*R^2*H
610 RETURN
```

```
620 REM ===== UP KEGEL =====
630 N¤="DES KEGELS"
640 PRINT "HOEHE ";N¤;
650 INPUT " H=",H
660 PRINT "RADIUS DER GRUNDFLAECHE";
670 INPUT " R=",R
680 V=P*R^2*H/3
690 RETURN
700 REM ===== UP KUGEL =====
710 N¤="DER KUGEL"
720 PRINT "RADIUS ";N¤;
730 INPUT " R=",R
740 V=4/3*P*R^3
750 RETURN
```

3.3.10. Funktionsdefinition

Außer den in Abschnitt 3.2.5. vorgestellten Standardfunktionen lassen sich in vielen BASIC-Versionen auch selbstdefinierte Funktionen verwenden.
Man definiert eine solche Funktion durch

DEF FN Variablenname [(einfache Variable [, einfache Variable])] = Ausdruck

Der Variablenname folgt ohne Leerzeichen auf FN und stellt zusammen mit FN den Funktionsnamen dar, er muß mit dem Typ des Ausdrucks übereinstimmen. Die einfachen Variablen (d. h., Feldvariablen sind nicht zugelassen) in den runden Klammern sind die sogenannte Parameterliste. Diese Parameterliste muß alle formalen Parameter des rechts vom =-Zeichen stehenden Ausdrucks enthalten.

Zum Berechnen des Zylindervolumens kann man sich selbst eine Funktion definieren:

```
10 DEF FNZ(X,Y)=3.141593*X^2*Y
```

Sie läßt sich im Programm genauso wie eine Standardfunktion verwenden. Das Volumen eines Zylinders mit der Höhe H=2.4 und dem Radius R=1.2 berechnet man durch

```
20 V=FNZ(1.2,2.4)
```

oder aber

```
15 H=2.4
17 R=1.2
20 V=FNZ(R,H)
```

Das bedeutet, daß der Interpreter hier die formalen Parameter X und Y der Funktionsdefinition durch die aktuellen Parameter des Programms ersetzt. Die aktuellen Parameter dürfen auch Ausdrücke sein, dasselbe Ergebnis wie oben liefert z. B. der Funktionsaufruf

```
20 V=FNZ(2.4/2,2+0.4)
```

Zur Ausgabe eines bestimmten Musters auf dem Bildschirm kann man sich auch eine Zeichenkettenfunktion definieren, etwa

```
10 DEF FNM (X¤, Y¤)=X¤+Y¤+Y¤+Y¤+X¤
```

und erhält z. B. durch die Aufrufe

```
20 PRINT FNM ("/*/","==")
30 PRINT FNM ("ANFANG","---")
```

die Bildschirmausschriften

```
/*/======/*/
ANFANG--------ANFANG
```

▼ *Übung 14:*

Schreiben Sie ein Programm zur Berechnung des Volumens von aus Quadern zusammengesetzten Körpern mit Hilfe einer selbstdefinierten Funktion zum Berechnen des Volumens der Teilquader.

3.3.11. Formatierte Ausgabe

Die einfache Ausgabe, wie sie in Abschnitt 3.3.6. vorgestellt wurde, ist nicht sehr komfortabel, wenn es darum geht, Tabellen

oder Schemata auszugeben. Will man ab einer ganz bestimmten Position auf dem Bildschirm etwas ausgeben, so hilft dabei die Tabulatorfunktion TAB(X), die nur im Zusammenhang mit der PRINT-Anweisung gebraucht werden darf. Das Argument X gibt dabei die Position an, ab der ausgegeben werden soll. Ist die augenblickliche Ausgabeposition bereits größer als X, so wird meistens ab Position X der nächsten Zeile weiter ausgegeben (einige Interpreter ignorieren eine solche Tabulation auch). X darf ein arithmetischer Ausdruck sein. Ist der Wert von X negativ, wird TAB(1) ausgeführt.

■ **Beispiel:**

```
10  PRINT TAB(10);"ABC"
20  A%=5
30  PRINT "ZUM BEISPIEL";TAB(2*A%);"DEF"
```

Es erscheint auf dem Bildschirm:

```
          ABC
ZUM BEISPIEL
          DEF
```

Nach ZUM BEISPIEL ist die Ausgabeposition bereits 13, so daß DEF erst in der nächsten Zeile ab Position 10 ausgegeben wird.
Die meisten größeren BASIC-Interpreter verfügen darüber hinaus über eine Ausgabeanweisung, bei der ein bestimmtes Ausgabeformat vorgegeben werden kann. Diese Anweisung lautet oft

```
PRINT USING "Ausgabeformat"; Ausdruck {[,oder;][Ausdruck]}
```

Die Realisierung der PRINT-USING-Anweisung ist jedoch recht unterschiedlich, insbesondere die zugelassenen Ausgabeformate variieren von Interpreter zu Interpreter. Es werden deshalb nur 3 häufige Ausgabeformate in Tabelle 3.8 vorgestellt.
Der Gebrauch der Ausgabeformatzeichen wird durch folgendes Beispiel deutlich:

Tabelle 3.8. Ausgabeformatzeichen

Formatierungszeichen	Bedeutung
#	Jedes #-Zeichen symbolisiert eine Ziffer. Eine Zahl wird rechtsbündig mit so vielen Stellen ausgegeben, wie #-Zeichen angegeben sind. Ist die Stellenanzahl der Zahl kleiner als die Anzahl der #-Zeichen, so wird linksseitig mit Leerzeichen aufgefüllt.
.	Kennzeichnet die Stellung des Dezimalpunktes, ##.### bedeutet z. B. Ausgabe von Zahlen mit 2 Stellen vor und 3 Stellen nach dem Dezimalpunkt.
^^^^	Zahlenausgabe in Exponentialdarstellung
!	Ausgabe des ersten Zeichens von Zeichenketten

■ Beispiel:

```
10  A=2.573
20  B=358.2
30  C=-0.07926
40  D%=23
50  E¤="ANFANG"
60  PRINT USING "##.####";A,B,C,D%
70  PRINT USING "#.##^^^^";A,B,C,D%
80  PRINT USING "!";E¤
```

Es wird ausgegeben:

```
2.5730        %358.2        -0.0793        23.0000
2.57E+00       3.58E+02     -7.93E-02       2.30E+01
A
```

Ein %-Zeichen vor einer Zahl bedeutet, daß die Zahl für das angegebene Format zu groß ist und sie deshalb unformatiert ausgegeben wird. Um im obigen Beispiel in Zeile 60 B korrekt auszugeben, müßte man das Format demnach auf mindestens 3 Stellen vor dem Dezimalpunkt erweitern.

Weiter soll hier auf die formatierte Ausgabe nicht eingegangen werden, da sie zwar vielfältige Möglichkeiten zur Ausgabegestaltung bietet, aber nur bei größeren Interpretern realisiert ist und

noch dazu recht unterschiedlich. Einige BASIC-Versionen benutzen zur Angabe des Ausgabeformats auch eine besondere Anweisung (z. B. IMAGE oder FORMAT).

3.3.12. Weitere BASIC-Anweisungen

Zum Testen von BASIC-Programmen ist die Programmunterbrechungsanweisung vorteilhaft. Sie lautet

```
STOP
```

Sie kann an beliebiger Stelle und beliebig oft in ein Programm eingefügt werden. Erreicht der Computer bei der Programmausführung eine STOP-Anweisung, so wird die Programmabarbeitung unterbrochen, und der BASIC-Interpreter meldet die Programmunterbrechung mit der Ausschrift
STOP AT Zeilennummer
oder
BREAK AT Zeilennummer
Durch die ausgegebene Zeilennummer weiß man sofort, an welcher Stelle das Programm unterbrochen wurde, auch wenn das Programm mehrere STOP-Anweisungen enthält. An dieser Stelle kann man sich z. B. die Werte von Variablen zeigen lassen und überprüft, ob das Programm tatsächlich so arbeitet, wie man es beabsichtigt hatte. Danach kann die Programmausführung ab dieser Stelle durch ein Fortsetzungskommando wieder aufgenommen werden (siehe dazu Abschnitt 4.).
Einige BASIC-Interpreter verfügen außer der im Abschnitt 3.3.8. beschriebenen Laufanweisung noch über eine weitere Möglichkeit zur Bildung von Programmschleifen. Diese Anweisung ist besonders dann nützlich, wenn die Anzahl der Schleifendurchläufe vor der Programmabarbeitung nicht bekannt ist, sondern nur von einer Wiederholungsbedingung abhängt. Statt der FOR-Anweisung steht dann

