CHIP
WISSEN

Software

Bernd Pol

Wie man in BASIC programmiert

Einführung · Techniken · Fallstudien

Ein Buch von CHIP, der Zeitschrift
für Mikrocomputer-Technik

Bernd Pol

Wie man in BASIC programmiert

CHIP WISSEN

Software

Bernd Pol

Wie man in BASIC programmiert

Einführung · Techniken · Fallstudien

2. Auflage



VOGEL-BUCHVERLAG
WÜRZBURG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Pol, Bernd:
Wie man in BASIC programmiert: Einf., Techniken, Fallstudien/Bernd Pol. — 2. Aufl. — Würzburg: Vogel, 1983.
(Chip-Wissen: Software)
ISBN 3-8023-0637-6

ISBN 3-8023-0637-6
2. Auflage. 1983
Alle Rechte, auch der Übersetzung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.
Printed in Germany
Copyright 1981 by Vogel-Buchverlag Würzburg
Herstellung: Vogel-Druck Würzburg

Inhaltsverzeichnis

Vorwort 9

Einführung 13

1 Ein erster Versuch 19
Der Anfang: Fertigzeichen, PRINT, CR 20, Grundstruktur von Befehlen in BASIC 22, Numerische Ausdrücke 23, Zahlen 24, Die Form von numerischen Ausdrücken: Priorität 29, Variable und LET 30, Kommandos und Anweisungen, NEW und RUN 36, Edieren des Programmtextes: LIST 40, Vorgabe von Daten: DATA und READ 45, Fehlermeldungen 50, Sprünge im Programm: GOTO 53, Entscheidungen treffen: IF...THEN 58, Etwas zum Druckbild: PRINT-Argumente 61, Noch mehr zum Druckbild: PRINT «Text» 68, Einige Vereinfachungen 72, Speichern des Programms: (C)SAVE, (C)LOAD 79

2 Systematisierung 83
Flußdiagramme 83, Die Strukturbeschreibung 92, Das Struktogramm 94, Darstellung von Schleifenstrukturen 95, Etwas zur Form von Strukturbeschreibungen 97, Programmiertechniken 100

3 Eine Fallstudie 105

3.1 Aufgabenstellung 106

3.2 Vorüberlegungen 106
Eingaben 107, Ausgaben 108, Grundaufgaben des Programms 108, Überprüfen des Ansatzes 109, Programmdokumentation 112, Die benötigten Daten 113, Darstellung der Daten 115

3.3 Eingabe über die Tastatur 118

3.4 Endlose Schleifen, Tips zur Fehlersuche: CONTROL-C, CONT und STOP 123

3.5 Anforderungsmeldungen 129
3.6 Alphanumerische Variable 132
Zeichenketten (Strings) 134
3.7 Eine Übersichtsplanung 137
Die Menütechnik 139
3.8 Unterprogramme: GOSUB und RETURN 143
3.9 Dokumentieren: REM 152
Nachteile des REM-Befehls 159
3.10 Etwas zum Umgang mit Unterprogrammen: Fehlermöglichkeiten, TRACE 161
3.11 Überlegungen zur Datenstruktur 168
Felder, DIM X(i), FRE(0) 171
3.12 Mehr über Zeichenketten 186
Zeichenkettenfelder 186, «Microsoft»- und «Northstar»-Versionen 187, Zeichenkettenfelder in «Northstar»-Versionen 188, Sicherung der Eingabedaten; VAL(X$) 194
3.13 Das Datumsproblem: LEFT$(X$), MID$(X$), RIGHT$(X$) 198
Vereinfachung der Datumsermittlung: INSTR, INDEX und Konkatenation 205
3.14 Steueranweisungen für Fortgeschrittene 210
FOR-NEXT-Schleifen 210, IF-THEN-ELSE-Strukturen 216, Mehrfachentscheidungen: ON...GOTO, ON...GOSUB 220
3.15 Planung der Teilprogramme 223
Beschreibungshilfsmittel: Die Programmhierarchie 226, Speicherplatzaufteilung: FRE, INT, DIM und CLEAR 229, Artikel neu aufnehmen: Zum Vorgehen bei der Programmerstellung 239, Artikeleintrag ändern 246, Eintrag in der Datei aufsuchen: Etwas über Parameter 250, Bestand verändern: Etwas zur Dialoggestaltung 253, Lagerübersicht zur Nachbestellübersicht: AND, OR, LPRINT 257
3.16 Probleme der Datenspeicherung 267
Numerische Felder abspeichern und laden: CSAVE* und CLOAD* 268, Zeichen, Zeichensatz und Zeichenkodes: ASC(X$), CHR$(X), INPUT-Feinheiten 271, Umwandlung von Zeichenketten in numerische Felder 280, Vom Umgang mit Diskettendateien 292
3.17 Hilfsprogramme: Parameterauswahl, Beziehungen zwischen Groß- und Kleinbuchstaben im ASCII-Kode 303

3.18 Umsetzung in BASIC 311
Übersicht 311, Umformung in Großschreibung: Zur Notwendigkeit der Programmoptimierung 315, Programmoptimierung, Flußdiagramm und Strukturbeschreibung 319, Programmtest 323, Bestätigung einholen 327

3.19 Testen für Fortgeschrittene 331

Ausblick 342

Anhang 1: Strukturbeschreibung der Dateimoduln 343

Anhang 2: Im Lagerdateiprogramm verwendete Variable 349

Anhang 3: Befehle, Anweisungen und Funktionen in BASIC 350
1. Kommandos 350
2. Anweisungen 351
3. Ein-/Ausgabeanweisungen 355
4. Numerische Funktionen 357
5. Zeichenkettenfunktionen 358
6. Sonstige Funktionen 359

Anhang 4: Der ASCII-Kode 360

Anhang 5: Tips, Tricks und sonstige Hinweise 365
(Ein Stichwortverzeichnis)

Vorwort

Seit einiger Zeit schon sind Mikrocomputer im größeren Maßstab verfügbar. Nicht nur in den verschiedensten Geräten versteckt findet man sie, sondern auch unmittelbar als Heim- oder Hobbycomputersysteme, wo sie zur Lösung kommerzieller oder technischer Aufgaben, für Forschungszwecke oder einfach nur zum kreativen Spielen dienen können. Ein ausgereifter Mikrocomputer ist ein Universalwerkzeug, das die Schwelle zur Zukunft bezeichnet. Und wie jedes Werkzeug muß er beherrscht werden, um seine vielfältigen Möglichkeiten ausschöpfen, aber auch um die aus seiner Anwendung hervorgehenden möglichen Gefahren erkennen zu können.

Nun ist Programmieren durchaus keine Hexerei. Und da die meisten Mikrocomputer mit BASIC eine einfache Programmiersprache anbieten, ist es auch ohne große Schwierigkeiten zu erlernen.

Dazu lassen sich zwei Wege einschlagen: Der erste – zunächst naheliegendere – ist, daß man sich eine möglichst umfangreiche Beschreibung der Sprache vornimmt und versucht, das Beste daraus zu machen.

Jedoch – es ist immer besser, unbekannte Möglichkeiten zunächst von innen heraus zu erproben, ein Gefühl für die Sache zu bekommen – selbst wenn man dabei nicht in die allerletzten Feinheiten der Materie eindringt. Kann man sich erst einmal sicher bewegen, ist es wesentlich einfacher (und lustvoller), den ganzen Boden auszuloten.

Nehmen wir uns darum gar nicht erst vor, den gesamten

Sprachschatz von BASIC kennenzulernen. Machen wir etwas ganz anderes: Versuchen wir anhand von ein paar einfachen Problemstellungen herauszubekommen, wie man programmiert. Denn darum soll es uns gehen: das Programmieren zu erlernen, und zwar mit Hilfe von BASIC. Wir werden so etwas wie eine kleine Reise unternehmen, auf der wir alle wichtigen Ansätze und Methoden erforschen, die zum Programmiervorgang als solchem gehören. Und damit das nicht zu abstrakt gerät, werden wir nach und nach die wichtigsten Bestandteile von BASIC untersuchen. Wir werden mit der Sprache experimentieren müssen, damit nicht nur klar wird, was es da alles gibt, sondern vor allem *wie* das angewendet wird und nicht zuletzt, *warum* es so ist, wie es uns vorliegt.

Wir werden auch untersuchen, was zu einem guten und modernen Programmierstil gehört. Wir werden nachweisen, daß es in BASIC nicht nur möglich, sondern im Grunde notwendig ist, *strukturiert* zu programmieren. Und wir werden zeigen, daß strukturiertes Programmieren nicht an die eine oder andere starre Regel gebunden ist, sondern das schöpferische und disziplinierte Anwenden einer Vielzahl von Werkzeugen zum Programmieren umfaßt. Die Regeln dazu werden wir natürlich kennenlernen, der Schwerpunkt der Darstellung wird aber darauf liegen, wie man sie anwendet.

Das Buch umfaßt im wesentlichen drei Kapitel von recht unterschiedlicher Länge und einen Anhang. Im ersten Kapitel werden wir uns an die Grundlagen des Programmierens mit BASIC herantasten. Im zweiten werden die wichtigsten Programmierhilfsmittel besprochen. Das dritte Kapitel schließlich macht den Löwenanteil aus: Hier sollen als Fallstudie ein (sehr) einfaches Lagerverwaltungsprogramm erstellt und daran die bis dahin erworbenen Erkenntnisse vertieft und erweitert werden.

Es liegt damit keine methodisch straff gegliederte Arbeit vor, die höchsten wissenschaftlichen Ansprüchen genügen mag. Wir werden uns von Erkenntnisschritt zu Erkenntnisschritt voranarbeiten. Und das soll nicht trocken geschehen, sondern wie gesagt:

Bereiten Sie sich auf eine kleine Entdeckungsreise durch das Reich des Programmierens — Unterabteilung BASIC — vor.

Prinzipiell sind keine besonderen Vorkenntnisse nötig, doch empfiehlt es sich, das Buch nicht nur einfach im «Trockenkurs» durchzuarbeiten (was allerdings durchaus möglich ist). Sehr viel sinnvoller wäre es, wenn Sie einen mit BASIC ausgerüsteten Computer zum Nachvollziehen der Beispiele und Überlegungen zur Hand hätten. Denn auch bei BASIC gilt der allgemeine Satz, daß man etwas nur dann wirklich lernt, wenn man schöpferisch und selbst aktiv an es herangehen kann. So läßt sich das Anliegen des Buchs auf einen kurzen Nenner bringen:

Lernen Sie Programmieren durch Programmieren!

Vorwort zur zweiten Auflage

Nicht zuletzt des großen Erfolgs der ersten Auflage wegen ist die zweite Auflage im wesentlichen unverändert geblieben. Lediglich einige Druckfehler wurden korrigiert. Das Konzept hat sich im großen und ganzen bewährt. Dennoch seien ein paar zusätzliche klärende Worte angefügt.

Zwei Punkte unterscheiden diese von anderen BASIC-Einführungen: Es wird nur wenig Gewicht auf Mathematik gelegt, und statt der sonst üblichen Fülle relativ unzusammenhängender Einzelbeispiele aus den verschiedensten Bereichen finden sich hier zwei umfangreiche Fallstudien, die dafür bis ins Detail untersucht werden. Dies hat gewichtige Gründe, insbesondere:

Es sollten nur wenige über das Allgemeinwissen hinausgehende Vorkenntnisse notwendig sein. Damit verbat sich ein ausführliches Eingehen auf mathematische Anwendungen. Allerdings ist dies nicht unbedingt als Nachteil zu werten, da die Übertragung üblicher mathematisch (rechnerisch) zu lösender Aufgaben in BASIC-Programme nicht sehr schwierig ist, hat man erst einmal die Grundzüge der Arbeit mit BASIC verstanden.

Zweitens und wichtiger ist, daß die meisten derzeit erhältlichen BASIC-Einführungen zwar durch die Fülle von Einzelbeispielen ein relativ breites Feld an Anwendungsbereichen abdecken, aber zu dem eigentlichen Problem kaum etwas aussagen, nämlich wie man von der ersten Idee bis zum abschließenden Funktionstest ein Programm erstellt. Die (notwendigerweise) relativ einfachen Einzelbeispiele gaukeln eine «heile Welt des Programmierens» vor und verführen gerade bei BASIC zu einem planlosen, letztlich chaotischen Drauflosarbeiten. Mag dies bei kleineren Projekten noch angehen, so wird der Aufwand bei umfangreicheren Arbeiten im Vergleich zum Ergebnis viel zu groß — und das Resultat ist nur zu oft wenig zuverlässig und kaum wartbar.

Dem soll hier durch die Konzentration auf ein der Praxis nachempfundenes Fallbeispiel vorgebeugt werden. Dabei wurde bewußt in Kauf genommen, daß das gewählte Beispiel nicht jeden interessiert. Es kam nicht darauf an, (halb)fertige Rezepte zu liefern, sondern ein Gefühl für den Programmierprozeß selbst zu schaffen, wichtige moderne Techniken bei der Programmerstellung vorzustellen und nicht zuletzt durch die intensive Detaildiskussion ein Vorstellungsgerüst aufzubauen, wie ein BASIC-Programm funktioniert.

Der Programmiervorgang selbst soll hier nähergebracht werden. Das bedeutet nicht, daß Einzelbeispiele sinnlos sind, doch sollten diese besser in jeweils fachorientierten «Nachschlagewerken» zusammengefaßt werden und so die eigentliche Programmiereinführung ergänzen. Aber auch das kann das Wichtigste, die eigene Erfahrung, nicht ersetzen. Programmieren lernt man nur durch die eigene Praxis, nur durch das eigene Auseinandersetzen mit der Fülle von Detailproblemen, die dabei auftreten. Und dazu soll die vorliegende Einführung eine erste Handreichung bieten.

Stuttgart *Bernd Pol*

Einführung

Von sich selbst aus macht ein Computer bekanntlich nichts – zumindest nichts sonderlich Sinnvolles. Will man etwas von ihm haben, muß man ihm erst einmal beibringen, was er zu tun hat. Er muß vor allem lernen, *Befehle* auszuführen:

Tu dieses – tu jenes – und wenn du fertig bist, sieh nach, was dort passiert ...

Leider versteht ein Computer, Maschine, die er ist, so etwas zunächst einmal überhaupt nicht. Die geläufige menschliche Sprache ist ihm zu kompliziert und zu umfangreich. Man muß sie erst einmal reduzieren, einschränken auf einige wenige, genau umrissene Befehle und auf Regeln, wie man mit diesen Befehlen umzugehen hat. Mit anderen Worten: Man muß eine *Sprache* schaffen, die der Computer versteht, und mit der man mit ihm Kontakt aufnehmen kann.

Nun ist ein Computer nicht ganz so unzugänglich, wie es erst den Anschein hat. Er versteht schon so etwas wie eine Sprache (so ist er gebaut) – aber diese «Sprache» hat mit unserer Umgangssprache so gut wie gar nichts zu tun. Als elektronisches Gerät versteht er im Grunde nur «Strom fließt – Strom fließt nicht» in allen zeitlichen Abfolgen, die dabei möglich sind. Wir wollen hier nicht darauf eingehen, wie das im einzelnen aussieht. Uns interessiert mehr die Tatsache, daß man eine solche «elektrische Sprache» so weit aufmöbeln kann, daß sie einfachen Befehlen aus der Umgangssprache zugänglich wird. So etwas geschieht durch besondere Übersetzungseinrichtungen, die diese dem Menschen

verständlichen Befehle in eben solche «Strom fließt – Strom fließt nicht»-Kombinationen übersetzt.

Es gibt dabei verschiedene Möglichkeiten, die sich darauf auswirken, wie diese dem Menschen verständliche «Computersprache» aussieht. Es gibt Sprachen, die jeweils nur von dem Computertyp verstanden werden können, für den sie entworfen sind. Wortschatz und grammatikalische Regeln sind hier noch der «Maschinensprache» stark angepaßt, und es gehört in der Regel eine genaue Kenntnis des Maschinentyps dazu, um sie sinnvoll anwenden zu können. Man nennt derartige Sprachen daher «maschinenorientiert». Sie stehen aber auch als Mittler zwischen der «elektrischen» Sprache der Maschine selbst und einer höheren Stufe von für den Verkehr mit dem Computer entworfenen Sprachen. Diese anderen können – geeignete Übersetzungseinrichtungen vorausgesetzt – im Prinzip von allen Computern verstanden werden, unterscheiden sich dabei aber ziemlich stark je nach angesprochenem Problemkreis.

Das ist verständlich. Wenn man die Umgangssprache schon einschränken muß, verwendet man sinnvollerweise nur solche Befehle und Regeln, die dem zu lösenden Problem angemessen sind. So hat der Mathematiker mit ALGOL eine ganz andere Computersprache zur Verfügung als der Geschäftsmann mit COBOL (ALGOL und COBOL sind nichts weiter als Namen für ganz bestimmte Computersprachen). Derartige zur sinnvollen Bearbeitung bestimmter Probleme entworfene Computersprachen bezeichnet man als «problemorientierte Sprachen».

Trotz aller Gebundenheit an die menschliche Sprache bleibt jede dieser Computersprachen zunächst einmal eine Fremdsprache, und wie alle Fremdsprachen hat sie den Nachteil, daß sie erst gelernt werden muß, bevor man mit ihr etwas anfangen kann.

Nun beruhen die meisten der gebräuchlichen derartigen Sprachen auf verwandten Befehlen und Regeln. Hat man diese erst einmal begriffen, so ist der Übergang von einer Computersprache zur anderen in vielen Fällen nicht sehr viel schwieriger als das Erlernen eines neuen Dialekts.

Es lag daher nahe, eine Art Anfängersprache zu schaffen, die alle wesentlichen Befehle und Regeln der üblichen Computersprachen enthält, einfach zu lernen ist und auf einen möglichst großen Kreis von Problemen anwendbar sein sollte. Ausgehend von der verbreiteten Computersprache FORTRAN, hat man so (bereits 1963) die Programmiersprache BASIC geschaffen. Dieser Name ist, wie die meisten, eine Abkürzung aus dem Englischen und heißt voll ausgeschrieben: Beginners All Purpose Symbolic Instruction Code (zu deutsch etwa: Symbolischer Allzweckkode für Anfänger). Diese Sprache gibt es in den verschiedensten Dialekten, für Großcomputer, für Minicomputer und schließlich auch für Mikrocomputer. Mit der Einführung der letzteren hat BASIC eine ungeheuer weite Verbreitung erfahren, in ein bis zwei ganz typischen Ausgestaltungen, denen wir uns im folgenden widmen wollen.

Zum Erlernen einer Computersprache gehört allerdings noch mehr, als sich nur ein Vokabular und Regeln zum Umgang damit anzueignen. Da es sich um ein *Werkzeug* zum Lösen bestimmter Probleme handelt, muß man auch lernen, wie man diese Probleme so aufbereitet, daß sie einer Lösung mit Hilfe der zu verwendeten Sprache zugänglich sind. Dieses Aufbereiten des Problems ist das eigentliche *Programmieren*. Ohne die Methoden zu kennen, mit denen man eine bestimmte Fragestellung angehen kann, läßt sich auch mit der besten *Programmiersprache*, wie man eine Computersprache auch nennt, nicht viel anfangen. Wir werden daher einen großen Teil des folgenden Textes darauf verwenden, zu lernen, wie man ein Problem aufbereitet und wie man es in eine Programmiersprache – eben in BASIC – umsetzt.

Um uns auf den eigentlichen Zweck der Arbeit konzentrieren zu können, sollen nur solche Aufgaben angegangen werden, die man nicht selbst erst wieder umständlich erklären muß. Das heißt nicht, daß BASIC nur zum Lösen relativ trivialer Aufgabestellungen geeignet ist. Ganz im Gegenteil hat die Praxis bewiesen, daß man BASIC noch in hochkomplexen Bereichen einsetzen kann. Diese Bereiche selbst sollen aber nicht Aufgabe des Buchs

hier sein. Wir wollen nur ein bißchen Routine erwerben, um später derartige Aufgaben gezielt und selbständig angehen zu können. Daß es dazu bei diesem Buch hier nicht bleiben kann, ist eine Binsenweisheit. Sie werden sich im Lauf der Zeit durch eine ganze Bibliothek durcharbeiten müssen. Was BASIC anbetrifft, sollten Sie diese weiterführende Literatur allerdings nach Durcharbeiten des vorliegenden Kurses auch verstehen können.

Eine letzte Bemerkung noch: Die verschiedenen Sprachen unterscheiden sich nicht nur nach ihrem Wortschatz und ihrer Struktur voneinander, sondern – oft auch bei ein und derselben Sprache – durch die Art, in der die Übersetzungseinrichtung arbeitet, mit der der Computer erst die Sprache versteht. Da gibt es zum einen die Form, die ein fix und fertig ausgearbeitetes Programm übernimmt, am Stück übersetzt und erst dann die Möglichkeit gibt, das Ganze abarbeiten zu lassen. (Diese Form wird als «Compiler» bezeichnet und der Prozeß der Übersetzung als «compilieren».) Daneben gibt es eine andere Möglichkeit, in der der Computer den Text, wie er vorliegt, untersucht und Schritt für Schritt abarbeitet, sobald etwas vorliegt – und sei es auch nur ein kleines Bruchstück eines Programms. Eine derartige «Interpretation» des Programms hat einige Nachteile, die die Effektivität (besonders die Abarbeitungsgeschwindigkeit) des fertigen Programms beeinträchtigen. Sie hat aber den für unsere Zwecke ungemein wichtigen Vorteil, daß man mit dem Programm *unmittelbar* experimentieren kann: Man setzt sich an die Rechnertastatur, tippt etwas ein und sieht sofort das Ergebnis. Auf diese Weise läßt sich *interaktiv* mit dem Rechner arbeiten, was für das Erlernen einer Sprache beträchtliche Vorteile bietet.

Die meisten BASIC-Versionen sind als derartige *Interpreter* konstruiert, bei denen man unmittelbar mit dem Rechner «sprechen» kann. Es gibt jedoch einige wenige Versionen, die *nicht* interpretierend, sondern compilierend arbeiten und so kein unmittelbares Experimentieren am Computer gestatten. Wir werden uns im ganzen folgenden Text auf eine interpretierende Version stützen — sollten Sie nur einen BASIC-Compiler besit-

zen, so werden Sie besonders zu Beginn viele Experimente nur im «Trockenkurs» oder aber mit einigen Schwierigkeiten verknüpft durchführen können. Am besten besorgen Sie sich irgendwie Zugriff auf eine interpretierende Version zum Üben. Mit Ihrer – sicherlich besseren – compilierenden Version werden Sie danach sehr viel besser zu Rande kommen können.

Doch fangen wir erst einmal an ...

1
Ein erster Versuch

Versuchen wir zum Einstieg etwas Einfaches: Nehmen wir an, Sie wollten sich eine Übersicht über den Benzinverbrauch Ihres Wagens verschaffen. Dazu haben Sie bei jedem Tanken den Stand des Kilometerzählers und die neu getankte Literzahl notiert (z.B. wie in Tabelle 1). Aus diesen Werten soll nun der Verbrauch je 100 km berechnet werden, und zwar für jedes Tanken einmal und dann noch der durchschnittliche Verbrauch. (Der Einfachheit halber nehmen wir an, der Tank wäre jedesmal leergefahren worden.)

Der jeweilige Verbrauch ergibt sich dann aus der zwischen zwei Tankintervallen gefahrenen Strecke und der jeweils getankten Benzinmenge (die wir im folgenden kurz mit B für «Benzin» bezeichnen wollen). Die zurückgelegte Strecke, die wir S nennen, ist gerade der Unterschied zwischen dem Stand des Kilometerzählers beim Tanken (diesen Stand nennen wir S1) und dem Stand, wenn diese Benzinmenge verfahren ist (was S2 sein soll).

In unserem Beispiel von Tabelle 1 würde sich der erste solche Verbrauchswert wie folgt ergeben. S1 erhält den Wert 41 956 km, d.h. den Stand des Kilometerzählers, bei dem B = 10 l Benzin getankt worden sind. Diese zehn Liter sind beim Kilometerzählerstand S2 = 42 090 km verfahren. Das heißt, sie haben gerade eine Strecke von S2 − S1 = 42 090 km − 41 956 km = 134 km gereicht. Je Kilometer gibt das einen Verbrauch von 10 l : 134 km = 0,0746 l/km. Und weil das ein bißchen unanschaulich klein ist,

Kilometer-stand	getankt (Liter)
41956	10
42090	15
42243	10
42340	25
42618	20
42845	20
43032	15
43155	10
43264	20
43465	15

Tabelle 1 Das Tankprotokoll

rechnen wir es auf 100 km um, nämlich 0,0746 l/km × 100 = 7,46 l/100 km oder ausgeschrieben: 7,46 Liter je 100 km.

Bezeichnen wir diesen Verbrauch mit V, so ergibt sich, daß wir folgende Formel verwendet haben:

$$V = \frac{B}{S2 - S1} \times 100 \qquad (1)$$

Diese Formel müssen wir jetzt dem Computer beibringen.

Der Anfang: Fertigzeichen, PRINT, CR

Gehen wir schrittweise vor: Zuerst einmal soll der Rechner die Differenz S2 − S1 ermitteln und das Ergebnis anzeigen, also z.B. die Frage beantworten, welche Strecke zurückgelegt wurde, wenn der Kilometerzähler beim ersten Tanken den Stand 41 956 und beim zweiten den Stand 42 090 gehabt hat.

Damit stoßen wir bereits auf ein Problem. Wir müssen dem Rechner die Aufgabe irgendwie verständlich machen, ihm also unseren Wunsch passend (über die Tastatur) *eingeben*. Nur – das können wir nicht zu jedem beliebigen Zeitpunkt tun. Es kommt in der Praxis oft zu sehr verwickelten Aufgaben, an denen einige Sekunden bis hin zu mehreren Stunden gearbeitet werden kann.

In der Regel ist der Rechner dann nicht in der Lage, irgendwelche Befehle entgegenzunehmen. Der Computer muß also von sich aus melden, wenn er zur Befehlsübernahme fertig ist. Dazu druckt er ein besonderes *Fertigzeichen* (englisch: prompt sign) aus, das besagt: «Jetzt bin ich mit Rechnen fertig und kann Befehle entgegennehmen.» Zumeist verwendet man hierzu einen Größer-als-Pfeil « > »; es sind aber auch andere Zeichen gebräuchlich (im Zweifelsfall müssen Sie das Handbuch der von Ihnen verwendeten BASIC-Version zu Rate ziehen).

Bleibt die Frage, was eigentlich befohlen werden muß. Dazu soll uns (wie bei allen folgenden Problemen) etwas Überlegung weiterhelfen:

BASIC ist eine einfache, auf den Dialog mit dem Benutzer orientierte Sprache, die auf dem englischen Wortschatz beruht. Die meisten Befehle in BASIC lassen sich unmittelbar aus der zu verrichtenden Tätigkeit herleiten. (Das gilt nicht unbedingt für alle höheren Programmiersprachen.)

Was ist also zu tun? Der Rechner soll den Wert einer Differenz berechnen und anschließend ausdrucken. Unser Beispiel läßt sich damit in einem Befehl zusammenfassen:

DRUCKE 42090 – 41956

oder englisch:

PRINT 42090 – 41956

Das sollten wir genauer betrachten.

In der Regel meldet sich der Computer nach dem Einschalten (oder Laden) von BASIC und nach jedem Ende eines Programmlaufs mit «READY», «OK» oder ähnlichem zurück. Nehmen wir das erstere an, so erhalten wir am Anfang die Meldung:

```
READY
>
```

und können nach dem Pfeil den Befehl eingeben:

```
READY
>PRINT 42090 - 41956
```

Nun müssen wir dem Rechner noch klar machen, daß das der ganze Befehl war und jetzt er an der Reihe ist. Das heißt, wir müssen den Befehl geeignet *abschließen*. Naheliegend (und allgemein üblich) ist es, zu diesem Zweck die Zeile zu beenden, was seinerseits am natürlichsten durch einen Wagenrücklauf erfolgt. Dieser hat im Englischen die Bezeichnung «carriage return» (oder kurz RETURN) und wird oft zu CR abgekürzt; wir werden ihn hier im Bedarfsfall durch «<CR>» symbolisieren.)

Also:

```
READY
>PRINT 42090 - 41956<CR>
 134
```

Grundstruktur von Befehlen in BASIC

Es lohnt sich, hier noch etwas zu verweilen. Was mag das bedeuten, wenn wir befehlen:

PRINT 42090 – 41956

Nun, wir haben da zunächst einmal ein *Befehlswort:* PRINT. Das allein gibt allerdings noch nicht viel her, wir müssen noch angeben, was zu drucken ist, eben «42090 – 41956». Man bezeichnet so etwas als *Befehlsargument*. Das ergibt die allgemeine Grundstruktur von BASIC-Befehlen:

BEFEHLSWORT Befehlsargument

=

Befehl

Diese Art, den Befehl anzugeben, ist typisch für Sprachen der Art von BASIC. (Es sei bemerkt, daß das nicht bei allen Programmiersprachen so aussehen muß.)

Numerische Ausdrücke

Was aber, um die Sache genau zu betrachten, haben wir oben als Argument des PRINT-Befehls angegeben? Nun, offensichtlich doch den Teil einer Formel, in dem die Operation «42090 minus 41956» ausgedrückt wird, einen *Ausdruck* also, der eine Zahl, d.h. ein numerisches Ergebnis, liefert. Und erst das vom Computer berechnete Ergebnis sollte durch PRINT-Befehl ausgegeben werden.

Das ist nicht ganz so trivial, wie es aussieht. Wir brauchen in BASIC den Wert eines Ausdrucks nicht erst gesondert zu berechnen, bevor wir ihn als Befehlsargument verwenden können. Es genügt, wenn wir in dem Argument den betreffenden Ausdruck angeben.

«42090 – 41956» ist ein Ausdruck, der ein numerisches Ergebnis, eben «134» liefert. Man bezeichnet so etwas daher als *numerischen Ausdruck* (numerical expression im Englischen). Wir werden unten noch andere, nichtnumerische Ausdrücke kennenlernen, daher ist die Unterscheidung wichtig. In einem numerischen Ausdruck kann man in der Regel *reelle Zahlen* miteinander verknüpfen. (Für Nichtmathematiker: Reelle Zahlen sind im Prinzip alle diejenigen, die sich als Dezimalzahl schreiben lassen, und unter «Verknüpfen» versteht man einfach die Tatsache, daß man mit ihnen eine mathematische *Operation* – also z.B. eine Subtraktion – durchführen kann, die ein bestimmtes Ergebnis liefert. Es empfiehlt sich, diese Begriffe irgendwie zu lernen; sie tauchen immer wieder auf.)

Einige BASIC-Versionen gestatten auch die Verarbeitung von *ganzen Zahlen* (also von Zahlen ohne gebrochenen Anteil, ohne Komma) in eingeschränktem Umfang. Das hat meist den Vorteil, daß die interne Verarbeitung im Computer schneller ist. Wir werden in unseren Beispielen jedoch, wenn nicht ausdrücklich etwas anderes gesagt ist, immer reelle Zahlen annehmen.

Eine solche Verknüpfung kann mit Hilfe der üblichen mathematischen *Operatoren* (d.h. der Rechenzeichen) angegeben wer-

den. Deren Form ist allerdings den Bedingungen der Tastatureingabe und dem meist verwendeten Zeichensatz zum Verkehr mit dem Computer angepaßt. (ASCII-Zeichen; das sind Zeichen aus dem «American Standard Code for Information Interchange» – «Amerikanischer Standardkode zum Informationsaustausch», der international als ISO-7-bit-Kode vereinbart ist [im Anhang findet sich eine Tabelle der verfügbaren ASCII-Zeichen].) Die in BASIC verwendeten mathematischen Operatoren finden Sie in Tabelle 2.

monadische Operatoren (1 Operand)			
höchste Priorität	+,-	Vorzeichen	
dyadische Operatoren (2 Operanden)			
Priorität:			
höchste	↑	Potenzieren	a↑b = a^b
	*,/	Mult.,Div.	a*b = a•b
niedrigste	+,-	Add.,Sub.	
Hilfszeichen			
	(,)	nur runde Klammern erlaubt	

Tabelle 2 Operatoren in BASIC

Zahlen

Wir wollen nun aber nicht die Wegstrecke wissen, sondern nach Formel (1) den Benzinverbrauch unseres Wagens berechnen. Diese Formel können wir aber leider nicht so eingeben, wie sie da steht, selbst wenn wir uns die Mühe machen wollten, einen Bruchstrich in den Rechner einzutippen. Das Problem liegt einfach darin, daß wir das Befehlsargument in einer Zeile mit dem Befehlswort zusammen angeben müssen. Um den Bruchstrich zu schreiben, müßten wir aber die Zeile abbrechen, und damit hätten

wir auch schon den Befehl beendet (probieren Sie das einmal an Ihrem Computer aus).

Man verwendet daher eine andere Darstellung, in der Zahlen und Rechenzeichen nacheinander über die Tastatur eingegeben werden können. Als Bruchstrich dient z.B. der auch sonst gebräuchliche *Schrägstrich* «/». Den Wert «Einhalb» erhalten wir z.B. so:

```
>PRINT 1/2
 .5
READY
```

Das heißt «0,5» – nur im amerikanischen Format geschrieben, wo man statt des Dezimalkommas einen Dezimal*punkt* verwendet und gerne eine führende Null unterschlägt. Wir müssen uns also ein wenig umgewöhnen.

Vor allem sollte statt des Dezimalpunkts kein Komma verwendet werden, denn das hat eine ganz andere Wirkung. Wenn wir z.B. die Ausgabe der Zahl 1,2 verlangen, dann erhalten wir

```
>PRINT 1,2
 1             2
READY
```

Das ist aber etwas ganz anderes als

```
>PRINT 1.2
 1.2
READY
```

Dagegen werden führende Nullen vor dem «Komma» toleriert:

```
READY
>PRINT 0.1
 .1
```

Der Rechner nimmt unsere Eingabe widerspruchslos entgegen, setzt aber bei der Wiedergabe seinen eigenen Kopf durch und unterschlägt die Null.

Anders sieht es natürlich mit den zum gebrochenen Anteil gehörenden Nullen aus:

```
READY
>PRINT 0.01
 .01
```

Doch wenn man das noch etwas weiterführt, gibt es eine Überraschung:

```
READY
>PRINT 0.001
 1E-03
```

Das soll den Wert «0.001» darstellen (wir übernehmen in Zukunft den Dezimalpunkt, um Verwechslungen vorzubeugen). Allerdings ist jetzt die Schreibweise verändert worden. Wenn die Zahlen kleiner als ein bestimmter Wert sind, dann verwenden die meisten BASIC-Versionen das sogenannte *wissenschaftliche Darstellungsformat*. Dieses Format ist von den meisten Taschenrechnern her geläufig. Hier wird die Zahl aufgeteilt in eine *Wertangabe* (die «1») und eine *Größenangabe* («E-03»). Das letztere ist nichts weiter als eine Abkürzung für «mal 10^{-3}». Damit läßt sich die Darstellung folgendermaßen übersetzen:

$$1E\text{-}03 = 1 \times 10^{-3} = 1 \times 0.001 = 0.001$$

Und was soll dieser Umstand? Nun – zum einen hilft diese Methode Platz einzusparen. Es ist ein Unterschied, ob man 0.0000000001 oder 1E-10 schreibt. Doch es steckt noch ein bißchen mehr dahinter. Experimentieren wir ein wenig:

```
READY
>PRINT 1/3
 .333333
READY
>PRINT 0.1/3
 .0333333
READY
>PRINT 0.01/3
 3.33333E-04
```

Jedesmal haben wir es mit einem unendlichen Dezimalbruch zu tun. Es fällt daher auf, daß immer *dieselbe Stellenzahl* für die Wertangabe verwendet wird: 6 (gültige) Ziffern in unserer BASIC-Version. (Andere Versionen können mehr Stellen verwenden, doch ändert das nichts am Prinzip.) Diese sechs Ziffern geben die *Rechengenauigkeit* wieder, in der unsere BASIC-Version arbeiten kann: Sechs Stellen, mehr nicht. Alles übrige wird einfach unterschlagen. Wird die Zahl nun kleiner, so daß sie mit sechs Stellen nicht mehr ausreichend erfaßt werden kann, dann teilt man einfach die Wertangabe von der Größenangabe und behält mit diesem Trick immer die volle Rechengenauigkeit über sechs Stellen bei.

Das gilt natürlich auch für große Zahlen. Experimentieren Sie ein wenig mit Ihrem Computer, damit Sie die Form der Zahlen in den Griff bekommen. Lassen Sie sich Zahlen aller Art und Größe ausdrucken, positive Zahlen und negative Zahlen. Vermerken Sie dabei nicht nur die Art der Darstellung, sondern machen Sie die Zahlen immer kleiner und immer größer, etwa so:

```
>READY
PRINT 1E-01
 .1
READY
>PRINT 1E-02
 .01
 . . .
READY
>PRINT 1E-10
 1E-10
READY
>PRINT 1E-20
 1E-20
READY
>PRINT 1E-30
 1E-30
READY
>PRINT 1E-40
 0
```

Hier wurde der *interne Zahlenbereich* unterschritten. Der Rechner kann mit der gegebenen BASIC-Version auch bei wissenschaftlichen Darstellungsformen keine beliebig kleinen Zah-

len darstellen. Er hat zur Erfassung der Größenordnung ebenfalls eine bestimmte Grenze. Wird die Zahl kleiner als der kleinste darstellbare (absolute) Wert, dann druckt er einfach 0 aus.

Ganz Entsprechendes gilt es nach oben hin:

```
>PRINT 1E10
 1E+10
READY
>PRINT 1E20
 1E+20
READY
>PRINT 1E30
 1E+30
READY
>PRINT 1E40
?OV ERROR
READY
```

Hier will der Rechner nicht weiter. Die Zahl ist ihm zu groß geworden, und da er hierfür keinen so bequemen Ersatz hat, wie die Null bei kleinen Beträgen, druckt er eine *Fehlermeldung* (?OV ERROR) aus. Wir werden uns mit solchen Fehlermeldungen später noch intensiver beschäftigen, hier genügt es festzustellen, daß OV ERROR für «Overflow Error» steht, was «Überlauffehler» bedeutet und besagt, daß der für die Zahl vorgesehene Arbeitsbereich im Rechner zu klein geworden, die Zahl schlichtweg «übergelaufen» ist.

Versuchen Sie, die Grenzen einzukreisen. Ermitteln Sie die größte und die kleinste Zahl, die ihre BASIC-Version noch verarbeitet. Es ist wichtig, ungefähre Vorstellungen über den verwendeten Zahlenbereich zu haben. Sie werden feststellen, daß diese Grenzen recht «krumme» Werte sind. Das liegt daran, daß die meisten BASIC-Computer intern nicht im uns gewohnten Dezimalsystem arbeiten. Lassen Sie sich dadurch nicht verwirren. Sie brauchen sich den darstellbaren Zahlenbereich keinesfalls *genau* zu merken. Aber so ungefähr sollten Sie schon wissen, was Sie Ihrem Rechner zutrauen können. Und wenn Sie einmal den genauen Wert benötigen, so finden Sie ihn mit Sicherheit in Ihrem Handbuch.

Eine letzte Bemerkung noch zu diesem Thema: Die Rechengenauigkeit, mit der intern gearbeitet wird, unterscheidet sich von BASIC-Version zu BASIC-Version. Das schlägt sich vor allem in der Genauigkeit der Ergebnisse nieder. Es kann sein, daß sich unsere Beispiele von den auf Ihrem Computer ermittelten ab der vierten oder fünften Stelle unterscheiden. Das wird um so eher der Fall sein, je komplizierter die Rechnungen sind. Sollten Ihre Experimente von den hier vorgeführten daher in dieser Größenordnung abweichen, dann muß nicht unbedingt etwas schiefgelaufen sein. Vielleicht rechnet Ihr Computer nur etwas genauer (oder auch ungenauer) als das für die Beispiele verwendete Gerät.

Die Form von numerischen Ausdrücken: Priorität

Wir müssen also Formel (1) so umformulieren, daß sie in einer Zeile mit den auf der Tastatur gegebenen Zeichen geschrieben werden kann. Dazu fassen wir die Teile geeignet durch Klammern zusammen, wie sie sich nacheinander in der Rechnung ergeben, etwa so:

$$V = (B / (S2 - S1)) * 100$$

Wenn wir hier die passenden Werte einsetzen, dann erhalten wir vom Rechner das richtige Ergebnis:

```
READY
>PRINT (10 / (42090 - 41956)) * 100
 7.46269
```

Allerdings sind die vielen Klammern etwas umständlich zu schreiben. Wie wäre es, wenn wir das äußere Paar einfach entfallen ließen:

```
READY
>PRINT 10 / (42090 - 41956) * 100
 7.46269
```

Und wenn wir die restlichen Klammern auch noch weglassen?

```
READY
>PRINT 10 / 42090 - 41956 * 100
-4195600
```

Das ist mit Sicherheit falsch. Offenbar beherrscht der Rechner die «Punkt- vor Strichrechnungs»-Regel, denn wir haben das Ergebnis aus 0,0002376 − 4105600, was der Computer einfach zu −4195600 gerundet hat. (0,0002376 ist das Ergebnis aus 10 / 42090, und das andere ergibt sich aus 41956 * 100.)