```
WHILE logischer oder Vergleichsausdruck
```

und die NEXT-Anweisung ersetzt man durch

```
WEND
```

■ Beispiel:

```
10  A=1
20  WHILE A<>0
30  INPUT "A=",A
40  M=M+A
50  N=N+1
60  WEND
70  M=M/(N-1)
80  PRINT "M=";M
```

Die Schleife wird so lange durchlaufen, bis der Bediener eine Null eingibt.

▼ *Übung 15:*

1) Was wird durch dieses Programm berechnet?
2) Warum ist Zeile 10 notwendig?

Sollen außer dem Programm auch Daten nach Abschalten des Rechners erhalten bleiben, müssen diese ebenfalls vom Arbeitsspeicher des Rechners auf einen externen Speicher geschrieben werden, um sie bei Bedarf wieder von dort in den Arbeitsspeicher einlesen zu können. Dazu muß man eine Datei (engl.: file) anlegen. Auf Grund der unterschiedlichen externen Speicher kann die Dateiarbeit hier nur angedeutet werden. Ein Kassettenrecorder ermöglicht nur das Schreiben oder Lesen aller Daten. Dazu benutzt man die Anweisungen CSAVE und CLOAD. Wie diese Anweisungen für den speziellen Fall zu handhaben sind, muß den entsprechenden Unterlagen entnommen werden. Bei anderen externen Speichern lassen sich Dateien auch stückweise (satzweise) speichern und lesen. Jede Datei erhält einen Namen, der im Dateiinhaltsverzeichnis des Datenträgers (Digitalkassette, Diskette usw.) gespeichert wird. Der Dateiname wird durch eine

Dateieröffnungsanweisung vom Nutzer festgelegt, sie heißt z. B. OPEN oder FILE. Danach sind meist Dateinummer, Dateiname und Dateinutzungsart anzugeben. Bei der Dateinutzungsart gibt es die Möglichkeiten Schreiben, Lesen oder beides. Daten lassen sich in einer Datei durch die Dateiausgabeanweisung ablegen. Das Schlüsselwort dieser Anweisung lautet je nach Interpreter z. B. PRINT #, WRITE # oder PUT #, wobei danach meist nur die Dateinummer sowie Angaben darüber, an welche Stelle der Datei die Daten geschrieben werden sollen, folgen. Für die Umkehrung, das Lesen von Daten, gibt es die Dateieingabeanweisung mit Schlüsselworten wie z. B. INPUT #, READ # oder GET #.
Damit man beim Lesen einer Datei nicht versehentlich über das Ende der Datei hinausschießt, gibt es noch die logische Dateiendefunktion EOF (engl.: end of file), die die Dateinummer als Argument hat und den Wert WAHR liefert, falls das Dateiende erreicht ist.
Wird die Arbeit mit einer Datei beendet, führt man als letztes eine CLOSE-Anweisung aus. CLOSE ohne Angabe einer Dateinummer schließt alle offenen Dateien. Ist eine Datei geschlossen, kann die Dateinummer (nicht aber der Dateiname!) neu vergeben werden.
Durch Kombination der EOF-Funktion mit der CLOSE-Anweisung läßt sich das Beenden der Dateiarbeit organisieren, etwa durch 1000 IF EOF(2) THEN CLOSE # 2
Nachdem die wichtigsten BASIC-Anweisungen erläutert wurden, seien abschließend noch 2 übliche Methoden der Gliederung von BASIC-Programmen empfohlen. Diese Gliederung macht die Programme übersichtlicher und hat sich bewährt.
Die häufigste Variante ist:

1. DIM-Anweisungen
2. Funktionsdefinitionen
3. Hauptprogramm
4. END-Anweisung
5. Unterprogramme
6. DATA-Anweisungen

Da BASIC-Interpreter die Eigenschaft haben, Unterprogramme

schneller abzuarbeiten, wenn diese weiter vorn im Programm stehen, hat sich für zeitkritische Programme auch folgende Variante durchgesetzt:

1. DIM-Anweisungen
2. Funktionsdefinitionen
3. GOTO-Sprung zum Anfang des Hauptprogramms
4. Unterprogramme
5. Hauptprogramm
6. DATA-Anweisungen

4. Korrektur und Test eines BASIC-Programms

Wie ein Programm eingegeben wird, wurde bereits im Abschnitt 3.1. beschrieben. Für die folgenden Ausführungen sei vorausgesetzt, daß sich ein BASIC-Programm im Arbeitsspeicher des Rechners befindet. Wird nach Start des Programms ein syntaktischer Fehler gemeldet, so schaut man sich die betreffende Zeile durch Eingabe des Kommandos