Mit anderen Worten: BASIC räumt den verschiedenen mathematischen Operationen unterschiedlichen Vorrang, verschiedene *Priorität* ein. «Punktrechnung», also Multiplikation und Division hat Vorrang vor der «Strichrechnung», d.h. vor Addition und Subtraktion. Sie wird zuerst ausgeführt, was man damit beschreibt, daß sie «höhere Priorität» habe. In Tabelle 2 sind die verschiedenen Prioritäten angegeben. Aus ihr können wir entnehmen, daß in einer Rechnung erst die Vorzeichen eingearbeitet werden, dann wird potenziert, dann folgen Multiplikation und Division und schließlich Addition und Subtraktion. Diese Prioritätsstufung läßt sich mit Hilfe einfacher runder Klammern umgehen, so wie wir es im vorletzten Beispiel getan haben, wo zuerst die Differenz (42090−41956) berechnet wurden, dann wurde geteilt und zum Schluß das Ganze mit 100 malgenommen. Unsere Formel erhält damit folgende allgemeine Gestalt:

$$V = B / (S2 - S1) * 100 \qquad (2)$$

Variable und LET

Um den Durchschnittsverbrauch zu erhalten, müssen wir die einzelnen Verbrauchswerte addieren und dann durch ihre Anzahl teilen. Das würde bedeuten, daß wir zunächst einmal alle Einzelwerte berechnen müßten, etwa so:

```
READY
>PRINT 10 / (42090 - 41956) * 100
 7.46269
READY
>PRINT 10 / (42243 - 42090) * 100
 9.80392
```

und so weiter. Diese Ergebnisse wären zu notieren, ihre Anzahl festzustellen und dann neu einzutippen. Bei den beiden eben berechneten Werten z.B. so:

```
READY
>PRINT (7.46269 + 9.80392) / 2
 8.63331
```

Das ist nicht ganz das, was einem so als Arbeit eines Computers vorschwebt. Richtiger wäre es doch, bräuchte man nur die Tabellenwerte irgendwie geeignet angeben und der Computer würde uns den Durchschnittswert mitteilen. Irgend etwas läuft noch nicht so, wie wir es erwarten würden. Aber was?

Nun, das Hauptproblem liegt offensichtlich daran, daß der Rechner bis jetzt alles, aber auch alles sofort wieder vergißt, wenn er die Befehlszeile abgearbeitet hat. Insbesondere gilt das für die einzelnen Teilergebnisse. Er druckt sie aus und fertig. Es wäre schon einiges geholfen, wenn wir diese einzelnen Zwischenwerte im Rechner geeignet festhalten, sie in ihm *speichern* könnten. Ja, das ist sogar eine der wichtigsten Voraussetzungen dafür, daß der Computer das tut, was wir von ihm erwarten, nämlich automatisch zu arbeiten.

Dazu aber benötigt man zweierlei:

a) einen *Speicherplatz* im Computer, in dem die betreffende Zahl festgehalten werden kann und
b) eine Methode, auf diesen Speicherplatz *zuzugreifen*, d.h. Werte in ihm abzulegen und wieder daraus hervorzuholen.

Zum ersten Punkt: Die üblichen BASIC-Versionen bieten eine große Zahl möglicher Speicherplätze an. Man hat sich darauf geeinigt, diese einfach mit Großbuchstaben A ... Z, eventuell noch mit einer nachgestellten Zahl (A0, ..., A9, B0, ..., Z9) zu bezeichnen. (Einige Versionen erlauben auch die Verwendung zweier Buchstaben, ja sogar beliebiger Wörter – das ist dem jeweiligen Handbuch zu entnehmen.) Gehen wir im folgenden davon aus, daß wir Speicherplätze mit den Bezeichnungen A, B, ..., Z, A0, A1, ..., A9, B0, ..., Z9 zur Verfügung hätten.

Damit bleibt noch die Frage, wie man diese Speichermöglichkeiten eigentlich einsetzt, d.h.:

1. Wie speichert man einen Wert?
2. Wie holt man einen Wert aus dem Speicher zurück?

Von Taschenrechnern her kennen wir Befehle der Art «STO n» (für das Englische «store in n») zum Ablegen eines Werts im Speicher Nummer n und «RCL n» (recall from n) für das Rückrufen des Werts an einer passenden Stelle im Rechengang. Das ist dort durchaus sinnvoll, aber es geht auch etwas einfacher. Grundlage dafür ist die Tatsache, daß man (in den angegebenen Grenzen) für den betreffenden Speicherplatz einen Namen frei aussuchen kann. Statt «Speichere den Wert xxx im Speicher V» können wir auch sagen:

Laß den Inhalt von V = xxx sein!

«den Inhalt von» können wir uns noch schenken, so daß wir erhalten:

Laß V = xxx sein!

oder englisch:

LET V = xxx.

Das entspricht formal einem BASIC-Befehl:

Befehlswort: LET
Befehlsargument: V = xxx

Noch einfacher können wir uns das Rückrufen des Wertes machen. Den ausdrücklichen Befehl «RCL n» brauchten wir beim Taschenrechner nur, weil der Speicher durch eine Nummer gegeben und damit mit seinem Inhalt verwechselbar war. Hier sollte es eigentlich genügen, einfach den Namen des Speichers anzugeben.

Das alles läßt sich ausprobieren. (Wir schenken uns aus Platzgründen das «READY» nach jedem Schritt.)

```
>LET V = 1234
>PRINT V
 1234
>LET V = 45678
>PRINT V
 45678
```

Damit aber haben wir den Namen des Speichers so verwendet, wie man in der Mathematik Variable einsetzt. D.h., wir haben der Variablen V zuerst einen Wert (1234 bzw. 45678) *zugewiesen* und danach den Speicherinhalt in einem Befehlsargument durch Angabe der Variablen V *abgerufen*.

Aus diesem Grund spricht man in BASIC (und den anderen problemorientierten höheren Programmiersprachen nicht von Speicherplätzen, sondern von den möglichen *Variablen*

A, ..., Z, A0 ..., A9, B0 ..., Z9

Das hat den Vorteil, daß man sich als Programmierer überhaupt nicht um die Verteilung der Speicherplätze im Rechner zu kümmern braucht, sondern das Problem in einer natürlichen Weise mathematisch behandeln kann. Es ist allerdings wichtig, zu wissen, daß jeder derartigen Variablen ein bestimmter Speicherraum zugeordnet werden muß. Dieser umfaßt je nach Auslegung der BASIC-Version 5 bis 12 Speichereinheiten (die man als *Bytes* bezeichnet). Würden wir alle oben angegebenen Namen ausnutzen, so bräuchten wir mindestens

$$5 \times (26 + (26 \times 10)) = 1430 \text{ Bytes.}$$

Nun ist Speicherplatz in einem Computer etwas recht Kostbares, mit dem man sparsam umgehen sollte. Aus diesem Grund hält man diese Speicherplätze nicht ständig bereit, sondern faßt die vorgegebenen Variablennamen als *Möglichkeiten* auf. D.h., man reserviert einen derartigen Speicherplatz erst dann, wenn die Variable im Programm ausdrücklich benannt worden ist. Dann braucht man bei einem Programm mit wenigen Variablen auch nur wenig Speicherraum im Computer und hat mehr Platz für die Problemlösung (die selbst Speicherraum benötigt).

Das können wir uns zunutze machen. Anstatt die einzelnen Teilergebnisse nach jeder Rechnung zu notieren und anschließend für die Berechnung des Durchschnitts wieder in den Rechner einzutippen – mit all den Fehlern, die dabei möglich sind – können wir sie ja auch in passenden Variablen speichern und für den letzten Rechenschritt einfach aus dem Speicher abrufen.

Das könnte z.B. so aussehen (wir beschränken uns wieder auf zwei Verbrauchswerte):

```
>LET A = 10 / (42090 - 41956) * 100
>LET B = 15 / (42243 - 42090) * 100
>PRINT (A + B) / 2
 8.63331
```

Diese Rechnerei läßt sich noch weiter vereinfachen. Warum nicht gleich die einzelnen Verbrauchswerte im Speicher aufaddieren? Dann brauchen wir nur die Einzelwerte zu zählen und zum Schluß nur noch den aufaddierten Betrag durch diese Anzahl zu dividieren. D.h., wir sparen uns die Summe (A + B + ···), die bei langen Tabellen einen ziemlichen Umstand machen kann. Nur, wie macht man so etwas?

Probieren wir es einfach aus. Was wir haben wollen, ist folgendes: Der Rechner soll zu einer Variablen, nennen wir sie V, jedesmal einen Wert addieren und diesen neuen Wert *danach* in derselben Variablen V ablegen. Formal gesehen sollte also folgendes geschehen:

$$V \leftarrow V + xxx$$

(Der Pfeil soll hier angeben, daß der Variablen V der rechts vom Pfeil stehende Wert *zugewiesen* wird, «xxx» steht für irgendeinen numerischen Ausdruck, der zum *alten* Wert von V addiert werden soll.)

Das können wir in einen BASIC-Befehl übersetzen, denn die Wertzuweisung «V ←» wird hier ja durch den Befehl «LET V = » ausgedrückt, also:

$$\text{LET V} = \text{V} + xxx$$

Überzeugen wir uns davon, daß das auch stimmt. Weisen wir der Variablen V einen bestimmten Wert zu und addieren danach einen anderen Wert. Das sieht dann so aus:

```
>LET V = 123
>PRINT V
 123
>LET V = V + 234
>PRINT V
 357
```

Ganz offensichtlich stimmt das mit unserer Annahme überein. Machen wir es uns zunutze:

Erst müssen wir die Variable *löschen*, d.h. ihren Wert auf Null setzen:

```
>LET V = 0
```

Dann können wir unsere Tabelle einrechnen:

```
>LET V = V + 10 / (42090 - 41956) * 100
>LET V = V + 15 / (42243 - 42090) * 100
>PRINT V / 2
 8.63331
```

(Natürlich hätten wir als ersten Schritt auch befehlen können

```
>LET V = 10 / (42090 - 41956) * 100
```

Dann würden wir den ersten *Initialisierungs*schritt

```
>LET V = 0
```

einsparen. Doch empfiehlt es sich, längere stumpfsinnige Berechnungen immer nach demselben Schema ablaufen zu lassen, also immer «V = V + xxx» zu schreiben. Das verringert die Fehleranfälligkeit.)

Kommandos und Anweisungen, NEW und RUN

Damit haben wir aber nur einen Teil unseres Problems gelöst. Denn noch immer kann von einer automatischen Computerarbeit keine Rede sein. Für jeden Tankvorgang, für jede Tabellenzeile müssen wir die Formel

```
LET V = V + B / (S2 - S1) * 100
```

neu eintippen. Und wenn wir hundertmal getankt haben, müssen wir sie auch hundertmal eingeben. In dieser Art und Weise hätten wir die Aufgabe auch – möglicherweise sogar bequemer – mit einem normalen Taschenrechner erledigen können.

Unser BASIC-Computer behält immer noch zuwenig. Nach jedem Befehl, unmittelbar nach Abschluß der Zeile durch <CR> (d.h. einen Wagenrücklauf), führt er ihn aus – und dann vergißt er ihn schlichtweg. Wir brauchen eine Möglichkeit, außer Daten auch *Befehle* im Rechner zu speichern und bei Bedarf im Zusammenhang abarbeiten zu lassen.

Im Grunde wollen wir folgendes:

1. Wir geben die Formel für die Summe der einzelnen Verbrauchswerte

   ```
   LET V = V + B / (S2 - S1) * 100
   ```

   in den Computer ein, ebenso einen Befehl zum Ausdrucken des Durchschnittsverbrauchs:

   ```
   PRINT V / N
   ```

   (wobei N die Anzahl der Einzelwerte ist, die der Rechner ja auch ohne unseren unmittelbaren Beistand ermitteln kann).
2. Dann geben wir auf irgendeine geeignete Weise unsere Tabellenwerte *einmal* in den Computer ein, damit er weiß, womit er zu arbeiten hat.
3. Schließlich befehlen wir dem Computer, daß er die Rechnungen bearbeiten soll, und erhalten so von ihm ohne weiteren Eingriff den Durchschnittsverbrauch.

4. Und das Ganze soll dann noch so abgefaßt sein, daß wir dem Rechner für eine andere Verbrauchsberechnung nichts weiter als die Tabellenwerte zu übermitteln brauchen. D.h., es sollte genügen, die Befehle aus Schritt 1 nur einmal anzugeben und dann immer über sie verfügen zu können.

Bis jetzt war es ja so, daß wir jeden Rechenschritt, jeden Befehl für sich angeben mußten, der dann unmittelbar, *direkt* nach der Eingabe vom Computer verarbeitet wurde. Man bezeichnet derart behandelte Befehle in BASIC daher als *Direkte Befehle* oder *Kommandos* (englisch: commands).

Unser Ziel aber ist es, diese Befehle geeignet zu einem *Programm* zusammengefaßt im Computer zu speichern und erst später «am Stück» abarbeiten zu lassen. Das bedeutet, daß wir einen Weg finden müssen, die Befehle so zu formulieren, daß sie vom Rechner als *Programmbefehle* und nicht als Direkte Befehle aufgefaßt werden. Ebenso wie für Direkte Befehle verwendet man auch für Programmbefehle eine eigene Bezeichnung: Man nennt sie *Anweisungen* (nach dem englischen Begriff «statements»).

Nun haben wir bis jetzt gesehen, daß ein Befehl (jedenfalls LET und PRINT) immer dann direkt abgearbeitet wird, wenn man die Zeile mit einem Befehlswort einleitet. Hier können wir ansetzen. Es gibt vermutlich zwei Möglichkeiten, das gewünschte Ziel zu erreichen:

Entweder signalisiert man dem Rechner mit einem besonderen Befehlswort: «Achtung! Alles Nachfolgende ist im Speicher als Programm abzulegen!» und behandelt den Programmtext als *Argument* zu diesem Befehl, oder man *kennzeichnet* jede Zeile durch eine besondere Zeichenfolge als nicht direkt abzuarbeitenden, sondern zu speichernden Befehl.

Beides ist üblich und hat seine besonderen Vorzüge (und Nachteile). Jedoch läßt sich die zweite Methode einfacher verwirklichen, weshalb sie in BASIC vorgezogen wird.

Wir brauchen damit eine *Zeilenkennzeichnung*, die folgenden Bedingungen genügt:

1. Sie muß rasch und eindeutig von Befehlswörtern zu unterscheiden sein.
2. Sie muß mit dem kleinstmöglichen Ausschnitt aus dem ASCII-Zeichensatz dargestellt werden können.
3. Aus der Angabe der Programmzeilen muß die spätere Reihenfolge der Befehlsabarbeitung hervorgehen.

Alle diese Bedingungen lassen sich einfach mit Zahlen erfüllen. Sie sind eindeutig von Befehlswörtern zu unterscheiden, die immer mit einem Buchstaben beginnen, und sie legen ohne weitere Maßnahmen eine Reihenfolge der Befehlszeilen im Programm fest. Wir numerieren dazu einfach die Zeilen in aufsteigender Reihenfolge durch. (Dabei müssen die Nummern nicht unmittelbar aufeinander folgen, sondern können auch in größeren Abständen – z.B. in Zehnerschritten – angegeben werden. Das wird sich unten noch als sehr nützlich herausstellen.)

Bevor wir das ausprobieren können, müssen wir allerdings noch einen technischen Punkt beachten, der den *Speicher* betrifft, in den wir das Programm einschreiben möchten. Zwar ist der Bereich dafür durch die verwendete BASIC-Version schon bestimmt, so daß wir uns nicht mehr um seine Anordnung im Computer selbst zu kümmern brauchen, aber es könnte dort z.B. bereits ein anderes Programm stehen, das mit den neu einzugebenden Befehlszeilen kollidieren würde. Oder der Speicher muß nach dem Einschalten (irgendwie – das hängt von der Art des BASIC ab) initialisiert werden. Jedenfalls müssen wir den Speicher erst für unser Programm (und nur für dieses Programm) aufnahmebereit machen.

Leider gibt es hierzu keinen verbindlich festgelegten BASIC-Befehl. Zumeist findet man die Angabe NEW (d.h., alle folgenden Befehle stellen ein *neues* Programm dar), aber auch direkte Löschbefehle, wie z.B. CLEAR (CLR) oder SCRATCH (SCR) werden angeboten. Einige BASIC-Versionen erlauben auch ein wahlweises Löschen des Speicherteils, in dem das Programm untergebracht ist, bzw. des Bereichs, in dem die Variablen stehen. (Auch

hier wieder: Sehen Sie bitte im Handbuch Ihrer BASIC-Version nach.) Wir wollen uns auf NEW einigen, was den gesamten Speicher löschen soll.

Sehen wir uns erst einmal an, ob diese Überlegungen wenigstens im Prinzip stimmen. D.h., versuchen wir zunächst, den Rechner das als Programm nachvollziehen zu lassen, was wir oben mit direkten Befehlen erreicht haben.

```
>NEW
READY
>10 LET V = 0
>20 LET V = V + 10 / (42090 - 41956) * 100
>40 PRINT V / 2
>30 LET V = V + 15 / (42243 - 42090) * 100
```

(Beachten Sie, daß wir Zeile 40 *vor* Zeile 30 eingegeben haben. Das ist kein Druckfehler. Wir wollen damit testen, ob die Zeilennummern wirklich für die Reihenfolge der Befehlsabarbeitung zuständig sind, oder ob nicht doch die Reihenfolge des Aufschreibens den Ausschlag gibt. Denn: Kontrolle ist – wie man weiß – allemal besser.)

Wir sehen zumindest einen Unterschied zu dem bis jetzt bekannten Verhalten: Die Befehle (außer NEW) wurden nicht direkt abgearbeitet. Das sieht schon ganz gut aus. Allerdings ist es noch lange kein Beweis dafür, daß die Befehle auch so, wie wir das beabsichtigt hatten, im Speicher stehen.

Das können wir auf die verschiedenste Weise nachprüfen. Die einfachste wäre: Wenn die Befehle richtig im Speicher stehen und wenn wir den Computer dazu bekommen, sie abzuarbeiten, dann müßte ja das Gewünschte ausgedruckt werden. Und wenn die Reihenfolge der Zeilennummern den Ausschlag gibt, dann müßten wir als Ergebnis den Wert 8.63331 erhalten. Wenn es dagegen die Reihenfolge des Aufschreibens ist, müßte der Computer die Hälfte des ersten Verbrauchswertes, nämlich 3.73134 ausgeben.

Wie bekommt man den Rechner dazu, ein Programm abzuarbeiten?

Auch hier sind wieder mehrere Möglichkeiten denkbar. Die meisten BASIC-Versionen verwenden hierfür das Kommando RUN (etwa: «Lauf los und arbeite die im Programmspeicher stehenden Befehle in der angegebenen Reihenfolge ab!»). Andere derartige Startbefehle wären «XEQ» (für «execute» – abarbeiten) oder «GO» (was dem RUN im Sinn vergleichbar wäre). Man findet sie aber nur selten und dann oft auch neben RUN mit besonderer Bedeutung (z.B. Lauf ohne vorheriges Löschen der Variablen). Wichtig ist, daß man fast immer RUN verwenden kann.

Damit können wir nachprüfen, ob unsere Programmeingabe sinnvoll war:

```
>RUN
 8.6333
```

Unsere Annahmen scheinen also zu stimmen: Der Rechner hat das erwartete Ergebnis richtig ausgedruckt. Das heißt, daß zum einen die Angabe von Zeilennummern in der Tat eine Programmzeile einleitet, und daß zum zweiten die Reihenfolge dieser Nummern für die Abarbeitungsfolge der Anweisungen entscheidend ist.

Edieren des Programmtextes: LIST

Ein unmittelbarer Weg, das in den Speicher eingegebene Programm zu kontrollieren, wäre, wenn wir es – so wie es im Speicher steht – auslisten lassen könnten. Nun, das Problem ist lösbar:

Alle (interpretierenden) BASIC-Versionen kennen das Kommando LIST, der in unserem Fall folgendes Ergebnis bringen würde:

```
>LIST
 10 LET V = 0
 20 LET V = V + 10 / (42090 - 41956) * 100
 30 LET V = V + 15 / (42243 - 42090) * 100
 40 PRINT V / 2
```

Wir sehen: Der Rechner hat die von uns bei der Eingabe vertauschten Zeilen 30 und 40 richtig sortiert, so wie wir es erwartet haben.

Was aber, wenn wir uns entscheiden, lieber noch einen Einzelwert in die Durchschnittsberechnung mit eingehen zu lassen? Dann müßten wir *vor* dem Ergebnisausdruck in Zeile 40 eine weitere Berechnung einfügen und natürlich die Formel für den Durchschnittswert von «V/2» auf «V/3» abändern. Wie macht man das aber, ohne das ganze Programm neu einzutippen?

Nun, wir müssen dafür sorgen, daß der Rechner die betreffende Befehlszeile *zwischen* Zeile 30 und Zeile 40 *einordnet*. Nach dem, was wir bis jetzt wissen, würde das bedeuten, daß wir diese Anweisung nur mit einer geeigneten Zeilennummer anzugeben brauchen. Tun wir das:

```
>35 LET V = V + 10 / (42340 - 42243) * 100
```

Allerdings müssen wir uns zuerst davon überzeugen, daß unsere Überlegung stimmt. Sehen wir uns dazu einfach an, was der Computer jetzt als Programm betrachtet:

```
>LIST
 10 LET V = 0
 20 LET V = V + 10 / (42090 - 41956) * 100
 30 LET V = V + 15 / (42243 - 42090) * 100
 35 LET V = V + 10 / (42340 - 42243) * 100
 40 PRINT V / 2
```

(Eben diese Möglichkeit war der Grund, weshalb wir die Befehlszeilen in Zehnerschritten numeriert hatten. Es empfiehlt sich, dies grundsätzlich zu tun, denn irgendeinen Fehler, irgendeine Ergänzung finden Sie nach Abschluß der Programmeingabe fast immer, besonders bei neu erstellten Programmen.)

Ganz fertig sind wir mit unserer Änderung nicht, denn noch müssen wir «PRINT V/2» in «PRINT V/3» abändern. Überlegen wir: Wenn wir durch Angabe der Zeilennummer 35 den Rechner dazu bekommen haben, eine neue Verbrauchsberechnung zwischen die Zeilen 30 und 40 einzufügen, dann könnte es doch sein,

daß man eine Zeile einfach dadurch korrigieren kann, daß man sie einfach *noch einmal* unter ihrer Nummer angibt.

Was zu beweisen wäre:

```
>40 PRINT V / 3
>LIST
 10 LET V = 0
 20 LET V = V + 10 / (42090 - 41956) * 100
 30 LET V = V + 15 / (42243 - 42090) * 100
 35 LET V = V + 10 / (42340 - 42243) * 100
 40 PRINT V / 3
```

Sehen wir uns noch an, ob das auch so abgearbeitet wird, wie wir es erwarten:

```
>RUN
 9.19196
```

(Überzeugen Sie sich selbst davon, daß dies in der Tat das richtige Ergebnis ist.)

Nun – der Mensch ist ein wankelmütiges Geschöpf – beschließen wir, daß es vielleicht doch besser wäre, den dritten Durchschnittwert wegzulassen. Also müssen wir Zeile 35 wieder aus dem Programm entfernen. Wie macht man das?

Die Überlegung folgt den gleichen Bahnen wie oben: Wenn wir eine zu korrigierende Zeile einfach mit einem neuen Text überschreiben können, vorausgesetzt, wir geben die richtige Zeilennummer mit an, so liegt die Vermutung nahe, daß man eine Zeile einfach dadurch aus dem Programm entfernt, indem man eine «Leerzeile», also nur die Zeilennummer und sonst nichts, angibt. Probieren wir es:

(Beachten Sie: «<CR>» steht für den Wagenrücklauf «RETURN»!)

```
>35<CR>
>LIST
 10 LET V = 0
 20 LET V = V + 10 / (42090 - 41956) * 100
 30 LET V = V + 15 / (42243 - 42090) * 100
 40 PRINT V / 3
```

Damit haben wir einiges Handwerkszeug zur Programmkorrektur, d.h. zum «Edieren» (oder auch «Editieren») des Programmtextes zur Hand:

1. Man kann eine Zeile *korrigieren,* indem man sie mit ihrer Zeilennummer zusammen neu eingibt.
2. Man kann eine Zeile *einfügen,* indem man sie mit einer zwischen den Nummern der betreffenden Befehlszeilen liegenden Zeilennummer eingibt.
3. Man kann eine Zeile *löschen,* indem man nur ihre Zeilennummer (und sonst nur noch RETURN!) eingibt.

Man braucht im übrigen nicht bei jedem Tippfehler gleich die gesamte Zeile neu einzugeben. Wird der Fehler noch rechtzeitig vor Abschluß der Zeile durch RETURN entdeckt, so kann man mit Hilfe bestimmter *Steuertasten* auf der Tastatur entweder das zuletzt eingegebene Zeichen oder die ganze gerade eingegebene Zeile löschen. Voraussetzung ist nur, daß die Zeile noch nicht abgeschlossen ist.

(Manche BASIC-Versionen bieten sogar noch weitgehendere *Edierfunktionen* an.)

Leider ist die Vereinbarung derartiger Edierfunktionen und Steuertasten von BASIC zu BASIC verschieden. Sie müssen das in Ihrem Handbuch nachschlagen. Wir werden im folgenden jedenfalls immer fehlerfreie Programme annehmen.

Wenden wir uns zum Abschluß noch einmal dem LIST-Befehl zu. So, wie wir ihn jetzt kennen, kann man immer nur das ganze Programm am Stück auslisten lassen. Das kann bei langen Programmen unangenehm werden. Nehmen wir an, wir wollten in einem BASIC-Programm von 1000 Zeilen (das ist gar nicht so viel) den irgendwo mittendrin stehenden Block von Zeile 450 bis Zeile 500 ansehen. Befehlen wir da nur einfach LIST, so kann es sein, daß das ganze Programm rasend schnell über den Bildschirm rauscht, und – bevor wir eingreifen können – nur noch das letzte Dutzend Anweisungen zu sehen ist. Schlimmer noch ist es, wenn wir mit einem der langsamen Fernschreiberterminals arbei-

ten. Hier kann es eine kleine Ewigkeit dauern, bis der gewünschte Programmausschnitt aufgelistet ist (es dauert immer kleine Ewigkeiten, wenn man es eilig hat) – ganz abgesehen einmal vom Lärm und Papierverbrauch.

Aus diesem Grund bieten die meisten BASIC-Versionen irgendeine Form des *selektiven Auslistens* an. Im allgemeinsten Fall kann man dem LIST-Befehl das gewünschte Zeilenintervall als Argument nachstellen, meist in folgender Form:

LIST aaa – eee

Dabei stehen «aaa» und «eee» für die Nummer der Anfangs- bzw. Endzeile, die noch ausgedruckt werden soll. Einfachere Versionen gestatten nur die Angabe der Anfangszeile:

LIST aaa

und verfügen über irgendeinen Mechanismus, mit dem man den Ausdruckvorgang unterbrechen kann. Wir wollen im folgenden immer die allgemeinere Version annehmen, die folgende drei Spezialfälle aufweisen möge:

a) Auslisten von Zeile aaa bis zum Programmende:

LIST aaa –

b) Auslisten vom Programmanfang bis Zeile eee:

LIST – eee

c) Auslisten einer einzigen Programmzeile mit der Nummer nnn:

LIST nnn

Das spricht für sich. Sehen Sie auf alle Fälle in Ihrem BASIC-Handbuch nach, welche LIST-Möglichkeiten zur Verfügung stehen und experimentieren Sie ein bißchen damit. Stellen Sie irgendein längeres «Nonsens»-Programm zusammen, z.B. so:

```
>NEW
>10 A0 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
>20 A1 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
>30 A2 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

und so weiter, bis Sie 30 oder 40 Zeilen zusammenhaben. Versuchen Sie dann, mit Ihren Möglichkeiten die LIST-Sonderfälle a) bis c) von oben nachzuvollziehen.

Vorgabe von Daten: DATA und READ

Ein wichtiges Problem haben wir damit gelöst: Wir können ein Programm in den Computer eingeben, ohne daß er es gleich wieder vergißt, und wir können den Programmtext relativ einfach nachprüfen und bei Bedarf abändern. Dieses Programm kann auf Befehl (d.h. durch RUN) beliebig oft abgearbeitet werden.

Damit sind aber noch nicht alle von unseren Anforderungen erfüllt. Wir müssen noch einen Weg finden, unsere Kilometer/Liter-Tabelle irgendwie sinnvoll im Programm festzuhalten.

Bis jetzt hatten wir immer nur die Befehle betrachtet, also «PRINT x» oder «LET y = x». Es ist selbstverständlich wichtig, diese Befehle genauestens festzulegen, aber ohne zu wissen, *was* für ein «x» da ausgedruckt oder zugewiesen werden soll, kommen wir nicht weit. Es fehlen die *Daten,* mit denen gearbeitet werden soll. Und diese Daten müssen wir irgendwie – mit möglichst geringem Aufwand – dem Programm übergeben. Solange es nur wenige Angaben sind, die sich noch dazu in den verschiedensten Einsatzfällen nicht ändern, ist das kein großes Problem: Man nimmt sie einfach in das betreffende Befehlsargument als *Konstante* auf. So stellt z.B. in unserer Formel

LET V = V + B / (S2 – S1) * 100

die Zahl 100 eine solche Konstante dar. Die anderen Argumentsbestandteile aber – V, B, S1 und S2 – ändern sich von

Rechnungsschritt zu Rechnungsschritt. Sie müssen anders vorgegeben werden. Gewiß, man könnte sie wie in unseren ersten Versuchen jeweils in einer eigenen Formel als konkrete Zahlenwerte angeben. Doch hat das beträchtliche Nachteile.

Zum Beispiel muß die betreffende Formel für jeden neuen Datenwert neu geschrieben werden. Nun kennt man oft aber die betreffenden Werte bei der Programmaufstellung noch gar nicht. Wie soll man dann die Formeln aufschreiben? Man könnte hier das Problem erst dann anpacken, wenn alle (!) Daten lückenlos vorliegen.

Außerdem werden viele Werte nicht nur einmal verwendet. In unserem einfachen Beispiel ist ein Kilometerstand einmal als *Endwert* S2 und im darauffolgenden Rechenschritt als *Anfangswert* S1 einzugeben. Wenn die Aufgaben komplexer sind, kann es sein, daß man einen bestimmten Wert sehr oft benötigt. So etwas macht natürlich das Ändern eines Programms bei neuen Daten sehr umständlich, fehleranfällig und manchmal schlichtweg unmöglich.

Kurz gesagt: So geht es nicht! Wir müssen die Formeln und sonstigen Anweisungen so allgemein abfassen können, daß wir im eigentlichen Verarbeitungsteil für die in Frage kommenden Daten nur Variable stehen haben. Die Werte, die das Programm aufarbeiten soll, sollen – als *Daten* gekennzeichnet – nur einmal vorzugeben sein. Und das Programm muß sich aus diesem Datensatz die benötigten Werte selbst herausholen und den in Frage kommenden Variablen schrittweise zuweisen. D.h., es muß die vorgegebenen Daten *lesen* und auswerten können.

Machen wir es konkret: Wir wollen für unsere Formel

```
LET V = V + B / (S2 - S1) * 100
```

den Benzinverbrauch und die Wegangaben so wie in Tabelle 1 vorgeben können. Aufgabe des Programms muß es sein, bei jedem Rechenschritt den jeweiligen Benzinverbrauch aus der Tabelle zu entnehmen und ihn der Variablen B zuweisen. Weiter muß es die beiden dazugehörigen Kilometerwerte aus der Tabelle

lesen und den Variablen S1 und S2 zuweisen. Erst wenn das geschehen ist, kann die Rechnung durchgeführt werden.

Dieses Problem tritt so häufig auf, daß jede einigermaßen ausgebaute BASIC-Version über derartige Möglichkeiten verfügt. Sehen wir sie uns der Reihe nach an.

Zunächst einmal sind die Daten im Programm festzuhalten. Auch das muß in Form einer BASIC-Anweisung erfolgen. D.h., man braucht einen Befehl, der besagt

«Die nachfolgenden Argumente sind DATEN
für das Programm.»

«DATEN» heißt im Englischen

DATA

und so lautet auch das betreffende BASIC-Befehlswort. Die jeweils gemeinten Datenwerte folgen dem Befehl DATA als Argument. Sehen wir uns das für die ersten paar Tabellenwerte an:

```
>NEW
```

(Natürlich müssen wir erst das alte Programm löschen!)

```
>1000 DATA 41956
```

Das ist der erste Kilometerstand aus unserer Tabelle. Wir geben ihn unter der relativ hohen Zeilennummer 1000 als Datenwert an, damit wir davor genügend Platz für das Programm haben.

```
>1010 DATA 10
```

So viel Liter haben wir nach der Tabelle beim erstenmal getankt. Und um den Verbrauch ausrechnen zu können, komplettieren wir die Datenliste noch um den Kilometerstand, bei dem die 10 Liter verfahren waren:

```
>1020 DATA 42090
```

Nun können wir mit dem Datensatz alleine genau so wenig anfangen, wie mit der Formel alleine. Wir müssen noch die Verbindung zwischen beiden herstellen, d.h. einen Weg finden, auf dem der Rechner die Datenwerte aus der Datenliste *ausliest* und den Variablen in der Reihenfolge S1, B und S2 zuweist. Das «Auslesen» selbst ist einfach, auf Englisch

READ

und findet sich so im BASIC-Befehlssatz vor. Doch ist das nur eine Hälfte der Anweisung, die wir brauchen. Es fehlt noch das Befehlsargument: Was soll da eigentlich wohin gelesen werden?

Was gelesen werden soll ist klar: Die einzelnen Datenangaben aus der Datenliste, und zwar hübsch eine nach der anderen, mit dem ersten Wert angefangen. Nun kennen wir aber beim Programmieren in der Regel die einzelnen Werte noch gar nicht und wissen auch nicht, wo sie genau im Programmtext zu stehen kommen. Doch können wir diese Arbeit getrost dem Computer überlassen, der ohne weitere Maßnahmen in der Lage sein dürfte, nacheinander die verschiedenen DATA-Anweisungen bei Bedarf aufzufinden.

Anders sieht es mit dem «Wohin» aus. Das ist etwas, was wir beim Programmieren bereits kennen:

Die jeweilige Variable, der ein Wert aus der Datenliste zugewiesen werden soll. Wenn wir also der Datenliste

```
>1000 DATA 41956
>1010 DATA 10
>1020 DATA 42090
```

den Befehl

```
>100 READ S1
```

voranstellen, so müßte der Rechner eigentlich den ersten Tabellenwert, d.h. 41956, aufsuchen und ihn der Variablen S1 zuweisen. Das können wir ganz einfach nachprüfen, wenn wir den Inhalt von S1 ausdrucken lassen:

```
>110 PRINT S1
```

Und um ganz sicher zu gehen, löschen wir S1 am Programmanfang:

```
>10 LET S1 = 0
```

Sehen wir uns das einmal an:

```
>RUN
 41956
```

So weit stimmt alles. Unsere Vermutung reicht aber noch etwas weiter: Wenn wir nach Übernahme des ersten Werts aus der Datenliste eine weitere READ-Anweisung in das Programm setzen, etwa so:

```
>120 READ B
```

Dann müßte der Variablen B der *nächste* Tabellenwert, also die Zahl 10, zugewiesen werden. Prüfen wir das wie oben nach:

```
>130 PRINT B
```

wobei wir B sicherheitshalber erst einmal löschen:

```
>20 LET B = 0
```

Das ergibt:

```
>RUN
 41956
 10
```

Und für den dritten Tabellenwert müßte dasselbe gelten:

```
>30 LET S2 = 0
>140 READ S2
>150 PRINT S2
>RUN
 41956
 10
 42090
```

Sehen wir uns das noch einmal im Zusammenhang an:

```
>LIST
10 LET S1 = 0
20 LET B = 0
30 LET S2 = 0
100 READ S1
110 PRINT S1
120 READ B
130 PRINT B
140 READ S2
150 PRINT S2
1000 DATA 41956
1010 DATA 10
1020 DATA 42090
```

In Zeile 10 bis 30 setzen wir die Variablenwerte zunächst auf 0, wir *initialisieren* (d.h. geben ihnen bestimmte Anfangswerte). Dann lesen wir in Zeile 100, 120 und 140 Schritt für Schritt die in der DATA-Tabelle in Zeile 1000 bis 1020 angegebenen Werte und weisen sie jeweils einer Variablen zu. Die Ergebnisse dieser Zuweisungen haben wir uns in den PRINT-Anweisungen in Zeile 110, 130 und 150 bestätigen lassen.

Damit haben wir bis jetzt folgendes gefunden:

1. Man kann die jeweils zur Arbeit benötigten Daten mit Hilfe von DATA-Anweisungen im Programm festhalten.
2. Diese Daten lassen sich mit Hilfe von READ-Anweisungen aus der Tabelle auslesen und beliebigen Variablen im Programm zuweisen.
3. Jede READ-Anweisung übernimmt einen anderen Datenwert aus der DATA-Liste, wobei mit der Tabellenspitze angefangen und von READ zu READ schrittweise zum Ende hin gearbeitet wird.

Fehlermeldungen

Und was geschieht, wenn wir mehr Werte aus der Tabelle auslesen wollen, als angegeben sind? Wir könnten unserem Programm ja beispielsweise hinzufügen:

```
>160 READ C
```

Wenn wir jetzt RUN befehlen, sind im wesentlichen vier Möglichkeiten denkbar:

- der letzte Wert der DATA-Liste wird erneut übernommen,
- die Übernahme beginnt wieder mit dem ersten DATA-Wert,
- es geschieht nichts, d.h., der READ-Befehl wird einfach ignoriert,
- der Rechner meldet, daß da irgend etwas nicht stimmen kann, d.h., daß er zu einer READ-Anweisung keinen DATA-Wert mehr vorfindet.

Der letzte Fall ist am sinnvollsten, denn in aller Regel werden für die Variablen jeweils eigene Werte benötigt. Wenn ein solcher in der Datentabelle nicht aufzufinden ist, muß ein Fehler aufgetreten sein, den der Computer gefälligst melden sollte. Alles andere kann zu sehr schwer aufzufindenden Abarbeitungsfehlern des Programms führen, da – gesetzt den Fall, die Tabelle wurde wirklich aus Versehen zu kurz angegeben – mit den falschen Variablenwerten weitergerechnet wird.

Lassen wir uns das einmal vorführen:

```
>RUN
 41956
 10
 42090
?OD ERROR IN 160
```

Folgendes ist da geschehen: Zunächst wurden die ersten drei Datenwerte aus der Tabelle übernommen und ausgedruckt. Dann aber folgt mit der Zeile

```
?OD ERROR IN 160
```

eine *Fehlermeldung* des Computers:

- Ein Fragezeichen für: «Ich weiß nicht weiter»,
- die Angabe des Fehlers (OD ERROR) und schließlich
- ein Hinweis darauf, wo der Fehler aufgetreten ist: IN 160.

Das letztere ist offensichtlich die betreffende Zeilennummer, wie wir uns ganz einfach überzeugen können:

```
>LIST 160
160 READ C
```

In dieser Zeile 160 ist ein «OD ERROR» aufgetreten. «ERROR» heißt nichts anderes als «Fehler», und «OD» ist eine Abkürzung für die Art des Fehlers. Die genaue Bedeutung läßt sich aus dem Handbuch entnehmen, wo man findet:

OD – Out of Data

Das bedeutet: «Keine Daten mehr (in der DATA-Liste) vorhanden». Eine READ-Anweisung steht im Programm, für die keine DATA-Anweisung gefunden werden konnte.

Diese Art der Fehlermeldung ist verbreitet, aber bei weitem nicht die einzige Möglichkeit. Einige BASIC-Versionen drucken die (englische) Fehlerbeschreibung im Klartext aus, wieder andere geben nur eine Zahl als *Fehlerkode* an, die man dann im Handbuch nachschlagen muß. Oder die *Form* weicht völlig ab. Einige Versionen sind ganz spartanisch und melden nur WHAT? (Was?) oder ähnliches bei einem Fehler. Im Zweifelsfall müssen Sie in Ihrem Handbuch nachschlagen und (eigentlich gilt das immer) etwas experimentieren, bis Sie genau wissen, welcher Fehler wodurch gemeldet wird.

Eines aber haben alle BASIC-Versionen gemeinsam: *Bei derartigen Fehlern wird die Programmabarbeitung abgebrochen.* Das ist in den meisten Fällen durchaus sinnvoll – kann aber manchmal Kopfzerbrechen bereiten, dann nämlich, wenn man den betreffenden Fehler durchaus tolerieren könnte oder Maßnahmen zu seiner Korrektur vorsehen kann. Man muß in einem solchen Fall das Programm so «narrensicher» wie möglich abfassen, d.h. um die Eigenschaften des BASIC-Computers (in diesem Fall die Fehlermeldungen mit Programmabbruch) «herum» programmieren. Das wird uns noch intensiver beschäftigen.

Sprünge im Programm: GOTO

Wir haben so die wichtigsten Kenntnisse zur Lösung unseres Problems, aus einem Tankverzeichnis wie in Tabelle 1 die einzelnen Verbrauchswerte und den Durchschnitt daraus zu errechnen, zusammen. Das beinhaltet die Möglichkeit, den Rechengang dauerhaft als Programm in den Computer eingeben zu können, im Kern also die Formel für den jeweiligen Benzinverbrauch je hundert Kilometer

LET V = B / (S2 − S1) ∗ 100

Weiter können wir die Tabellenwerte bequem in Form von DATA-Anweisungen im Programm festhalten. Und wir haben einen Weg gefunden, sie – mit Hilfe von READ-Anweisungen – gezielt wieder aus der DATA-Liste auszulesen und den jeweils in Frage kommenden Variablen zuzuweisen.

Nehmen wir an, die Liste sei bereits geeignet in DATA-Anweisungen untergebracht und zwar genau in der Form, in der wir sie in Tabelle 1 festgehalten haben: Kilometerstand – getankte Benzinmenge – Kilometerstand – getankte Benzinmenge usw. Diese Information müssen wir geeignet auslesen und den Variablen B, S1 und S2 zuweisen, was etwa so geschehen kann:

```
>NEW
>100 READ S1
>110 READ B
>120 READ S2
```

Daraus können wir einen einzelnen Verbrauchswert berechnen. Damit wir wie gefordert auch die Einzelwerte ausdrucken können, weisen wir den jeweiligen Verbrauchswert einer *Hilfsvariablen* V1 zu:

```
>130 LET V1 = B / (S2 - S1) * 100
```

Diesen Wert lassen wir ausdrucken:

```
>140 PRINT V1
```

Außerdem müssen wir ihn natürlich zum Gesamtverbrauch V aufsummieren, also:

```
>150 LET V = V + V1
```

Und der Computer muß sich merken, wieviele Einzelwerte er bearbeitet hat. Das soll in der Variablen N geschehen, die in jedem Schritt um 1 erhöht wird:

```
>160 N = N + 1
```

N und V müssen natürlich zunächst einmal einen festen Ausgangswert, nämlich den Inhalt Null besitzen. Dazu sind sie am Programmanfang geeignet zu initialisieren (also auf diesen Anfangswert Null zu setzen):

```
>10 LET N = 0
>20 LET V = 0
```

Damit haben wir den Kern unseres Programms bereits zusammen:

```
>LIST
10 LET N = 0
20 LET V = 0
100 READ S1
110 READ B
120 READ S2
130 LET V1 = B / (S2 - S1) * 100
140 PRINT V1
150 LET V = V + V1
160 N = N + 1
```

Es fehlt natürlich noch die Datenliste in den verschiedenen DATA-Anweisungen, z.B. so:

```
1000 DATA 41956
1010 DATA 10
1020 DATA 42090
```

und so weiter.

Für einen einzelnen Verbrauchswert arbeitet das durchaus richtig, wie wir uns rasch überzeugen können:

```
>RUN
 7.46269
READY
>
```

Das ist der erste Verbrauchswert. Und damit ist auch schon Schluß, wie das «READY» beweist.

Wir können auch noch nachprüfen, welche Werte die Variablen N und V erhalten haben:

```
>PRINT N
 1
>PRINT V
 7.46269
```

Doch wie ist das nun mit allen folgenden Berechnungen?

Nach dem, was wir bis jetzt wissen, müßten wir für jeden weiteren Rechenschritt zunächst einmal der Variablen S1 den Wert der Variablen S2 zuweisen (denn in jedem Schritt wird ja der ursprüngliche Endstand des Kilometerzählers zum Anfangsstand der neu durchfahrenen Strecke):

```
>170 LET S1 = S2
```

Und dann müßten wir den ganzen Abschnitt von Zeile 110 bis Zeile 170 noch einmal aufschreiben, nur mit anderen Zeilennummern:

```
180 READ B
190 READ S2
200 LET V1 = B / (S2 - S1) * 100
210 PRINT V1
220 LET V = V + V1
230 N = N + 1
240 LET S1 = S2
```

Für den nächsten Durchschnittswert wäre das zu wiederholen und dann noch einmal und noch einmal... So geht das ganz sicher nicht. Wir brauchen irgendeine Möglichkeit, nach dem Variablenaustausch in Zeile 170 die ganze Anweisungsfolge von Zeile 110 bis Zeile 170 bei jedem Rechengang automatisch in einer Art *Programmschleife* neu durchlaufen zu können.

Sehen wir uns das genauer an:

```
>LIST 110-170
110 READ B
120 READ S2
130 LET V1 = B / (S2 - S1) * 100
140 PRINT V1
150 LET V = V + V1
160 N = N + 1
170 LET S1 = S2
```

Wenn der Computer mit dem Abarbeiten des Programms bis hier gekommen ist, so sucht er nach der Zeile mit der nächsthöheren Befehlsnummer. In unserem Fall wäre das die erste DATA-Anweisung in Zeile 1000. (Zeile 180 und folgende haben Sie ja hoffentlich noch nicht eingetippt. Falls doch, dann müssen Sie sie wieder löschen. Sie wissen ja, wie das geht: Zeilennummer angeben und gleich darauf RETURN drücken.)

Unser Problem ist, daß der Computer bis jetzt immer nur *vorwärts* arbeitet, von niedrigen Zeilennummern hin zu höheren. Wir brauchen aber etwas anderes, nämlich nach Zeile 170 einen *Sprung zurück zu Zeile 110.* Mit anderen Worten: Es fehlt uns eine Anweisung nach Zeile 170, die besagt: «Gehe zu Zeile 110 und mache dort mit der Programmabarbeitung weiter.»

Das Problem, im Programm *verzweigen* zu können, ist so allgemein, daß alle BASIC-Versionen eine solche *Sprunganweisung* «Gehe zu (Zeile) xxx» besitzen. Auf englisch ergibt das den Befehl

GOTO

(Beachten Sie, daß hier die beiden Worte GO TO (gehe zu) zu dem *einen* Wort GOTO zusammengezogen sind!)

Als Befehlsargument dient die Zeilennummer, zu der gesprungen werden soll. In unserem Fall ergäbe das die Anweisung

GOTO 110

Dabei unterliegt das Argument strengeren Bedingungen als sonst üblich: *Die angegebene Zeile muß mit Sicherheit existieren.*

Würden wir die Nummer aus einem numerischen Ausdruck berechnen lassen, so könnte infolge von Rundungsfehlern auch 109 oder 111 herauskommen. Und wehe, wenn in diesen Zeilen irgendwelche anderen Befehle stehen! Wir könnten nur unter Einhaltung striktester Regeln einen wiederholbaren Programmlauf gewährleisten, was die Fehleranfälligkeit des Programmentwurfs natürlich erhöht. Aus diesem Grund verlangen die meisten BASIC-Versionen bei Sprungbefehlen der Art von GOTO von vornherein die ausdrückliche Angabe der Nummer der Zielzeile im Argument.

Probieren wir das Ganze einmal aus:

```
>180 GOTO 110
>RUN
 7.46269
 9.80392
 10.3093
 8.99281
 8.81057
 10.6952
 12.1951
 9.17431
 9.95025
?OD ERROR IN 120
```

Das sind alle unsere Einzelverbrauchswerte, wie sie sich aus der Tabelle ergeben. Der Computer hat unser BASIC-Programm genau so abgearbeitet, wie wir es erwartet hatten – bis ans Ende der DATA-Liste. Dort hat er keinen neuen Kilometerstand vorgefunden und folglich einen Fehler gemeldet. Überzeugen Sie sich davon, daß dieser Fehler wirklich beim Versuch der Übernahme eines Werts für S2 aufgetreten ist.

Wir können uns den Durchschnittsverbrauch einfach mittels

```
>PRINT V/N
 9.71046
```

ausdrucken lassen, womit unser Problem – fast – gelöst ist.

Entscheidungen treffen: IF ... THEN

Aber eben nur «fast». Unsere Forderung lautet, daß der Computer den Durchschnittsverbrauch *ohne* weiteren Eingriff in den Rechengang ermittelt. Im Idealfall müßte jemand ohne alle Computerkenntnisse diese Rechnung durchführen können. Wir sagen ihm lediglich, wie er die Daten in den Rechner bekommt – und dann muß auf RUN alles weitere von selbst geschehen. Irgendwelches Verständnis für Fehlermeldungen der Art von ?OD ERROR können wir bei einem Computerlaien beim besten Willen nicht voraussetzen.

Wir brauchen also noch mehr Möglichkeiten. Der Rechner muß im Programm das Ende der Datenliste erkennen können und daraufhin den Durchschnittswert ausdrucken. Das wirft zwei Fragen auf:

1. Wie soll man das Tabellende markieren?
2. Wie kommt der Rechner am Tabellenende aus der Programmschleife zur Berechnung der Einzelwerte heraus?

Die Markierung des Tabellendes ist relativ einfach. Da man die Tabelle nicht, wie wir es getan haben, «offen» lassen kann (das führt ja gerade zur Fehlermeldung ?OD ERROR), fügen wir einfach einen besonderen «Endwert» hinzu, d.h. irgendwelche Daten, die in dieser Form in der Tabelle nicht auftauchen können. Das kann z.B. eine negative Zahl in einer Tabelle mit lauter positiven Werten sein, oder man verwendet einfach eine Null an einer Stelle, die sinnvollerweise nicht Null sein kann, oder es wird sonst eine passende Markierung eingesetzt. Was man verwendet, hängt immer vom Tabellenaufbau im gegebenen Fall ab. Markieren wir hier das Tabellende einfach dadurch, daß wir als nächsten Kilometerstand eine Null angeben. Wenn unsere Datenliste z.B. so aufhört:

```
>LIST 1170-
1170 DATA 20
1180 DATA 43465
1190 DATA 15
```

so fügen wir einfach als letzten Schritt an:

```
1200 DATA 0
```

Das eigentliche Problem kommt jetzt erst. Wie soll der Computer erkennen können, daß das, was er gerade aus der Datenliste übernommen hat, *kein* für die Berechnung dienender Wert mehr ist? Und wie bekommen wir ihn dazu – vorausgesetzt, er hat den Wert Null erkannt – nicht mehr mit der Schleife zur Berechnung der einzelnen Verbrauchswerte weiterzumachen, sondern den Durchschnittsverbrauch auszudrucken und dann anzuhalten?

Packen wir die Sache von hinten an. Da wäre als letzte Operation der Durchschnittsverbrauch auszurechnen und auszudrucken. Das läßt sich durch eine einfache PRINT-Anweisung erledigen, die wir am besten im eigentlichen Verarbeitungsteil unseres Programms folgen lassen:

```
>190 PRINT V/N
```

Nach dieser Anweisung folgt nur noch die DATA-Liste. Aus unseren Experimenten wissen wir, daß der Rechner hier mit der Programmabarbeitung aufhört. Damit wäre der letzte Teil der Aufgabe gelöst.

Gehen wir zum vorletzten Problemteil über. Wie kommt der Rechner von der Übernahme des S2-Werts Null zur Berechnung des Durchschnittsverbrauchs in Zeile 190? Die Antwort ist einfach: Durch einen Sprungbefehl GOTO 190, den wir geeignet in die Programmschleife einbauen müssen.

Um dieses «geeignet» rankt sich unser eigentliches Problem. Denn dieser Sprung zu Zeile 190 darf ja nur erfolgen, *wenn* die Variable S2 den Wert Null übernommen hat. Im Gegensatz zu der Anweisung in Zeile 180 darf hier nur gesprungen werden, wenn eine bestimmte *Bedingung* erfüllt ist. Diese Bedingung muß vom Computer bei der Programmabarbeitung *getestet* werden. Ist sie erfüllt, so soll er springen. Ist sie nicht erfüllt, dann ist mit dem nächsten Befehl in der Schleife weiterzumachen.

Wir brauchen also sinngemäß folgende Anweisung:

«WENN S2 = 0 ist, DANN springe zu Zeile 190, sonst mache mit dem nächsten Programmbefehl weiter.»

Nun würde der Rechner unter normalen Bedingungen ohnehin mit dem nächsten Programmbefehl weitermachen, so daß sich das verkürzen läßt zu:

«WENN S2 = 0 ist, DANN springe zu Zeile 190.»

WENN ... DANN läßt sich im Englischen durch

IF ... THEN

ausdrücken, und so findet man denn in fast allen BASIC-Versionen diese beiden Befehlsworte IF und THEN. Damit können wir unser Problem durch Einfügen eines *bedingten Sprungs* in das Programm lösen. Diese Anweisung muß logischerweise unmittelbar auf die Übernahme des Werts für die Variable S2 aus der Datenliste folgen. Setzen wir sie in Zeile 125:

```
>125 IF S2 = 0 THEN GOTO 190
```

Das sollte folgendes bewirken. Der Rechner sieht sich den Wert der Variablen S2 an. Wenn (IF) er gleich Null (S2 = 0) ist, dann (THEN) springt er zu Zeile 190 (GOTO 190). Und wenn nicht, so möge die Abarbeitung mit der unmittelbar folgenden (Zeile (130) fortgesetzt werden.

Sehen wir uns an, ob das stimmt. (Wir unterschlagen der Übersichtlichkeit halber ein paar Einzelwerte.)

```
>RUN
 7.46269
 . . .
 9.17431
 9.95025
 9.71046
```

Vergleichen Sie das mit unseren bisherigen Ergebnissen: Es funktioniert!

Etwas zum Druckbild: PRINT-Argumente

Wir haben damit zwar alle Forderungen erfüllt, die wir zu Beginn an die Verbrauchsberechnung gestellt hatten, doch irgendwie befriedigt das Ergebnis noch nicht so richtig. Gewiß, wir können die einzelnen Verbrauchswerte berechnen lassen, der Rechner ermittelt auch den gewünschten Durchschnittsverbrauch und das Ganze praktisch «auf Knopfdruck», ohne daß außer der Angabe der Datenliste in DATA-Anweisungen (und dem RUN-Kommando) weitere Eingriffe notwendig sind. Was nicht so richtig passen will, ist die *Form der Ausgabe.*

Da ist zum Beispiel die Tatsache, daß der Durchschnittsverbrauch an die Einzelwerte «angeklebt» ist. Besser wäre es, diese Angabe optisch durch eine *Leerzeile* von der Liste der Einzelwerte zu trennen.

So etwas ist in BASIC gar nicht so schwierig zu realisieren. Betrachten wir, was beim normalen Abarbeiten eines PRINT-Befehls geschieht: Die Zahl (in unserem Fall) wird gedruckt, und dann fährt der Wagen (oder der Kursor) in die nächste Zeile. Also müßte es doch genügen, wenn *nichts* ausgedruckt wird, so daß als einzige Operation nur das Anfahren der nächsten Zeile übrig bleibt. Das ist durchaus sinnvoll (wenngleich andere Programmiersprachen das Problem anders lösen können), und die meisten BASIC-Versionen bieten denn auch die Möglichkeit, den PRINT-Befehl ohne ausdrücklich angegebenes Argument zu verwenden.

In unserem Fall könnte das so aussehen: Wir schreiben den Programmteil für den Durchschnittsausdruck etwas um, so daß vor der Ausgabe des Verbrauchsdurchschnitt eine Leerzeile ausgedruckt wird. Dazu schieben wir einfach die bisherige Zeile 190 auf die Position 200 und geben als neue Zielzeile für den bedingten Sprung eine «leere» PRINT-Anweisung an:

```
>200 PRINT V/N
>190 PRINT
```

(Das ersetzt die alte Zeile 190 mit einer neuen Anweisung.)

Wenn wir das ausprobieren, erhalten wir (wieder ein paar Einzelwerte unterschlagen):

```
>RUN
 7.46269
  .  .  .
 9.17431
 9.95025

 9.71046
```

Jetzt kann man den Durchschnitt auf den ersten Blick von den übrigen Werten unterscheiden.

(Es gibt jedoch einige wenige BASIC-Versionen, die einen PRINT-Befehl ohne Argument einfach unterschlagen. In diesem Fall muß man irgendein «Hilfsargument» einsetzen. Eine praktische Möglichkeit dazu werden wir im nächsten Abschnitt betrachten.)

Was aber, wenn wir die Berechnung in ein paar Monaten wieder in die Finger bekommen? Wie sollen wir da noch wissen, was sich worauf bezieht? Was ist das, was da steht? Wenn wir auch noch unsere Originaltabelle (Tabelle 1) verloren haben, dann ist das im Grunde nicht viel mehr als ein bedruckter Fetzen Papier. Wenn wir doch wenigstens die übrigen Tabellenwerte – in eigenen Spalten – mit ausdrucken lassen könnten. Und wenn dann auch noch die jeweils verfahrene Strecke ausgedruckt würde, wäre das bestimmt auch nicht schlecht.

Das Problem ist, die einzelnen Werte in getrennten *Spalten* ausdrucken zu lassen. So wäre für unser Problem eine Aufteilung wie folgt angebracht:

Kilometerstand getankt gefahren Verbrauch

Die dazu notwendigen Daten stehen alle zur Verfügung bzw. können einfach berechnet werden. Warum sollte man sie nicht mit angeben? Die Frage ist nur, ob das möglich ist, und wenn ja, wie man so etwas macht.

Nun – einen Hinweis auf die Lösung dieses Problems haben

wir bereits erhalten. Erinnern Sie sich noch an unseren Versuch, die Zahl 1,2 auszudrucken?

```
>PRINT 1,2
 1             2
```

Das ist nicht «1,2». Das sind *zwei* Argumente, «1» und «2», durch einen Zwischenraum von 13 Zeichen getrennt. Dieser Zwischenraum hat seine Bedeutung. Er ist gerade so groß, daß eine Zahl in wissenschaftlicher Notation (einschließlich Vorzeichen) angegeben werden kann. Offenbar bewirkt das Komma diese Spaltenaufteilung. Probieren wir es aus:

```
>PRINT -0.001/3,-0.001/3,-0.001/3
-3.33333E-04  -3.33333E-04  -3.33333E-04
```

Wenn Sie nachzählen: Jede Spalte nimmt einen Raum von genau 14 Stellen ein. Das reicht für die Zahldarstellung selbst und einen «Sicherheitsabstand» von zwei Leerschritten. (Dieser Spaltenabstand ist von BASIC-Version zu BASIC-Version verschieden definiert. Er hängt natürlich von der Darstellungsgenauigkeit der Zahlen ab: Wenn Ihre BASIC-Version mehr Ziffern wiedergibt dann werden wahrscheinlich auch die Druckspalten breiter sein. Andere Versionen orientieren sich an der Zahl von Zeichen, die üblicherweise in einer Zeile unterzubringen sind. Da das oft Mehrfache von 16 sind [bei Zeilenbreiten von 64, 80 oder 96 Zeichen], verwenden viele Versionen eine Spaltenbreite von 16 Zeichen.)

Unser Experiment hat noch mehr ergeben: Wir können mehr als ein Argument an den PRINT-Befehl anfügen, wenn wir die verschiedenen Angaben nur geeignet voneinander unterscheiden. In unserem Fall dienten Kommas als Trennungsmarkierungen. Genau betrachtet stellen diese Kommas wieder eigene Befehle dar, die besagen:

«Das nächste Argument ist in der nächsten Spalte zu drukken.»

Und wenn kein Argument folgt? Machen wir einen Versuch:

```
>PRINT 1,,2
 1                              2
```

Offensichtlich bewirkt das Komma im PRINT-Befehl nicht unmittelbar den Ausdruck eines Arguments in der jeweils folgenden Spalte. Sehen wir uns dazu an, was geschieht, wenn einem Komma überhaupt kein Argument vorangeht:

```
>PRINT ,2
               2
```

(Einige ältere BASIC-Versionen gestatten die Angabe eines Kommas ohne vorheriges Druckargument nicht. Sollte das bei Ihnen der Fall sein, verzweifeln Sie nicht. Im nächsten Abschnitt werden wir eine Möglichkeit besprechen, wie man dieses Manko umgehen kann.)

Wir können ein Komma im PRINT-Befehl folglich als eine Art Unterbefehl ansehen, durch welchen der Wagen oder Kursor in die jeweils nächste Spalte gesetzt wird. Ab dieser Position erfolgt dann der Ausdruck des PRINT-Arguments wie sonst auch.

Das können wir uns zunutze machen. Doch noch ist eine Frage zur Rolle des Kommas offen: Was geschieht, wenn das Komma am Ende des PRINT-Befehls steht?

Was wird geschehen? Bei Kommandos, also direkt abgearbeiteten Befehlen, ist kein Unterschied zum sonstigen Verhalten festzustellen:

```
>PRINT 2,
 2
READY
```

Das liegt daran, daß für die «READY»-Meldung auf alle Fälle eine neue Zeile angefangen wird. Wie verhält es sich aber mit Anweisungen? Setzen wir zum Test unserem Programm einfach eine Zeile voran:

```
>1 PRINT 2,
 RUN
 2             7.46269
 9.80392
 10.3093
 . . .
```

Was ist geschehen? Nun – offensichtlich hat der Befehl in Zeile 1 nach Ausdrucken des zugehörigen Arguments «2» den Wagen auf die zweite Spalte gesetzt, wie es das Komma vorschreibt. *Und dort blieb er stehen.* Der nächste Druckbefehl setzte nämlich das Argument in *dieselbe* Zeile, nur eben eine Spalte weiter – obwohl die Anweisung in Zeile 1 längst abgearbeitet war.

Das ist eine wichtige Erweiterung der Möglichkeiten des PRINT-Befehls: Man kann das Anfahren einer neuen Zeile verhindern, indem man als letztes Argument einer PRINT-Anweisung ein Komma einsetzt. Unangenehm ist in einigen Fällen eigentlich nur, daß dabei gleich eine neue Spalte angefahren wird. Man müßte den Wagen auch einfach dort stehen lassen können, wohin er beim Ausdrucken des letzten PRINT-Arguments gelangt ist.

Diese Möglichkeit ist in vielen Anwendungsfällen bedeutungsvoll, so daß sie sich auch in BASIC findet. Wie ein Komma den Druckwagen oder Bildschirmkursor auf eine neue Spalte setzt und ihn dort stehen läßt, gestattet ein *Strichpunkt* es, den Wagen dort stehen zu lassen, wo er gerade angelangt ist. Insbesondere wird dadurch das Anfahren einer neuen Zeile verhindert. Probieren wir das an unserem letzten Beispiel aus:

```
>1 PRINT 2;
 RUN
 2 7.46269
 9.80392
 10.3093
 . . .
```

Der Strichpunkt hat in einem PRINT-Befehl noch eine weitere wichtige Funktion. Wir können nämlich mehrere PRINT-Argu-

mente nicht ohne weitere Maßnahmen hintereinander aufreihen:

```
>PRINT 1 2
 1
?SN ERROR
```

Das führt zu einer Fehlermeldung (SN ERROR = Syntaxfehler, d.h. Verletzung der Regeln des Sprachaufbaus): Der Computer suchte nach dem Ausdrucken des ersten Arguments («1») nach einem neuen Befehl, fand dort aber nur eine Zahl vor, mit der er nichts anzufangen wußte. Setzen wir dagegen einen Strichpunkt zwischen die beiden Argumente:

```
>PRINT 1;2
 1 2
```

so werden sie wie gewünscht unmittelbar hintereinander ausgedruckt.

Insgesamt haben wir also folgende Zusatzmöglichkeiten zur Verwendung des PRINT-Befehls gefunden:

- Wird der PRINT-Befehl ohne Argument angegeben, so wird eine neue Zeile angefahren.
- Ein Strichpunkt trennt mehrere unmittelbar hintereinander zu druckende Argumente voneinander und bewirkt, daß der Wagen – auch am Befehlsende – an der erreichten Stelle stehen bleibt.
- Ein Komma trennt mehrere Befehle voneinander, die in aufeinanderfolgenden Spalten von (in unserem Fall) 14 Zeichen Breite ausgedruckt werden sollen. Am Befehlsende eingesetzt, stellt es den Wagen auf die nächste Druckspalte und unterdrückt das Anfahren einer neuen Zeile.

Von BASIC-Version zu BASIC-Version verschieden ist allerdings das Verhalten, wenn beim Anfahren oder Auffüllen einer Spalte die Schirm- oder Walzenbreite überschritten wird. Die meisten Versionen beginnen in diesem Fall einfach eine neue Zeile und beenden dort gegebenenfalls das angefangene PRINT-Argument.

Andere Versionen unterschlagen das nicht in den Rahmen passende Argumentende, und wieder andere drucken ohne Rücksicht auf Verluste weiter – und wenn es auf derselben Stelle ist. Machen Sie sich damit vertraut, wie sich Ihre Version in so einem Fall verhält. Wir werden immer annehmen, daß ein angefangenes Argument auf einer neuen Zeile fertiggestellt und danach bei Bedarf (d.h. bei folgendem Komma) in der *nächsten* Druckzeile die nächste Spalte angefahren wird.

Alles das können wir uns zunutze machen, um unsere Verbrauchstabelle übersichtlich mit mehr Inhalt zu versehen. Löschen wir zunächst die Testzeile von vorhin:

```
>1
```

und führen dann den Ausdruck aller Einzeldaten für einen Rechengang durch, sobald alle Informationen bereitstehen. Das ist nach Zeile 160 der Fall. Fügen wir das als Zeile 165 ein:

```
>165 PRINT S1,B,S2-S1,V1
```

Natürlich müssen wir dann den Ausdruck der Einzelergebnisse in Zeile 140 löschen:

```
>140
```

Außerdem sollten wir der Übersichtlichkeit halber den Durchschnittsverbrauch als Abschluß der Spalte der Einzelverbrauchswerte angeben. Dazu müssen wir Zeile 200 so ändern, daß vor Ausdruck von V/N der Wagen auf die vierte Spalte gesetzt wird, d.h., wir müssen dem Argument die richtige Zahl von Kommas voransetzen. Ein einziges Komma am Argumentanfang setzt den Wagen auf die *zweite* Druckspalte. Damit benötigt man zum Anfahren der N-ten Druckspalte N-1 Kommas, für die vierte Spalte also drei Stück:

```
>200 PRINT ,,,V/N
```

```
41956        10          134         7.46269
42090        15          153         9.80392
42243        10          97          10.3093
42340        25          278         8.99281
42618        20          227         8.81057
42845        20          187         10.6952
43032        15          123         12.1951
43155        10          109         9.17431
43264        20          201         9.95025

                                     9.71046
```

Bild 1 Eine erste Verbrauchsauswertung

Wie Bild 1 zeigt, ergibt das genau die gewünschte Form des Ausdrucks. (Im übrigen hätten wir diese Anweisung auch nach Zeile 150, *nicht* aber nach Zeile 170 einfügen können. Da S2 = S1 gesetzt wird, würden wir dort als Entfernungsangabe S2 – S1 immer den Wert Null erhalten.)

Noch mehr zum Druckbild: PRINT «Text»

So schön unser Druckergebnis von Bild 2 auch sein mag, es fehlt immer noch etwas: Was ist das, was die einzelnen Zahlenkolonnen bezeichnen? Gewiß, wir könnten das alles im nachhinein von Hand mit «Kilometerstand», «getankt» usw. beschriften, doch – warum sollen eigentlich wir diese Arbeit machen? Wenn der Computer schon etwas ausdrucken kann, warum sollen es immer nur Zahlen sein? Warum nicht auch Text ausdrucken?

Das ist aber gar nicht so einfach, denn mit unseren bis jetzt erarbeiteten Hilfsmitteln können wir nur Zahlen und nichts als Zahlen ausdrucken lassen. Mit dem Befehl

```
>PRINT Kilometerstand
```

erhalten wir nämlich je nach BASIC-Version bestenfalls als Antwort eine Reihe von Nullen oder Fehlermeldungen der Form:

```
?SN ERROR
```

```
10 LET N = 0
20 LET V = 0
50 PRINT "Kilometer-","getankt","gefahren","Verbrauch"
60 PRINT "stand","(l)","(km)","(l/100km)"
70 PRINT
100 READ S1
110 READ B
120 READ S2
125 IF S2 = 0 THEN GOTO 190
130 LET V1 = B / (S2 - S1) * 100
150 LET V = V + V1
160 LET N = N + 1
165 PRINT S1,B,S2-S1,V1
170 LET S1 = S2
180 GOTO 110
190 PRINT
200 PRINT "Durchschnitt",,,V/N
1000 DATA 41956
1010 DATA 10
1020 DATA 42090
1030 DATA 15
1040 DATA 42243
1050 DATA 10
1060 DATA 42340
1070 DATA 25
1080 DATA 42618
1090 DATA 20
1100 DATA 42845
1110 DATA 20
1120 DATA 43032
1130 DATA 15
1140 DATA 43155
1150 DATA 10
1160 DATA 43264
1170 DATA 20
1180 DATA 43465
1190 DATA 15
1200 DATA 0
```

Bild 2 Ein Programm zur Verbrauchsauswertung

In dieser Situation bieten sich mehrere Auswege an.
Entweder vereinbart man einen neuen Befehl, der besagt:

«Nimm das folgende Argument als wörtlich auszudruckenden Text!»

Oder man bevorzugt – wie wir es schon öfter getan haben – die Angabe im Befehlsargument. Dabei ist die Forderung an den PRINT-Befehl im Grunde dieselbe wie bei der ersten Alternative: Das Argument soll als wörtlich auszudruckender Text behandelt werden. Nur soll jetzt die Information dazu nicht im Befehlswort, sondern im Argument selbst stecken.

«Wörtlich» ist der Schlüssel: Wenn man etwas wörtlich wiedergeben, etwas zitieren will, dann wird es (auch im Englischen) in Anführungsstriche eingeschlossen. Und da BASIC sich an der Umgangssprache orientiert, sollten wir das einmal ausprobieren:

```
>PRINT "Kilometerstand"
Kilometerstand
READY
>
```

Genau diese Möglichkeit benötigen wir, um unsere Tabelle zu vervollständigen. Wir drucken einfach Spalte für Spalte eine Überschriftszeile über die Zahlenkolonnen – wenn das spaltenweise Ausdrucken bei Textargumenten funktioniert. Machen wir einen Test:

```
>PRINT "Kilometerstand","getankt","gefahren"
Kilometerstand              getankt       gefahren
```

Das paßt noch nicht ganz. Unsere Tabelle hat eine etwas andere Aufteilung:

```
Kilometerstand              getankt       gefahren
 41956          10           134           7.46269
```

Der Grund für diese mangelhafte Übereinstimmung liegt daran, daß «Kilometerstand» mit 14 Zeichen Breite genau eine Spalte einnimmt. Der Wagen steht für das nächste Druckzeichen

bereits in der zweiten Spalte, und auf *diese* Stellung bezieht sich das Komma: Der Wagen wird in die dritte Spalte gestellt. Es ist daher wichtig, beim Einsatz eines Kommas zur Spaltenaufteilung von Texten genau auf die Wortbreite zu achten: Füllt ein Text die ganze Spalte aus, dann wird durch das Komma nicht die nächste, sondern die *übernächste* Druckspalte angelaufen.

Am einfachsten umgehen wir die Schwierigkeit, wenn wir die Spaltenüberschriften auf zwei Zeilen aufteilen. Diese Überschriften sollten vor Eintritt in die Rechnung bereits ausgedruckt sein. Programmieren wir die zugehörigen Anweisungen in Zeile 50 und 60:

```
>50 PRINT "Kilometer-","getankt","gefahren","Verbrauch"
>60 PRINT "stand","(l)","km","(l/100km)"
```

und fügen wir noch eine Leerzeile ein:

```
>70 PRINT
```

Zu guter Letzt sollten wir noch vermerken lassen, daß es sich bei der letzten Druckzeile um die Angabe des Durchschnittsverbrauchs handelt. Ändern wir daher Zeile 200 ab in:

```
>200 PRINT "Durchschnitt",,,V/N
```

Bild 2 zeigt das so entstandene Programm in voller Länge, und Bild 3 gibt die damit erhaltene Tabelle wieder, mit der man im großen und ganzen zufrieden sein kann.

Die Möglichkeit, Text ausdrucken zu lassen, kann im übrigen ein Problem aus dem vorigen Abschnitt überwinden helfen. Wir hatten dort angemerkt, daß einige BASIC-Versionen eine PRINT-Anweisung ohne Argument schlichtweg ignorieren, anstatt eine neue Zeile anzulaufen, oder daß andere Versionen keine Kommas ohne vorangehendes Argument erlauben. Dem läßt sich einfach begegnen, wenn man einen «leeren» Text als Druckargument einsetzt, wenn man beispielsweise einen einfachen Leerschritt

Kilometer-stand	getankt (l)	gefahren (km)	Verbrauch (l/100km)
41956	10	134	7.46269
42090	15	153	9.80392
42243	10	97	10.3093
42340	25	278	8.99281
42618	20	227	8.81057
42845	20	187	10.6952
43032	15	123	12.1951
43155	10	109	9.17431
43264	20	201	9.95025
Durchschnitt			9.71046

Bild 3 Die endgültige Verbrauchsauswertung

ausgeben läßt. Die Anweisung in Zeile 70 in unserem Programm würde so z.B. die Form

```
70 PRINT " "
```

erhalten, ebenso Zeile 190, und Zeile 200 müßte dann lauten:

```
200 PRINT "Durchschnitt"," "," ",V/N
```

Einige Vereinfachungen

So wie unser Programm jetzt steht, ist es eigentlich fertig: Wir erhalten das gewünschte Ergebnis in einer brauchbaren Form und mit vertretbarem Aufwand. Man hat jedoch im Laufe der Zeit einige Vereinfachungen bei der Angabe von BASIC-Programmen geschaffen, welche die Schreibarbeit verringern und das Programm kompakter abzufassen gestatten. Man begegnet ihnen in fast allen modernen BASIC-Programmen, so daß wir sie hier etwas näher betrachten sollten.

LET

Da ist zunächst einmal die Beobachtung, daß die *Zuweisung* eines Werts an eine Variable durch den Befehl LET die bei weitem häufigste Operation ist, die in einem Programm durchgeführt wird. Man hat daher vereinbart, daß in solchen Fällen (allerdings auch nur in diesen!) das Befehlswort *entfallen* kann. Zum Beispiel erhält Zeile 150 unseres Programms statt

```
>LIST 150
150 LET V = V + V1
```

jetzt die Form

```
>150 V = V + V1
```

Durch dieses Entfallen des Befehlsworts LET können kaum Mißverständnisse auftreten. Die Form A = B bedeutet immer eine *Zuweisung* des Werts von B an die Variable A, wenn die Anweisung eigenständig ist. Sie bezeichnet immer dann eine *Vergleichsoperation* «A ist gleich B», wenn sie im Rahmen von IF-Anweisungen auftritt:

IF A = B THEN...

heißt

WENN A gleich B ist, DANN...

Dieser Unterschied geht einem rasch in Fleisch und Blut über, und man zieht die Kompaktkeit durch das Entfallen von «LET» bald der Exaktheit der Befehlsangabe und der (unwesentlichen) Verlängerung der Befehlsabarbeitungszeit in einigen BASIC-Versionen vor.

Bedingte Sprünge

Eine ähnlich häufige Konstruktion ist der *bedingte Sprung* der Art

IF ... THEN GOTO nnn

wobei «nnn» irgendeine Zeilennummer im Programm angibt. Man hat hier vereinbart, in einem solchen Fall das «GOTO» einfach entfallen zu lassen, was die Anweisung verkürzt zu

IF... THEN nnn

Das wirkt sich in unserem Programm in Zeile 125 aus, die jetzt die Form

```
>125 IF S2 = 0 THEN 190
```

erhält. Auch hier ist eigentlich sofort ersichtlich, was gemeint ist. Einige BASIC-Versionen gestatten im übrigen auch Anweisungen der Form

IF ... GOTO nnn

die in ihrer Wirkung der eben betrachteten voll entsprechen. Probieren Sie an einem einfachen Beispiel aus, welche Möglichkeit Ihre BASIC-Version bietet. (In vielen Fällen werden Sie beide Angaben gleichberechtigt nebeneinander einsetzen können.)

Mehrfachargumente

Ähnlich wie man in einem PRINT-Befehl mehrere Argumente durch Einsatz passender Trennoperatoren in einer Anweisung zusammenfassen kann, gestatten einige BASIC-Versionen, insbesondere die moderneren unter ihnen, die Angabe mehrere Argumente beim READ-Befehl. Dabei werden die einzelnen Argumente einfach durch Kommas getrennt. In unserem Programmbeispiel können wir das in Zeile 110 ausnutzen:

```
>110 READ B, S2
```

Das übernimmt wie bisher zuerst den Wert für B und darauf den Wert für S2 aus der DATA-Liste, wobei wir natürlich die separate Übernahme von S2 in Zeile 120 wieder streichen müssen:

```
>120
```

Beachten Sie, daß wir hier nicht alle drei READ-Operationen zu einer Zeile zusammenfassen können. Zeile 110

```
110 READ S1
```

dient ja nur zur Vorgabe eines Anfangswerts für die Schleife zur Berechnung der Einzelwerte. S1 erhält seinen jeweils sonst gültigen Wert in Zeile 170 durch die Zuweisung

```
>170 S1 = S2
```

Diese Zusammenfassung mehrerer READ-Argumente in einer Anweisung ist allerdings nicht in allen BASIC-Versionen erlaubt. Probieren Sie aus, ob es in Ihrer Version möglich ist. Wenn nicht, so schlagen Sie am besten in Ihrem Handbuch nach, ob vielleicht eine andere Schreibweise des Trennungsoperators vorgeschrieben ist.

Mehrfachbefehle

Eine letzte Vereinfachung besteht in der Möglichkeit, mehrere Befehle in einer Zeile zusammenzufassen. So etwas vereinfacht nicht nur die Schreibarbeit, es kann – überlegt eingesetzt – auch die Lesbarkeit verbessern.

Allerdings kann man nicht ohne weiteres zwei Befehle in einer Zeile zusammenfassen. Z.B. führt

```
>10 N = 0    V = 0
```

zu einer Fehlermeldung:

```
>RUN
?SN ERROR IN 10
```

Der Grund liegt darin, daß die einzelnen BASIC-Befehle eindeutig voneinander unterscheidbar sein müssen. Man braucht auch hier besondere Trennungsoperatoren. In der Regel verwendet man hierzu einen *Doppelpunkt,* einige BASIC-Versionen

schreiben auch einen *umgekehrten Schrägstrich* (englisch: backslash) vor. Wir bleiben hier bei der häufiger anzutreffenden Doppelpunkt-Version, mit der Zeile 10 die Form

```
>10 N = 0 : V = 0
```

erhält. (Untersuchen Sie Ihre Version daraufhin, ob Mehrfachanweisungen möglich sind und welche Form das Trennungssymbol gegebenenfalls haben muß.)

Mehrfachbefehle sind besonders im Fall von Kommandos nützlich. Man kann mit ihrer Hilfe recht einfach komplexe Konstruktionen mit einer einzigen Zeile untersuchen. Es gibt Fälle, in denen dies der einzige Weg ist, von der direkten Befehlsabarbeitung Gebrauch zu machen.

In Programmen muß man beim Einsatz von Mehrfachbefehlen allerdings Vorsicht walten lassen. Wenn man auf-Teufel-komm-raus durch Zusammenfassen in Mehrfachbefehlszeilen aus dem Programm alle nur erdenkbare «Luft» herausläßt, ist das Ergebnis zumeist ungenießbar. Man kann nur noch schwer verstehen, was da vorgeht. Und Änderungen können ganz beträchtlichen Aufwand machen.

Man findet diese Form hin und wieder in Programmen, die für Rechner mit geringen Speichermöglichkeiten geschrieben sind. Hier wird von der Tatsache Gebrauch gemacht, daß die Zeilennummern selbst Platz im Rechner benötigen, was man durch Komprimieren des Programms in Mehrfachbefehlszeilen umgehen kann. In den allermeisten Fällen sind es aber nur schlechte Gewohnheiten, die zu einem solchen Programmstil führen, weniger der aktuelle Platzbedarf.

Sinnvoll wird die Zusammenfassung dort, wo sie die *Struktur des Programms* hervorhebt. Z.B. können wir, wie oben bereits geschehen, die Initialisierung von N und V in einer Zeile zusammenfassen. Allerdings sollte dann die zusätzliche Initialisierung von V in Zeile 20 gestrichen werden:

```
>20
```

Sinnvoll wäre es auch, den Ausdruck der Leerzeile und des Durchschnittverbrauchs zu einer Einheit zusammenzufassen:

```
>190 PRINT : PRINT "Durchschnitt",,,V/N
```

wobei Zeile 200 gelöscht werden muß:

```
>200
```

Möglich, aber nicht sehr übersichtlich, wäre auch die Zusammenfassung von Zeile 130 und 150. Die ursprüngliche Fassung ist jedoch klarer.

Gar nicht sinnvoll wären dagegen Zusammenfassungen der Form

```
165 PRINT S1,B,S2-S1,V1 : S1 = S2
```

die den Befehl «S1 = S2» als zum PRINT-Argument gehörig erscheinen lassen. So etwas führt leicht zu Verwechslungen.

Wie bei allen Gestaltungsmöglichkeiten zur Abfassung eines Programms sollte auch hier immer auf möglichst klare Gliederung und eindeutige Aussage geachtet werden. Denken Sie daran, daß Sie Ihr Programm auch in einem Jahr noch verstehen müssen, oder daß Außenstehende mit Ihrem Programm umgehen, es warten und weiterentwickeln müssen. Erschweren Sie sich und anderen nicht unnötig die Arbeit durch allzu vordergründiges Feilschen um Geschwindigkeit und Platz. Bedenken Sie, daß der Aufwand zum Verstehen und Warten eines Programms oft sehr viel größer wird, wenn der Programmtext unübersichtlich abgefaßt ist. *Fassen Sie nur zusammen, was logisch zusammengehört.* Und wenn Sie wirklich einmal das letzte bißchen Speicherbedarf herausquetschen müssen, dann sorgen Sie durch eine passende *Begleitdokumentation* dafür, daß das Ganze verstehbar bleibt. Ein Programm, das sich weitgehend selbst dokumentiert, ist jedoch auf alle Fälle besser als eine noch so gute Begleitdokumentation, die erst mühsam mit dem Programm zur Deckung gebracht werden muß.

```
10 N = 0 : V = 0
50 PRINT "Kilometer-","getankt","gefahren","Verbrauch"
60 PRINT "stand","(l)","(km)","(l/100km)"
70 PRINT
100 READ S1
110 READ B, S2
125 IF S2 = 0 THEN 190
130 V1 = B / (S2 - S1) * 100
150 V = V + V1
160 N = N + 1
165 PRINT S1,B,S2-S1,V1
170 S1 = S2
180 GOTO 110
190 PRINT : PRINT "Durchschnitt",,,V/N
1000 DATA 41956
1010 DATA 10
1020 DATA 42090
1030 DATA 15
1040 DATA 42243
1050 DATA 10
1060 DATA 42340
1070 DATA 25
1080 DATA 42618
1090 DATA 20
1100 DATA 42845
1110 DATA 20
1120 DATA 43032
1130 DATA 15
1140 DATA 43155
1150 DATA 10
1160 DATA 43264
1170 DATA 20
1180 DATA 43465
1190 DATA 15
1200 DATA 0
```

Bild 4 Das endgültige Programm zur Verbrauchsauswertung

Wir werden auf die Frage des Programmierstils noch oft zurückkommen. Versuchen Sie, sich gleich von vornherein einen sauberen, übersichtlichen Stil anzueignen. Schlechte Gewohnheiten sind immer leichter von vornherein zu vermeiden, als später gezwungenermaßen abzulegen.

Wenn wir noch die verbliebenen LET-Befehlsworte streichen,

dann kommen wir zu dem in Bild 4 wiedergegebenen Programm. Überzeugen Sie sich davon, daß es wirklich dasselbe Ergebnis wie die Version von Bild 2 liefert. Vergleichen Sie beide Versionen und entscheiden Sie selbst, welche von beiden besser verstehbar ist. Untersuchen Sie weiterhin Beispiele von Dritten (etwa in Büchern oder Fachzeitschriften) hinsichtlich ihrer Gliederung. Überlegen Sie, ob die dort gewählte Aufteilung und Zusammenfassung verständlich und sinnvoll ist, und versuchen Sie, es nach Möglichkeit noch besser zu machen. Es ist besonders bei komplexeren Problemen ein gut Teil Geschmacksache dabei. Versuchen Sie im Laufe der Zeit Ihren eigenen «Stil» zu finden, und bleiben Sie dann dabei, gleich, ob Sie eigene Programme schreiben oder Programme von anderen übernehmen. Das einzige, worauf Sie achten sollten, ist, daß der Stil, den Sie sich aneignen, auch wirklich ein guter – d.h. ein verständlicher und sauberer – Stil ist.

Speichern des Programms: (C)SAVE, (C)LOAD

Wenn man mit einiger Mühe ein Programm in den Rechner eingetippt, von Fehlern befreit und zum Laufen gebracht hat, so möchte man die ganze Tipparbeit – besonders bei längeren Programmen – nur ungern noch einmal machen. Leider haben die heute verwendeten Mikrocomputer die unangenehme Eigenschaft, nach Ausschalten des Geräts alles restlos zu vergessen. Außerdem gestattet der Speicherplatz (und die Arbeitsweise der meisten BASIC-Versionen) in der Regel nur einem Programm den Aufenthalt im Speicher. Will man ein anderes Programm verwenden, so muß man das vorangehende (z.B. durch NEW) aus dem Speicher löschen. Kurz gesagt, man braucht eine Möglichkeit, ein Programm bei Bedarf irgendwo außerhalb des Rechners speichern zu können.

Alle Mikrocomputer, die BASIC-Programme verarbeiten, bieten deshalb irgendeine Form von *externem Speicherplatz* an. Das können ganz gewöhnliche Tonbandkassetten auf einem Kasset-

tenrekorder sein. Ausgebautere (und teurere) Modelle bieten als Alternative die Speicherung auf Disketten (sogenannte Floppy-Disks) verschiedener Größe an. Wir wollen hier nicht zu tief in die technischen Einzelheiten hinuntersteigen. Es genügt, daß die Möglichkeit zum externen Abspeichern von Programmen vorhanden ist.

Zum Aufzeichnen eines Programms auf ein derartiges externes Speichermedium bedarf es aber zweierlei:

- Man muß mit dem Gerät selbst umgehen können.
- Man muß BASIC den Befehl geben, das jeweilige Programm auf den im Computersystem vorhandenen Speicher aufzuzeichnen.

Zum ersten Punkt läßt sich hier wegen der Vielfalt an Möglichkeiten kaum etwas sagen. Studieren Sie Ihr Handbuch oder lassen Sie sich das Vorgehen von jemandem zeigen, der mit Ihrem System umgehen kann. Zur Not müssen Sie sich an den Verkäufer Ihres Computers wenden.

In der Regel sind es recht einfache Schritte, die zu vollziehen sind. Eine Kassettenaufzeichnung verlangt z.B. das Einlegen des richtigen Bands in den Rekorder, Einstellen des Bands so, daß keine andere Aufzeichnung beschädigt (überschrieben oder gelöscht) werden kann, Einstellen der Lautstärke und Tonhöhe zum Auslesen der Aufnahme vom Band und Einschreiben in den Computerspeicher und ähnliches. Bei Disketten mag es in den meisten Fällen sogar genügen, die richtige Diskette in das richtige Laufwerk einzusetzen. Wie gesagt, informieren Sie sich anhand Ihres Systemhandbuchs, bei Kollegen oder wo auch immer.

Nehmen wir an, daß alle diese vorbereitenden Schritte erledigt sind. Dann muß BASIC der Befehl gegeben werden:

«Rette das im Computerspeicher stehende Programm auf das externe Speichermedium (bevor es verlorengeht)!»

«Retten» heißt im Englischen «to save», und so lautet der allgemeine Befehl für diesen Vorgang in vielen BASIC-Versionen

SAVE

oder speziell bei Aufzeichnung auf Tonbandkassetten

CSAVE

(was für «Cassette SAVE» – Programm auf Kassette speichern – steht).

Manchmal kann man der Aufzeichnung auch einen Titel geben, der je nach Version zwischen einem und acht Buchstaben umfassen kann und der (C)SAVE als Argument nachgestellt wird. Ziehen Sie dazu am besten wieder Ihr Handbuch zu Rate.

Der umgekehrte Vorgang betrifft das Übernehmen eines Programms vom externen Speichermedium in den Computer. Vorausgesetzt, daß alle Geräteanforderungen erfüllt sind, kann man das Programm durch den BASIC-Befehl

LOAD

oder

CLOAD

(was «LADEN» oder «von Kassette LADEN» bedeutet), eventuell unter zusätzlicher Angabe des Titels der Aufzeichnung, in den Rechner bekommen. Das kann bei Kassettenaufzeichnungen gerade wie beim Abspeichern des Programms seine Zeit dauern und je nach Geräteauslegung sogar mehrere Male wiederholt werden müssen, bis alles richtig gelaufen ist. Wir wollen uns hier nicht mit möglichen Ladefehlern und ihrer Abhilfe befassen, das ist wiederum Aufgabe Ihres Systemhandbuchs oder sonstigen Geräteunterstützung. Setzen wir einfach voraus, es würde funktionieren.

War das durch (C)SAVE gerettete Programm einwandfrei, so können Sie das mittels (C)LOAD wieder in den Computer geladene Band unmittelbar durch RUN starten. Insbesondere ist in der Regel vor dem Ladevorgang kein Löschen des Speichers durch NEW o.ä. notwendig. Das geschieht in diesem Fall automatisch.

Sie können aber auch das eben in den Rechner geladene Programm mit den üblichen Ediermethoden verändern, zusammenstreichen oder ergänzen. In unserem Fall wird man z.B. vor dem RUN-Befehl erst ab Zeile 1000 die zu verarbeitende Tabelle in Form von DATA-Anweisungen vorgeben müssen. Die Änderungsmöglichkeit gestattet es insbesondere, auch noch nicht fertig eingetippte Programme für alle Fälle zwischenzuspeichern. Es empfiehlt sich, so etwas vor allem beim Eintippen längerer Programme regelmäßig – etwa nach jedem Sinnabschnitt – vorzunehmen. Stellen Sie sich vor, was geschieht, wenn kurz vor Beendigen der Tipperei der Strom für einen Moment ausfällt oder wenn Sie aus Versehen bei der Erprobung statt RUN den Speicher durch NEW oder die RESET-Taste Ihres Geräts wieder löschen. (Sagen Sie nicht, das könnte Ihnen nicht passieren – nach drei Stunden Tipperei am Gerät kommen die unmöglichsten Sachen vor!) Kurz gesagt: Machen Sie regelmäßig *Sicherheitskopien* (englisch: back-up copies) nicht nur von den Programmen, die sie in den Rechner gerade eintippen, sondern auch von allen Änderungen, die Sie an Programm oder Daten vornehmen. Das mag Platz und Zeit kosten – auf alle Fälle aber spart es Nerven (und unter Umständen viel Geld).

2
Systematisierung

Um ein Gefühl für die Sache zu bekommen, haben wir bis jetzt ziemlich «drauflos» gearbeitet. Es ist an der Zeit, unsere Erkenntnisse ein wenig zu systematisieren. Dazu wollen wir das, was wir bis jetzt erreicht haben, etwas globaler betrachten.

Flußdiagramme

Versuchen wir, unser Programm von Bild 4 ein wenig zu veranschaulichen, indem wir jeden Befehl in einen Kasten übertragen und den Übergang von einer Anweisung zur nächsten durch Pfeile wiedergeben. Dann ergibt sich nach Bild 5a:

Wir steigen mit RUN in das Programm ein, das zunächst (zweiter Kasten) die Variablen N und V auf Null initialisiert. Dann werden die Spaltenüberschriften ausgedruckt, gefolgt von einer Leerzeile (dritter Kasten). Die Übernahme des ersten Werts aus der Tabelle in die Variable S1 leitet die Rechnung ein (vierter Kasten), worauf – deutlich zu erkennen – in die Schleife zur Berechnung der Einzelwerte eingetreten wird. Hier werden die Werte der anderen beiden Variablen aus der Tabelle übernommen, getestet, ob das Tabellenende erreicht ist und – falls nicht – der Einzelverbrauch berechnet, aufsummiert, gezählt und ausgedruckt. Die Zuweisung des alten Endwerts (S1) an die Anfangsvariable S1 der Verbrauchsberechnung leitet bereits den nächsten

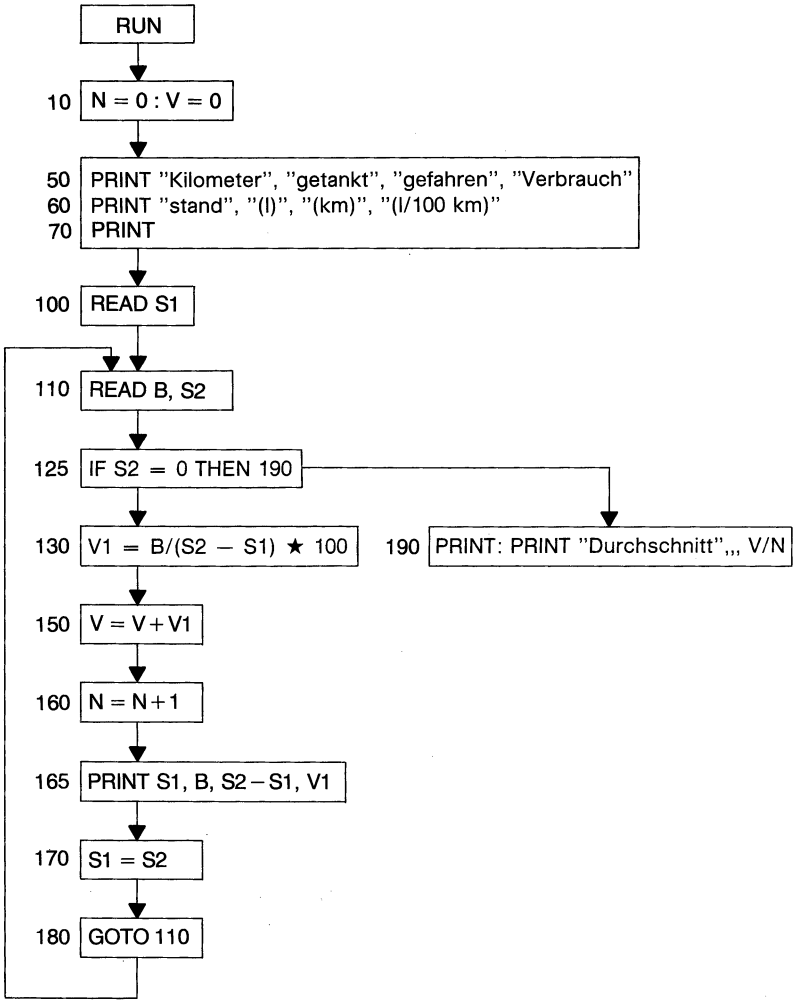


Bild 5a Veranschaulichung des logischen Programmflusses

Schleifendurchgang ein, der mit dem Sprung zurück zum fünften Kasten durchgeführt wird. Ist das Tabellenende schließlich erreicht, so wird – wiederum deutlich erkennbar – aus der Schleife herausgesprungen und zum Programmabschluß der Durchschnittswert geeignet ausgegeben.

Die bisherige «aufgefädelte» Darstellung unseres Programms ist übersichtlicher geworden. Dadurch tritt der logische Fluß der Befehle klarer hervor, weshalb man eine solche Darstellung auch als *Flußdiagramm* bezeichnet.

Dabei ist noch einiges zu verbessern. Zum Beispiel: Wozu braucht man eigentlich die Zeilennummern an den Kästen? Das Diagramm hätte doch sicher dieselbe Form, würden wir das Programm ab Zeile 5000 und in Dreierschritten ablegen. Offensichtlich besteht ein Unterschied zwischen dem *logischen Aufbau* des Programms, den wir durch das Flußdiagramm verdeutlichen können, und der Umsetzung in BASIC-Zeilen, der *Implementierung* des Problems als BASIC-Programm. Die Zeilennummern beziehen sich ausschließlich auf eine von vielen möglichen Implementierungen; uns interessiert hier jedoch der allgemeine Grundgedanke des Programms. Streichen wir die Zeilennummern also.

Damit wird allerdings auch der Befehl «GOTO 110» hinfällig. Das ist nicht weiter schlimm, denn – genau betrachtet – brauchen wir ihn gar nicht mit anzugeben. «GOTO xxx» besagt ja nur, daß wir als nächstes den Befehl in Zeile xxx abarbeiten wollen. Das wird im Flußdiagramm aber ohnehin durch den Pfeil zu dem betreffenden Kasten angezeigt.

«GOTO xxx» ist ein typischer *Steuerbefehl,* der die Beschränkungen der linearen Programmdarstellung (eine Befehlszeile nach der anderen) überwinden soll. Seine ausdrückliche Angabe in einer Darstellung, die diese Einschränkung nicht kennt, würde nur Verwirrung stiften.

Wir können noch mehr tun: RUN, PRINT usw. sind eigentlich Bestandteile der BASIC-Implementation. Der Zusammenhang erschließt sich besser, wenn wir in den Kästen die gemeinte Operation angeben, also z.B. «Spaltenüberschriften ausgeben», «1.

Kilometerstand (S1) lesen» usw. Wir entfernen uns damit zwar einen Schritt weiter von der konkreten Implementierung des Programms, gewinnen dafür aber mehr Einsicht in die logische Struktur der Lösung.

Beschäftigen wir uns noch ein wenig mit einem weiteren Steuerbefehl: der *logischen Entscheidung* nach der Wertübernahme aus der Tabelle. Logische Entscheidungen sind so etwas wie Marksteine im Flußdiagramm: Man kann sich an ihnen rasch über die Grundstruktur des Programms orientieren. Um die Besonderheit der Entscheidung, als *Verzweigung* einen Eingang und (mindestens) zwei Ausgänge zu haben, im Diagramm deutlich hervortreten zu lassen, verwendet man statt des Kastens eine auf der stumpfen Ecke stehende Raute und schließt die jeweiligen Pfeile an den Ecken an, während die Entscheidungsbedingung über den einzuschlagenden Weg in die Raute selbst geschrieben werden.

Dabei bleibt allerdings noch ein Problem: Wie stellt man fest, welche der beiden möglichen Verzweigungen eingeschlagen werden soll? Man könnte in unserem Beispiel in die Raute schreiben: «Wenn S2 = 0 ist, dann gehe nach rechts.» Es empfiehlt sich aber, das nicht zu tun. Wir würden nur wieder an einer bestimmten Problemdarstellung – der gerade aufgemalten Zeichnung – kleben. Nicht nur, daß wir bei Änderungen der Zeichnung den Text möglicherweise mit ändern müßten (wenn wir aus Platzgründen nach links herauskämen), wichtiger ist, daß das Entscheidungskriterium unübersichtlich wird.

Wenn wir nun abhängig von einer bestimmten Aussage über die Verzweigungsbedingung einen bestimmten Weg einschlagen möchten, warum geben wir das nicht einfach an dem zugehörigen Pfeil an? In unserem Beispiel könnten wir als Bedingung «S2 = 0?» in die Raute schreiben, was besagt, daß der Rechner vor Weitergehen im Programm den Wert von S2 ansehen muß. Ist er gleich Null, so ist der Weg zur Durchschnittsausgabe einzuschlagen, also bezeichnen wir den zugehörigen Pfeil einfach mit «ja», denn hier ist die Bedingung «S2 = 0» erfüllt. Ganz entsprechend bezeichnen wir den anderen Ausgang aus der Raute mit «nein».

Ersetzen wir noch das «RUN» am Programmanfang durch das, was dort inhaltlich ausgedrückt werden soll, nämlich «Anfang» und ergänzen einen Hinweis auf das Programmende nach dem Kasten zur Ausgabe des Durchschnittswerts, dann gelangen wir zu der Darstellung in Bild 5b, die doch ein ganzes Stück übersichtlicher als unsere ursprüngliche Version ist.

Genau betrachtet lassen sich hierbei vier verschiedene Funktionsgruppen unterscheiden:

- Verbindungen nach außen, zur «Umgebung» des Programms (Anfang, Ende),
- Übernahme und Ausdrucken von Werten,
- die Verarbeitung von Werten und
- die Verzweigung nach einer logischen Entscheidung (S2 = 0?).

Nun konnten wir bereits durch Einführen eines besonderen Symbols für die logische Entscheidung das Diagramm übersichtlicher machen. Also liegt es nahe zu fragen, ob dies nicht auch für die anderen oben aufgeführten Gruppen geht. Und wenn wir schon fragen: Natürlich geht es. Man hat die hierfür zu verwendeten Zeichen sogar genormt. Ganz allgemein ist vereinbart, daß jeder logische Arbeitsschritt in einen eigenen Kasten kommt und daß man die einzelnen Gruppen überall da hervorhebt, wo das sinnvoll für die Übersicht ist.

Nehmen wir unser Beispiel zur Grundlage. Da haben wir zunächst einmal die *Endpunkte* «Anfang» und «Ende». Diese «Terminals» werden immer in ein Oval eingeschlossen. Dabei bezeichnet man sie in der Regel nicht mit «Anfang» und «Ende» (das ist ohnehin klar), sondern gibt am Anfang einen *Kurznamen* für das betreffende Programmstück an. Das hat den Vorteil, daß man sich von anderen Teilen eines umfangreichen Programms einfach auf dieses Stück beziehen kann. Ebenso am Ende: Hier kann man, wenn nach Abarbeiten des betrachteten Programmteils an einer *bestimmten* anderen Stelle im Programm weitergemacht werden soll, in das Endoval den Namen dieser Stelle ein-

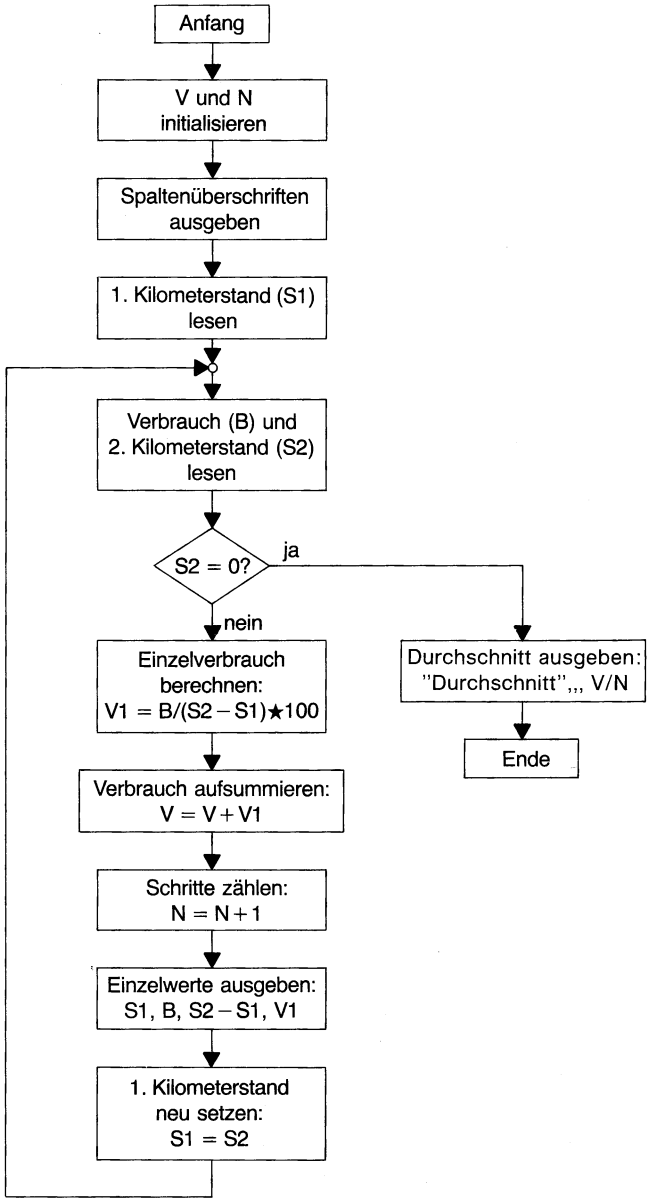


Bild 5b Ein ausführliches Flußdiagramm

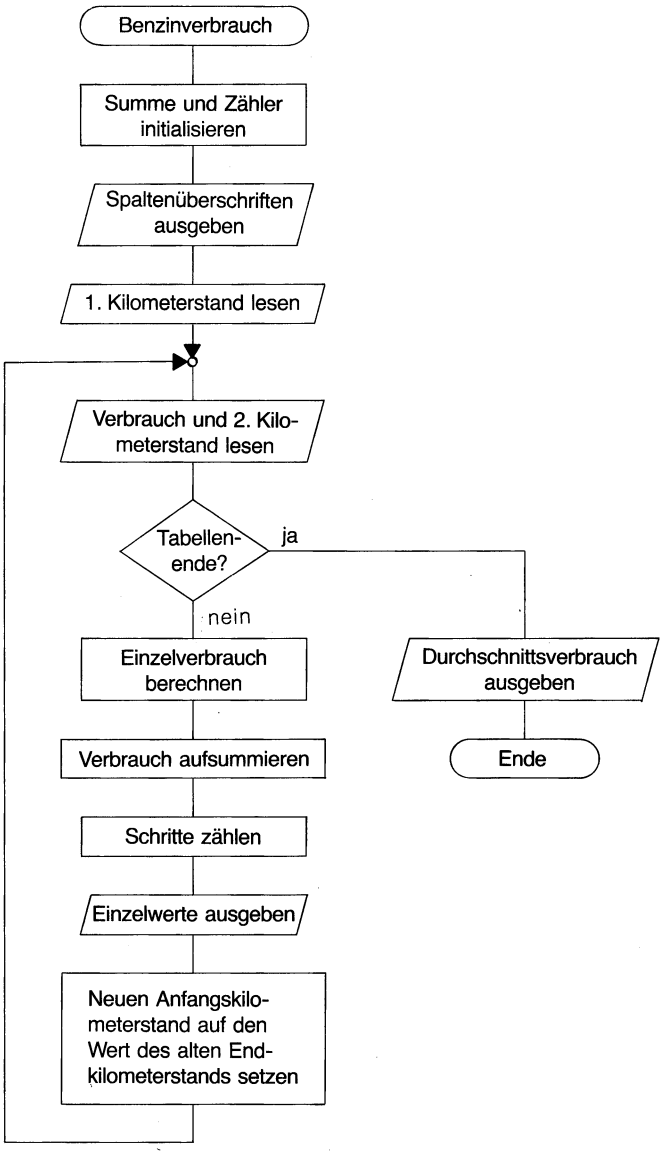


Bild 5c Endgültige Form des Flußdiagramms

getragen. In unserem Fall können wir bei «Ende» bleiben, «Anfang» wollen wir der Illustration halber durch «Benzinverbrauch» ersetzen.

Ein- und Ausgabeoperationen werden im allgemeinen in einen *schrägen Kasten* eingeschlossen. Man hat hier darüber hinaus die Möglichkeit, zwischen verschiedenen Arten von Operationen zu unterscheiden, je nachdem, von wo die eingegebenen Daten kommen bzw. wohin ausgegeben werden soll. Diese Spezialsymbole sind in Bild 6 wiedergegeben. Man sollte allerdings sparsam mit ihnen umgehen, da ein Zuviel an verschiedenen Symbolen in einem Flußdiagramm wieder verwirrt. In unserem Fall bleiben wir bei dem einfachen schrägen Kasten.

Vereinbaren wir noch eine weitere Vereinfachung, die Pfeile im Diagramm betreffend. Wir können hier Arbeit sparen, wenn wir die *Pfeilspitzen* nur dann zeichnen, wenn Verwechslungen möglich sind. Insbesondere kann man die Spitzen dann weglassen, wenn der logische Fluß der Leserichtung – also von links nach rechts bzw. von oben nach unten – entspricht.

Anders an den Stellen, an denen *mehrere Wege zusammentreffen,* wie in unserem Beispiel am Schleifenanfang. Hier muß man eindeutig zwischen ankommenden Verbindungen und abfließenden unterscheiden können. Mit anderen Worten: Man muß sofort sehen können, wie es weitergeht. Deshalb werden hier die *zur Verbindung* (als kleiner Kreis angegeben) hinführenden Pfeile *immer* gezeichnet.

Wenn wir das alles beachten, kommen wir zur Darstellung von Bild 5c. Beachten Sie, daß wir hier die Operationsangaben so allgemein wie nur möglich gehalten haben. Es wird tatsächlich nur der logische Inhalt des Programms und nichts von seiner konkreten Implementation (z.B. den schließlich verwendeten Variablenbezeichnungen) wiedergegeben.

Eine solche äußerste Verallgemeinerung ist allerdings nur bedingt sinnvoll. Man wird diese Form in erster Linie am Anfang der Programmierarbeit antreffen, dort, wo es wirklich nur um den Programminhalt und überhaupt noch nicht um die konkrete Dar-

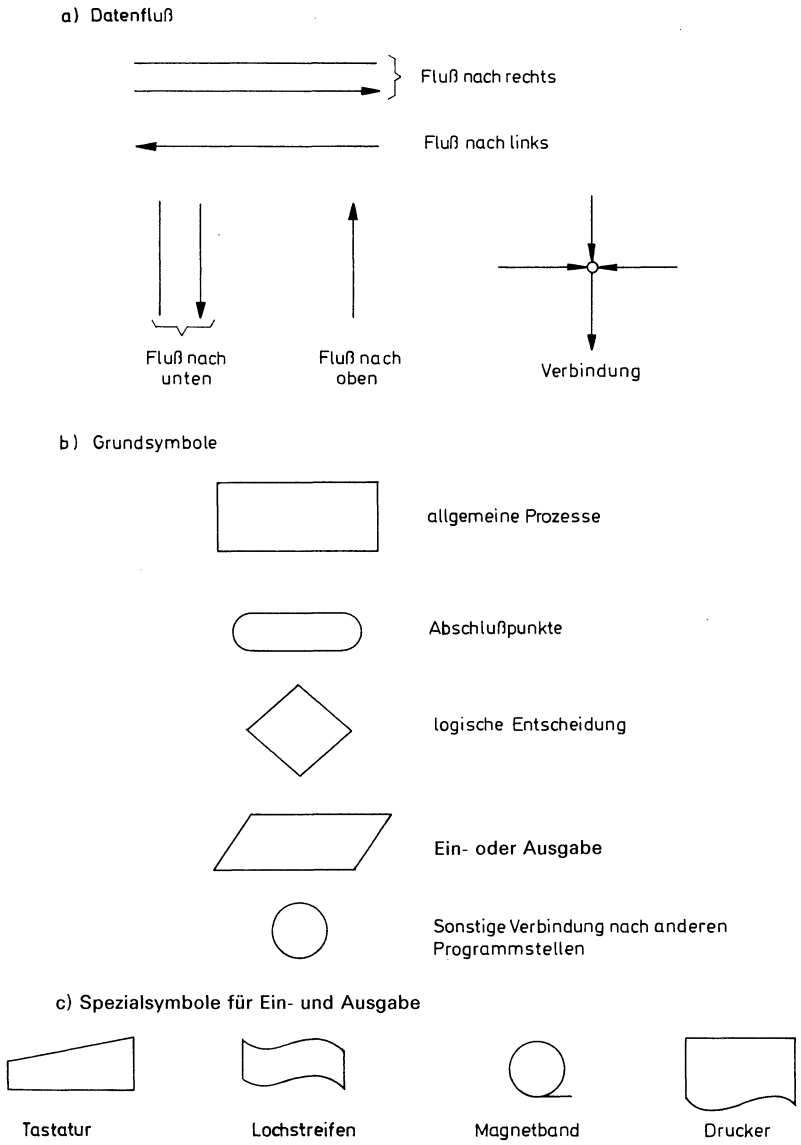


Bild 6 Symbole in Flußdiagrammen

stellung als BASIC-Programm geht. Ein solches allgemeines Flußdiagramm könnte auch in jeder anderen Sprache implementiert werden. Es ist überall dort angebracht, wo man eine Programm*idee* weitervermitteln möchte. Ganz allgemein gilt:

Fassen Sie ein Flußdiagramm so knapp und allgemein, wie es die Übersicht erfordert, aber so ausführlich, daß es ohne zusätzliche Anstrengung verstehbar und mit der konkreten Implementation in Verbindung zu bringen ist.

Die Strukturbeschreibung

Ein Flußdiagramm erhält seine besondere Bedeutung beim Erstellen eines Programms und wird dort um so wichtiger, je komplexer die zu lösende Aufgabe ist. Es ist aber nicht das einzige Hilfsmittel, das hier zur Verfügung steht. Ein solches Flußdiagramm erleichtert das Eindringen in einen möglichen Ablauf der Problemlösung zwar erheblich, doch drängt es einen in eine bestimmte Denkrichtung hinein, die für sich alleine genommen an der optimalen Problemlösung vorbeiführen kann. Man müßte noch andere Möglichkeiten haben, eine Aufgabe für das Programm aufzubereiten.

Wie wäre es, wenn man eine *rein verbale Beschreibungsform,* eine Darstellung der Zusammenhänge nur mit Worten versuchen würde? Unser Programmbeispiel ließe sich z.B. so beschreiben:

```
Benzinverbrauch:
  Summe und Zähler initialisieren,
  Spaltenüberschriften ausgeben,
  Anfangskilometerstand lesen,
  Schleife:
    Verbrauch lesen,
    Endkilometerstand lesen,
    Wenn Tabellenende, dann:
      Durchschnittsverbrauch ausgeben,
      Programm beenden.
```

Einzelverbrauch berechnen,
Verbrauch aufsummieren,
Schritte zählen,
Einzelwerte ausgeben,
Anfangskilometerstand ← Endkilometerstand.
Nächster Schleifendurchgang.
Ende.

Im Prinzip steht hier das gleiche wie in unserem Flußdiagramm von Bild 5c. Jedoch liegt der Schwerpunkt anders. Während wir uns in den Diagrammen auf den *Fluß* der Ereignisse konzentrieren, geht es hier um die *Gliederung* der Aufgabe in Unteraufgaben, um ihre innere Struktur, weshalb man eine solche Darstellung als «Strukturbeschreibung» bezeichnet.

Typisch für eine solche Strukturbeschreibung ist ihre intensive Gliederung: Jede Teilaufgabe bildet eine Ebene für sich, die von den übergeordneten Aufgaben umschlossen wird. So haben wir eine Art *Schachtelprinzip:* Höchste Ebene ist die allgemeine Aufgabenstellung «Benzinverbrauch:» ... «Ende.», welche die Unterebene «Schleife:» ... «Nächster Schleifendurchgang.» als Teilstruktur umfaßt. Diese wiederum enthält den bei erfüllter Aussprungbedingung gültigen Programmteil «Durchschnittsverbrauch ausgeben,» und «Programm beenden.» als weitere Unterstruktur.

Die Strukturbeschreibung ist vor allem bei der Erstellung umfangreicher und komplexer Programme wertvoll. Man kann die Aufgabe mit ihrer Hilfe schrittweise in immer feinere Untereinheiten zerlegen, bis man schließlich auf einer Ebene angelangt ist, die unmittelbar in ein Programm übersetzt werden kann. Wir werden das später noch eingehend veranschaulichen.

Aus dem Vergleich der Strukturbeschreibung mit den Flußdiagrammen von Bild 5 wird ein wichtiger Unterschied zwischen den beiden Methoden zur Problemaufarbeitung deutlich. Bei der Strukturbeschreibung bleibt die allgemeine Aufgabenstellung jeder Unterabteilung erhalten, die man sich aus dem Flußdia-

gramm manchmal mühsam rekonstruieren muß. Auf der anderen Seite hat das Flußdiagramm den Vorzug, daß man durch die zweidimensionale Darstellung sofort sieht, was als nächstes geschieht. So schließen sich Strukturbeschreibung und Flußdiagramm nicht etwa gegenseitig aus, sondern ergänzen sich und können – jedes an seiner Stelle verwendet – ein beachtlich leistungsfähiges Werkzeug bei der Programmerstellung bieten.

Das Struktogramm

Bei aller Leistungsfähigkeit hat die Strukturbeschreibung doch einen Nachteil: Sie ist streng linear aufgebaut. Beziehungen können nicht wie beim Flußdiagramm durch räumliches Nebenein-

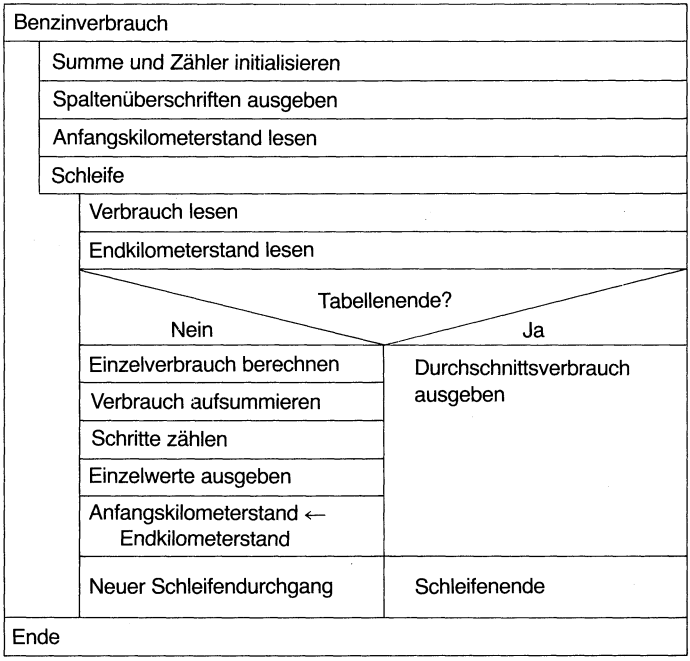


Bild 7a Erste Form eines Struktogramms zur Verbrauchsauswertung

anderstellen verdeutlicht werden. Dadurch wird sie bei tiefen Verschachtelungen mit vielen Verzweigungen leicht unübersichtlich. Man hat daher nach Möglichkeiten gesucht, die intensive Programmgliederung mit einer räumlich übersichtlichen Darstellungsweise zu verbinden. Eine der Lösungsmöglichkeiten stellt das sogenannte Nassi-Schneiderman-Diagramm oder *Struktogramm* dar.

Bild 7a zeigt das unserer eben erstellten Strukturbeschreibung entsprechende Struktogramm. Das auffälligste an der Darstellung ist die Wiedergabe der logischen Entscheidung «Tabellenende?» zusammen mit den zugeordneten Programmabschnitten für die Antwortmöglichkeiten «Ja» und «Nein». Das Grundprinzip ist hier, daß man die in der einfachen Strukturbeschreibung *aufeinanderfolgenden* Programmzweige nach der logischen Entscheidung *nebeneinander* stellt und in einer gemeinsamen Kopfzeile die Bedingung und die möglichen Entscheidungswege angibt. Das Ganze wird dann noch geeignet grafisch zu Blöcken zusammengefaßt.

Das Struktogramm vermag durch den einfachen Trick der Zusammenfassung des Nacheinander verschiedener Programmzweige in ein Nebeneinander sehr viel mehr Überblick über den Programmzusammenhang geben, als es die einfache Strukturbeschreibung tut. Nachteilig ist allerdings, daß das Erstellen eines sauberen Struktogramms umständlicher ist, und Änderungen sehr großen Aufwand bereiten können. So eignet sich das Struktogramm besser zur Darstellung fertiger Lösungen oder für die Fälle, in denen man sich im Programmierprozeß eine zusätzliche Übersicht verschaffen möchte. Wir werden in den meisten Fällen bei den einfacher darstellbaren und bei einiger Sorgfalt ausreichend übersichtlichen Strukturbeschreibungen bleiben.

Darstellung von Schleifenstrukturen

Wenn man es genau betrachtet, dann ist unsere Schleifendarstellung in der Strukturbeschreibung wie im Struktogramm von



Bild 7b Endgültiges Struktogramm zur Verbrauchsauswertung

Bild 7a ziemlich umständlich. Sie umfaßt ja eigentlich nur die folgenden Befehle:

Verbrauch lesen,
Endkilometerstand lesen,
Einzelverbrauch berechnen,
Verbrauch aufsummieren,
Schritte zählen,
Einzelwerte ausgeben,
Anfangskilometerstand ← Endkilometerstand.

Die Schleife selbst ist durch die Angabe der Bedingung «Bis Tabellenende» ausreichend bestimmt. Die zugehörigen Operationen zur Berechnung der einzelnen Verbrauchswerte werden wiederholt, bis das Tabellenende erreicht ist. Die besondere Art und Weise, in der wir das Tabellenende im Programm ausfindig machen, ist eigentlich eine Frage der konkreten Implementation und hätte auch ganz anders gelöst werden können. Mit der eigent-

lichen Programm*logik* hat die Methode, das Ende aufzufinden, nur am Rande zu tun.

Die Programmlogik, die innere «Struktur» des Programms ist es aber, die wir in der Strukturbeschreibung festhalten wollen. Warum machen wir die Bedingungen der Schleife dann nicht deutlicher? Das könnte z.B. so aussehen:

Bis Tabellenende:
  Verbrauch lesen,
  Endkilometerstand lesen,
  Einzelverbrauch berechnen,
  Verbrauch aufsummieren,
  Schritte zählen,
  Einzelwerte ausgeben,
  Anfangskilometerstand ← Endkilometerstand.
Schleifenende.
Durchschnittsverbrauch ausgeben.

Das macht doch viel deutlicher, was wir eigentlich beabsichtigt haben: Berechnung der Einzelwerte, bis die Tabelle erschöpft ist und dann erst Ausgabe des Durchschnittsverbrauchs.

Bild 7b zeigt die Darstellung einer solchen Schleife in einem Struktogramm. Sie entspricht im großen und ganzen unserer Strukturbeschreibung, nur daß der *Schleifenkörper* durch die zusätzliche Einschachtelung noch einmal besonders gegen die *Schleifenbedingung* abgehoben ist, was die Übersichtlichkeit weiter verbessert. Man sieht deutlich, daß eine solche Schleife einen Unterabschnitt in sich bildet, eine Tatsache, die sich aus einem Flußdiagramm fast nie ableiten läßt.

Etwas zur Form von Strukturbeschreibungen

Es empfiehlt sich, bei der Abfassung von Strukturbeschreibungen von vornherein einige kleine Formalitäten einzuhalten, durch welche die Darstellung übersichtlich und eindeutig gehalten werden kann.

Besonders bei längeren Problemlösungen wichtig ist die Regel:

Gliedern Sie eine Strukturbeschreibung nur so tief, wie zum Verständnis unbedingt nötig ist!

Zu viele Einzelheiten machen die Beschreibung unübersichtlich, und zu lange Beschreibungen lassen den Zusammenhang verlorengehen. Jede Strukturbeschreibung sollte nach Möglichkeit nicht mehr als drei bis vier – durch Einrücken gekennzeichnete – Unterebenen enthalten und sollte maximal eine Seite einnehmen. Wenn eine Untereinheit in sich selbst wieder sehr viele Einzelheiten verlangt, dann ist es besser, sie als eigene Strukturbeschreibung aufzustellen. Das muß allerdings in der übergeordneten Beschreibung geeignet vermerkt werden, etwa so:

Aktion ...:
(gesondert ausgeführt).

Das «gesondert ausgeführt» ist dabei in *Klammern* gefaßt, um zu verdeutlichen, daß es sich dabei um keine eigene Aktion, sondern nur um einen *Verweis* auf eine andere Stelle handelt. Wir werden Klammern in der Strukturbeschreibung zu verschiedenen Zwecken einsetzen, nämlich

a) um Verweise auf Aktionen von den eigentlichen Aktionen zu unterscheiden;
b) um innerhalb einer Zeile mehrere Aktionen gegeneinander abzusetzen;
(Das geschieht in erster Linie bei Fallunterscheidungen, die mit «und» oder «oder» mehrere Möglichkeiten zusammenfassen, z.B.:

Wenn ((X>0 und D gesetzt)
oder
(X<0 und D gelöscht))
dann:

Dabei gibt es keine festen Regeln. Man sollte auch mit Klammern sparsam umgehen und sie nur dort setzen, wo sie zum besseren Verständnis unbedingt notwendig sind.)

c) wie üblich in Formeln und ähnlichem, wo es im Sinnzusammenhang hingehört.

Umklammern Sie alle größeren Einheiten durch eine Anfangs- und eine Endzeile.

Hier ist «größeren» zu betonen. Das bezieht sich vor allem auf Schleifen und selbständige Blöcke. Allerdings ist es unübersichtlich, jeden Block durch eine besondere Anfangs- und Endzeile hervorzuheben. Wir werden, wo immer es möglich ist, derartige Gliederungen durch Einrücken und geeignete Zeichensetzung sichtbar machen. So könnte in unserem Beispiel die Endklammer «Schleifenende» auch entfallen, da hier ohnehin alles klar ist:

```
Bis Tabellenende:
  Verbrauch lesen,
  Endkilometerstand lesen,
  Einzelverbrauch berechnen,
  Verbrauch aufsummieren,
  Schritte zählen,
  Einzelwerte ausgeben,
  Anfangskilometerstand ← Endkilometerstand.
Durchschnittsverbrauch ausgeben.
```

Gliedern Sie die Beschreibung optisch so gut wie möglich.

Das betrifft vor allem die Einrückungen von Untereinheiten. Zusätzlich kann man die Beschreibung durch Leerzeilen geeignet auflockern. Allerdings muß man hier vorsichtig sein, da eine zu intensive Verwendung von Leerzeilen das Gesamtbild zerreißen und dadurch wieder unübersichtlich machen kann. Es hängt von den jeweiligen Umständen (Druckbild u.ä.) ab, ob eine Gliederung durch Leerzeilen sinnvoll ist.

Verwenden Sie eine eindeutige Zeichensetzung.

Wir haben uns hier von einer der Schrift ähnlichen Gliederung in «Abschnitte» und «Sätze» leiten lassen. Jeder «Abschnitt» (d.h. jeder Block von Unteraktionen) hat eine «Überschrift», die durch einen *Doppelpunkt* abgeschlossen ist. Er soll signalisieren: «Hier folgt die auszuführende Aktion.»

Diese wird im folgenden eingerückt. Einfache Aktionen (d.h. Einzelzeilen) werden durch *Kommas* voneinander getrennt, das soll die Aufzählung verdeutlichen und klar machen, daß man sich immer noch im selben Unterblock befindet.

Die letzte Unteraktion des Blocks wird wie ein Satz durch einen *Punkt* abgeschlossen. Es empfiehlt sich, den Punkt immer als Abschlußzeichen zu setzen, selbst wenn nur eine einzige Zeile im Block vorhanden ist.

Zusammengefaßt unterscheiden wir in der Strukturbeschreibung also folgende drei Zeichenarten:

a) Einleitungszeichen «:»
b) Trennungszeichen «,»
c) Endzeichen «.»

Das ist natürlich nur ein Vorschlag. Es empfiehlt sich aber, von vornherein eine klar gegliederte Darstellung mit einer sinnvollen Zeichensetzung zu wählen. Das erleichtert nicht nur die Programmierarbeit, sondern macht auch einen später eventuell notwendigen Übergang zu anderen «blockorientierten» Programmiersprachen (wie z.B. PASCAL) einfacher, da man sich schon eine gewisse Denkdisziplin angewöhnt hat.

Programmiertechniken

Flußdiagramm, Strukturbeschreibung und Struktogramm sind drei Werkzeuge, die in den Bereich der Programmiertechniken gehören, der die Methoden und Hilfsmittel bei der Programmierung umfaßt.

Es gibt viele Möglichkeiten, ein Programm zu erstellen. Man kann zum Beispiel von den ersten Einfällen ausgehend einfach drauflos programmieren und hoffen, irgendwie irgendwann ans Ziel zu gelangen. Hin und wieder schafft man das auch – aber fragen Sie nicht, wie und wann das sein wird.

Der Nachteil dieser «Methode» liegt darin, daß es eigentlich gar keine Methode ist. Man versenkt sich in das Stückchen Kode, das man gerade bearbeitet, und hat keinen Blick für die Probleme um sich herum. Für den Augenblick mag das am wenigsten anstrengend sein, betrachtet man aber den Gesamtaufwand von der ersten Programmidee bis zum fertigen (und funktionierenden!) Programm, so muß man sagen, daß man die Zeit und Energie besser woanders investiert hätte.

Das Problem ist nicht neu, und es wurde recht viel Forschungsarbeit investiert, um herauszufinden, wie man am schnellsten mit geringstem Aufwand zuverlässige und übersichtliche Programme erstellen kann. Im Lauf der Zeit kristallisierten sich dazu einige Vorgehensweisen heraus, die sich bewährt haben.

Zerlegen sie die Aufgabenstellung schrittweise von oben nach unten in immer kleinere Teilprobleme, bis Sie überall eine genügend einfache Beschreibungsebene erreicht haben.

Diese sogenannte «Top-down»-Programmierung (d.h. Programmierung «von oben nach unten») entspricht dem Aufstellen von Strukturdiagrammen für das gesamte Problem von der ersten groben Gliederung in Unterprobleme bis zur letzten Einzelaktion. Der Vorteil dieses Vorgehens besteht darin, daß man immer den Kontakt zum Gesamtsystem behält. Allerdings muß man einige Konzentration in die Erarbeitung eines solchen Programmplans investieren. Führt man nämlich dieses Prinzip streng durch, so entsteht der größte Teil des Programms ausschließlich am Schreibtisch, ohne Rückmeldungen vom Gerät und Hinweise auf mögliche Fehler. Man muß sich schon sehr sicher sein und viel Geduld und Disziplin haben, um dieses Vorgehen ohne greifbare Ergebnisse bei umfangreichen Problemstellungen auch wirklich bis zum Schluß durchzuhalten.

Ist die Aufgabe jedoch erst einmal genügend in Einzelheiten zerlegt, dann kann man mit der eigentlichen Programmerstellung beginnen:

Erstellen Sie das Programm in einzelnen in sich abgeschlossenen und getesteten Bausteinen schrittweise von der untersten Ebene an, bis das gesamte Problem gelöst ist.

Man bezeichnet das als «Bottom-up»-Programmierung (als Programmierung «von unten nach oben»). Hier wird das ganze Programm aus einzelnen – möglicherweise bereits vorhandenen – Bausteinen von unten nach oben zusammengestellt. Das hat den Vorteil, daß die zu lösende Teilaufgabe immer übersichtlich bleibt, daß man jeden Schritt für sich austesten und dann eine Stufe höher mit ausreichender Sicherheit verwenden kann.

Allerdings muß man – will man sich nicht verrennen – an diesem Punkt bereits das Gesamtproblem systematisch aufgearbeitet haben. Man muß sich über den Programmaufbau im klaren sein, bevor man mit der ersten Kodezeile anfängt. Geht man streng vor, so hat man auch hier ein gut Teil Unsicherheit, ob man wirklich auf dem richtigen Weg ist.

So können beide Ansätze – «Top-down» und «Bottom-up» – eines nur bedingt: Hilfestellung für einen brauchbaren Lösungsansatz liefern. Man muß sich für das endgültige Programm bereits genau im klaren sein, was man will und wie man es will. In den meisten Fällen ist es aber gerade so, daß man zu Beginn der Arbeit (auch an den Teilproblemen) nur eine höchst verschwommene Vorstellung davon hat, was eigentlich gemacht werden soll und wie die vorhandenen Möglichkeiten genutzt werden können.

Früher, als Computerzeit noch sehr teuer war, blieb einem nichts übrig, als sich auf Papier und Bleistift zu beschränken, bis das Programm endlich zum ersten Probelauf – in der als «endgültig» gemeinten Fassung – fertig war. Das garantiert durchaus nicht für eine optimale Lösung. Man verbohrt sich gar zu leicht in einen ersten Ansatz und wendet letztlich sehr viel Zeit auf, um all die Klippen zu umschiffen, die ein etwas unglücklich gewählter Lösungsansatz liefert.

Mikrocomputer bieten dank ihrer Unmittelbarkeit hier eine ganz neue Möglichkeit an, eine Art «Programm-Brainstorming». Wenn Sie sich über einen Lösungsansatz nicht ganz klar sind, wenn Sie eben gerade noch nicht eine saubere Strukturbeschreibung nahezu mühelos aufschreiben können, wenn Sie plötzlich an Einzelproblemen «hängen», wenn verschiedene Lösungsalternativen gegeneinander abzuwägen sind – dann nehmen Sie doch einfach den Computer her und probieren aus, was Ihnen an Ansätzen in den Sinn kommt. *Gerade eine solche scheinbare Spielerei, ein Drauflosprogrammieren nach den ersten verschwommenen Vorstellungen, ein Experimentieren mit den verschiedensten Möglichkeiten, die einem dabei plötzlich klar werden, wirkt ungemein befruchtend!* Es schafft das nötige Material, mit dem man erst ernsthaft an die Lösung gehen kann. Dann allerdings, wenn der Weg klar ist, dann ist *strengste Disziplin* geboten, dann sind die oben genannten Methoden zusammen mit den passenden Werkzeugen Flußdiagramm, Strukturbeschreibung, Struktogramm usw. strengstens einzuhalten, will man brauchbare Ergebnisse erzielen.

Doch genug der «Theorie». Sehen wir uns im nächsten Kapitel an, wie man ein Programmierproblem in der Praxis angehen kann.

3
Eine Fallstudie

Traditionell gibt es zwei grundlegende Einsatzbereiche von Computern: Da sind zum einen komplizierte Rechnungen zu verarbeiten, wie sie vor allem im technisch-wissenschaftlichen Bereich auftreten, und da sind zum anderen große Datenmengen zu verwalten und zu verarbeiten, wie es z.B. im kommerziellen Bereich notwendig ist. Die modernen BASIC-Versionen werden beidem gerecht, wobei im zweiten Fall mehr Programmieraufwand notwendig ist, da diese Daten in geordneter Form gespeichert, wieder aufgefunden, kombiniert und die Ergebnisse daraus dem Benutzer in festgelegter Form (z.B. als Tabelle) mitgeteilt werden müssen.

Sehen wir uns ein einfaches Beispiel für das Sammeln und Verarbeiten von Daten an. Es geht uns hier in erster Linie darum, wie man ein Programm von den ersten Problemskizzen bis hin zur endgültigen Lösung aufbaut. Dabei werden wir bewußt keine «glatte», problemlose Darstellung geben, sondern den mannigfaltigen Überlegungen und Versuchungen folgen, wie sie im tatsächlichen Programmierprozeß auftreten. Und wir wollen natürlich die Sprache selbst tiefer ergründen, also weitere BASIC-Befehle und das Umgehen mit ihnen kennenlernen.

(Um es vorweg zu sagen: Man braucht bereits zur Verarbeitung des Datenanfalls in einem Kleinbetrieb sehr viel mehr Rüstzeug. Die Verarbeitung und Verwaltung großer Datenmengen ist eine

Kunst für sich und umfaßt viele Teilgebiete. Wir können hier nur die allereinfachste Methode illustrieren.)

3.1 Aufgabenstellung

Nehmen wir an, wir hätten ein kleines Lager zu verwalten, das ungefähr zweihundert Arten von Gegenständen umfaßt. Unsere Aufgabe ist es, eine Übersichtsliste über den Lagerbestand zu führen, die für jede Entnahme und jedes Hinzufügen von Lagerteilen auf den neuesten Stand gebracht werden muß. Außerdem sind die betreffenden Artikel rechtzeitig, d.h. wenn die gelagerte Stückzahl einen bestimmten vorgegebenen Wert unterschritten hat, nachzubestellen. Schließlich und endlich müssen regelmäßig Übersichten über den Lagerbestand und den Wert der gelagerten Ware zusammengestellt werden.

All das kann von Hand ziemliche Mühe machen und erfordert einigen Zeitaufwand, den man besser in wichtigere Arbeiten investiert. Sehen wir uns deshalb an, ob und wie das Problem mit Hilfe eines BASIC-Programms gelöst werden kann.

3.2 Vorüberlegungen

Was ist zu tun? Da haben wir zunächst einmal die gelagerten Artikel selbst nach Bezeichnung, Menge und Wert zu erfassen. Das sind die vorgegebenen Werte, die *Eingabedaten*. Diese Daten sind geeignet zu *verarbeiten*, d.h. aufzubereiten und je nach Bedarf in verschiedenen Listen zusammengefaßt wiederzugeben. Das sind die erarbeiteten Werte, die *Ausgabedaten*. Diese drei Grundeinheiten

- Eingabe,
- Ausgabe und
- Verarbeitung

treten in jedem Programm auf.

Es ist sehr wichtig, sich vor jeder intensiveren Beschäftigung mit einem Programmierproblem möglichst genau darüber klarzuwerden, was wann *eingegeben* und was wann *ausgegeben* werden soll. Denn Eingabe und Ausgabe legen Ausgangspunkt und Ziel des zu lösenden Problems fest. Sind sie nicht richtig formuliert, so kann auch die ausgefeilteste Verarbeitung der Werte nicht das gewünschte Ergebnis liefern.

Legen wir also erst einmal genauer fest, was uns in der Aufgabenstellung gegeben ist.

Eingaben

Die Eingabewerte in unser Programm beschreiben die verschiedenen gelagerten Artikel. Das umfaßt folgende Angaben:

- Name des Gegenstands
  Wir wollen hier annehmen, daß jeder Lagerartikel eine unverwechselbare Bezeichnung besitzt, die mit maximal 15 Buchstaben und/oder Ziffern voll erfaßt werden kann.
- Entnahme und Neuaufnahme
  Das ist die eigentliche Größe, die wir zu verwalten haben. Entnahmen und Neuaufnahmen beeinflussen zunächst einmal die Anzahl der vom betreffenden Artikel gelagerten Exemplare. Damit hängt die nächste Eingabegröße eng zusammen:
- Mindestmenge auf Lager
  Von jedem Artikel muß eine bestimmte Mindestmenge vorhanden sein, um den Betrieb geordnet aufrechtzuerhalten. Wird dieser Wert unterschritten, so muß nachbestellt werden. Wir erwarten, daß uns das Programm irgendwie Auskunft darüber gibt, was wann bestellt werden muß.
- Einzelwert
  Diese Größe brauchen wir für die Übersicht über den Wert, den das Lager repräsentiert. Gelagerte Gegenstände sind brachliegendes Kapital, dessen Größe sich aus ihrer Anzahl und ihrem Einzelwert ergibt.

Ausgaben

Diese Eingabewerte sollen vom Programm so verarbeitet werden, daß wir die Übersicht über den Zustand des Lagers behalten und diesen Zustand in guter Ordnung erhalten können. Das bedeutet, daß wir zumindest zwei verschiedene Ausgaben benötigen:

- eine Lagerübersicht,
  in der alle gelagerten Artikel zusammen mit dem Wert, den sie darstellen, aufgeführt sind;
- eine Nachbestellübersicht,
  in der regelmäßig alle Artikel aufgeführt sind, deren Mindestlagermenge unterschritten wurde.

Diese Übersichten sollen in Form von Listen ausgedruckt werden, welche im ersten Fall die Bezeichnung, die Lager- und Mindestmenge, den Einzel- und den Gesamtwert aufführen. Im zweiten Fall genügen die Bezeichnung und die gegenwärtige Lagermenge zur Information.

Grundaufgaben des Programms

Aus all dem ergeben sich folgende Grundaufgaben, die unser Programm erfüllen muß:

- Neuaufnahme von Artikeln,
  also Einfügen von Name, Mindestmenge und Einzelwert in die Datei, eventuell auch Aufnahme der gerade gegebenen Lagermenge;
- Ändern der Einträge,
  was in erster Linie Entnahmen aus dem und Zugänge in das Lager betrifft, doch sollten auch die übrigen Einträge (einschließlich des Namens) korrigiert werden können;
- Löschen von Einträgen
  ist notwendig, um auslaufende Artikel berücksichtigen zu können und die Datei in übersichtlicher Form zu halten;

- Ausgeben der Gesamtübersicht
  beinhaltet das Berechnen des Lagerwerts und das Ausdrucken der verschiedenen Einträge in den richtigen Spalten;
- Ausgeben der Bestellübersicht
  schließlich umfaßt das Auffinden der Artikel, deren Lagermenge unter den Mindestwert gefallen ist und das Ausdrucken des zugehörigen Eintrags wie oben gefordert.

Überprüfen des Ansatzes

Wenn man mit der Formulierung der Aufgabenstellung so weit gekommen ist, dann läßt man am besten erst einmal alles eine Zeitlang (zumindest über Nacht) liegen und nimmt sich das Ganze später noch einmal vor. Warum? Nun – wir haben hier unsere ersten, noch ziemlich unklaren Vorstellungen formuliert, haben eine erste Ordnung in unsere Gedanken gebracht. Genauso, wie man keinen Menschen nur nach dem ersten Eindruck beurteilen soll, genauso falsch wäre es, würde man von dieser ersten Konzeption an mit dem Ausarbeiten des eigentlichen Programms beginnen. Die Gefahr ist bereits bei so einfachen Aufgabenstellungen wie der unsrigen zu groß, daß man sich in einen Seitenweg verrennt, aus dem man nur mit Mühe (und Kosten) wieder herausfindet. Das Sicherste ist es, erst einmal Abstand zu dem Erarbeiteten zu gewinnen, um es dann später mit neuen Augen betrachten zu können. Es fällt einem dann manches auf, was vorher schlichtweg übersehen wurde – und was unter Umständen umfangreiche Eingriffe in das fertiggestellte Programm erfordern kann, welche die ganze Struktur zerstören und das Programm praktisch unwartbar machen. Noch besser ist es, wenn man den Ansatz mit anderen durchdiskutieren und an alternativen Lösungsmöglichkeiten messen kann (bei umfangreicheren Projekten ist das sogar unumgänglich). Je sorgfältiger dieser erste Schritt bei der Programmerstellung durchgeführt ist, um so leichter und sicherer läßt sich das Programm selbst schreiben.

Sehen wir uns also unseren Ansatz noch einmal sorgfältig

durch: Erfüllt er alle Anforderungen, die wir an ein solches Lagerverwaltungsprogramm stellen müssen? Die Frage gilt auch anders herum: Werden hier nicht auch Probleme angegangen, die man besser auf anderer Ebene, mit einem anderen Programm lösen würde?

Wir können mit unserer gegenwärtigen Konzeption

- Artikel neu aufnehmen,
- Einträge ändern, insbesondere Zu- und Abgänge festhalten,
- Einträge löschen,
- eine Gesamtübersicht und
- eine Bestellübersicht ausgeben.

Wir haben darüber hinaus einige Feinheiten festgelegt, die wir noch genauer betrachten sollten. Dazu untersuchen wir am besten, ob die Endergebnisse des Programms, also die beiden Übersichten, auch wirklich alles ergeben, was gewünscht wird. So umfaßt die vom Programm erarbeitete Gesamtübersicht Informationen über

- die Bezeichnung der Artikel,
- die Lager- und Mindestlagermenge,
- den Einzel- und Gesamtwert.

Das ergibt zwar in der Tat eine Gesamtübersicht über den gegenwärtigen Lagerzustand, sagt aber kaum etwas über die *Bewegungen* im Lager aus, d.h. z.B. darüber, welche Artikel sehr lange im Lager liegen oder was sehr rasch umgeschlagen wird. Eine brauchbare Information in dieser Richtung wäre wichtig, denn wenn ein Artikel sehr lange im Lager liegt, wird man sinnvollerweise seine Mindestmenge herabsetzen, und wird er sehr rasch umgesetzt, so kann man u.U. Kosten sparen, wenn man höhere Mengen am Lager hat. D.h. wir brauchen noch eine Datumsangabe für die letzte Entnahme und das letzte Aufstocken in unserem Datensatz. Schraubt man die Anforderungen nicht gar so hoch, so genügt es, das Datum der letzten Änderung festzuhalten.

Daraus läßt sich auf die eigentliche Lagerbewegung zurückschließen. Diese eine Datumsangabe ist allerdings für eine brauchbare Lagerübersicht unbedingt notwendig.

Ergänzen wir unsere Liste der einzugebenden Größen also noch durch das

- Datum der letzten Änderung.

Wenn wir an der Bewegung der Waren im Lager interessiert sind, dann interessiert zumeist nur, welche Gegenstände in einem bestimmten Zeitrahmen umgeschlagen worden sind. Z.B. ist es sinnvoll, sich nur die Einträge ausdrucken zu lassen, die seit einem bestimmten Datum nicht geändert worden sind, oder die Artikel, die in einem bestimmten Zeitraum (z.B. in der letzten Woche) umgesetzt wurden. Es wäre sehr unpraktisch, müßte man zu diesem Zweck immer die ganze Liste durchforsten. Wir brauchen also noch eine weitere Ausgabemöglichkeit:

- Ausdrucken einer Liste der Einträge, die in einem bestimmten Zeitraum (d.h. zwischen einem Anfangs- und Enddatum) verändert worden sind.

Ganz entsprechende Überlegungen könnte man für die Liste der nachzubestellenden Artikel anstellen. Z.B. wäre es u.U. sinnvoll, eine Mindestbestellmenge anzugeben, oder alle Informationen mit aufzunehmen, durch welche die Bestellungen gleich automatisch aufgegeben werden können (Herstelleradresse, Bestellnummer des Herstellers usw.). Wir wollen hier nicht weiter darauf eingehen. Machen Sie sich aber selbst einmal Gedanken, was man noch sinnvoll mit aufnehmen könnte und welche Aufgaben des Programms das nach sich ziehen würde.

Stellen Sie sich in diesem Zusammenhang auch die umgekehrte Frage: Gehört alles das, was gefordert ist, auch wirklich in das betrachtete Programm, oder stellt man nicht besser ein gesondertes Programm für bestimmte Teilbereiche auf? In unserem Beispiel müßte die Lagerführung beispielsweise mit der Buchhal-

tung verzahnt werden, die wieder ganz eigene Daten- und Programmstrukturen braucht. Und umgekehrt: Wenn ein Programm zur Buchhaltung bereits vorliegt, so muß das Lagerverwaltungsprogramm den dort verwendeten Daten- und Programmstrukturen angepaßt werden.

Programmdokumentation

Sie werden häufig finden, daß Sie bei der Lösung komplexer Probleme den ursprünglichen Ansatz etwas abändern, unter Umständen sogar von Grund auf umformulieren müssen. Machen Sie es sich trotzdem zur Regel, das jeweilige Problem so genau zu analysieren, wie es Ihnen möglich ist, selbst auf die Gefahr hin, daß Sie irgendwann alles umstoßen müssen. In diesem Fall ist ein Neuansatz auf alle Fälle einfacher, als wenn Sie nur drauf los arbeiten würden – immerhin haben Sie das Problem schon eingehend durchdacht und wissen so, *warum* Sie den ursprünglichen Ansatz aufgeben mußten. Und: Halten Sie unbedingt schriftlich fest, was Sie erarbeitet haben. Nichts ist unangenehmer, als irgendwann den Faden zu verlieren (das geschieht leichter, als einem lieb ist) und dann nicht mehr zu wissen, wo man ansetzen soll. Sie haben Ihr Projekt immer dann am besten im Griff, wenn Sie es jederzeit einem anderen übergeben können und dieser nur aufgrund Ihrer Unterlagen in kürzester Frist den Faden dort wieder aufnehmen kann, wo Sie abbrechen mußten.

Diese schriftlichen Unterlagen werden im Lauf der Arbeit an dem Programm alles erfassen, was irgendwie zum Verständnis und zum Programmeinsatz notwendig ist. Man nennt das die *Dokumentation* des Programms. Eine gute Dokumentation erst macht das Programm versteh- und wartbar. D.h., man kann es nur mit ausreichender Begleitinformation versehen richtig einsetzen, abändern und eventuell später auftretende Fehler beseitigen. Eine gute Dokumentation entsteht aber auch gleichzeitig mit dem Programm, wenn alle wichtigen Unterlagen von Anfang an sorg-

fältig formuliert und gesammelt werden. Dann ist die Abschlußdokumentation, die das fertige Programm begleiten soll, nur noch eine Art Reinschrift des bereits Vorhandenen und verliert auf diese Weise viel von ihrem Schrecken für den Programmierer. Wir werden auf dieses Thema immer wieder zurückkommen.

Die benötigten Daten

Ist die Analyse der Problemstellung selbst zu einem gewissen Grade abgeschlossen, so ist es Zeit, sich über die zu verarbeitenden Daten Gedanken zu machen. Drei Fragen spielen hier eine Rolle:

- Welche Daten werden gebraucht?
- Wie werden die Daten dargestellt?
- Wie werden die Beziehungen zwischen den Daten wiedergegeben?

Diese Fragen sind in der Regel durchaus nicht einfach und eindeutig zu beantworten, insbesondere wenn es um größere Datenkomplexe geht, für die unser Lager ein (sehr) einfaches Beispiel abgibt. Schon die erste Frage nach den benötigten Daten wirft einige Probleme auf. Das beginnt mit einer einfachen *Sammlung aller Datenangaben,* die voraussichtlich benötigt werden, in unserem Fall also

- Bezeichnung des Gegenstands,
- gegenwärtige Menge auf Lager,
- Entnahme oder Aufnahme,
- Datum der Änderung,
- Mindestmenge auf Lager,
- Einzelwert und
- Gesamtwert der verschiedenen Artikel sowie
- Gesamtwert des Lagerbestands.

Das läßt sich in verschiedene Gruppen aufteilen:

1. Daten, die für längere Zeit festzuhalten sind:
   - Bezeichnung des Gegenstands
   - Menge auf Lager
   - Datum der Änderung
   - Mindestmenge auf Lager
   - Einzelwert der verschiedenen Artikel
2. Daten, die nur im Verlauf der Bearbeitung benötigt werden
   - Entnahme oder Aufnahme
   - Anfangs- und Enddatum für den Ausdruck eines Lagerauszugs
3. Daten, die hieraus ermittelt werden können:
   - Gesamtwert der verschiedenen Artikel
   - Gesamtwert des Lagerbestands

Damit sind aber nur die vorher ausdrücklich festgelegten Ein- und Ausgabedaten erfaßt. Wenn wir unsere Aufgabenstellung betrachten, dann können wir jetzt schon sagen, daß noch mehr notwendig ist. Woher weiß z.B. der Computer, welchen Programmteil (Einträge ändern, Lagerbestand ausdrucken usw.) er gerade ausführen soll? Wir müssen ihm das schon irgendwie mitteilen, d.h. wir brauchen irgendeine Form von Befehlen, aufgrund deren der gewünschte Programmteil bearbeitet wird. Diese Befehle können wir als eine bestimmte Art von Daten auffassen, die nur im Verlauf der Rechnung selbst benötigt werden, so daß sich als weitere Gruppe ergibt:

4. Daten, die im Einzelfall vorzugeben sind:
   - Befehle zur Auswahl unter den Programmteilen

Dazu werden im Verlauf der Programmformulierung noch einige Daten kommen, die nur Zwischenergebnisse darstellen, die aber auch irgendwo im Rechner festgehalten werden müssen. Unsere Liste, welche die benötigten Daten erfaßt, ist bei weitem noch nicht vollständig.

Darstellung der Daten

Diese Daten haben jeweils ganz bestimmte Formen. So ist z.B. die Warenbezeichnung ein aus Buchstaben, Ziffern und Zeichen bestehender Text, das Datum muß Jahr, Monat und Tag wiedergeben, die Befehle brauchen eine bestimmte Form, und der Rest besteht aus Zahlenwerten. Das alles muß exakt formuliert werden.

Fangen wir mit dem Einfachsten an. Die Größen

- Entnahme und Aufnahme,
- gegenwärtige Menge auf Lager
- Mindestmenge auf Lager
- Einzelwert der verschiedenen Artikel
- Gesamtwert der verschiedenen Artikel
- Gesamtwert des Lagerbestands

liegen alle in Form von Zahlen vor. Ordnen wir sie den folgenden Variablen zu:

M – Menge auf Lager
M1 – Mindestmenge auf Lager
W1 – Einzelwert der verschiedenen Artikel
W – Gesamtwert der verschiedenen Artikel
W0 – Gesamtwert des Lagerbestands

Wir haben diese Variablennamen so gewählt, daß sie (bei etwas Fantasie) einen Aufschluß über das damit Bezeichnete geben. Wenn Sie in Ihrer BASIC-Version längere Variablennamen bilden können, dann sollten Sie sich nicht scheuen, die Variablen so genau wie möglich zu benennen, z.B. durch MENGE, MINDESTMENGE, EINZELWERT, WERT, GESAMTWERT usw. Achten Sie aber darauf, daß die meisten BASIC-Versionen nur die ersten beiden Zeichen auswerten. Die Namen müssen exakt unterscheidbar bleiben. (EINZELMENGE und EINZELWERT z.B. werden von solchen BASIC-Versionen beide zu EI reduziert und bezeichnen somit hier dasselbe.) Auch sind oft der Länge der

Namen Grenzen gesetzt (meist 6 oder 8 Zeichen). Informieren Sie sich genau in Ihrem Handbuch, was für Anforderungen Ihre BASIC-Version an die Variablenbezeichnungen stellt.

Die Datumsangabe bereitet schon größere Schwierigkeiten. Wie soll man z.B. den 30. Oktober 1981 im Computer speichern? Man könnte das beispielsweise getrennt nach Tag, Monat und Jahr tun und drei Variable bereitstellen, doch ist das genau betrachtet eine ziemliche Verschwendung und umständlich zu handhaben. Warum sollte man die Angabe 30. 10. 1981 nicht einfach zu einer Zahl zusammenfassen? Man braucht z.B. nur die Punkte wegzulassen:

30101981

Schön – nur können die meisten BASIC-Versionen so lange Zahlen gar nicht bis zur letzten Stelle genau wiedergeben. In unserer Version sieht das z.B. so aus:

```
>PRINT 30101981
 3.0102E+07
```

Also streichen wir erst einmal die Jahrhundertangabe (19), was ergibt:

```
>PRINT 301081
 301081
```

In dieser Form TTMMJJ (T = Tages-, M = Monats-, J = Jahresziffer) könnten wir die Datumsangaben im Computer als numerischen Wert in einer Variablen festhalten. Doch ist das immer noch unpraktisch. Bedenken Sie, daß wir die Datumsangabe brauchen, um die Eintragungen, die sich in einem bestimmten Zeitintervall geändert haben, ausdrucken zu lassen. D.h., wir brauchen einen einfachen Weg, festzustellen, ob eine Datumsangabe vor oder nach einer bestimmten Grenze liegt. Wenn wir das Datum schon in Form einer Zahl verschlüsseln, so bietet sich an, eine solche Sortierung durch Größenvergleiche vorzunehmen. So ist z.B. die Darstellung für den 30. Oktober 1982 mit

301082

bestimmt größer als die für den 30. Oktober 1981. Für den 1. November 1981 klappt das aber nicht mehr:

11181 ist *kleiner* als 301081

Das Problem liegt darin, daß die Tagesangaben im Datum den *geringsten Wert* besitzen, in unserer Darstellung aber den Platz der Zehn- und Hunderttausenderstelle, also der *höchstwertigen* Ziffern einnehmen. Doch, wer sagt, daß wir das Datum in derselben Reihenfolge verschlüsseln müssen, wie man es schreibt? Stellen wir das Ganze doch einfach um und schreiben beispielsweise für den 30. Oktober 1981:

```
>PRINT 811030
 811030
```

Diese Darstellung in der Form JJMMTT genügt allen unseren Anforderungen – bis auf eine Kleinigkeit: Wir können einem Lagerarbeiter nicht zumuten, das Datum in einer so «verrückten» Form anzugeben. Wir müssen noch das Problem lösen, wie man die Angabe «30. 10. 1981» in die Form «811030» bringt. Halten wir das vorläufig als eine Teilaufgabe des Programms fest:

- Das Programm muß die gewohnte Datumsangabe in die Form JJMMTT bringen.

Wir können unsere Variablenliste jedenfalls um drei weitere Festlegungen erweitern:

D – Datum der letzten Änderung eines Eintrags
D1 – Untere Datumsgrenze beim Ausdruck von Lagerauszügen
D2 – Obere Datumsgrenze beim Ausdruck von Lagerauszügen

Damit kommen wir zum schwierigsten Problem: Wie gibt man die Bezeichnung der Lagerartikel wieder? Man kann es sich einfach machen und jedem Artikel seine eigene Nummer geben,

unter der er in der Datei geführt wird. Doch ist das wirklich sinnvoll? Es ist einfach für den Programmierer, aber es kann eine ständige Fehlerquelle bedeuten. Wer hat schon immer 200 Nummern und mehr im Kopf? Wer vertippt sich nicht einmal bei der Eingabe? Es ist durchaus nicht erfreulich, wenn statt tausend Stück des Pfennig-Artikels 137 tausend Stück des Artikels 173 bestellt werden, der zehnmal so teuer ist. Hätte man statt dessen beispielsweise «SCHRAUBEN M3 X 25» bzw. «PASSSTUECKE XYZ» angeben müssen, so wäre dieser Fehler nicht passiert. Verzichten wir der Sicherheit halber also auf eine Verschlüsselung der Artikelbezeichnungen in Lager- oder Bestellnummern und verwenden statt dessen wie geplant voll ausgeschriebene Namen, wobei wir uns auf maximal 15 Buchstaben, Ziffern oder Zeichen beschränken wollen.

Nur – mit unseren bekannten Mitteln können wir in BASIC nur Zahlen und nichts als Zahlen erfassen und verarbeiten. Wir müssen uns also zunächst die Sprache selbst auf weitere Darstellungsmöglichkeiten hin anzusehen. Dazu ist etwas weiter auszuholen.

3.3 Eingabe über die Tastatur

Wir müssen sogar recht weit ausholen. Mit unseren bisherigen BASIC-Kenntnissen können wir nämlich nur einen – ziemlich kleinen – Teil der anstehenden Aufgaben lösen. Hauptproblem ist zunächst:

Wie bekommen wir die zu verarbeitenden Eingabedaten in den Rechner hinein?

In unserem Beispiel zur Berechnung des Benzinverbrauchs hatten wir einen Sonderfall betrachtet. Hier waren die zu verarbeitenden Daten *vor dem Programmlauf* alle bekannt, wir konnten sie also als Datenliste in Form von DATA-Anweisungen unmittelbar in den Programmtext mit aufnehmen. Das geht jetzt nicht

mehr. In unserer neuen Aufgabenstellung sind die meisten Daten beim Aufstellen des Programms selbst unbekannt. Mehr noch – die Aufgabenstellung besagt ausdrücklich, daß wir solche immer neuen Daten *durch das Programm* aufnehmen und verarbeiten müssen. Mit anderen Worten:

Unser BASIC-Computer nutzt uns nicht viel, wenn wir keine Möglichkeit finden, die zu verarbeitenden Daten je nach Bedarf beim Programmlauf selbst in den Rechner einzugeben. Und das soll in unserem Fall über die Tastatur des Geräts geschehen.

Das Problem hat allgemeine Bedeutung. Bei den wenigsten Problemstellungen ist eine feste Vorgabe (aller) Daten im Programmtext selbst sinnvoll. Man hat also einen Befehl geschaffen, der die *Eingabe* von Daten durch den Benutzer des Programms zum Ziel hat. «Eingabe» heißt auf englisch «input», und so lautet das Befehlswort auch:

INPUT

Das ist allerdings nur die eine Hälfte der Angelegenheit. Was soll eingegeben werden, und wohin soll das gelangen? Für das «Was» stehen prinzipiell zwei verschiedene Möglichkeiten offen: Zahlen oder Texte. Entscheiden wir uns zunächst einmal für Zahlen, denn mit ihnen wissen wir bereits umzugehen. So ist nur noch die Antwort auf die Frage nach dem «Wohin» offen. Nun hat der INPUT-Befehl im Programm ja die gleiche Aufgabe wie der READ-Befehl in unserem ersten Beispiel: Daten sollen übernommen werden. Der Unterschied liegt nur in der Herkunft dieser Daten, der *Datenquelle*. Beim READ-Befehl ist das die im Programm selbst festgehaltene Datenliste, beim INPUT-Befehl ist es das, was der Benutzer bei der Programmabarbeitung über die Tastatur eingibt. Der so gefundene Wert wird dann durch den Befehl einer Variablen – bei Zahleneingaben einer *numerischen Variablen* – zugewiesen. Auf sie beziehen wir uns später im Programm, wenn die Eingabe weiterverarbeitet werden soll. Damit erhält die Anweisung die Form

INPUT x

wobei «x» hier symbolisch für einen der nutzbaren Variablennamen steht, in unserem Fall also für irgend etwas aus dem Bereich zwischen A und Z9.

Probieren wir das erst einmal aus. Zunächst löschen wir den alten Inhalt des Programmspeichers:

```
>NEW
```

und machen dann den folgenden kleinen Versuch:

```
>10 INPUT A
>20 PRINT "Sie haben die Zahl ";A;" eingegeben."
>30 GOTO 10
```

Das soll in der ersten Zeile einen Wert von der Tastatur in die Variable A übernehmen. Die zweite Zeile bestätigt diese Eingabe, und dann läuft das Programm durch den GOTO-Befehl von Zeile 30 in einer Schleife zum Anfang zurück, wo eine neue Zahl übernommen wird.

Was wird wohl geschehen, wenn wir das durch RUN starten? Nun, der Computer wird sich die erste Zeile vornehmen und feststellen, daß er zunächst einen (numerischen) Wert anfordern muß, bevor er weiterarbeiten kann. Also wird er irgendeine Aufforderung ausdrucken, damit wir seine Bereitschaft zur Wertübernahme erkennen können. Erst wenn dieser Wert eingegeben ist, dürfte der Rest des Programms abgearbeitet werden.

Sehen wir uns das einmal an:

```
>RUN
?
```

So, und jetzt wartet er auf einen passenden Wert. Das Fragezeichen dient in den meisten BASIC-Versionen als Anforderung einer Eingabe über die Rechnertastatur. Man findet hin und wieder auch andere Zeichen für diesen Zweck, etwa einen Doppelpunkt, oder man kann selbst etwas Passendes definieren. Das läßt sich dem jeweiligen Handbuch entnehmen.

Tippen wir also eine Zahl ein. (Denken Sie daran, daß diese Eingabe wie üblich durch einen Wagenrücklauf abzuschließen ist.)

```
? 12
Sie haben die Zahl  12  eingegeben.
?
```

Das hat geklappt. Der Rechner hat die Zahl übernommen, der Variablen A zugewiesen und diese Zuweisung wie verlangt nach außen gemeldet. Dann ist er zum Anfang zurückgelaufen und wartet jetzt auf einen neuen Wert.

```
? -24
Sie haben die Zahl -24  eingegeben.
?
```

Negative Zahlen werden also auch richtig übernommen. Wie steht es dann mit Dezimalzahlen? (Denken Sie daran, daß ein Dezimal*punkt* eingegeben werden muß.)

```
? 12.345
Sie haben die Zahl  12.345  eingegeben.
?
```

Und was geschieht beim wissenschaftlichen Datenformat?

```
? 1E10
Sie haben die Zahl  1E+10  eingegeben.
?
```

Jetzt sind Sie an der Reihe! Finden Sie heraus, wie Ihre BASIC-Version in folgenden Grenzfällen reagiert:

- wenn der Zahlenbereich *überschritten* wird,
- wenn der Zahlenbereich *unterschritten* wird,
- wenn Sie nur einen *Punkt* eingeben,
- wenn Sie *gar nichts* (d.h. nur RETURN) eingeben,
- wenn Sie nur einen *Leerschritt* (und natürlich einen Wagenrücklauf RETURN) eingeben,
- wenn Sie ein *Komma* eingeben,
- wenn Sie einen *Text* statt einer Zahl eingeben (wählen Sie etwas Kurzes, z.B. «HALLO»),
- wenn sie *sonst ein Zeichen* eingeben.

Notieren Sie die Ergebnisse (egal, was passiert; falls das Programm mit einer Fehlermeldung «aussteigt», dann starten Sie es einfach neu durch RUN). Vergleichen Sie dann das, was Sie herausbekommen haben, mit dem, was das Handbuch Ihrer BASIC-Version dazu sagt. Der Zweck dieser Übung ist, daß Sie mit Ihrem System so gut wie möglich vertraut werden. Gerade das Verhalten bei der Eingabe ist sehr wichtig für unsere weiteren Aufgaben. Sie sollten so genau wie möglich darüber Bescheid wissen.

Die verschiedenen BASIC-Versionen reagieren höchst unterschiedlich auf diese Grenzfälle. Manchmal wird die Eingabe schlichtweg ignoriert und noch einmal – eventuell mit zwei Fragezeichen – angefordert. Manchmal erhält man eine Fehlermeldung, und das System «steigt aus». In einigen Fällen wird die Abarbeitung insbesondere dann abgebrochen, wenn nur ein Wagenrücklauf (RETURN, ENTER o.ä.) eingegeben worden ist. Einige moderne Versionen drucken bei allen Fehlbedienungen eine Meldung wie

```
?REDO FROM START
?
```

aus und erwarten, daß Sie die Eingabe wiederholen. Das ist zwar sehr freundlich von ihnen, aber leider in Englisch und kann manchmal einiges Kopfzerbrechen bereiten.

Wir werden uns im folgenden um die genaue Form der Reaktion in der Regel nicht sonderlich kümmern. Sie müssen sich das Ergebnis des Versuchs vergegenwärtigen, wenn Ihre Version einmal nicht genau das macht, was wir hier vorführen. Im Zweifelsfall gilt immer wie bisher: Schlagen Sie in Ihrem Handbuch nach.

3.4 Endlose Schleifen; Tips zur Fehlersuche: CONTROL-C, CONT und STOP

Das Beispiel hat eine bemerkenswerte Eigenschaft: Solange kein Eingabefehler vorliegt, arbeitet der Rechner die Schleife immer wieder neu ab. Es gibt keine Möglichkeit, «legal» aus ihr herauszukommen. Man bezeichnet so etwas als *endlose Schleife.* Insbesondere bei den BASIC-Versionen, die Fehleingaben tolerieren und die Anforderung einfach wiederholen, hilft nur das Anhalten des ganzen Computers, um zu einer anderen Arbeit zu kommen.

Endlose Schleifen werden oft bewußt im Programm verwendet, um immer wiederkehrenden Vorgängen gerecht zu werden. Noch häufiger aber entstehen sie aus Versehen, wenn beispielsweise das betrachtete Programm intensiv durch Löschen, Einfügen und Ändern von Programmzeilen ediert worden ist. In einem solchen Fall «geht der Computer durch», und es bleibt nur das Abschalten, um ihn in seinem Amoklauf zu stoppen.

Abschalten aber ist so eine Sache. Viele Computermodelle vergessen dabei das vorliegende Programm, was zumindest einigen Umstand macht, wenn es neu von der Kassette oder Diskette geladen werden muß. Und wehe, wenn man bei umfangreicheren Programmen vergessen hat, sich eine solche Sicherungsaufzeichnung zu machen! Die meisten BASIC-Versionen (zumindest, wenn sie das Programm interpretierend abarbeiten) kennen daher eine Möglichkeit, die Programmabarbeitung jederzeit von außen abzubrechen, ohne den Speicherinhalt zu verändern.

Dazu enthält die Tastatur besondere *Steuertasten,* auf englisch «control keys». Oft werden dazu die normalen Buchstabentasten durch eine besondere Umschalttaste, die sogenannte «CONTROL»-Taste (auch durch «CTRL» abgekürzt), auf solche Steuerfunktionen umgeschaltet. Das funktioniert ähnlich wie die normale Großbuchstabenumschaltung einer Schreibmaschine: Man drückt CONTROL und die betreffende Buchstabentaste *gleichzeitig,* was dem Computer dann den gewünschten Steuerbefehl

übermittelt. Dementsprechend bezeichnet man diese Funktionen oft als CONTROL-A (für gleichzeitiges Bedienen von CONTROL und A-Taste), CONTROL-B, CONTROL-C usw. Wir werden das der Kürze halber bei Bedarf symbolisch durch einen vorgesetzten Aufwärtspfeil vor dem Buchstaben verdeutlichen (diese Schreibweise ist weit verbreitet):

CONTROL-A: ↑A
CONTROL-B: ↑B
CONTROL-C: ↑C

und so weiter.

In den meisten BASIC-Versionen (die mit Normaltastaturen arbeiten) hat man sich auf CONTROL-C als Steuerfunktion zum Programmabbruch geeinigt. (Computer mit Spezialtastaturen besitzen oft eigene Tasten zu diesem Zweck, die z.B. mit BREAK (abbrechen) o.ä. bezeichnet sind.) Die Wirkungsweise läßt sich anschaulich testen, wenn wir in unserem Beispiel den INPUT-Befehl einmal durch eine einfache Zuweisung ersetzen:

```
>10 A = A+1
```

Setzen wir den Wert von A vor der Schleife noch ausdrücklich auf Null (in einigen BASIC-Versionen ist das notwendig; andere machen das automatisch, doch sicher ist sicher):

```
>5 A = 0
```

und starten das Ganze:

```
>RUN
Sie haben die Zahl  1  eingegeben.
Sie haben die Zahl  2  eingegeben.
Sie haben die Zahl  3  eingegeben.
```

Und so geht das immer weiter. Mit CONTROL-C (oder BREAK o.ä. – Ihr Handbuch gibt dazu Auskunft) läßt sich das anhalten.