LIST Zeilennummer (ENTER)
an und sucht den Fehler mit Hilfe des Fehlermeldungsverzeichnisses der Anwenderdokumentation. Der Fehler läßt sich auf verschiedene Art und Weise beseitigen. Man kann z. B. einfach die gesamte Programmzeile neu eintippen. Dabei wird die fehlerhafte Programmzeile automatisch gelöscht.
Oft ist aber nur ein Zeichen zu ändern. Dann wäre es recht aufwendig, die gesamte Zeile neu einzugeben. Dazu gibt es das Kommando
EDIT Zeilennummer (ENTER)
welches die fehlerhafte Zeile anzeigt und das Ändern, Löschen oder Einfügen einzelner Zeichen in dieser Zeile ermöglicht. Dafür werden meist Steuertasten benötigt. Dieses Kommando ist rechner- und interpreterabhängig. Nähere Hinweise kann man deshalb nur den entsprechenden Rechnerunterlagen entnehmen.
Mitunter kommt es auch vor, daß sich einige völlig falsche oder überflüssige Programmzeilen eingeschlichen haben, die ersatzlos gestrichen werden sollen. Eine einzelne Programmzeile wird am einfachsten durch die Eingabe
Zeilennummer (ENTER),
gelöscht. Für mehrere Zeilen oder ganze Programmabschnitte wäre dieses Verfahren aber zu aufwendig. Hier eignet sich das Löschkommando DELETE (oft auch abgekürzt DEL). Nach dem Kommandowort DELETE ist anzugeben, ab bzw. bis zu welcher Zeile das Programm gelöscht werden soll. Auch diese Angabe ist interpreterabhängig.

Manchmal möchte man eine weitere Programmzeile an eine Stelle des Programms einfügen, an der dies nicht mehr möglich ist, weil schon so viele Einfügungen gemacht wurden, daß die Zeilennummern bereits in Einerschritten numeriert sind. Hier hilft das Kommando RENUMBER (mitunter auch nur RENUM oder REN), welches das gesamte Programm wieder neu numeriert, und zwar mit dem Zeilenabstand 10. Dabei ändert es automatisch auch alle Sprungziele nach GOTO, GOSUB usw. Anschließend ist wieder Platz für die gewünschte Einfügung. Nach dem Kommandowort RENUMBER läßt sich meist auch eine beliebige Schrittweite angeben. Außerdem ist es möglich anzugeben, ab welcher alten Zeilennummer und beginnend mit welcher neuen Zeilennummer umnumeriert werden soll. Das Kommando RENUMBER sollte man jedoch nur benutzen, wenn es unumgänglich ist. Da sich beim Umnumerieren auch alle Sprungziele ändern, muß man sich anschließend ständig neu im Programm orientieren. Das Umnumerieren ist auch relativ zeitaufwendig.

Syntaxfehler sind mit Hilfe der Fehlermeldungen des Interpreters relativ leicht zu finden. Schwieriger ist es schon mit logischen Fehlern oder gewissen Flüchtigkeitsfehlern, die vom Rechner nicht bemerkt werden können. Hat man etwa versehentlich statt eines <-Zeichens ein >-Zeichen oder statt eines +-Zeichens ein *-Zeichen eingetippt, liefert der Rechner plötzlich andere Ergebnisse als erwartet. Wie kann man solche Fehler lokalisieren und ausschalten?

Sobald alle syntaktischen Fehler beseitigt sind, werden mit dem Programm einige Testrechnungen durchgeführt. Man bedient sich dazu einfacher Beispiele, die man leicht nachrechnen kann. Somit weiß man, ob die Ergebnisse, die das Programm liefert, richtig oder falsch sind. Ist ein Ergebnis falsch, werden nach «verdächtigen» Berechnungs- oder Sprunganweisungen STOP-Anweisungen (siehe Abschnitt 3.3.12.) eingefügt, um an diesen Stellen die Zwischenergebnisse zu kontrollieren bzw. um festzustellen, ob Sprünge richtig ausgeführt werden. Dazu ein simples Beispiel. Das folgende Programm soll die Fakultät n! einer natürlichen Zahl n berechnen.

```
10  INPUT "N=",N%
20  F%=1
30  IF N%=0 OR N%=1 THEN 70
40  FOR I%=2 TO N%
50  F%=F%*N%
60  NEXT I%
70  PRINT N%;"!=";F%
```

Dabei ist in Zeile 50 ein Fehler unterlaufen, den der Rechner nicht bemerken kann. Es wurde versehentlich N% statt I% geschrieben.
Man testet das Programm für 3 Beispiele: n=2, n=3 und n=4. Diese Fakultäten lassen sich schnell ausrechnen, denn $2!=1\cdot2=2$, $3!=1\cdot2\cdot3=6$ und $4!=1\cdot2\cdot3\cdot4=24$. Das Programm liefert nach dem 1. Start das Ergebnis 2!=2, es rechnet also scheinbar richtig! Beim 2. Lauf liefert es jedoch die Ausschrift 3!=9 und beim 3. schließlich 4!=64. Wie man sieht, kann aus einem erfolgreichen Test noch nicht auf ein fehlerfreies Programm geschlossen werden.
In diesem Miniprogramm ist die Fehlersuche recht einfach, der Fehler kann nur entweder in Zeile 40 (falsche Laufvariable) oder Zeile 50 (falsche Berechnungsvorschrift) stecken. Man fügt deshalb die Zeile
45 STOP
55 STOP
in das Programm ein. Jetzt wird das Programm erneut für n=3 gestartet und hält mit der Ausschrift
STOP AT 45
an. An dieser Stelle kann man kontrollieren, ob die Laufvariable I% mit dem richtigen Wert 2 belegt ist. Es wird ohne Zeilennummer (man sagt dazu auch «im Tischrechnermodus» oder «im Direktmodus») der Befehl
PRINT I% (ENTER)
eingetippt, und der Rechner gibt sofort den richtigen Wert 2 aus.

Hier steckt der Fehler also nicht, und die Programmabarbeitung wird durch das Fortsetzungskommando
CONTINUE (ENTER)
wieder aufgenommen (oft statt CONTINUE auch nur CONT oder CON). Das Programm hält nun mit
STOP AT 55
an. Jetzt gibt man den Tischrechnerbefehl
PRINT F% (ENTER)
ein und erhält den falschen Wert 3 vom Rechner angezeigt. Richtig müßte von der Berechnung $3!=1\cdot2\cdot3$ ja an dieser Stelle die 1. Multiplikation $1\cdot2$ ausgeführt werden und F% demnach den Wert 2 haben. Der Fehler ist gefunden, und Zeile 50 läßt sich korrigieren. Nach einer Programmänderung ist aber keine Fortsetzung mit dem CONTINUE-Kommando möglich, da das Programm neu übersetzt werden muß. Zum Neustart verwendet man RUN. Stimmen nach der Fehlerkorrektur auch die Zwischenergebnisse, können die STOP-Anweisungen wieder aus dem Programm entfernt werden. Nach Fehlerkorrekturen sind stets noch einige weitere Testrechnungen zu empfehlen, um sicher zu gehen, daß auch alle Fehler beseitigt wurden.
An diesem Beispiel sollten nur die prinzipiellen Möglichkeiten des Programmtests klar werden. In solch einfachen Fällen ist die geschilderte Verfahrensweise recht umständlich. Statt der STOP-Anweisungen lassen sich hier auch sofort PRINT-Anweisungen zur Ausgabe der Zwischenergebnisse einbauen, also
45 PRINT I%
55 PRINT F%.

Lösungen der Übungsaufgaben

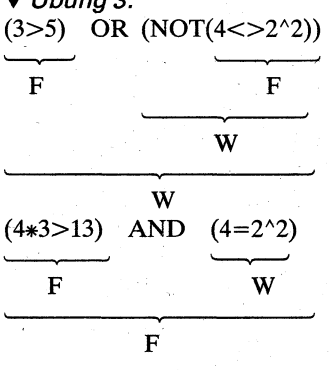
▼ *Übung 1:*

3+4>=5	W
(3−2∗4)^3>0	F
3^2=4+5	W

▼ *Übung 2:*

"MUELLER"<="MEIER"	F
"3.6.1986"<>"3.6.86"	W
"ABSTAND">"AB STAND"	W
"(ET1)">="[ET1]"	F

▼ *Übung 3:*



▼ *Übung 4:*

1) LN(5)
   EXP(3∗A)
   SIN(30∗0.017453)
   ABS(EXP(−X)−EXP(Y))

2) Runden der Zahl X auf eine ganze Zahl erfolgt durch INT(X+0.5).
3) LEN("") hat den Wert 0
LEN(" ") hat den Wert 1
VAL(STR¤(138)+chr¤(46)+"35 PFENNIGE") hat den Wert 138.35
ASC("%") hat den Wert 37
MID¤(LEFT¤("PAUSENZEICHEN",5),2,3) hat den Wert "AUS"

▼ *Übung 5:*

1) (A+B)*(A−B)
2) 3*(SIN(A)+5)
3) A^2−B^2/(N+1)
4) 1. B^3
   2. 2*A
   3. 2*A/B^3
5) 1. A+B
   2. (A+B)/2
   3. SIN((A+B)/2)
   4. 3*SIN((A+B)/2)
   5. 3*SIN((A+B)/2)/4
   6. 2−3*SIN((A+B)/2)/4
6) 1. D+2
   2. ABS(D+2)
   3. A*B
   4. A*B/C
   5. 2*ABS(D+2)
   6. A*B/C−2*ABS(D+2)
7) 1. SQR(A)
   2. SQR(A)+3
   3. 1/(SQR(A)+3)
   4. 1/(SQR(A)+3)−5
   5. ABS(1/(SQR(A)+3)−5)

▼ *Übung 6:*
1) W
2) W
3) F
4) W

▼ *Übung 7:*
1) 10 DIM A%(3,4)
2) 10 DIM B¤(8,12)

▼ *Übung 8:*
Die Zeilen 20, 50, 70 und 80 sind fehlerhaft.

▼ *Übung 9:*
Die Wertzuweisungen erfolgen in dieser Reihenfolge: A(1)=1, A(2)=1, A(3)=3, B=1, A(3)=1, B=2, A(1)=4, A(2)=5, A(3)=1, B=7. Am Ende enthält A die Werte 4 5 1, und B hat den Wert 7

▼ *Übung 10:*
Eine Möglichkeit ist:

```
142 IF A¤="NEIN" THEN 150
144 PRINT "BITTE NUR JA ODER NEIN ANTWORTEN!"
146 GOTO 120
```

▼ *Übung 11:*

```
130 INPUT "WEITERE ZYLINDER ? (0/1): ",A%
135 ON A%+1 GOTO 40,150
140 PRINT "BITTE NUR 0(=NEIN) ODER 1(=JA) EINGEBEN!"
145 GOTO 120
```

▼ *Übung 12:*

1) Zeile 220 ist äquivalent mit:
   220 FOR I=3 TO 5 STEP 0.5
   Die Laufvariable I nimmt demzufolge nacheinander die Werte 3, 3.5, 4, 4.5 und 5 an.

2)
```
10 PRINT "X","X^2","X^3"
20 FOR X%=1 TO 10
30 PRINT X%,X%^2,X%^3
40 NEXT X%
```

▼ *Übung 13:*

1)
```
 11  FOR I%=1 TO 3
 12  M%(I%)=0
 13  NEXT I%
 14  FOR I%=1 TO 9
 15  Z%(I%)=0
 16  NEXT I%
```

2)
```
710  FOR I=1 TO 24
```

3)
```
 10  P=3.141593
 20  INPUT "ANZAHL DER TEILZYLINDER: ",N%
 30  FOR I%=1 TO N%
 40  GOSUB 90
 50  G=G+V
 60  NEXT I%
 70  PRINT "GESAMTVOLUMEN DER ZYLINDERWELLE: ";G
 80  END
 90  REM ===== UP ZYLINDERVOLUMEN =====
100  PRINT "HOEHE ";I%;". TEILZYLINDER ";
```

```
110  INPUT "H=",H
120  PRINT "RADIUS ";I%;". TEILZYLINDER ";
130  INPUT "R=",R
140  V=P*R^2*H
150  RETURN
```

▼ *Übung 14:*

```
 10 DEF FNQ(A,B,C)=A*B*C
 20 INPUT "ANZAHL DER TEILQUADER: ",N%
 30 FOR I%=1 TO N%
 40 PRINT "LAENGE ";I%;". TEILQUADER ";
 50 INPUT "L=",L
 60 PRINT "BREITE ";I%;". TEILQUADER ";
 70 INPUT "B=",B
 80 PRINT "HOEHE ";I%;". TEILQUADER ";
 90 INPUT "H=",H
100 V=V+FNQ(L,B,H)
110 NEXT I%
120 PRINT "GESAMTVOLUMEN DES KOERPERS: ";V
```

▼ *Übung 15:*

1) Es wird der arithmetische Mittelwert M der eingegebenen Werte A berechnet.
2) Fehlte Zeile 10, so wäre die Bedingung A<>0 von Anfang an nicht erfüllt und die WHILE-Schleife würde gar nicht durchlaufen. Es würde dann M=0 ausgegeben.

ASCII-Code
(American Standard Code for Information Interchange)

Die Steuerzeichen werden bei Mikrorechnern oft anders genutzt, als es der ASCII-Code vorschlägt. Es sind deshalb nur die bei vielen Rechnern übereinstimmenden Funktionen erläutert.

ASCII-Code	Zeichen	Bedeutung	
0	NUL	Null	Steuerzeichen
1	SOH	Start Of Heading	
2	STX	Start Of Text	
3	ETX	End Of Text	
4	EOT	End Of Transmission	
5	ENQ	Enquiry	
6	ACK	Acknowledge	
7	BEL	Bell (akustisches Signal)	
8	BS	Backspace (letztes Zeichen löschen)	
9	HT	Horizontal Tab	
10	LF	Line Feed (Zeilenvorschub)	
11	VT	Vertical Tab	
12	FF	Form Feed (Seitenvorschub)	
13	CR	Carriage Return (Wagenrücklauf)	
14	SO	Shift On	
15	SI	Shift In	
16	DLE	Data Link Escape	
17	DC1	Device Control 1	
18	DC2	Device Control 2	
19	DC3	Device Control 3	
20	DC4	Device Control 4	
21	NAK	Negative Acknowledge	
22	SYN	Synchronous Idle	
23	ETB	End Transmission Block	
24	CAN	Cancel	
25	EM	End Of Medium	
26	SUB	Substitute	
27	ESC	Escape	
28	FS	File Separator	

ASCII-Code	Zeichen	Bedeutung	
29	GS	Group Separator	
30	RS	Record Separator	
31	US	Unit Separator	
32	SP	Space (Leerzeichen)	Sonderzeichen
33	!	Ausrufungszeichen	
34	"	Anführungszeichen	
35	#, £	Nummern-, Pfundzeichen	
36	¤, $	Währungs-, Dollarzeichen	
37	%	Prozentzeichen	
38	&	Et-Zeichen	
39	'	Apostroph	
40	(	runde Klammer auf	
41	)	runde Klammer zu	
42	*	Stern	
43	+	Pluszeichen	
44	,	Komma	
45	−	Minuszeichen	
46	.	Punkt	
47	/	Schrägstrich	
48	0		Ziffern
49	1		
50	2		
51	3		
52	4		
53	5		
54	6		
55	7		
56	8		
57	9		
58	:	Doppelpunkt	Sonderzeichen
59	;	Semikolon	
60	<	Kleiner-Zeichen	
61	=	Gleichheitszeichen	
62	>	Größer-Zeichen	

ASCII-Code	Zeichen	Bedeutung	
63	?	Fragezeichen	
64	@	At-Zeichen	
65	A		Großbuchstaben
66	B		
67	C		
68	D		
69	E		
70	F		
71	G		
72	H		
73	I		
74	J		
75	K		
76	L		
77	M		
78	N		
79	O		
80	P		
81	Q		
82	R		
83	S		
84	T		
85	U		
86	V		
87	W		
88	X		
89	Y		
90	Z		
91	[	eckige Klammer auf	Sonderzeichen
92	\	umgekehrter Schrägstrich	
93	]	eckige Klammer zu	
94	^, ↑, ¬	Zirkumflex, Pfeil nach oben, Negationszeichen	
95	_, ←	Unterstreichung, Pfeil nach links	
96	`	Akzent	

ASCII-Code	Zeichen	Bedeutung	
97	a		Kleinbuchstaben
98	b		
99	c		
100	d		
101	e		
102	f		
103	g		
104	h		
105	i		
106	j		
107	k		
108	l		
109	m		
110	n		
111	o		
112	p		
113	q		
114	r		
115	s		
116	t		
117	u		
118	v		
119	w		
120	x		
121	y		
122	z		
123	{	geschweifte Klammer auf	Sonderzeichen
124	|	senkrechter Strich	
125	}	geschweifte Klammer zu	
126	~	Überstreichung, Tilde	
127	DEL	Delete	Steuerzeichen

Literaturverzeichnis

[1] *S. Müller*, Programmieren mit BASIC. Reihe Automatisierungstechnik Bd. 216, Berlin 1985

[2] *H. Gutzer*, Das kann der Mikrocomputer. Leipzig/Jena/Berlin 1985

[3] *S. Stief*, BASIC – Systematische Darstellung für den Anwender. Reihe Datenverarbeitung, München/Wien 1980

[4] *W. Schneider*, Strukturiertes Programmieren in Basic. Reihe Programmieren von Mikrocomputern Bd. 13, Braunschweig/Wiesbaden 1985

[5] Anwenderdokumentation BASIC-Interpreter für SIOS 1526. Karl-Marx-Stadt 1982

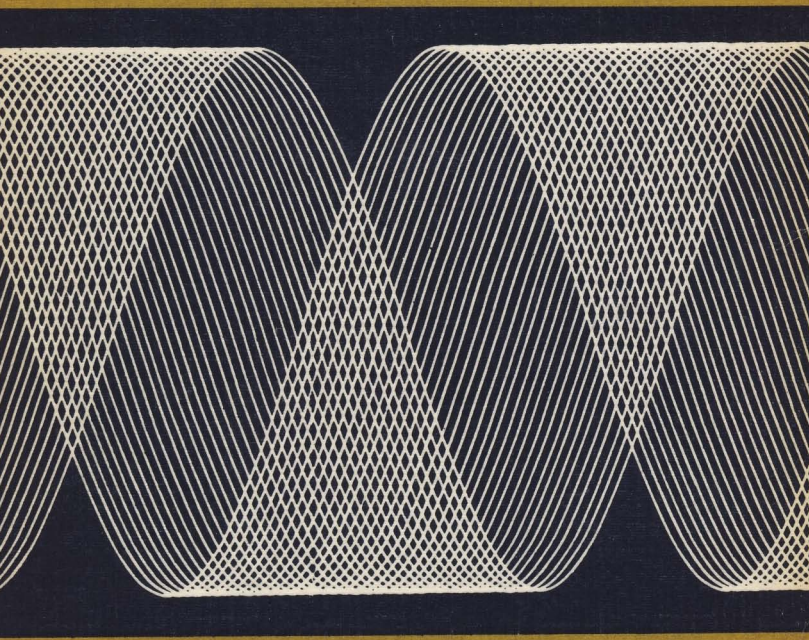
[6] Anwenderdokumentation BASIC-Interpreter für SCPX 1526. Karl-Marx-Stadt 1984

[7] *G. Keller*, Basic für Heimcomputer Z9001. In: radio fernsehen elektronik Heft 9/1984, S. 586–593

[8] *R. Heyne*, Basic-Compreter für U880. In: radio fernsehen elektronik Heft 3/1984, S. 150–152

[9] *K. L. Boon*, BASIC für Tischcomputer. München 1984

[10] *G. Merkel*, Internationaler Stand und Trend der CAD/CAM-Systeme sowie Überblick über Basissysteme des VEB Kombinat Robotron. In: 31. Internationales Wissenschaftliches Kolloquium 1986 / TH Ilmenau, Heft 4, S. 205–208



ISBN 3-327-00348-3