```
Sie haben die Zahl  4  eingegeben.
↑C
BREAK IN 20
```

Der Rechner hat die Abarbeitung abgebrochen und meldet dies nach außen unter Angabe der Zeile, die gerade abgearbeitet wurde: «BREAK IN 20» bedeutet «Abbruch in Zeile 20».

Bleibt noch nachzuprüfen, ob so das Programm wirklich nicht beeinflußt wurde. Nichts leichter als das, wir starten es einfach neu:

```
>RUN
Sie haben die Zahl  1  eingegeben.
Sie haben die Zahl  2  eingegeben.
Sie haben die Zahl  3  eingegeben.
↑C
BREAK IN 20
```

Der gezielte Programmabbruch durch ↑C hat aber noch eine weitere wichtige Eigenschaft: Normalerweise bleiben dadurch die gerade bestehenden Variablenwerte erhalten. Das ist ein wichtiges Hilfsmittel beim *Austesten* eines Programms. Man kann die Abarbeitung an beliebigen Stellen unterbrechen und sich anhand der Variablenwerte ansehen, was der Rechner gerade tut. In unserem Fall können wir beispielsweise folgenden direkten Befehl eintippen.

```
>PRINT A
 3
```

Das ist gerade der Wert, der als letztes gemeldet wurde. Um Zufälligkeiten auszuschließen, können wir das auch für andere Werte von A untersuchen:

```
Sie haben die Zahl  13  eingegeben.
Sie haben die Zahl  14  eingegeben.
Sie haben die Zahl  15  eingegeben.
↑C
BREAK IN 20
>PRINT A
 15
```

Nachteilig an dieser Methode ist lediglich, daß man nach dem Abbruch nach unseren bisherigen Kenntnissen das Programm jedesmal wieder ganz von vorne an laufen lassen muß. Dabei zeigt

sich der Wert des gezielten Abbruchs vor allem bei der Abarbeitung von längeren Programmen, bei denen sich über Minuten nichts von außen Erkennbares tut. Gerade bei der Programmerprobung stellt sich dann die Frage, ob der Rechner eigentlich das macht, was gemeint ist, oder ob er sich nicht schon längst in einer unendlichen Schleife «aufgehängt» hat. Mit CONTROL-C ließe sich das ganz leicht nachprüfen – könnte man das Programm anschließend an der unterbrochenen Stelle so fortsetzen, als ob nichts geschehen wäre.

Das Problem ist allgemein genug. Fast alle BASIC-Versionen verfügen – soweit es sich um Interpreter handelt – über einen derartigen Fortsetzungsbefehl. «Fortsetzen» heißt im Englischen «to continue», das diese Aktion bewirkende Befehlswort ist in der Regel eine Abkürzung davon:

CONT

Ein Befehlsargument ist nicht notwendig, denn die Stelle, an der die Arbeit aufgenommen werden soll, liegt ja bereits fest.

Wenden wir das auf unser Beispiel an:

```
>RUN
Sie haben die Zahl  1  eingegeben.
Sie haben die Zahl  2  eingegeben.
Sie haben die Zahl  3  eingegeben.
↑C
>PRINT A
 3
>CONT
Sie haben die Zahl  4  eingegeben.
Sie haben die Zahl  5  eingegeben.
```

Wie das Beispiel zeigt, wird die Programmfortsetzung nicht davon beeinflußt, daß wir den Wert einer Variablen – in unserem Fall von A – untersuchen. Wir können sogar noch einen Schritt weitergehen und den Variablenwert nach der Unterbrechung gezielt verändern:

```
>RUN
Sie haben die Zahl  1  eingegeben.
Sie haben die Zahl  2  eingegeben.
Sie haben die Zahl  3  eingegeben.
↑C
>PRINT A
 3
>A=20
>CONT
Sie haben die Zahl  21  eingegeben.
Sie haben die Zahl  22  eingegeben.
 . . .
```

Das ist sehr wichtig für das Austesten komplexer Programme. Man kann sie sich mit dieser Hilfe Stück für Stück vornehmen, geeignete Variablenwerte vorgeben und sich dann nachher ansehen, was daraus geworden ist.

Eine Schwäche hat die Methode aber immer noch: Man ist beim Abbruch mit CONTROL-C darauf angewiesen, daß die gewünschte Stelle im Programm zufällig getroffen wird. (Versuchen Sie einmal, unser Beispiel so anzuhalten, daß gerade Zeile 10 abgearbeitet wird.) In den meisten Fällen will man bei der Fehlersuche das Programmverhalten aber an einer bestimmten Stelle untersuchen oder beeinflussen. Das geht nur, wenn man an dieser Stelle einen Befehl zum Anhalten in das Programm einfügt. Auch dieser Befehl ist in den meisten BASIC-Versionen vorhanden. Er lautet schlichtweg:

STOP

Machen wir auch damit einen Versuch. Fügen wir das vor den Ausdruck von A in das Programm ein:

```
>15 STOP
>RUN
BREAK IN 15
>PRINT A
 1
>CONT
Sie haben die Zahl  1  eingegeben.
BREAK IN 15
>PRINT A
 2
```

und so weiter.

Hin und wieder stört es, daß das Programm *jedesmal* anhält, wenn die Zeile mit dem STOP-Befehl angetroffen wird. Das läßt sich ganz einfach umgehen: Man macht das STOP von einer Bedingung abhängig. Sagen wir, das Programm solle anhalten, wenn A den Wert 5 erreicht hat. Dazu ändern wir Zeile 15 in eine IF-THEN-Anweisung um:

```
>15 IF A=5 THEN STOP
```

Sie erinnern sich: Ist die Bedingung nach IF erfüllt, dann wird der auf THEN folgende Teil der Zeile abgearbeitet, andernfalls macht der Rechner gleich mit der nächsten Zeile weiter. In unserem Fall «sieht» der Rechner den STOP-Befehl also nur dann, wenn A den Wert 5 erreicht hat. In allen anderen Fällen wird die Schleife wie vorher durchlaufen.

Wenn wir nach dieser Änderung allerdings mit CONT das Programm fortsetzen wollen, verweigert der Rechner uns den Dienst:

```
>CONT
?CN ERROR
```

Diese Fehlermeldung besagt «continue error», auf gut Deutsch also «Fehler, die Abarbeitung ist nicht fortsetzbar». Das ist eine Schutzmaßnahme, die immer dann durchgeführt wird, wenn der *Programmtext* nach dem Anhalten verändert worden ist. Dieses Verändern beeinflußt eine ganze Menge interner Vorgänge, die bei einer normalen Programmfortsetzung alle möglichen Fehler in die Arbeit einfließen lassen würden – bis hin zur Zerstörung des Programms im Speicher. Aus diesem Grund ist man gezwungen, das Programm nach jeder Veränderung wieder ganz von vorne laufen zu lassen. (Beachten Sie aber, daß das nur für den Programm*text* selbst, nicht für die vom Programm *erarbeiteten* Werte gilt.)

Wir müssen unser Testprogramm also von vorne an laufen lassen:

```
>RUN
Sie haben die Zahl  1  eingegeben.
Sie haben die Zahl  2  eingegeben.
Sie haben die Zahl  3  eingegeben.
Sie haben die Zahl  4  eingegeben.
BREAK IN 15
>PRINT A
 5
```

Behalten Sie diese Möglichkeiten im Gedächtnis. Sie werden Ihnen große Dienste leisten, wenn größere Programme auszutesten sind. (Nur – vergessen Sie nicht, nach dem Programmtest *alle* derartigen STOP-Befehle wieder herauszunehmen!)

3.5 Anforderungsmeldungen

Wenn der Computer einen INPUT-Befehl im Programm vorfindet, dann meldet er das nach außen, indem er ein Fragezeichen ausdruckt. Nun können wir mit einer derart spartanischen Eingabeanforderung im praktischen Betrieb nicht allzuviel anfangen. Es ist wichtig, daß der Benutzer weiß, was da gerade eingegeben werden soll. Ist es das Datum? Ein Artikelname? Oder die gerade entnommene Menge? All das wird an verschiedenen Stellen im Programm und zu verschiedenen Zeiten angefordert werden. Um klar zu machen, was gerade gebraucht wird, muß die Anforderung also in Form einer ausdrücklichen *Anforderungsmeldung* erfolgen, etwa so:

```
Welches Datum haben wir heute?
Welcher Artikel soll bearbeitet werden?
Wieviel Stueck wurden entnommen?
```

Beachten Sie die Form dieser Anforderungen: Es sind alles Fragesätze. Das führt ganz von selbst zu einer möglichen Programmierung solcher Meldungen. Das Fragezeichen stammt aus dem INPUT-Befehl selbst, und der Text davor kann mit einem

PRINT-Befehl ohne Wagenrücklauf ausgegeben werden. Erweitern wir unser INPUT-Testprogramm z.B. um die ausdrückliche Anforderung:

```
Welche Zahl wollen Sie eingeben?
```

dann kann das so geschehen:

```
>10 PRINT "Welche Zahl wollen Sie eingeben";
>15 INPUT A
```

Vergessen Sie den Strichpunkt nach dem PRINT-Argument nicht. Wenn er fehlt, dann sieht die Anforderung plötzlich so aus:

```
Welche Zahl wollen Sie eingeben
?
```

Einen Computerlaien kann so etwas ziemlich verwirren.

In dieser Form läßt sich die Anforderungsmeldung in fast allen BASIC-Versionen fassen (soweit die Eingabe durch ein Fragezeichen angefordert wird). Man muß sich etwas Kopfzerbrechen machen, um für die gerade erforderliche Eingabe sinnvolle Fragesätze zu finden. Natürlich hätte man beispielsweise im letzten Fall auch formulieren können:

```
Geben Sie eine Zahl ein?
```

oder wenigstens das Fragezeichen in die nächste Zeile setzen:

```
Geben Sie eine Zahl ein.
?
```

Doch das verwirrt den ungeübten Benutzer unnötig. Man sollte sich schon die Mühe machen, die Bedienung so sinnvoll wie möglich zu halten. Und dazu gehört auch eine saubere Textformulierung.

Ganz entsprechendes gilt natürlich auch für die BASIC-Versionen, welche die Eingabe nicht durch ein Fragezeichen, sondern beispielsweise durch einen Doppelpunkt anfordern. In diesem Fall ist natürlich die zweite Fassung unseres Beispiels sinnvoller:

```
Geben Sie eine Zahl ein:
```

Nutzen Sie auf alle Fälle die Möglichkeiten (und die Beschränkungen) Ihrer BASIC-Version voll aus.

Unser Testprogramm hat jetzt folgende Form:

```
>LIST
 10 PRINT "Welche Zahl wollen Sie eingeben";
 15 INPUT A
 20 PRINT "Sie haben die Zahl ";A;" eingegeben."
 30 GOTO 10
```

Wenn wir das genauer betrachten, so ergibt sich, daß in Zeile 10 und 15 ähnliche Funktionen ausgeführt werden: Beidemale wird etwas ausgedruckt, in Zeile 10 der Anforderungstext und in Zeile 15 die eigentliche Eingabeanforderung, ein Fragezeichen. Warum sollte man eigentlich nicht diese beiden Funktionen in einer Zeile zusammenfassen können. Denn wenn schon beim INPUT-Befehl etwas ausgedruckt wird, warum dann nicht gleich eine vollständige Meldung?

Viele der modernen BASIC-Versionen gestatten ein solches Vorgehen. Hier wird in den INPUT-Befehl einfach der Anforderungstext wie ein PRINT-Argument mit aufgenommen:

```
>10 INPUT "Welche Zahl wollen Sie eingeben";A
>15
```

(Zeile 15 muß natürlich gelöscht werden.)

Beachten Sie, daß das Fragezeichen der Meldung vom INPUT-Befehl nach wie vor automatisch eingefügt wird, wovon wir uns rasch überzeugen können:

```
>RUN
Welche Zahl wollen Sie eingeben? 15
Sie haben die Zahl  15  eingegeben.
Welche Zahl wollen Sie eingeben?
```

In einigen BASIC-Versionen kann auch der Strichpunkt zwischen dem Textargument und der Variablen, der die Eingabe zugewiesen werden soll, entfallen, denn es ist aus dem Zusam-

menhang ohnehin deutlich, was gemeint ist. Man muß das wissen, um fremde Programme ohne größere Schwierigkeiten übernehmen zu können. Probieren Sie selbst einmal aus, ob Ihre Version diese Möglichkeit bietet.

3.6 Alphanumerische Variable

Mit Zahleneingaben allein können wir unser Problem allerdings noch nicht lösen. Wir brauchen auch die Möglichkeit, z.B. Artikelnamen über die Tastatur einzugeben oder die Datumsgrenzen für einen Lagerauszug festzulegen. D.h., wir müssen *Texte,* Kombinationen aus Ziffern und Buchstaben und Zeichen, eingeben können. Das ist nicht ohne weiteres möglich, wie wir aus dem Experiment von oben wissen. In vielen Fällen antwortet der Rechner auf die Eingabe von Buchstaben so:

```
Welche Zahl wollen Sie eingeben? X
?TM ERROR IN 10
```

«TM» steht für «type mismatch» und bedeutet sinngemäß «erwartete und vorgefundene Größenart passen nicht zusammen». Nur – was ist das, eine «Größenart», ein «type»?

Wir haben im INPUT-Befehl mit der Variablen A einen *numerischen* Wert vorgeschrieben. Eingegeben haben wir jedoch einen Buchstaben, einen *alphanumerischen* Wert (das ist zusammengezogen aus «alphabetisch» und «numerisch»). Der Rechner hat diese Eingabe offensichtlich zunächst einmal akzeptiert und sie auf ihre Verträglichkeit mit der numerischen Variablen A untersucht. Erst an dieser Stelle kann eine Fehlermeldung TM verursacht worden sein. Das wiederum bedeutet, daß es irgendeine Möglichkeit geben muß, derartige alphanumerische Werte zu verarbeiten (sonst würde man wohl kaum ausdrücklich die Größenart untersuchen).

Das wirft eine ganze Reihe von Fragen auf, z.B. danach, was für eine *Form* derartige alphanumerische Daten für den Compu-

ter (genauer: für die im Computer verwendete BASIC-Version) haben müssen, wie die *Variablen* zur Aufnahme solcher Daten gearbeitet sein müssen, was für *Operationen* mit alphanumerischen Daten möglich sind und endlich, welche *Beziehungen* zwischen alphanumerischen und numerischen Daten bestehen.

Zunächst einmal müssen wir derartige alphanumerische Daten im Rechner irgendwie festhalten können. Das bedeutet, wir müssen wissen, welche formalen Anforderungen an die Variablen zur Aufnahme alphanumerischer Daten zu stellen sind. Wie üblich gibt es hier die verschiedensten Möglichkeiten. Eines ist jedenfalls sicher: Mit der üblichen Bezeichnungsmethode für numerische Variable (d.h. für Variable zur Speicherung numerischer Werte) kommen wir nicht weiter.

Gesetzt den Fall, unsere Schlußfolgerungen aus dem letzten Experiment waren richtig, dann stehen im wesentlichen drei Möglichkeiten zur Auswahl:

1. Bestimmte Variablenbezeichnungen sind für alphanumerische Daten reserviert.
2. Man muß für eine Variable vor der ersten Verwendung besonders angeben, ob sie alphanumerische Daten aufnehmen soll.
3. Es gibt besondere Zeichen, mit denen alphanumerische Variablennamen von numerischen unterschieden werden können.

Der erste Fall ist ziemlich unwahrscheinlich, denn er schränkt die Zahl der zur Aufnahme alphanumerischer Daten geeigneten Variablen stark ein. Auf diese Weise würden die meisten Möglichkeiten wieder verschenkt, die sich bei alphanumerischer Datenverarbeitung ergeben.

Die Möglichkeit, für jede Variable besonders zu *deklarieren,* welche Art von Daten sie aufnehmen soll, hat demgegenüber einiges für sich. In der Tat verwenden die meisten Programmiersprachen dieses Konzept. Es hat den Vorteil, daß man sich beim Programmieren auf die Verarbeitung der Daten selbst konzentrieren kann, ohne sich jedesmal mit dem *Typ* der Variablen, d.h.

der Art der Daten, die sie festhalten kann, herumschlagen zu müssen.

Dennoch hat man für fast alle BASIC-Versionen den dritten Weg gewählt. In erster Linie hat das praktische Gründe: Der Rechner kann bei der üblichen BASIC-Programmabarbeitung einfacher erkennen, was mit einer bestimmten Variablen gemeint ist, wenn ihr Name eine besondere Kennzeichnung trägt. Fordert man eine besondere Variablendeklaration, so muß der Rechner sich für jede verwendete Variable deren Typ merken und ihn bei jeder Verwendung nachschlagen. Das erfordert einigen Aufwand, der bei einer so einfachen Programmiersprache wie BASIC nicht gern gemacht wird. Erst die jüngsten BASIC-Versionen bieten alternativ zur besonderen Variablenkennzeichnung auch die Möglichkeit der Typendeklaration an. Sie können das gegebenenfalls Ihrem Handbuch entnehmen. Dort findet sich auf alle Fälle auch folgende Vorschrift:

Variable zur Aufnahme alphanumerischer Daten werden durch Anfügen eines Dollarzeichens ($) an den sonst wie üblich gebildeten Variablennamen besonders gekennzeichnet.

Zeichenketten (Strings)

Damit stellt sich als nächstes die Frage, was der Begriff «alphanumerische Daten» eigentlich alles umfaßt. Insbesondere müßte geklärt werden, *welche* und *wieviele* Zeichen eine alphanumerische Variable festhalten kann. Das kann von jeweils einem einzigen Zeichen aus einem eingeschränkten Alphabet bis hin zu unbeschränkt vielen aus allen mit einem BASIC-Computer darstellbaren Zeichen reichen.

Versuchen wir erst einmal, so etwas in den Rechner zu bekommen:

```
>NEW
READY
>10 INPUT "Welches Zeichen jetzt";A$
>20 PRINT A$
>30 GOTO 10
```

```
>RUN
Welches Zeichen jetzt? A
A
Welches Zeichen jetzt? B
B
```

und so weiter. Wir sparen es uns hier, die ganze Tastatur durchzugehen. Doch Sie sollten das unbedingt tun, aber immer nur ein Zeichen auf einmal. Notieren Sie sich dabei alle vom Normalen abweichenden Antworten.

Sie werden finden, daß alle Buchstaben und alle Ziffern anstandslos übernommen werden, ebenso der größte Teil der Satzzeichen. In aller Regel werden Sie beim Komma, Doppelpunkt, Anführungszeichen und eventuell auch bei Doppelpunkt und dem umgekehrten Schrägstrich (\, «backslash») entweder gar keine Reaktion oder besondere Meldungen bekommen haben. (Das hängt wiederum stark von der eingesetzten BASIC-Version ab.)

Wie gesagt, notieren Sie sich alle Anomalien. Schlagen Sie aber noch nicht in Ihrem Handbuch nach. Wir wollen erst versuchen, noch etwas mehr über die Grundlagen herauszufinden. Ganz wichtig ist z.B. die Frage, was geschieht, wenn wir mehrere Zeichen eingeben:

```
Welches Zeichen jetzt? ABCDEFGH
ABCDEFGH
Welches Zeichen jetzt? 1234567890
1234567890
```

Offensichtlich lassen sich nicht nur Einzelzeichen, sondern ganze *Zeichenketten* (sogenannte «strings») im Rechner festhalten. Wichtig ist weiter, daß auch Ziffern übernommen werden können. Das wirft allerdings sofort die Frage auf, ob so eine Eingabe auch als Zahl verstanden wird. Wir können das nachprüfen, indem wir versuchen, mit so einer Variablen zu rechnen. Stoppen wir daher den Rechner durch CONTROL-C (↑C) oder sonst eine angebrachte Methode und fügen wir eine passende Zeile ein:

```
Welches Zeichen jetzt? ↑C
READY
>25 PRINT 2*A$
```

Wenn der Rechner eine Zifferneingabe in A$ als Zahl versteht, so ist anzunehmen, daß er hier das Rechenergebnis ausdruckt. Sehen wir uns das an:

```
>RUN
Welches Zeichen jetzt? 123
123
?TM ERROR IN 25
```

Das kennen wir bereits. Ganz offensichtlich nimmt der Rechner die in A$ übernommenen Ziffern nicht – jedenfalls nicht ohne weiteres – als Zahl an. Das heißt allerdings nicht, daß man eine solche Eingabe prinzipiell nicht als Zahl betrachten könnte. Möglicherweise gibt es Mittel und Wege, eine solche Zeichenkette in eine Zahl umzuwandeln. Wir werden das später genauer untersuchen müssen. Wichtig wäre es allerdings zu wissen, ob man *alle* vom Rechner numerisch verarbeiteten Zahlendarstellungen auch als «String» eingeben kann, in erster Linie also solche mit Dezimalpunkt und solche in Exponentialnotation. Wenn das möglich ist, und wenn man irgendwie die alphanumerische «Stringvariable» in eine numerische umwandeln, d.h. dem Rechner irgendwie den gemeinten Zahlenwert verfügbar machen kann, dann haben wir ein wichtiges Hilfsmittel gefunden, Fehlbedienungen erkennen und korrigieren zu können.

```
>25
>RUN
Welches Zeichen jetzt? 123.45
123.45
Welches Zeichen jetzt? -123.45E-12
-123.45E-12
```

Damit haben wir im wesentlichen folgendes gefunden:

Man kann in den Rechner durch mit einem Dollarzeichen bezeichnete «Stringvariable» (daher das «$»-Zeichen; A$ liest sich als «A-String») mit Ausnahme einiger Sonderzeichen beliebige Zeichen-

ketten ein- und diese durch einen einfachen PRINT-Befehl wieder ausgeben lassen. Numerische und Zeichenkettenwerte müssen allerdings streng getrennt voneinander behandelt werden. Insbesondere kann man einer numerischen Variablen keinen alphanumerischen Wert und einer Stringvariablen keinen (unmittelbar) numerischen Wert zuweisen.

3.7 Eine Übersichtsplanung

Nachdem wir so eines der wichtigsten Werkzeuge zur Lösung unseres Problems erarbeitet haben, ist es an der Zeit, sich ein wenig um die Systematik der Aufgabe zu kümmern. Wir müssen das Problem dergestalt *strukturieren,* daß wir das eigentliche Programm ohne weitere größere Schwierigkeiten erstellen können. Vergegenwärtigen wir uns dazu zunächst noch einmal die Aufgabenstellung:

Es ist ein Programm zur Verwaltung eines kleinen Lagers, das ungefähr 200 Artikel umfaßt, mit folgenden Eigenschaften zu erstellen:

a) Der aktuelle Lagerbestand muß durch einfache Angabe der entnommenen bzw. hinzugefügten Artikel und Artikelmengen auf dem laufenden gehalten werden, wobei nachzubestellende Artikel rechtzeitig gemeldet werden.
b) Der jeweilige Lagerbestand muß in Form von Gesamt- und terminlich eingegrenzten Übersichten wiedergegeben werden können.

Das Problem läßt sich damit in zwei verschiedene Komplexe zerlegen, die *Lagerführung* und die *Lagerübersicht.* In einer Strukturbeschreibung kann das versuchsweise so gefaßt werden:

```
Lagerverwaltung:
  Lagerführung,
  Lagerübersicht.
Ende.
```

Nur – dieser Ansatz beschreibt nicht genau das, was das Programm leisten soll. Betrachten wir das Problem genauer, so ergibt sich, daß die beiden Aufgabenkomplexe Lagerführung und Lagerübersicht unabhängig voneinander bearbeitet werden sollten. Es handelt sich im Grunde genommen um zwei verschiedene eigenständige Programme, die lediglich mit denselben Daten arbeiten. Unsere Strukturbeschreibung sagt aber etwas anderes aus, nämlich:

Im Rahmen des Programms «Lagerverwaltung» wird *zuerst* der Komplex «Lagerführung» und *danach* der Komplex «Lagerverwaltung» abgearbeitet – und zwar bei jedem Programmaufruf durch RUN.

Die beiden Grundaufgaben hängen hier also zeitlich voneinander ab, und sie werden jedesmal alle beide abgearbeitet. Mit anderen Worten: Wenn wir eine Lagerübersicht haben wollen, dann müssen nach dieser Struktur erst Maßnahmen zur Lagerführung getroffen werden. Und wenn wir nur einen Artikel im Lager verändern wollen, dann geraten wir anschließend immer in das Programm für die Lagerübersicht.

Wir müssen eine andere Beschreibungsform finden, die uns zum einen die Zusammenfassung in einem einzigen Programm «Lagerverwaltung» bewahrt, andererseits aber die Unabhängigkeit der beiden Teilaufgaben gewährleistet. Das Problem liegt darin, daß der Rechner nach dieser Strukturierung noch nicht weiß, ob er nun den ersten oder den zweiten Aufgabenkomplex abarbeiten soll.

Eine Möglichkeit wäre beispielsweise, wenn wir jedem Teilprogramm eine ausdrückliche Anfrage vorausstellen, ob das Programm abgearbeitet werden soll, etwa so:

```
Lagerverwaltung:
  Lagerführung:
    «Ist der Lagerbestand zu verändern?»,
    Wenn «Ja», dann:
      Lagerführung abarbeiten.
```

Lagerübersicht:
«Ist eine Lagerübersicht auszugeben?»,
Wenn «Ja», dann:
Lagerübersicht abarbeiten.
Ende.

Sind nur relativ wenige Alternativen gegeben, wie in unserem Fall, dann ist das durchaus sinnvoll. Anders aber, wenn das Programm mehr unabhängige Teilaufgaben umfaßt. Es ist sehr umständlich, wenn man unter zehn Möglichkeiten auf diese Weise die letzte auswählen will. Und es ist recht fehlerintensiv: Gar zu leicht tippt man auf die Anfrage nach dem gewünschten Programmteil auch – aus lauter Routine – ein «Nein» ein. Und dann kann man wieder ganz von vorne anfangen. Es muß also noch eine andere Möglichkeit der Programmauswahl geben.

Die Menütechnik

Nehmen wir doch einfach die «Auswahl» ernst. Stellen wir dem Programm einen Katalog voran, anhand dessen der Benutzer bestimmen kann, mit welchem der unabhängigen Programmteile er gerade arbeiten möchte. In unserem Beispiel könnte am Programmanfang folgende Meldung auf dem Schirm erscheinen:

```
Sie haben folgende Moeglichkeiten:
   1. Veraendern des Lagerbestands
   2. Ausgeben einer Lageruebersicht

Welche Moeglichkeit wuenschen Sie (1 oder 2)?
```

Der Benutzer kann daraufhin eine Eins oder eine Zwei eintippen, je nach Aufgabe, die er bearbeiten möchte. Bei einer Eins leitet der Rechner das Programm zur Lagerführung ein, bei einer Zwei ist die Lagerübersicht abzuarbeiten.

Kennzeichen dieser Methode ist, daß dem Benutzer die gegebenen Alternativen wie auf einer Speisekarte angeboten werden. Er braucht lediglich auszuwählen, was er haben möchte. Aus diesem Grund bezeichnet man diesen Ansatz als *Menütechnik*.

In der Strukturbeschreibung unseres Problems läßt sich das so formulieren:

```
Lagerverwaltung:
  Menü anbieten,
  Auswahl anfordern,
  Wenn 1 ausgewählt, dann:
    Lagerführung.
  Sonst:
  Wenn 2 ausgewählt, dann:
    Lagerübersicht.
Ende.
```

Das erweitern wir sinnvollerweise gleich ein wenig. Wir haben nicht berücksichtigt, daß für das Beenden des Programms einige Zusatzarbeit notwendig werden kann (so müssen auf alle Fälle die Daten geeignet außerhalb des Computers – z.B. auf Band – abgespeichert werden). Das muß mit festgehalten werden:

```
Lagerverwaltung:
  Menü anbieten,
  Auswahl anfordern,
  Wenn 1 ausgewählt, dann:
    Lagerführung.
  Sonst:
  Wenn 2 ausgewählt, dann:
    Lagerübersicht.
  Programm beenden.
Ende.
```

In dieser Form wiederum ist das nicht sehr sinnvoll, denn es kann vorkommen, daß wir nur aus Versehen aus dem Teil zur Lagerführung ausgestiegen sind oder daß nach der Lagerübersicht gleich Änderungen im Lagerbestand vorgenommen werden sollen oder daß zwischendurch eine Übersicht über den Lagerbestand benötigt wird. In all diesen Fällen lohnt sich der Umstand einer Aufzeichnung der Daten nicht unbedingt. Im Gegenteil

kommt es darauf an, so schnell wie möglich mit der eigentlichen Arbeit weitermachen zu können. Wir nehmen daher am besten die zum Beenden des Programms notwendigen Arbeiten als eigenen Menüpunkt auf und bilden das Lagerverwaltungsprogramm selbst als Schleife aus:

```
Lagerverwaltung:
  Solange Programm nicht beendet:
    Menü anbieten,
    Auswahl anfordern,
    Wenn 1 ausgewählt, dann:
      Lagerführung.
    Sonst:
    Wenn 2 ausgewählt, dann:
      Lagerübersicht.
    Sonst:
    Wenn 3 ausgewählt, dann:
      Programm beenden.
  Schleifenende (Programm beendet).
Ende.
```

Noch eine Ergänzung wäre sinnvoll. Der Programmteil zur Lagerführung ermittelt als «Nebenprodukt», ob die Mindestlagermenge für einen gegebenen Artikel unterschritten ist. Wenn ja, dann kann eine passende Nachbestellmeldung ausgegeben werden. Dies geschieht sinnvollerweise aber erst nach Abschluß der Lagerführungsarbeiten, so daß wir den Fall 1 wie folgt erweitern sollten:

```
Wenn 1 ausgewählt, dann:
  Lagerführung,
  Nachbestellmeldung ausgeben.
```

Diese Menütechnik läßt sich auf jeder Programmebene einsetzen. Nehmen wir an, wir hätten die Lagerführung gewählt. Dann stehen uns wieder mehrere Möglichkeiten offen, z.B.:

1. Artikel neu aufnehmen
2. Artikel ändern
3. Bestand verändern

Und wenn wir hier Fall 3 wählen, dann haben wir noch einmal eine Auswahlmöglichkeit:

1. Exemplare entnehmen
2. Exemplare hinzufügen

In der Strukturbeschreibung würde sich das so niederschlagen:

```
Lagerverwaltung:
  Solange Programm nicht beendet:
    Menü «Lagerverwaltung» anbieten,
    Auswahl anfordern,
    Wenn 1 ausgewählt, dann:
      Lagerführung:
        Menü «Lagerführung» anbieten,
        Auswahl anfordern,
        Wenn 1 ausgewählt, dann:
          Artikel neu aufnehmen.
        Sonst:
        Wenn 2 ausgewählt, dann:
          Artikel ändern.
        Sonst:
        Wenn 3 ausgewählt, dann:
          Bestand verändern:
            Menü «Bestand verändern»,
            Auswahl anfordern,
            Wenn 1 ausgewählt, dann:
              Exemplare entnehmen.
            Sonst:
            Wenn 2 ausgewählt, dann:
              Exemplare einfügen.
          Ende (Bestand verändern).
```

```
        Ende (Lagerführung),
        Nachbestellmeldung.
    Sonst:
    Wenn 2 ausgewählt, dann:
        Lagerübersicht.
    Sonst:
    Wenn 3 ausgewählt, dann:
        Programm beenden.
Ende.
```

Wir haben hier den verschiedenen Menüs eigene Titel gegeben, die sinnvollverweise mit auf dem Bildschirm erscheinen sollten. Der Benutzer weiß dann auch in komplizierten Programmen mit vielen derartigen Verzweigungen immer, wo er sich gerade befindet.

3.8 Unterprogramme: GOSUB und RETURN

Wie verwirklicht man nun so etwas? Sehen wir uns dazu zunächst den einfachsten Fall an:

```
Lagerverwaltung:
  Solange Programm nicht beendet:
    Menü anbieten,
    Auswahl anfordern,
    Wenn 1 ausgewählt, dann:
      Lagerführung,
      Nachbestellmeldung.
    Sonst:
    Wenn 2 ausgewählt, dann:
      Lagerübersicht.
    Sonst:
    Wenn 3 ausgewählt, dann:
      Programm beenden.
  Schleifenende (Programm beendet).
Ende.
```

Naheliegend wäre, das so, wie es da steht, hintereinander weg in BASIC-Anweisungen zu fassen. Doch – ist das sinnvoll? Je Aufgabenkomplex (Lagerführung, Nachbestellmeldung oder Lagerübersicht) werden möglicherweise ein paar hundert Programmzeilen notwendig, ausgelistet also mehrere Seiten Papier. Ganz grob skizziert würde das Programm z.B. so aussehen (mit A als Auswahlvariable, «< >» steht in BASIC für «ungleich»):

```
100 (Menü anbieten und Auswahl in A übernehmen)
 . . .
1000 IF A<>1 THEN GOTO 5000
 . . .
(mehrere Seiten Lagerführungsprogramm)
(mehrere Seiten Nachbestellmeldung)
 . . .
5000 IF A<>2 THEN GOTO 10000
 . . .
(mehrere Seiten Übersichtsausgabe)
 . . .
10000 IF A<>3 THEN GOTO 100
 . . .
(Programm beenden)
 . . .
```

Das funktioniert zwar, aber unsere mühsam erarbeitete Strukturbeschreibung ist beim Programmieren zum Teufel gegangen mit dem Ergebnis, daß das Programm selbst schwerer lesbar wird, denn die Fallunterscheidung aufgrund der Menüauswahl ist auf das ganze Programm verstreut. Woher soll man nach mehreren Seiten Programmtext noch wissen, was eigentlich mit der Variablen A gemeint war? Wollen wir die Übersicht bewahren, dann müssen wir die Entscheidung anders formulieren. Anstatt sie in der Aufeinanderfolge von Teilprogrammen zu begraben, sollten wir sie in einem eigenen Programmteil zusammenfassen, wie es durch die Strukturbeschreibung vorformuliert ist:

Lagerverwaltung:
  Solange Programm nicht beendet:
    Menü anbieten,
    Auswahl anfordern,

```
    Wenn 1 ausgewählt, dann:
      (Lagerführung),
      (Nachbestellmeldung).
    Sonst:
    Wenn 2 ausgewählt, dann:
      (Lagerübersicht).
    Sonst:
    Wenn 3 ausgewählt, dann:
      (Programm beenden).
  Schleifenende (Programm beendet).
Ende.
```

Wir haben hier die Teilaufgaben in Klammern eingeschlossen, um deutlich zu machen, daß es sich nicht um die Arbeiten selbst handelt, sondern nur um die *Steuerung* der Abarbeitung. «(Lagerführung)» ist von dem Programmteil zu unterscheiden, in dem die Lagerführung tatsächlich erfolgt. «(Lagerführung)» stellt eine Art Superbefehl dar, eine Anweisung innerhalb der Auswertung der Benutzerantwort auf das Menüangebot, aufgrund deren irgendwo sonst im Programm der Teil zur Lagerführung selbst abgearbeitet wird. Sinngemäß soll hier nämlich folgendes geschehen:

- Der Rechner kommt bei der Auswertung einer Entscheidung (IF A=1 THEN ...) zu dieser Stelle und findet «(Lagerführung)» zusammen mit der Angabe, wo er das zugehörige Teilprogramm suchen soll, vor.
- Er springt zu diesem «Unterprogramm» und führt die dort vorgeschriebenen Arbeiten aus.
- Mit dieser Aufgabe fertig, springt er zur Ablaufsteuerung zurück und sieht sich an, welche Aufgabe als nächste kommt.

Mit anderen Worten: In der Menüauswertung, der *Ablaufsteuerung* des Programms, werden die jeweiligen Teilaufgaben als selbständige Untereinheiten des Gesamtprogramms lediglich

«aufgerufen». Es fällt somit in der Schleife selbst an Arbeit nur an, was für einen geordneten Ablauf des Programms notwendig ist, sonst nichts. Will man sich einen raschen Überblick über die Programmstruktur, über den Zusammenhang der verschiedenen Teilaufgaben verschaffen, so braucht man sich nur noch die Steuerschleife anzusehen. Das ist wesentlich angenehmer und erfolgversprechender, als wenn man sich durch das ganze Programm hindurchwühlen müßte.

Wie lassen sich diese Überlegungen nun in die Praxis umsetzen? Nehmen wir an, der Programmteil «Lagerführung» würde von Zeile 1000 bis Zeile 3500 reichen. Dann hätten wir in der Ablaufsteuerung (der Menüauswertung) beispielsweise stehen:

```
300 GOTO 1000
310 (nächster Befehl der Ablaufsteuerung)
```

In Zeile 1000 würde die Lagerführung beginnen und in Zeile 3500 mit einem Sprung zum nächsten Befehl in der Ablaufsteuerung abgeschlossen:

```
3500 GOTO 310
```

Auf den ersten Blick ist das ganz akzeptabel. Doch – was ist, wenn wir aus irgendeinem Grund in der Ablaufsteuerung den Unterprogrammaufruf verschieben müssen, z.B. so:

```
310 GOTO 1000
311 (nächster Befehl der Ablaufsteuerung)
```

Dann muß auf alle Fälle der Rücksprungbefehl im aufgerufenen Unterprogramm mit geändert werden:

```
3500 GOTO 311
```

Solange das Programm so einfach und übersichtlich wie in unserer bisherigen Annahme aufgebaut ist, macht das keine

Schwierigkeiten. Anders sieht es aus, wenn ein fix und fertiges Programm – möglicherweise gar nicht von uns selbst erstellt – nach einer derartigen Änderung des Programmtexts auf die neuen Zeilennummern umgestellt werden muß. Es gibt keine Garantie dafür, daß das betrachtete Unterprogramm nur einen einzigen Rücksprungbefehl hat. Irgendwo könnte ein von einer bestimmten Bedingung abhängiger Rücksprung stehen. Und nun suchen Sie mal in einem dicht gedrängten Programmausdruck von mehreren hundert Zeilen, das Sie womöglich noch gar nicht bis ins letzte Detail verstanden (oder es wieder vergessen) haben! Es ist fast unausweichlich, daß dann irgendwo eine solche Rücksprunganweisung übersehen wurde – unter Umständen macht sich das erst nach ein paar Monaten mit einem völlig unerklärlichen Fehlverhalten bemerkbar.

Ein anderer Punkt kommt noch dazu. Oft wird dieselbe Aufgabe in einem Programm an verschiedenen Stellen benötigt. Beispielsweise müssen bei der Auswertung von angloamerikanischen Unterlagen bestimmte Werte häufig ins metrische System umgerechnet werden. Dazu sind jedesmal ein paar wenige Programmzeilen nötig. Wenn das zwei- oder dreimal auftritt, so mag es nicht sonderlich stören, sonst aber summieren sich die paar Zeilen ganz schön auf. Außerdem wird der eigentliche Rechengang durch die ständigen Umrechnungseinschübe unübersichtlich.

Abhilfe: Wir lagern die Umrechnungsarbeiten in ein Unterprogramm aus, das von Programmzeile 4000 bis 4050 reichen möge. Nehmen wir nun an, der erste Aufruf dieses Unterprogramms käme von Zeile 1000, was folgende Situation ergäbe:

```
1000 GOTO 4000
1010 (nächste Programmzeile)
 . . .
 . . .
4000 (Beginn der Umrechnung)
 . . .
4050 GOTO 1010
```

So weit, so gut. Doch hier hört das Programm nicht auf. Sagen wir, die nächste Umrechnung sei in Zeile 2000 fällig:

```
1000 GOTO 4000
1010 (nächste Programmzeile)
 . . .
2000 GOTO 4000
2010 (nächste Programmzeile)
 . . .
 . . .
4000 (Beginn der Umrechnung)
 . . .
4050 GOTO 1010
```

Das hat seinen ganz eigenen Effekt: Sprung von 1000 zu 4000 – Umrechnung – Sprung von 4050 zu 1010 – Abarbeitung des Hauptprogramms – Sprung von 2000 zu 4000 – Umrechnung – Sprung von 4050 zu 1010 ... Eine perfekte endlose Schleife! *So geht es jedenfalls nicht.*

Damit sind wir etwas in der Klemme. Auf der einen Seite soll das Programm übersichtlich gegliedert und so kurz wie möglich sein, auf der anderen Seite wären wir gezwungen, für jeden Aufruf ein Unterprogramm neu aufzuschreiben, nur weil sich dessen letzter Befehl, «GOTO xxxx», von Aufruf zu Aufruf ändert. Ja, könnten wir Variable als Zieladressen verwenden, etwa «GOTO X» schreiben, was einen Sprung zu der Zeile bewirken würde, die durch den Wert von X gegeben ist, dann wäre die Sache schon eher machbar:

```
1000 X=1010 : GOTO 4000
1010 (nächste Programmzeile)
 . . .
 . . .
2000 X=2010 : GOTO 4000
2010 (nächste Programmzeile)
 . . .
 . . .
4000 (Beginn der Umrechnung)
 . . .
4050 GOTO X
```

In diesem Fall übergeben wir dem Unterprogramm in der «Rücksprungvariablen» X einfach bei jedem Aufruf die Zeilennummer, zu der es zurückzukehren hat, und alle unsere Sorgen sind vorbei.

Oder nicht? Leider hat die Sache einen kleinen Haken: Wir können aus gutem Grund keine Variable als Sprungziel angeben. Die Möglichkeit eines Programmierfehlers ist einfach viel zu groß. Nur einmal der Rücksprungvariablen durch einen Tippfehler oder durch irgendwelche Nebenwirkungen anderer Rechnungen einen falschen Wert geben, und die ganze Arbeit bis dahin ist zum Teufel.

Man hat daher, wie wir früher schon einmal besprochen hatten, ausschließlich ganzzahlige Konstanten, eben die ausdrückliche Angabe der betreffenden Zeilennummer, als Sprungziel in einem GOTO-Befehl zugelassen. Andererseits aber ist eine derartige Unterprogrammtechnik ein viel zu leistungsfähiges Werkzeug, um die Möglichkeit dazu einfach unter den Tisch fallen zu lassen. Also hat man etwas für BASIC Typisches gemacht: Man definierte einen eigenen Befehl dafür. Anstatt nur «Springe zu Zeile xxxx» zu sagen, faßt man sich genauer: «Springe zu dem Unterprogramm in Zeile xxxx». «Unterprogramm» heißt auf Englisch «subroutine», so daß man eigentlich sagen müßte:

GOTO SUBROUTINE xxxx

was natürlich viel zu lang ist. Man hat sich daher auf die Kurzform:

GOSUB xxxx

geeinigt.

«GOSUB xxxx» unterscheidet sich von «GOTO xxxx» nur in einem Punkt: Der Rechner merkt sich, woher der Sprung zum Unterprogramm kam, tut also gerade das, was wir oben mit Hilfe der Variablen X versucht hatten. Damit wäre die erste Hälfte unseres Problems gelöst. Bleibt nur noch ein Weg zu finden, dem Rechner klarzumachen, wann das Unterprogramm beendet ist

und er zum aufrufenden Programm zurückzukehren hat. Nun – man nimmt einfach einen weiteren Befehl zu Hilfe:

Kehre zum aufrufenden Programm zurück!

kurz:

Kehre zurück!

was auf Englisch

RETURN

ergibt.

GOSUB und RETURN gehören zusammen mit GOTO zur Gruppe der Steuerbefehle. Ein Steuerbefehl dient dazu, den üblichen linearen Programmablauf zu ändern. Das hatten wir oben bereits besprochen. Nun hat das Paar GOSUB – RETURN gegenüber dem einfachen GOTO den großen Vorteil, daß man mit seiner Hilfe das Programm viel bequemer strukturieren kann. Wir hatten es ja eben entwickelt: Jede Grundaufgabe des Programms wird getrennt für sich behandelt. Die Verbindungen zwischen den einzelnen Programmteilen erfolgt nur über die Daten, mit denen gearbeitet wird. Die Teile selbst sollten dabei so in sich abgeschlossen sein, daß kein GOTO-Sprung von einem Teilprogramm zum anderen notwendig wird. Das geht – wie wissenschaftlich nachgewiesen ist – bei allen Problemstellungen zu machen.

Braucht man die Ergebnisse eines anderen Programmteils für eine bestimmte Teilaufgabe, so ruft man ihn einfach als Unterprogramm auf. Ja, es gibt Fälle, in denen sich ein Unterprogramm für bestimmte Aufgaben selbst aufruft (sogenannte *rekursive Programmierung*). Allerdings ist hier in BASIC einige Vorsicht geboten. Auf alle Fälle aber kehrt die Abarbeitung mit absoluter Sicherheit wieder zum aufrufenden Programm zurück, was – wie wir eben gesehen haben – bei einfachen GOTO-Befehlen nicht so ohne weiteres der Fall ist.

Noch einen Vorteil bietet diese Unterprogrammtechnik gegenüber dem vielleicht zunächst naheliegenderen Verketten der verschiedenen Programmteile durch einfache GOTO-Sprünge: Man kann Einzelheiten viel leichter ändern. Jedem Unterprogramm, das mit GOSUB – RETURN abgearbeitet wird, entspricht in unserer Strukturbeschreibung eine Gliederungsebene. Muß in einem Teil die Abarbeitung geändert werden, etwa weil sich die Aufgabenstellung geändert hat, so braucht man nur dieses eine Stück umzuschreiben; muß also höchstens noch die Adresse im GOSUB-Befehl ändern. Das ist alles. Und nun stellen Sie sich einmal vor, welcher Aufwand notwendig wäre, wollte man bei einer intensiven Verkettung des Programms allen GOTO-Befehlen folgen – einmal ganz abgesehen davon, daß in so einem Fall die Änderung eines Programmteils völlig unerwartete Fehler in einem anderen erzeugen kann (und in der Praxis auch erzeugt!), der mit ihm eng verkettet ist. Es braucht z.B. für eine Teilaufgabe nur eine Addition in eine Subtraktion verwandelt zu werden, weil das die geänderte Aufgabenstellung für einen Programmabschnitt so verlangt. Da aber auch noch andere Programmteile unmittelbar darauf beruhen – und es gar nicht so einfach zu ersehen ist, welcher Teil in einem solchen monolithischen Block mit welchem anderen verbunden ist – macht das Ganze nach der Änderung Mist (wenn Sie Pech haben, noch dazu nur sporadisch, ein- oder zweimal die Woche). Dann suchen Sie mal schön ...

Und noch ein drittes kommt hinzu: Da man jedes Programmstück als völlig in sich geschlossene Einheit auffassen kann, ist ein viel intensiveres und zuverlässigeres Austesten des Programms möglich. Man beginnt bei der eigentlichen Kodierung am besten mit den Programmteilen der tiefsten Schachtelebenen und prüft diese Unterprogramme zunächst einmal auf Herz und Nieren. Erst wenn man sich ihrer korrekten Funktion sicher sein kann, lohnt es sich, zur nächsten Ebene überzugehen. Hier benutzt man die vorher ausgetesteten Programmteile mittels «GOSUB xxxx» wie einen eigenständigen Befehl. Arbeiten sie wirklich einwandfrei (und stimmt der logische Aufbau des Gesamtprogramms),

dann braucht man sich nicht mehr um sie zu kümmern. Genauso verfährt man mit den höheren Schachtelebenen, bis man zum Schluß ein Programm stehen hat, auf das man sich – beinahe – hundertprozentig verlassen kann. Mit anderen Worten: Unterprogramme bieten sich ganz natürlich zur «Bottom-up-Kodierung» an, die wir weiter oben angesprochen hatten.

3.9 Dokumentieren: REM

Durch GOSUB und RETURN ist das Verwenden der Unterprogrammtechnik zur Lösung von Problemen mit BASIC möglich, ein beträchtlicher Schritt vorwärts zu übersichtlicheren, rasch verstehbaren Programmen. Doch damit ist nur ein Teil des hierfür Notwendigen erreicht. Sehen wir uns einmal an, wie der folgende Ausschnitt aus unserem Problem programmiert werden könnte:

```
Lagerverwaltung:
    Solange Programm nicht beendet:
        Menüauswahl,
        Wenn 1 ausgewählt, dann:
            Lagerführung,
            Nachbestellmeldung.
        Sonst:
        Wenn 2 ausgewählt, dann:
            Lagerübersicht.
        Sonst:
        Wenn 3 ausgewählt, dann:
            Programm beenden.
    Schleifenende (Programm beendet).
Ende.
```

Dabei haben wir unter «Menüauswahl» das Anbieten des betreffenden Menüs und die Auswahlanforderung zusammengefaßt. Die Auswahl eines Programmteils durch den Benutzer sei in

der Variablen A festgehalten. Setzen wir ferner voraus, die Unterprogramme seien bereits kodiert und würden an folgenden Zeilen beginnen:

```
500 (Menüauswahl)
1000 (Lagerführung)
3500 (Nachbestellmeldung)
5000 (Lagerübersicht)
10000 (Programm beenden
```

Die zugehörigen Anweisungen der Ablaufsteuerung mögen mit Zeile 100 beginnen. Mit diesen Informationen können wir uns an eine erste Formulierung wagen:

```
100 GOSUB 500
110 IF A<>1 THEN GOTO 150
120 GOSUB 1000
130 GOSUB 3500
140 GOTO 100
150 IF A<>2 THEN GOTO 180
160 GOSUB 5000
170 GOTO 100
180 IF A<>3 THEN GOTO 100
190 GOTO 10000
```

Hand aufs Herz: Haben Sie auf Anhieb verstanden, was da vorgeht? Das Programm ist zwar kurz und bündig, aber sonderlich verständlich ist es nicht. Was bedeuten z.B. die Sprungziele 500, 1000, 3500, 5000 oder 10000, wenn man gerade dabei ist, sich mit dem Programm vertraut zu machen? Zunächst einmal sind das böhmische Dörfer. Ihre Bedeutung läßt sich erst durch intensive Lektüre der anzuspringenden Unterprogramme – hoffentlich – herausfinden. Das ist nicht nur sehr mühsam, es ist auch unhaltbar, wenn man bedenkt, daß ein solches Programm «gepflegt», d.h., ohne viel Aufwand an die verschiedensten Einsatzsituationen angepaßt werden muß. Eigentlich müßten wir da stehen haben:

```
100 GOSUB Menüauswahl
110 IF A<>1 THEN GOTO nächste Möglichkeit
120 GOSUB Lagerführung
130 GOSUB Nachbestellmeldung
140 GOTO Schleifenanfang
```

```
150 IF A<>2 THEN GOTO nächste Möglichkeit
160 GOSUB Lagerübersicht
170 GOTO Schleifenanfang
180 IF A<>3 THEN GOTO Schleifenanfang
190 GOTO Programm beenden
```

Wir müssen hier zwei Punkte unterscheiden. Das eine sind die Zeilennummern wie 500, 1000, 3500 usw. Diese sind für den Rechner da und dirigieren seine Arbeit. Das andere sind Kommentare, Anmerkungen zum Programm, wie «Menüauswahl», «Lagerführung», «Nachbestellmeldung» u.ä. Sie sind ausschließlich für den Menschen gedacht, der mit dem Programm umgehen muß,und sollten – ginge es nach dem Computer – gar nicht erst im Programmtext stehen. Da derartige Ergänzungen für ein sauberes Programmieren aber unumgänglich sind, muß es einen Weg geben, dem Computer mitzuteilen:

Achtung, hier folgt eine Anmerkung! Diesen Text ignorieren!

Kern des Ganzen ist «Anmerkung», englisch «remark», was zu dem BASIC-Befehlswort

REM

abgekürzt wird.

Dieser REM-Befehl kann uns jedenfalls aus der Klemme helfen. Wir fügen einfach passende Kommentare in den Programmtext ein. Programmieren wir das einmal zur Übung:

```
>NEW
READY
>100 GOSUB 500 : REM Menueauswahl
>110 IF A<>1 THEN GOTO 150 : REM naechste Moeglichkeit
>120 GOSUB 1000 : REM Lagerfuehrung
>130 GOSUB 3500 : REM Nachbestellmeldung
>140 GOTO 100 : REM Schleifenanfang
>150 IF A<>2 THEN GOTO 180 : REM naechste Moeglichkeit
>160 GOSUB 5000 : REM Lageruebersicht
>170 GOTO 100 : REM Schleifenanfang
>180 IF A<>3 THEN GOTO 100 : REM Schleifenanfang
>190 GOTO 10000 : REM Programm beenden
```

Wir brauchen noch etwas für die betreffenden Unterprogramme, wollen wir ausprobieren, ob das stimmt. Dazu beschränken wir uns auf ein paar einfache Meldungen, die anzeigen, daß wir tatsächlich in dem gewünschten Programmteil sind. Das kann z.B. so aussehen:

```
>500 REM *** Menueauswahl ***
>510 INPUT "Welches Menue jetzt"; A
>520 RETURN
>1000 REM *** Lagerfuehrung ***
>1010 PRINT "Lagerfuehrung"
>1020 RETURN
>3500 REM *** Nachbestellmeldung ***
>3510 PRINT "Nachbestellmeldung"
>3520 RETURN
>5000 REM *** Lageruebersicht ***
>5010 PRINT "Lageruebersicht"
>5020 RETURN
>10000 REM *** Programm beenden ***
>10010 PRINT "Programm wird beendet."
>10020 END
```

Das sieht alles sehr schön aus. Doch funktioniert es auch? Wir sollten es unbedingt ausprobieren:

```
>RUN
Welches Menue jetzt? 1
Lagerfuehrung
Nachbestellmeldung
Welches Menue jetzt? 2
Lageruebersicht
Welches Menue jetzt? 0
Welches Menue jetzt? 4
Welches Menue jetzt? 3
Programm wird beendet.
READY
```

So weit sind unsere Überlegungen also in Ordnung. Der REM-Befehl bewirkt demnach, daß der Computer den nachfolgenden Text – egal was es ist – ignoriert. Egal was es ist? Das haben wir noch nicht nachgeprüft. Was geschieht z.B., wenn nach REM ein anderes Befehlswort folgt? Sehen wir es uns an. (Löschen Sie den Speicher *nicht* für dieses Experiment! Wir fügen die Zeilen einfach vor den jetzt vorliegenden Text ein.)

```
>10 REM PRINT "xxx"
>11 PRINT "Ende"
>12 END
>RUN
Ende
READY
```

Der Befehl wurde ignoriert. Und wenn wir ihn durch einen Doppelpunkt abtrennen?

```
>10 REM Hier folgt ein Befehl : PRINT "xxx"
>RUN
Ende
READY
```

Sie können hiermit noch ein wenig experimentieren. Das Ergebnis dürfte bei allen BASIC-Versionen dasselbe sein:

Der REM-Befehl bewirkt, daß der ganze bis zum Zeilenende folgende Text vom Rechner bei der Abarbeitung ignoriert wird.

Mit anderen Worten: Wenn der Computer bei der Abarbeitung eines BASIC-Programms auf einen REM-Befehl stößt, so springt er sofort zur nächsten Zeile und macht dort mit der Arbeit weiter. Alles in der Zeile *vor* REM Stehende wird dagegen normal abgearbeitet, wie unser Beispiel zeigt.

Insbesondere bedeutet diese Eigenschaft von REM, daß wir in der grafischen Gestaltung der Kommentare völlig frei sind. Das haben wir oben bereits ausgenutzt, indem wir den verschiedenen Unterprogrammen *Titel* gegeben haben:

```
>LIST 500-
500 REM *** Menueauswahl ***
510 INPUT "Welches Menue jetzt"; A
520 RETURN
1000 REM *** Lagerfuehrung ***
1010 PRINT "Lagerfuehrung"
1020 RETURN
3500 REM *** Nachbestellmeldung ***
3510 PRINT "Nachbestellmeldung"
3520 RETURN
5000 REM *** Lageruebersicht ***
5010 PRINT "Lageruebersicht"
5020 RETURN
```

```
10000 REM *** Programm beenden ***
10010 PRINT "Programm wird beendet."
10020 END
```

Das Einschließen in Sterne hebt die Bedeutung als Überschrift besonders hervor und gestattet ein rasches Auffinden bestimmter Programmteile im Ausdruck. Bei komplexen Programmen sollte man das Prinzip weiterführen und eine Gliederung durch «Zwischentitel» schaffen, in der zur optischen Unterscheidung beispielsweise anstelle der Sterne Doppelstriche genommen werden. So ließe sich z.B. die Lagerführung weiter unterteilen in die Einheiten:

```
1000 REM *** Lagerfuehrung ***
1010 (Menüanforderung)
 . . .
1500 REM === Artikel aufnehmen ===
1510 (Unterprogramm zur Artikelneuaufnahme)
 . . .
2000 REM === Artikel loeschen ===
2010 (Unterprogramm zum Löschen eines Artikels)
 . . .
2500 REM === Bestand veraendern ===
2510 (Menüanforderung)
 . . .
2750 REM --- Exemplare entnehmen ---
2760 (Unterprogramm zur Exemplarentnahme)
 . . .
3000 REM --- Exemplare einfuegen ---
3010 (Unterprogramm zur Exemplareinfügung)
```

Das läßt sich auch nach oben hin fortsetzen. Z.B. sollte man dem ganzen Programm einen möglichst auffallenden Titel geben, was so geschehen kann:

```
>10 REM ****************************************
>11 REM ***   L a g e r v e r w a l t u n g    ***
>12 REM ****************************************
```

Schließlich und endlich können wir REM zur rein optischen Gliederung des Programmausdrucks verwenden, indem einfach nur der Befehl REM und sonst nichts angegeben wird. Das ist besonders dann sehr wichtig, wenn man sich in einer engbedruck-

ten Programmliste rasch orientieren oder auf dem Bildschirm eine Stelle schnell auffinden möchte. Diese «Leerzeilen» wirken wie Kapitelunterteilungen; man kann mit ihnen die inhaltliche Gliederung des Programms auch optisch hervortreten lassen. In unserem Beispiel könnte man das so machen:

```
>13 REM
>14 REM
>109 REM
>149 REM
>179 REM
>198 REM
>199 REM
>999 REM
>3499 REM
>4999 REM
>9999 REM
```

was dann dieses Bild liefern würde:

```
>LIST
10 REM ****************************************
11 REM ***   L a g e r v e r w a l t u n g    ***
12 REM ****************************************
13 REM
14 REM
100 GOSUB 500 : REM Menueauswahl
109 REM
110 IF A<>1 THEN GOTO 150 : REM naechste Moeglichkeit
120 GOSUB 1000 : REM Lagerfuehrung
130 GOSUB 3500 : REM Nachbestellmeldung
140 GOTO 100 : REM Schleifenanfang
149 REM
150 IF A<>2 THEN GOTO 180 : REM naechste Moeglichkeit
160 GOSUB 5000 : REM Lageruebersicht
170 GOTO 100 : REM Schleifenanfang
179 REM
180 IF A<>3 THEN GOTO 100 : REM Schleifenanfang
190 GOTO 10000 : REM Programm beenden
198 REM
199 REM
500 REM *** Menueauswahl ***
510 INPUT "Welches Menue jetzt"; A
520 RETURN
999 REM
1000 REM *** Lagerfuehrung ***
1010 PRINT "Lagerfuehrung"
1020 RETURN
3499 REM
```

```
3500 REM *** Nachbestellmeldung ***
3510 PRINT "Nachbestellmeldung"
3520 RETURN
4999 REM
5000 REM *** Lageruebersicht ***
5010 PRINT "Lageruebersicht"
5020 RETURN
9999 REM
10000 REM *** Programm beenden ***
10010 PRINT "Programm wird beendet."
10020 END
```

Nachteile des REM-Befehls

Bei aller Bequemlichkeit und Zeitersparnis beim Warten und Pflegen des Programms besitzt der REM-Befehl zwei wichtige Nachteile, die sich in Extremsituationen auswirken können:

1. *Jede REM-Anweisung belegt Speicherplatz*

   Das ist besonders bei vielen einfacheren Heimcomputern von Bedeutung, deren Programmspeicher recht klein ist. Hier muß man wohl oder übel mit Platz knausern. Das bedeutet unter anderem, daß bei längeren Programmen alle REM-Anweisungen und alle überflüssigen Leerschritte gestrichen sowie so viele Mehrfachbefehlszeilen wie möglich eingesetzt werden müssen.

2. *Jede REM-Anweisung kostet Abarbeitungszeit*

   In den meisten Fällen ist das nebensächlich, da BASIC-Programme in der Regel ohnehin relativ langsam abgearbeitet werden. Es gibt aber Grenzfälle, in denen man – beispielsweise eine sehr häufig durchlaufene Schleife – so rasch wie möglich abarbeiten möchte. In diesem Fall sollten nicht nur die REM-Anweisungen entfernt, sondern auch alle anderen oben angegebenen Maßnahmen durchgeführt werden.

Das Programm muß also *für den Computer* aufbereitet werden, und zwar um so intensiver, je umfassender und komplexer es ist.

Das wiederum bedeutet, daß der eigentliche Programmtext, wie man ihn durch einen Programmausdruck mittels «LIST» gewinnt, völlig ungenießbar werden kann. Und damit steigt der Zeitaufwand rasch an, wenn ein solches komplexes Programm gepflegt werden soll.

Streichen Sie daher nicht einfach drauflos. Versuchen Sie statt dessen, von der Gliederung so viel wie nur möglich zu erhalten. Streichen Sie erst die nebensächlichsten Anmerkungen, dann die weniger wichtigen, dann erst (wirklich) überflüssige Zwischenräume. Fassen Sie möglichst nur logisch Zusammenhängendes in einer Zeile zusammen usw. Und wenn sich nach intensiver Überarbeitung herausstellen sollte, daß (auch für den Programmlauf selbst, die Variablen brauchen auch Platz im Speicher!) noch Speicherplatz frei sein sollte, dann scheuen Sie sich nicht, diesen Platz durch geeignet gewählte Gliederungs- und Anmerkungsmaßnahmen zu nutzen.

Und wenn Sie einmal wirklich alle «Luft» aus einem Programm herauslassen mußten, dann belassen Sie die Dokumentation nicht beim LIST-Ausdruck. Fügen Sie nachträglich Anmerkungen von Hand ein. Gliedern Sie den Text, indem Sie Zusammengehöriges farbig hervorheben. Geben Sie Verweise auf Strukturbeschreibungen, Flußdiagramme und ähnliche Hilfsmittel an. Kurz: Tun Sie alles, damit das Programm auch in einem Vierteljahr noch verständlich ist. Selbst wenn Sie das Programm nie aus der Hand geben wollen: Legen Sie seine Dokumentation so an, daß sich ein Außenstehender jederzeit ohne große Mühe zurechtfinden kann.

3.10 Etwas zum Umgang mit Unterprogrammen: Fehlermöglichkeiten, TRACE

Widmen wir uns noch ein wenig dem GOSUB-RETURN-Paar. Durch die besondere Funktion von Steueranweisungen im Programm sind auch eine Reihe besonderer Programmierfehler möglich, die man kennen und erkennen sollte. Da ist z.B. der Fall, daß die *Zielzeile* eines GOSUB- oder GOTO-Befehls im Programm nicht existiert. Sehen wir uns das einmal an unserem oben ausgearbeiteten Beispiel an.

Die erste Anweisung der Ablaufsteuerung besagt:

```
GOSUB 500
```

Was geschieht nun, wenn wir Zeile 500 streichen?

```
>500
>RUN
?UL ERROR IN 100
```

«UL» steht für «undefined line» und meldet, daß die vorgeschriebene Zielzeile nicht im Programm vorhanden ist. Anders als beim LIST-Kommando macht der Rechner nicht einfach mit der nächsten Zeile weiter, sondern meldet den Fehler nach außen. So wird einem Programm- oder Datenverlust durch logische oder Tippfehler entgegengewirkt. Der zugrundeliegende Mechanismus ist dabei einfach:

Jedesmal, wenn der Rechner im Programm eine Verzweigungsanweisung (GOTO oder GOSUB) vorfindet, sucht er im Programmtext die Zeilennummer der zugehörigen Zielzeile. Ist sie vorhanden, so wird die Abarbeitung auf diese Stelle umgelenkt. Fehlt sie, dann gibt der Rechner eine UL-Fehlermeldung (oder das der jeweiligen BASIC-Version Entsprechende) aus und bleibt stehen.

Und was geschieht, wenn das Unterprogramm zwar vorhanden, aber der zugehörige GOSUB-Befehl vergessen worden ist? Nun – dann wird das Unterprogramm gar nicht erst aufgerufen.

So etwas kommt in der Praxis häufig beim Abtippen fremder Programme vor. In diesem Fall hilft in den meisten BASIC-Versionen nur eines:

Machen Sie es wie oben. Lassen Sie jedes in Frage kommende Unterprogramm eine Meldung ausdrucken, ob es während der Abarbeitung aufgerufen worden ist. Zumeist findet sich eine Zeile, in der man so einen Zusatzbefehl unterbringen kann. Starten Sie dann das Programm, eventuell unter geeigneten Rahmenbedingungen, und sehen Sie sich am Bildschirm oder besser noch mit Hilfe eines Druckers an, wer da was aufruft.

Einige der ausgebauteren BASIC-Versionen unterstützen das durch die sogenannte «TRACE-Option». Das ist die Möglichkeit, sich vom Rechner in einem bestimmten Programmbereich alle Zeilennummern ausdrucken zu lassen, die gerade bearbeitet werden. Man erhält so eine «Spur» (auf englisch: trace) der Programmabarbeitung für diesen Bereich, aus der man auf das Programmverhalten schließen kann. Es ist allerdings nicht ganz einfach, diese Möglichkeit auch sinnvoll einzusetzen, ohne daß man in einem Berg von Papier ertrinkt. Am besten kombiniert man dies mit der Möglichkeit, die fraglichen Unterprogramme eine ausdrückliche Meldung ausdrucken zu lassen. Man braucht etwas Praxis, um derartige Fehlersuchmöglichkeiten auch wirklich zu nutzen. Wir wollen hier jedenfalls nicht näher darauf eingehen.

Und was geschieht, wenn ein RETURN fehlt? Nun – dann macht das Programm einfach weiter, bis zum nächsten RETURN, bis zum Programmende oder bis es sich irgendwo «aufgehängt» hat. Wir können das illustrieren, wenn wir in unserem Beispiel ein RETURN streichen. Doch zunächst sollten wir Zeile 500 wieder einfügen:

```
>500 REM *** Menueauswahl ***
```

Streichen wir dann den Rücksprung nach dem Unterprogramm zur Lagerführung:

```
>1020
>RUN
Welches Menue jetzt? 1
Lagerfuehrung
Nachbestellmeldung
Nachbestellmeldung
Welches Menue jetzt? 3
Programm wird beendet.
READY
```

«Nachbestellmeldung» kommt zweimal vor: einmal aus dem Aufruf des Unterprogramms in Zeile 1000 und einmal aus dem Aufruf des Unterprogramms in Zeile 3500. Und wenn wir statt des RETURN-Befehls von Zeile 1020 ein END einsetzen?

```
>1020 END
>RUN
Welches Menue jetzt? 1
Lagerfuehrung
READY
```

Hier wurde das Programm schon vor dem Teil zur Nachbestellung abgebrochen.

Zwei wichtige Folgerungen können wir aus diesen Experimenten ziehen:

1. Man braucht ein Unterprogramm nicht unbedingt mit dessen erster Zeile aufzurufen. Jede Zeile kann als Einsprung dienen, solange der zugehörige RETURN-Befehl noch erreicht wird.
2. Man kann die Abarbeitung auch in einem Unterprogramm durch END abschließen.

Beides läßt sich gezielt bei der Programmierarbeit einsetzen. Jedoch ist Vorsicht geboten. Derartige Programmiertricks werden bei Änderungen oder Programmergänzungen leicht übersehen, was insbesondere für den ersten Fall zutrifft. Es passiert leicht, daß man die Zielzeilen mitten im Unterprogramm löscht oder – schlimmer noch – unabsichtlich ändert, so daß das Programm plötzlich unerwartete Fehler aufweist. Wenn die betreffende Stelle dann nur noch sporadisch vom Programm angesprungen

wird, können Wochen bei der Fehlersuche draufgehen. Machen Sie sich daher zur Regel:

Vermeiden Sie Unterprogramme mit mehreren Einsprungstellen wo immer möglich! Sollte das in bestimmten Ausnahmefällen nicht gehen, so vermerken Sie den Einsprung durch eine REM-Anweisung ausdrücklich im Unterprogramm! Drittens: Springen Sie immer nur zu einer Kommentarzeile im Programm! Das verzögert die Abarbeitung nur minimal, bietet aber die Sicherheit, daß eine solche zusätzliche Zielzeile nicht versehentlich gelöscht wird.

(Im letzten Punkt gehen die Meinungen im übrigen auseinander: Man findet oft auch die Empfehlung, *niemals* eine Kommentarzeile anzuspringen, damit ein Streichungswütiger nicht alle Sprungziele aus dem Programm entfernt. Dieses Argument ist zwar nicht völlig von der Hand zu weisen, doch wird ein Fehler durch eine entfallene Zielzeile automatisch vom Programm gemeldet. Unsere Empfehlung bietet mehr: Es bietet die Sicherheit, daß das Programm auch nach einer Änderung verständlich bleibt! Und wenn wirklich einmal alle REM-Anweisungen gestrichen werden müssen, dann ist es ein Leichtes, die dem Kommentar folgende Zeile umzunumerieren, so daß sie als Ziel dienen kann.)

Bleibt noch eine letzte Fehlermöglichkeit: Was geschieht, wenn ein Unterprogramm ohne ausdrücklichen GOSUB-Befehl abgearbeitet wird? Wir müssen hier zwei Fälle unterscheiden:

1. Das Herkunftsprogramm ist selbst ein Unterprogramm.
2. Das Herkunftsprogramm ist kein Unterprogramm.

Ein derartiger Fehler tritt bevorzugt unter zwei Umständen auf. Dann nämlich, wenn statt eines GOSUB ein GOTO verwendet wird, und dann, wenn ein END-Befehl fehlt, der einen Programmteil vor einem anderen abschließt, und das Programm so statt zu stoppen in das Unterprogramm hineinläuft. Schreiben wir hierzu unser Testprogramm ein wenig dahingehend um, daß das Unterprogramm zur Ausgabe einer Lagerübersicht zusätzlich das

Unterprogramm für die Ausgabe einer Nachbestellübersicht aufruft. Und um zu kontrollieren, daß das auch wirklich geschehen ist, fügen wir noch eine Meldung «Nach UP-Aufruf» ein.

Zunächst sollten wir allerdings den RETURN-Befehl von Zeile 1020 wiederherstellen:

```
>1020 RETURN
```

Dann ändern wir folgende Zeilen:

```
>5020 GOSUB 3500
>5030 PRINT "Nach UP-Aufruf"
>5040 RETURN
```

Der geänderte Programmabschnitt lautet also:

```
>LIST 3500-5040
3500 REM *** Nachbestellmeldung ***
3510 PRINT "Nachbestellmeldung"
3520 RETURN
4999 REM
5000 REM *** Lageruebersicht ***
5010 PRINT "Lageruebersicht"
5020 GOSUB 3500
5030 PRINT "Nach UP-Aufruf"
5040 RETURN
```

Normalerweise würde das so laufen:

```
>RUN
Welches Menue jetzt? 2
Lageruebersicht
Nachbestellmeldung
Nach UP-Aufruf
Welches Menue jetzt?
```

Die «Lagerübersicht» hat die «Nachbestellmeldung» aufgerufen, die nach Abarbeitung wieder zur «Lagerübersicht» zurückgekehrt ist. Was geschieht nun, wenn wir in Zeile 5020 versehentlich statt GOSUB 3500 einen normalen Sprung «GOTO 3500» fordern würden?

```
>5020 GOTO 3500
>RUN
Welches Menue jetzt? 2
Lageruebersicht
Nachbestellmeldung
Welches Menue jetzt?
```

Der Rechner hat jetzt den RETURN-Befehl in Zeile 3520 als Rücksprung zur Ablaufsteuerung interpretiert, anstatt wie beabsichtigt zum Unterprogramm «Lagerübersicht» zurückzukehren. So etwas läßt sich zur Kodeersparnis einsetzen, wenn man ein Unterprogramm als letzten Schritt in einem Teilprogramm ausführen möchte. Anstatt «GOSUB xxxx» mit anschließendem «RETURN» kann man hier auch «GOTO xxxx» programmieren – aber die Lesbarkeit des Programms wird dadurch eingeschränkt. Man sollte aber davon wissen, da dieser Trick in der Praxis häufig zu finden ist.

Anders sieht es aus, wenn der einfache Sprung tatsächlich versehentlich statt eines GOSUB-Befehls verwendet worden ist. Dann hat man einen besonders heimtückischen Fehler, der zu keiner (unmittelbaren) Fehlermeldung beim Abarbeiten führt. Man muß das Programm bis in alle Einzelheiten kennen, um ihm auf die Spur zu kommen.

Sehen wir uns zum Abschluß noch an, was geschieht, wenn im *Hauptprogramm* ein GOSUB mit einem GOTO vertauscht worden ist:

```
>120 GOTO 1000
>RUN
Welches Menue jetzt? 1
Lagerfuehrung
?RG ERROR IN 1020
```

Jetzt hat das Programm ein RETURN vorgefunden, ohne daß eine Zeilennummer als Rücksprungadresse gespeichert war. «RG» heißt, wie ein Blick ins Handbuch zeigt, «RETURN without GOSUB» – RETURN ohne GOSUB. Das wird gemeldet – allerdings mit Angabe der Zeile, in der das fehlerhafte RETURN steht, und nicht mit der Zeile, von welcher der Aufruf kam. Das

kann in verwickelten Fällen Schwierigkeiten machen, weil man jetzt durch alle möglichen Schachtelebenen *rückwärts* nach dem fehlerhaften Aufruf suchen muß. Hier kann die oben erwähnte «TRACE-Option» zusammen mit dem Ausdruck von Meldungen durch die in Frage kommenden Unterprogramme ungemein hilfreich sein.

Bleibt noch eines nachzutragen: Unser Beispiel zeigt die *Kodierung* «von oben nach unten», die dem weiter oben besprochenen «Top-Down-*Entwurf*» parallel gehen kann. Diese Methode ist geeignet, wenn man den allgemeinen logischen Programmfluß untersuchen möchte. Durch vorläufige Meldungen an Stelle der tatsächlichen Unterprogramme kann man sich auf die Hauptsache konzentrieren, kann Nebenwirkungen untersuchen und verschiedene Strukturmöglichkeiten rasch gegeneinander abwägen. Der Nachteil dieser Methode liegt vor allem darin, daß das Austesten einzelner Teilprogramme sehr viel komplexer wird. Man wird daher in der praktischen Programmentwicklung normalerweise die Kodierung «von unten nach oben» wählen und die in diesem Abschnitt vorgestellte Methode nur bei besonderen Problemen – parallel zum übrigen Programmentwurf – einsetzen. Sie empfiehlt sich vor allem dann, wenn logische Fehler im Programm*fluß* aufgedeckt werden sollen, die durch die «Bottom-up-Kodierung» nur schwer zu finden sind.

Wichtig ist auch festzuhalten, daß sich die verschiedenen Methoden weder gegenseitig ausschließen, noch daß es irgendein «Patentrezept» zur Programmerstellung gibt. Je größer die Vielzahl an Programmiermethoden ist, die man beherrscht, um so effektiver kann man im Ernstfall vorgehen!

3.11 Überlegungen zur Datenstruktur

Mit Eingabeanweisung und Unterprogrammen haben wir einige wesentliche Voraussetzungen für die Lösung unseres Problems betrachtet. Um systematisch weiter voranschreiten zu können, fehlt uns aber noch ausreichende Kenntnis über ein weiteres Gebiet, nämlich über die Form, Art und Weise, in der die Daten im Rechner festgehalten werden sollen. Kurz: Wir müssen die *Datenstruktur* zu unserem Problem festlegen.

Einige Festlegungen hatten wir schon zu Beginn unserer Untersuchung getroffen, wo wir folgende Gliederung aufstellten:

1. Daten, die für längere Zeit festzuhalten sind:
   - Bezeichnung des Gegenstands
   - Menge auf Lager
   - Datum der Änderung
   - Mindestmenge auf Lager
   - Einzelwert der verschiedenen Artikel
2. Daten, die nur im Verlauf der Bearbeitung benötigt werden
   - Entnahme oder Aufnahme
   - Anfangs- und Enddatum für den Ausdruck eines Lagerauszugs
3. Daten, die hieraus ermittelt werden können:
   - Gesamtwert der verschiedenen Artikel
   - Gesamtwert des Lagerbestands
4. Daten, die im Einzelfall vorzugeben sind:
   - Befehle zur Auswahl unter den Programmteilen

Ferner haben wir für diese Daten auch schon einige Variablennamen festgelegt:

D – Datum der letzten Änderung eines Eintrags
D1 – Untere Datumsgrenze beim Ausdruck von Lagerauszügen

D2 – Obere Datumsgrenze beim Ausdruck von Lagerauszügen
M – Menge auf Lager
M1 – Mindestmenge auf Lager
W1 – Einzelwert der verschiedenen Artikel
W – Gesamtwert der verschiedenen Artikel
W0 – Gesamtwert des Lagerbestands

Das können wir sofort um eine weitere Variable ergänzen, die bei der Dateneingabe benötigt wird:

M0 – Menge, die gerade entnommen oder hinzugefügt wurde

Ordnen wir das zusammen, so ergibt sich folgendes Bild:

1. Daten, die für längere Zeit festzuhalten sind:
   - Bezeichnung des Gegenstands
   - Menge auf Lager: Variable M
   - Datum der Änderung: Variable D
   - Mindestmenge auf Lager: Variable M1
   - Einzelwert der verschiedenen Artikel: Variable W1
2. Daten, die nur im Verlauf der Bearbeitung benötigt werden
   - Entnahme oder Aufnahme: Variable M0
   - Anfangs- und Enddatum für den Ausdruck eines Lagerauszugs: Variablen D1, D2
3. Daten, die hieraus ermittelt werden können:
   - Gesamtwert der verschiedenen Artikel: Variable W
   - Gesamtwert des Lagerbestands: Variable W0
4. Daten, die im Einzelfall vorzugeben sind:
   - Befehle zur Auswahl unter den Programmteilen

Auf den ersten Blick sind damit fast allen Daten Variable zugeordnet, in denen die betreffenden Werte im Rechner festgehalten werden können (mit den noch fehlenden Festlegungen

werden wir uns weiter unten noch intensiv zu beschäftigen haben). Allerdings ist das nur auf den ersten Blick so, denn wenn wir näher hinsehen, dann fällt auf, daß wir es mit zwei ganz verschiedenen Arten von Daten zu tun haben.

Da sind zunächst einmal die Daten, die im Verlauf der Programmabarbeitung nur in einem Exemplar zur Zeit auftreten, wie beispielsweise der Gesamtwert von Artikel- bzw. Lagerbestand. Dann haben wir da aber auch noch Daten festgelegt, die nicht nur in einem Exemplar vorkommen, alles das nämlich, was einen konkreten Lagergegenstand beschreibt und auf Dauer festgehalten werden muß, insbesondere also alle diejenigen Daten, die in Gruppe 1 stehen:

1. Daten, die für längere Zeit festzuhalten sind:
   - Bezeichnung des Gegenstands
   - Menge auf Lager: Variable M
   - Datum der Änderung: Variable D
   - Mindestmenge auf Lager: Variable M1
   - Einzelwert der verschiedenen Artikel: Variable W1

Ganz offensichtlich reicht die Variablenfestlegung hier nicht aus, denn mit einer einzigen Variablen können wir auch nur einen einzigen Wert festhalten, in der Variablen M z.B. die Menge aller Schrauben M3 x 20 auf Lager, auf keinen Fall aber gleichzeitig die Menge aller Reißzwecken. Und doch müssen wir sowohl die Schrauben als auch die Reißzwecken auf Anforderung zur Verfügung haben. Das heißt, wir brauchen mindestens so viele verschiedene Variable von der Art «Menge auf Lager», wie verschiedene Artikel im Lager vorhanden sind – jedenfalls solange wir diese Werte alle gleichzeitig im Computerspeicher stehen haben, und dazu bleibt uns gegenwärtig keine andere Wahl. Wir müssen uns also damit auseinandersetzen, wie so etwas mit BASIC erreicht werden kann.

Felder, DIM X(i), FRE(0)

Sind nur relativ wenige Werte zu speichern, dann kann man zur Not für jeden eine eigene Variable erfinden, für die Lagermenge z.B. (da M0 und M1 schon anderweitig belegt sind) M2, M3, M4 usw. Doch das hört bei M9 bereits auf. Mehr noch: Auf diese Weise wird der ganze Variablenvorrat, den man unter Umständen für die verschiedenen Rechnungen dringend benötigt, aufgebraucht.

Das Problem stellt sich bereits bei ganz einfachen Aufgaben, so daß schon die primitivsten BASIC-Versionen die Möglichkeit bieten, *Felder* zur Variablenspeicherung einzusetzen. Im einfachsten Fall ist das nichts anders als eine Vielzahl von unter dem gleichen Namen zusammengefaßten Variablen, die einfach mit Hilfe von Indizes durchgezählt werden, etwa so:

$$M_1, M_2, M_3, \ldots$$

Allerdings stehen wir hier wieder vor der alten Schwierigkeit: Wie schreibt man denn so etwas? Indizes durch Tieferstellen anzudeuten, geht ja bei den üblichen Eingabegeräten nicht, ohne vorher die Zeile – und damit den ganzen Befehl – abzubrechen. Bevor wir lange herumraten: Es hat sich allgemein eingebürgert, derartige Indizes durch *Einschließen in Klammern* anzugeben. Viele Programmiersprachen verwenden hierzu der Eindeutigkeit halber eckige Klammern. In BASIC dagegen hält man sich an den kleinstmöglichen Zeichensatz: Man verwendet ganz normale *runde Klammern* für diesen Zweck:

M(1), M(2), M(3), ...

Dabei ist eines wichtig: M1, M2, M3 ist etwas grundlegend anderes als M(1), M(2), M(3) usw. Im ersten Fall haben wir drei *verschiedene* Variable benannt, die jeweils Speicherplatz für *eine* Zahl zur Verfügung stellen. Mit einem Feld aber benennen wir nur *eine* Variable (in unserem Beispiel ist das die Variable M), die aber die besondere Eigenschaft hat, Speicherplatz für *mehrere*

Zahlen bereitzustellen. Und um die verschiedenen Speicherplätze dort unterscheiden zu können, zählt man sie *innerhalb* der Feldvariablen durch. Mathematisch entspricht dies einem Vektor bzw. einer eindimensionalen Matrix, weshalb man derartige Felder auch als *Matrizen* bezeichnet.

Wie geht man nun mit einer derartigen Matrix um? Eigentlich dürfte da kein großer Unterschied zum bisher Gewohnten sein. Neu ist nur, daß man die Feldelemente mit ihren Indizes bezeichnen muß. Testen wir zunächst, ob überhaupt ein Zugriff auf ein derartiges Element möglich ist:

```
>PRINT M(3)
 0
```

Damit haben wir sichergestellt, daß bei Angabe eines Feldelements in der beschriebenen Art und Weise kein Syntax- oder sonstiger Fehler vorliegt. Weiter wissen wir, daß das Element mit der Nummer 3 im Feld M – wenn es überhaupt existiert – den Wert 0 hat. Versuchen wir also, ihm einen anderen Wert zuzuweisen:

```
>LET M(3)=1234.56
READY
```

Soweit ist das in Ordnung. Wir müssen nur noch nachprüfen, ob die Zuweisung das gewünschte Ergebnis gebracht hat:

```
>PRINT M(3)
 1234.56
```

Das Prinzip stimmt. Das bedeutet, daß wir derartige Felder für unsere Problemlösung ins Auge fassen können. Allerdings müssen wir noch ein bißchen mehr über den Umgang mit einem solchen Feld wissen.

Unangenehm ist im Moment noch, daß wir die Indizes bis jetzt ausdrücklich als Zahl angeben müssen: M(1), M(2), M(3) usw. Das engt die Anwendungsmöglichkeiten ziemlich ein. Viel sinnvoller wäre es, könnte man die Indizes durch Variable benennen,

z.B. M(I), und bräuchte dann nur diese Variable I durchzuzählen. Denn das entspricht ja unserem Problem, eine (beim Programmieren) unbekannte Anzahl von Werten aufeinanderfolgend speichern zu müssen. Nun, so etwas läßt sich herausbekommen:

```
>NEW
READY
>10 I = 1
>20 M(I) = 2*I+1
>30 PRINT I;":";M(I)
>40 I = I+1
>50 IF I<=20 THEN 20
>60 END
```

Das ist eine Schleife, die von Zeile 20 bis Zeile 50 reicht und schrittweise von 1 angefangen, 20 Feldelemente von M verschiedene Werte zuweist. Diese Zuweisung wird in Zeile 30 überprüft, wo die Elementnummer und, von ihr durch einen Doppelpunkt getrennt, der Wert des betreffenden Feldelements ausgegeben wird. Und damit es gar nicht erst zu Verwechslungen kommen kann, bekommt jedes Feldelement einen anderen Wert zugewiesen, nämlich die ungeraden Zahlen, beginnend mit 3 in aufsteigender Folge.

Lassen wir das einmal laufen:

```
>RUN
 1:3
 2:5
 3:7
 4:9
 5:11
 6:13
 7:15
 8:17
 9:19
 10:21
?BS ERROR IN 20
```

Irgend etwas stimmt da noch nicht. Doch bevor wir im Handbuch nachschlagen, was BS ERROR nun schon wieder ist, sollten wir untersuchen, was wir bis dahin erhalten haben. Zunächst:

Unsere Forderung, die Feldelemente durch Variable indizieren zu lassen, scheint erfüllbar. Bis zum Feldelement 10 haben wir genau das herausbekommen, was wir erwartet hatten: Eine schrittweise Zuweisung der ungeraden Zahlen an die aufeinanderfolgenden Elemente.

Doch was hat es nun mit dem BS ERROR auf sich? Der Fehler ist in Zeile

```
20 M(I) = 2*I+1
```

aufgetreten, nachdem vorher alles in Ordnung zu sein schien. Überlegen wir: Das einzige, was sich in diesem Schritt ändert, ist der Wert der Variablen I (von 10 auf 11). Es ist mehr als unwahrscheinlich, daß in dem Ausdruck $2 * I + 1$ etwas nicht stimmen sollte, so daß nur in der Zuweisung

$$M(I) = \ldots$$

die Fehlerquelle zu suchen sein dürfte.

Nun – was heißt «M(I) =»? Das ist doch faktisch nichts anderes, als daß wir dem I-ten Element des Felds M einen Inhalt geben, was im Rechner zusätzlich Speicherplatz belegt. Mit dieser Speicherplatzanforderung muß unser Problem zusammenhängen. Wie, so stellt sich damit die Frage, reserviert der Rechner eigentlich für eine Variable den notwendigen Platz im Speicher? Oder anders herum gefragt: Wie greift er eigentlich auf einen bestimmten Variablenwert im Speicher zu?

Ohne allzutief in die technischen Einzelheiten einzudringen, läßt sich dazu folgendes sagen: Wenn der Rechner eine normale, nicht indizierte Variable zum erstenmal im Programm entdeckt, so reserviert er für sie das nächst erreichbare Stück im Speicher. Diesen Ort behält die Variable dann bei, solange das Programm läuft. Er hat seinen festliegenden Anfang und – was noch wichtiger ist – einen eindeutig gegebenen Platzbedarf: Eine einzige Zahl, nicht mehr und nicht weniger, kann dort festgehalten werden.

Anders bei Feldvariablen: Wenn der Rechner z.B. das Feld M

im Programm entdeckt, so kann er zwar nach wie vor den Anfang des dafür benötigten Speicherplatzes festlegen – eben das nächst erreichbare Stück. Über dessen Länge aber schweigen sich die Götter aus. Denn – wie soll man so ein Feld eigentlich organisieren? Das naheliegendste und üblicherweise auch angewendete ist folgendes Verfahren: Der Rechner legt den *Anfang* des dafür benötigten Speicherplatzes fest. Des weiteren kennt er, wie bei allen anderen Variablen, die *Länge eines Feldelements,* d.h. den Platz, den ein solches Element im Speicher in Beschlag nimmt. Und alle Feldelemente werden in der Reihenfolge ihrer Indizes *nacheinander* im Speicher abgelegt: M(2) folgt auf M(1), M(3) und M(2) usw.

Das genügt für den Zugriff auf einen bestimmten Wert: Um das I-te Element im Feld zu erreichen, *berechnet* der Computer dessen Platz, bezogen auf den Feldanfang. D.h., er multipliziert die Länge eines Einzelelements mit der Anzahl der vor dem gesuchten Feldelement stehenden Teile, addiert diese Größe zum Feldanfang und kann so schnell und einfach auf jedes Feldelement ohne langes Suchen zugreifen (vgl. Bild 8).

Und was ist, wenn der Rechner *nach* Festlegen einer Feldvariablen eine weitere Variable im Programm vorfindet? Ist es eine einfache Variable, so könnte man sie zur Not noch vor dem Feld einfügen. Dazu müßte man die von dem Feld belegten Speicherplätze geeignet verschieben, um den nötigen Platz zu schaffen. Im Prinzip wäre das machbar, ist aber mit einigem Aufwand verbunden. Und was, wenn *noch eine Feldvariable* im Programm auftaucht? Hier ist guter Rat teuer. Denn eigentlich müßte für sie der nächste freie Speicherraum belegt werden. Doch wo fängt dieser nächste freie Speicherraum an? Der Rechner weiß zwar, wo das *zuletzt* definierte Feld beginnt, kennt auch einen Weg, ein beliebiges Element aus diesem Feld zu erreichen – aber so wie die Dinge jetzt stehen, hat er keine Ahnung, wieviele Elemente das Feld letztlich umfassen wird. Es können 3 sein, aber auch 3000. Mit anderen Worten: Es ist nahezu unmöglich, dem nächsten Feld auf eine sinnvolle Art und Weise einen Platz zuzuweisen.

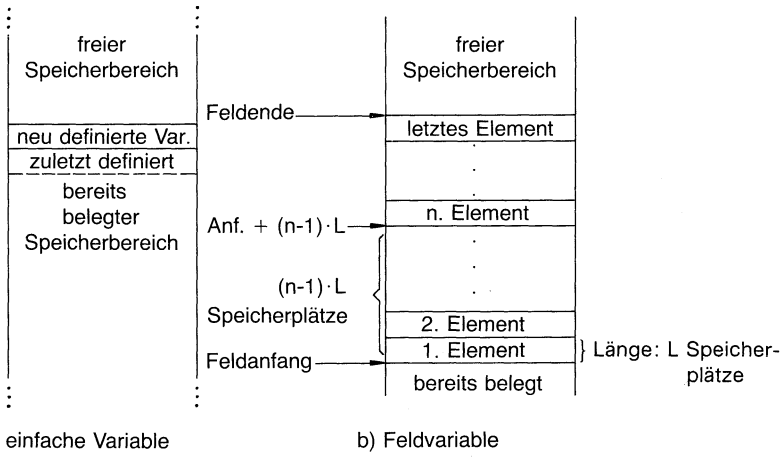


Bild 8 Organisation von Feldern und Feldzugriff im Rechner

Was tun? Es sind verschiedene Lösungsmöglichkeiten denkbar. Zum Beispiel könnte sich der Rechner auf die bis dahin angegebene größte Elementzahl für das vorhergehende Feld zunächst beschränken und weiteren Platz je nach Bedarf schaffen. Das würde jedoch jedesmal ein Verschieben der nachfolgenden Variablen im Speicher erfordern, was einigen Umstand macht und die Arbeit insgesamt verlangsamt. Die Alternativen wären dann darin zu suchen, daß der Rechner *vor* der ersten Angabe des Felds bereits dessen Länge kennt. So könnte der belegbare Raum vom BASIC her bereits vorgegeben sein, was allerdings unpraktisch ist, da man auf diese Art einerseits bei kurzen Feldern mit wenigen Elementen Speicherplatz vergeudet, auf der anderen Seite bei größerem Bedarf aber ziemliche Umstände treiben muß, um die festen Grenzen irgendwie zu umgehen. Bleibt nur noch die Möglichkeit, durch einen *besonderen Befehl* rechtzeitig anzugeben, wieviel Platz ein bestimmtes Feld voraussichtlich einnehmen wird.

Kehren wir zum Ausgangspunkt dieser Überlegungen zurück, zu der Frage nämlich, was es wohl mit der Fehlermeldung «BS ERROR» auf sich haben möge. Irgendwie, so ist zu vermuten, hat das etwas mit der Feldgröße zu tun. Bevor wir nachschlagen, sollte noch eine Beobachtung verzeichnet werden: Bis zu I = 10 lief alles wie gewünscht. D.h. – zumindest in der hier verwendeten BASIC-Version – ein Feld hat entweder *mindestens* 10 Elemente, oder es wird mangels sonstiger Angaben eine Feldgröße von 10 Elementen *angenommen,* oder man kann nur mit Feldern von *maximal* 10 Elementen arbeiten. Von diesen Möglichkeiten ist die letzte am unwahrscheinlichsten, da das die Arbeit mit BASIC beträchtlich einschränken würde. Auch gibt es genau betrachtet keinen Grund, ausgerechnet eine *Mindest*grenze für ein Feld festzulegen – warum sollte man bei Feldern mit beispielsweise fünf Elementen unnötig Platz verschenken? Dagegen hat die Möglichkeit, daß eine bestimmte Feldgröße zunächst einmal vom Rechner angenommen wird, durchaus etwas für sich.

Damit wird es nun wirklich Zeit, im Handbuch nachzuschlagen, so sich folgende Erläuterung findet:

BS – Bad Subscript (Indexangabe falsch)

«Es wurde der Versuch gemacht, ein Feldelement zu erreichen, das außerhalb der *Dimension,* d.h. der Grenzen des angegebenen Felds, liegt.»

Immerhin waren wir auf der richtigen Spur. Doch noch steht das Problem offen, was es wohl mit der «magischen Grenze» I = 10 auf sich hat. Offensichtlich ist der Begriff der Feld-*Dimension* der Schlüssel hierzu. Nehmen wir die wahrscheinlichste unserer Vermutungen über die Feldgröße als Anhalt, so müßte es einen Befehl geben, der das *Dimensionieren,* das Festlegen der Feldgrenzen gestattet. Und wenn wir unsere bisherigen Kenntnisse über BASIC-Befehle weiter zu Rate ziehen, so müßte der Befehl mit dem Wort «Dimension» – das ebenso im Englischen vorhanden ist – etwas zu tun haben. Sehen wir daraufhin das

Befehlsverzeichnis im Handbuch durch, so findet sich dort ein Befehl

DIM

der möglicherweise mit unserem Problem etwas zu tun hat.

Aufgabe eines solchen DIM-Befehls müßte unserer Vermutung nach das Bereitstellen von genügend Speicherplatz für Feldvariable sein. Auf der anderen Seite aber wissen wir mittlerweile, daß die Feldgröße von der Zahl der darin enthaltenen Elemente abhängt. Also ist zu erwarten, daß der DIM-Befehl als Argument irgendwie eine Angabe der Maximalzahl von Elementen für das betrachtete Feld benötigt. Und außerdem muß da noch irgendwie eine Angabe der Feldvariablen vorhanden sein.

Mit diesem Rüstzeug aus Vermutung und Hoffnung können wir uns an die Befehlsbeschreibung im Handbuch wagen und finden dort sinngemäß folgendes:

DIM X(i)

«Stellt den Speicherplatz für das Feld X mit i Feldelementen zur Verfügung. Ist kein DIM-Befehl für ein Feld angegeben, so wird eine Feldgröße von 10 Elementen angenommen.»

Das sollten wir zunächst einmal an unserem Testprogramm erproben. Wenn alles stimmt, was wir uns oben überlegt haben, dann müßte der Zusatz

DIM M(20)

vor der ersten Verwendung des Felds M das fehlerfreie Abarbeiten bis einschließlich I = 10 ermöglichen. Das sollten wir nachprüfen (wir unterschlagen aus Platzgründen ein paar Druckzeilen):

```
>5 DIM M(20)
>RUN
 1:3
 2:5
 3:7
 . . .
 18:37
 19:39
 20:41
READY
```

Einverstanden? Nun – ganz sicher können wir unserer Sache noch nicht sein. Wir müssen der Vollständigkeit halber noch nachprüfen, ob die Angabe der maximalen Elementzahl 20 für das Feld M tatsächlich bestimmend ist. Das läßt sich machen. Ändern wir die Dimensionierung doch einfach um in 19 und sehen uns an, was dann passiert.

```
>5 DIM M(19)
>RUN
 1:3
 2:5
 3:7
 . . .
 18:37
 19:39
?BS ERROR IN 20
```

Das dürfte eindeutig genug sein. Man kann den Speicherplatzbedarf aber auch unmittelbar bestimmen. Es gibt nämlich in den meisten BASIC-Versionen einen besonderen Befehl, mit dem sich der frei zur Verfügung stehende Speicherplatz ermitteln läßt. Dieser Befehl ist nicht standardisiert, in einigen Versionen heißt er FREE, in den meisten anderen hat er die Form einer Funktion, wie z.B. in der Version im Testgerät:

FRE(0)

(Auf BASIC-Funktionen werden wir weiter unten noch ausführlich eingehen. Hier genügt es festzuhalten, daß jede Funktion aus formalen Gründen in den meisten BASIC-Versionen ein in Klammern nachgestelltes Argument braucht, für das wir oben einfach eine 0 gesetzt haben – es hätte auch jeder beliebige andere numerische Wert sein können.)

FRE(0) ist insofern eine Funktion, als es einen Wert liefert, den man z.B. ausdrucken kann:

```
>NEW
>PRINT FRE(0)
 3564
```

Das ist die Zahl von Speicherplätzen (sogenannten Bytes), die gegenwärtig noch frei sind. Beim gegebenen Testgerät sind das nicht gerade sehr viel: Gerade 3564 Bytes an Speicherplatz stehen hier dem Anwender zur Verfügung, wenn kein Programm im Speicher steht. Sie werden in der Regel ein gutes Stück mehr erhalten.

Diese FRE-Funktion können wir jetzt ganz einfach ausnutzen. Da sie uns immer anzeigt, wieviel Speicherplatz gerade frei ist, können wir sie auch umgekehrt dazu verwenden, den durch irgendeinen Befehl belegten Platz zu ermitteln. Nutzen wir das, um herauszufinden, wieviel Speicherplatz durch eine Variable belegt wird:

```
>10 A = FRE(0)
>20 PRINT A : REM - Das liefert den derzeit freien Platz
>30 B = 1234 : REM - Der Rechner reserviert Platz für B
>40 PRINT A-FRE(0) : REM - Dieser wird hier angezeigt
>50 A = FRE(0) : REM - Wir setzen die Grenze neu und
>60 C(1)=234 : REM - wiederholen dasselbe für ein Feld
>70 PRINT A-FRE(0)
>RUN
 3477
 7
 62
```

Das bedeutet folgendes: Das Programm hat noch 3477 Bytes im Speicher frei gelassen. (Die Anmerkungen wurden für den Test nicht mit eingetippt – sonst wäre durch das Programm selbst mehr Platz belegt worden.) Davon wurden für die Variable B vom Rechner 7 Bytes belegt – was unsere Behauptung vom Anfang unserer Untersuchung belegt, daß der Rechner für eine Variable immer erst dann Speicherplatz reserviert, wenn er sie zum erstenmal im Programm vorfindet. Das Feld C seinerseits hat 62 Bytes im Speicher belegt.

Wenn Sie nun nachrechnen, so ergibt 62 geteilt durch 7 runde 8,9 – jedenfalls weniger, als für 10 einzelne Variable notwendig wäre. Ist damit unsere Überlegung von oben bezüglich der Speicherplatzreservierung falsch? Nun – prüfen wir das nach, indem wir ausdrücklich Platz für 10 Feldelemente reservieren:

```
>55 DIM C(10)
>RUN
 3465
 7
 62
```

Das entspricht dem ersten Ergebnis. (Beachten Sie, daß der Rechner für das Programm jetzt mehr Speicherplatz benötigt.) Die Vermutung liegt nahe, daß nicht nur Platz für den Variablenwert selbst, sondern auch zusätzlich Speicherplätze zur Verwaltung der Daten benötigt werden. Auch das läßt sich herausbekommen. Dimensionieren wir das Feld einmal für ein einziges Element:

```
>55 DIM C(1)
>RUN
 3466
 7
 17
```

Und für zwei Elemente:

```
>55 DIM C(2)
>RUN
 3466
 7
 22
```

Diese Ergebnisse können wir mit unseren Kenntnissen wie folgt deuten:

- Der Wert einer numerischen Variablen belegt beim Beispielrechner 22 − 17 = 5 Bytes im Speicher.
- Für eine einfache numerische Variable werden in dieser BASIC-Version 7 − 5 = 2 Bytes zur Datenverwaltung benötigt.
- Für ein Feld werden dagegen 17 − 5 = 12 Bytes zur Datenverwaltung belegt, dies aber nur einmal je Feld.

Wir können sogar entschlüsseln, was es (in der gegebenen BASIC-Version) mit dem Datenverwaltungsaufwand auf sich hat. Diese Verwaltung hat ja als Hauptaufgabe, die betreffende Variable im Speicher aufzufinden. Dazu benötigt sie den Namen dieser

Variablen. Dieser Variablenname wiederum kann zwei Zeichen – bei uns einen Buchstaben und eine Ziffer – umfassen, die ihrerseits jeweils ein Byte belegen. Es steht also zu vermuten, daß die beiden Verwaltungsbytes gerade diesen Namen festhalten.

Sie sollten sich, bevor Sie weitergehen, noch ein wenig mit dem FRE-Befehl befassen. Er ist ein ungemein wichtiges Werkzeug, wenn man in das Innere der betreffenden BASIC-Version eindringen möchte. Blättern Sie das Buch noch einmal von vorne an durch, und versuchen Sie, alle unsere Vermutungen und Behauptungen über den Speicherplatz zu belegen. Notieren Sie sich die Versuchsergebnisse, und versuchen Sie, zu formulieren, was sie bedeuten könnten, gerade so, wie wir es oben getan haben. Sehen Sie sich insbesondere an, wie das mit den Feldern und Dimensionierungen eigentlich in punkto Speicherbedarf vor sich geht. Und wenn Sie dann an dieser Stelle hier wieder angekommen sind, dann schlagen Sie einmal in Ihrem Handbuch nach, ob und was es zur Speicherplatzbelegung aussagt. Wenn sich die Angaben mit Ihren Vermutungen decken – sehr gut. Wenn sie davon abweichen, dann dürfte das eher zu verstehen sein, als wenn Sie es nur gelesen hätten. Und wenn das Handbuch zu diesem Thema beharrlich schweigt, dann sollten Sie die gefundene Information irgendwo geeignet nachtragen.

Eine Frage drängt sich in dem Zusammenhang allerdings noch auf: Wie ist das mit dem Variablennamen? D.h., wenn beispielsweise ein Feld durch DIM M(15) dimensioniert worden ist, kann man dann noch die einfache (skalare) Variable M verwenden oder nicht? Die Antwort darauf ist durchaus nicht eindeutig. Sie hängt stark davon ab, wie die betreffende BASIC-Version konstruiert worden ist. Am besten probieren wir das einmal aus. Wir nehmen an, daß skalare und Feldvariable unabhängig voneinander definiert werden können, daß also M gefolgt von einem Index in Klammern einen anderen Speicherplatz bezeichnet als ein einfaches M. Das läßt sich ganz einfach nachprüfen, indem wir zwei derartigen Variablen verschiedene Werte zuweisen und uns anschließend ansehen, was der Rechner daraus gemacht hat.

```
>NEW
>10 DIM M(15)
>20 M(1) = 123.45
>30 M = 5432
>40 PRINT M(1),M
>RUN
 123.45         5432
```

Wenn Sie hier eine Fehlermeldung bekommen haben, dann kann Ihre BASIC-Version nicht zwischen skalaren und Feldvariablen unterscheiden. Bei einigen älteren Versionen ist das der Fall. Sie müssen in diesem Fall auf die Namen noch mehr Sorgfalt verwenden als sonst schon nötig. Insbesondere gilt das, wie so oft, für die Übernahme fremder Programme.

In unserer BASIC-Version wird zwischen skalaren und Feldvariablen zwar unterschieden, doch empfiehlt es sich auch hier der Deutlichkeit halber, nach Möglichkeiten verschiedene Bezeichnungen zu wählen. Im übrigen funktioniert das auch ohne ausdrückliche Dimensionierung des zu verwendenden Felds. Die einfache Angabe von Klammern nach dem Variablennamen genügt auch, solange der Index unter 10 bleibt. Der Unterschied liegt darin, daß der Rechner im Fall der ausdrücklichen Dimensionierung den Speicherplatz für das Feld vor der ersten Verwendung im Programm reserviert, während im anderen Fall der Speicherplatz dann (für 10 Feldelemente) reserviert wird, wenn zum erstenmal ein Bezug auf die Feldvariable im Programm auftaucht.

```
>10
>LIST
 20 M(1) = 123.45
 30 M = 5432
 40 PRINT M(1),M
>RUN
 123.45         5432
```

Wir haben damit die wichtigsten Grundlagen zum Umgang mit Feldern erarbeitet – so viel jedenfalls, wie wir voraussichtlich zur Lösung unserer Aufgabe benötigen. Dennoch haben wir mit unserer Betrachtung gerade die Oberfläche etwas angekratzt.

Es gibt noch eine Menge Fragen zu klären, wie z.B.:

- Gibt es eine Obergrenze für die Zahl der Feldelemente? Wenn ja, wo liegt sie?
  Das läßt sich einfach anhand des DIM-Befehls herausbekommen. Es ist anzunehmen, daß eine Fehlermeldung ausgegeben wird, wenn die Höchstgrenze für die Anzahl der Feldelemente überschritten wird.
- Können auch *Mehrfachindizes* verwendet werden?
  Das würde mehrdimensionalen Matrizen entsprechen, in denen ein Element beispielsweise durch einen Dreifachindex bestimmt wird: $M_{i,j,k}$, in BASIC: M(I,J,K).
- Falls ja, wieviele Indizes (d.h. Felddimensionen) sind in der gegebenen BASIC-Version möglich?
- Was geschieht, wenn ein Feld mehr als einmal dimensioniert wird?
  Viele BASIC-Versionen gestatten nur einen DIM-Befehl je Feld. Einige andere erlauben aber auch das Umdimensionieren, sofern dabei der belegte Speicherbereich nicht vergrößert wird. So läßt sich z.B. das durch DIM A(2,5) dimensionierte zweidimensionale Feld A, das $2 \times 5 = 10$ Elemente umfaßt, in manchen Versionen in ein eindimensionales Feld A(10) umdimensionieren. Das berührt nur die Art und Weise, in der auf die Feldelemente *zugegriffen* wird, nicht aber die im Speicher festgehaltenen Werte und ihre Reihenfolge. So etwas ist für Initialisierungszwecke recht brauchbar und taucht hin und wieder in veröffentlichten Programmen auf. Man sollte es erkennen und deuten können.
- Wie lautet der kleinstmögliche Index, der vom System angenommen wird (0 oder 1)?
  Diese Frage ist wichtig, da sie für die Übernahme fremder Programme entscheidend ist. Die meisten BASIC-Versionen verwenden 1 als niedrigsten Feldindex, manche lassen auch 0 zu, und einige wenige der neueren Versionen gestatten die Festlegung, welche Indexuntergrenze verwendet werden kann.

- Müssen die Feldindizes immer mit 0 oder 1 anfangen, oder kann man auch einen bestimmten Anfangspunkt vorwählen?
  Viele der höheren Programmiersprachen gestatten beispielsweise die Indexvorgabe (4:15), was heißt, daß das niedrigste Feldelement den Index 4 und das höchste den Index 15 besitzt. Die meisten BASIC-Versionen können das nicht, doch ist nicht auszuschließen, daß die eine oder andere ausgebaute Version die Festlegung der Indexgrenzen gestattet. In einigen Anwendungen wäre das eine bedeutende Hilfe.

Wir werden auf alle diese Fragen hier nicht weiter eingehen, vor allem deswegen, weil sich die verschiedenen BASIC-Versionen in diesen Einzelheiten sehr stark unterscheiden. Sie sollten sich jedoch unbedingt damit beschäftigen. Es ist sehr wichtig, daß Sie mit Ihrer BASIC-Version vertraut werden. Dazu können Sie in Ihrem Handbuch nachschlagen, besser aber ist es, wenn Sie versuchen, es selbst herauszufinden:

Experimentieren Sie selbst!

Was unsere Aufgabe betrifft, so können wir jetzt die ständig festzuhaltenden Variablen genauer definieren. Wir wandeln die bisherigen (skalaren) Angaben einfach in eindimensionale Felder um. Dazu allerdings müssen wir eine weitere Variable benennen, in welcher die Nummer des jeweils betrachteten Feldelements festgehalten ist. Nennen wir sie einfach

F – Nummer des gerade betrachteten Feldelements

Damit erhalten wir folgende Datenstruktur:

1. Daten, die für längere Zeit festzuhalten sind:
   - Bezeichnung des Gegenstands
   - Menge auf Lager: Variable M(F)
   - Datum der Änderung: Variable D(F)
   - Mindestmenge auf Lager: Variable M1(F)
   - Einzelwert der verschiedenen Artikel: Variable W1(F)

2. Daten, die nur im Verlauf der Bearbeitung benötigt werden
   - Nummer des gerade betrachteten Feldelements: Variable F
   - Entnahme oder Aufnahme: Variable M0
   - Anfangs- und Enddatum für den Ausdruck eines Lagerauszugs: Variablen D1, D2
3. Daten, die hieraus ermittelt werden können:
   - Gesamtwert der verschiedenen Artikel: Variable W
   - Gesamtwert des Lagerbestands: Variable W0
4. Daten, die im Einzelfall vorzugeben sind:
   - Befehle zur Auswahl unter den Programmteilen

3.12 Mehr über Zeichenketten

Zeichenkettenfelder

Es fehlt uns noch die Möglichkeit, den jeweiligen Artikelnamen in unserer Lagerdatei festzuhalten. Die erste Überlegung dazu wäre, die Felddefinitionen einfach auf Zeichenketten zu übertragen. Nennen wir die Variable z.B. N$, dann können wir unsere Definition erweitern um

N$(F) – Bezeichnung des Gegenstands

wobei F wie oben die Lage der betreffenden Zeichenkette im Feld betrifft. Nur – das funktioniert durchaus nicht bei allen BASIC-Versionen.

Der Hauptgrund liegt darin, daß Zeichenketten ursprünglich nicht zum Verarbeitungsbereich von BASIC gehörten. Man hat die Möglichkeiten dazu erst relativ spät in die bestehenden BASIC-Versionen eingefügt, und das geschah von Hersteller zu Hersteller verschieden. So unterscheiden sich die einzelnen BASIC-Versionen z.B. auch darin, ob und wie Zeichenkettenfelder definiert werden können.

Es gibt einfachere BASIC-Versionen, die zwar numerische Felder kennen, in denen aber Zeichenketten von vornherein nicht vorgesehen sind. Dann gibt es durchaus ausgefeilte Versionen, die aber vom Prinzip her Schwierigkeiten mit Zeichenkettenfeldern haben. Und schließlich ist da noch eine große Gruppe von BASIC-Versionen im Mikrocomputerbereich, welche die Definition von Zeichenkettenfeldern so ermöglichen, wie wir das eben versucht hatten.

Da es wichtig ist, mit Zeichenketten richtig umgehen zu können und vor allem auch die Lösungen von anderen Programmen richtig auf die eigene Version zu übertragen, bleibt uns nichts weiter übrig, als uns dieses Gebiet genauer anzusehen.

«Microsoft»- und «Northstar»-Versionen

Zunächst einmal: Was sind eigentlich Zeichenketten?

Nun – Zeichenketten sind nichts weiter als Ketten von alphanumerischen Zeichen, die irgendwie geeignet im Computer festgehalten werden. An diesem «irgendwie geeignet festgehalten» scheiden sich jedoch die Geister. Es gibt nämlich zwei grundlegend verschiedene Möglichkeiten, eine solche Zeichenkette zu betrachten. Nach den frühesten Anbietern dieser Versionen unterscheidet man sie im Mikrocomputerbereich als «Microsoft»- bzw. «Northstar»-Version. (Nach den Hauptanbietern im Minicomputerbereich bezeichnet man sie auch als «DEC»-[Digital Equipment Corporation-] bzw. «HP»-[Hewlett Packard-]Versionen.)

Der Unterschied liegt darin, daß die «Microsoft»-Versionen eine Zeichenkette als eine *Einheit* ansehen, während man sie in den «Northstar»-Versionen als *Feld* aus Einzelzeichen betrachtet. Damit aber bekommt der DIM-Befehl, auf Zeichenketten angewendet, eine völlig unterschiedliche Bedeutung.

```
DIM A$(15)
```

bewirkt in «Microsoft»-Versionen, daß ein *Feld,* bestehend aus 15

verschiedenen Zeichenketten, reserviert wird. In «Northstar»-Versionen hingegen wird dadurch Speicherplatz für *eine einzige* Zeichenkette mit (maximal) 15 Zeichen belegt. In ausgebauteren BASIC-Versionen nach dem Northstar-Prinzip kann man Zeichenkettenfelder durch Vorgabe zweidimensionaler Matritzen im DIM-Befehl definieren. Für unsere Aufgabe würde das etwa so aussehen:

```
DIM N$(F,15)
```

was Platz für F verschiedene Zeichenketten von je 15 Zeichen Länge reserviert. Das würde sich im großen und ganzen mit der «Microsoft»-Definition treffen. Leider geht das aber nur in einigen wenigen ausgebauteren Versionen. In den meisten Fällen von «Northstar»-artigen BASIC-Versionen sind Zeichenkettenmatrizen dieser Form nicht vorgesehen. Hier muß man sich anders behelfen.

Zeichenkettenfelder in Northstar-Versionen

Der Trick geht so: Wir vollziehen das ausführlich nach, was der Rechner in anderen Versionen beim Zugriff auf ein Feldelement von selbst macht. Wir fassen alle als Feldelemente vorgesehenen Zeichenketten zu einer einzigen großen Kette zusammen. Für den Zugriff auf eine solche Teilkette berechnen wir ihren Ort aus der Platznummer und der Länge der Einzelketten und isolieren sie dann aus der Gesamtkette.

Der Reihe nach: Zunächst müssen wir das «Feld» dimensionieren. Bei 15 Zeichen je Name und maximal F1 Einträgen erfordert das den Dimensionierungsbefehl:

```
DIM N$(F1*15)
```

Der erste Eintrag belegt dann die Plätze 1 bis 15 (oder 0 bis 14, wenn die betreffende Version mit Null zu zählen beginnt, wir wollen der Einfachheit halber aber bei Eins bleiben), der zweite

reicht von 16 bis 30 usw. Um den F-ten Eintrag zu erreichen, müssen wir also die Zeichenkette von Platz Nummer $(F-1)*15+1$ bis $(F-1)*15+15$ herausheben und besonders behandeln können. Wie macht man so etwas?

Nun – in diesem Fall ist das nicht besonders schwierig. Wenn die Zeichenkette wie ein Feld behandelt werden kann, dann müßte man auf die einzelnen Zeichen dadurch zugreifen können, daß man einfach die Platznummer des Zeichens als «Index» angibt. Und genau so geht das,

```
N$(1)
```

liefert das erste Zeichen von N$,

```
N$(13)
```

das dreizehnte und allgemein

```
N$((F-1)*15+I)
```

in unserem Fall das I-te Zeichen der F-ten Zeichenkette.

Nun wäre es ziemlich umständlich, müßte man eine längere Zeichenkette derart Stückchen für Stückchen zurückgewinnen. Man hat daher die Möglichkeit geschaffen, eine ganze Teilkette am Stück dadurch zu bestimmen, indem man den Platz des ersten und den des letzten Zeichens des gewünschten Ausschnitts bestimmt. So liefert z.B.

```
N$(16,30)
```

in unserem Beispiel mit Zeichen Nummer 16 bis Zeichen Nummer 30 gerade die zweite Teilkette, die in N$ festgehalten ist, und mit

```
N$((F-1)*15+1,(F-1)*15+15)
```

erhalten wir die F-te Teilkette.

Deren Wert können wir einer anderen (richtig dimensionierten!) Variablen zuweisen, z.B. durch

```
N1$ = N$((F-1)*15+1,(F-1)*15+15)
```

was auch umgekehrt geht:

```
N$((F-1)*15+1,(F-1)*15+15) = N1$
```

Wir müssen dabei nur darauf achten, daß beide Seiten gleich lang sind, d.h. dieselbe Anzahl von Zeichen enthalten. Bei N$((F − 1) * 15 + 1,(F − 1) * 15 + 15) ist das klar: Hier sind die Argumente so gewählt, daß gerade 15 Zeichen Platz finden. Anders ist das bei N1$. Nehmen wir an, wir hätten es auf 30 Zeichen dimensioniert:

```
DIM N1$(30)
```

Was passiert dann, wenn wir N1$ einen Wert zuweisen, sagen wir

```
N1$ = "Test"
```

enthält N1$ jetzt den Wert «Test», also nur diese vier Zeichen? Oder wurde N1$ bei der Dimensionierung gleich auch (auf 30 Leerschritte) *initialisiert* und enthält jetzt das Wort

```
"Test                          "
```

oder

```
"                              Test"
```

oder sonst etwas? Die Antwort darauf liegt nicht von vornherein fest. Es gibt in der Tat BASIC-Versionen, die eine «leere» Zeichenkette auf Leerzeichen initialisieren und den eigentlichen Text dann links- oder rechtsbündig einfügen – doch das ist selten. Die meisten heute gebräuchlichen Versionen enthalten das, was man ihnen ausdrücklich zuweist, nämlich gerade die vier Zeichen T-e-s-t in unserem Falle.

Wir können die Frage relativ rasch entscheiden, wenn es eine Möglichkeit gibt, die *Länge* der Kette zu ermitteln. Das läßt sich mit Hilfe einer besonderen Funktion machen, die sich der englischen Bezeichnung für «Länge»

length

gemäß

LEN

nennt und die als Argument noch eine Angabe der Zeichenkette benötigt, deren Anzahl festgehaltener Zeichen ermittelt werden soll. Wenden wir das auf N1$ an

```
>PRINT LEN(N1$)
 4
```

so besagt das, daß die Zeichenkette tatsächlich nur die vier Zeichen T-e-s-t enthält, die wir ihr zugewiesen haben. Lautet das Resultat dagegen 30, dann liegt eine der exotischen Versionen vor, die eine Zeichenkette nur «voll» behandeln kann. In diesem Fall muß man besondere Vorsicht walten lassen.

Es ist dennoch nicht so weit hergeholt, die Zeichenkette jeweils voll zu initialisieren. Wo liegt – vom Speicherplatz her gesehen – der Unterschied zwischen den Zeichenkettenvariablen N1$ mit dem Inhalt

```
"Test"
```

in der einen und

```
"Test                              "
```

in der anderen Version, wobei beide wohlgemerkt mit DIM N1$(30) dimensioniert worden sind.

DIM N1$(30) reserviert auf alle Fälle 30 Speicherplätze für den möglichen Inhalt der Zeichenkette. Im zweiten Fall wird diesem sofort ein bestimmter Wert, nämlich 30 Leerzeichen, gegeben. Im ersten Fall ist das anders. Hier haben die Speicherstellen irgendwelchen zufälligen Inhalt. Dieser Unsinn darf natürlich beim Verarbeiten der betreffenden Variablen nicht weitergegeben werden. Das heißt, es muß irgendeine Information über die Zahl der gültigen Zeichen in der Kette vorliegen, eine

Angabe, welche die gültigen Zeichen von den ungültigen abgrenzt.

Solange nur mit einer Kette alleine gearbeitet wird, ist das kein Problem. Anders wird das, wenn man einem *Teil* einer bestimmten Kette den Inhalt einer anderen zuweisen möchte. Nehmen wir an, die Zeichenkette N$ wäre wie folgt initialisiert worden:

```
>N$ = "12345678901234567890123456789012345678901234567890"
```

N1$ sei wie oben der Inhalt «Test» zugewiesen worden:

```
>N1$ = "Test"
```

Was geschieht nun, wenn wir der vier Zeichen langen Teilkette N$(5,8) den Wert von N1$ zuweisen?

```
>N$(5,8) = N1$
```

Nichts besonders Überraschendes, wenn wir uns das Ergebnis ansehen:

```
>PRINT N$
1234Test90123456789012345678901234567890
READY
```

Was mag aber geschehen, wenn die Zielkette im Original *leer* ist?

```
>N$ = ""
>N$(5,8) = N1$
>PRINT N$

READY
```

Nun – was ist geschehen? Zunächst: Die Zeichenkette N$ belegt im Speicher nach wie vor 30 Plätze. Zweitens: Durch N$ = «» haben wir die Längenmarkierung so gesetzt, daß alle diese 30 Speicherinhalte als ungültig erklärt werden. Drittens: Durch N$(5,8) = N4 müßte eigentlich die Längenmarkierung so umdefiniert worden sein, daß jetzt alle Zeichen bis hin zum achten gültig sind. *Das geht aber nicht!* Denn die Zeichen auf den

Plätzen 1, 2, 3 und 4 besitzen nach wie vor ungültigen Inhalt, und die Längenmarkierung bezeichnet immer nur den Platz vor dem *ersten* nicht mehr gültigen Zeichen. Das Ergebnis: Obwohl wir den Speicherstellen 5 bis 8 den Inhalt «Test» gegeben haben, erscheint die Variable N$(5,8) *als Gesamtes* leer.

Einige Versionen melden einen Fehler, wenn man versucht, derart ein «Loch» in einer Zeichenkette zu erzeugen. Einige wenige andere erkennen die Situation, akzeptieren die Zuweisung und füllen das Loch mit Leerzeichen auf. Das kann bei langen Ketten aber einige Zeit benötigen und fällt deshalb in der Regel fort. Wir wollen hier eine mehr durchschnittliche Version annehmen.

Aus dem Ganzen folgt für unser eigentliches Problem, daß wir

1. die Zeichenkette N$ auf die ganze Länge mit Leerzeichen füllen müssen, wenn das System das nicht automatisch macht, und
2. immer nur genau 15 Zeichen lange Abschnitte zuweisen dürfen, soll nicht versehentlich ein Loch erzeugt oder die Arbeit unnötig verlängert werden (indem der Rest zeitraubend vom System ergänzt wird) oder – schlimmer noch – die alte Information stehen bleiben und uns so alles verderben. (Es ist ein Unterschied, ob nach dem Inhalt «Kirschbaum» durch Zuweisung von «Aepfel» in der Teilkette nun «Aepfel » oder «Aepfelbaum» steht!)

So viel einmal zur Nachbildung eines Felds in «Northstar»-BASIC-Versionen. Wir werden uns im wesentlichen nur mit der verbreiteteren «Microsoft»-Version befassen. Dennoch sollte man wissen, was im gegebenen Fall womit gemeint ist und wie man um Einschränkungen der eigenen BASIC-Version herumprogrammiert. Sollten Sie eine der Versionen mit «Northstar»-ähnlicher Zeichenkettenhandhabung besitzen, so müssen Sie im folgenden auf eigene Faust experimentieren, um die von uns behandelten Anweisungen Ihrer BASIC-Version anzupassen.

Sicherung der Eingabedaten: VAL(X$)

Bleiben wir noch etwas bei Zeichenketten. Wir müssen nämlich die Eingabeoperationen in unserem Programm so narrensicher wie möglich machen. Das betrifft in erster Linie die Eingabe von Zahlenwerten. Ein normaler Lagerarbeiter dürfte bei folgendem Dialog vermutlich sauer reagieren (der von ihm eingetippte Wert ist unterstrichen):

```
Wieviele Exemplare sind entnommen worden? w
?TM ERROR in 2480
READY
>
```

Was ist passiert? Nun – bei der Eingabe hat sich der Gute vertippt und ist statt auf die «2» auf das kleine «w» gekommen. Damit muß man immer rechnen, doch daß dadurch erstens eine völlig unverständliche Meldung erfolgt ist und zweitens der Rechner darüber hinaus angehalten wurde – dafür kann man im Lager wohl kaum Verständnis aufbringen. So wendet sich das, was zum Schutz des Anwenders gedacht war, hier gegen ihn. Sicher, wir könnten eine Betriebsanleitung dazu legen, die besagt, daß bei ?TM ERROR mit CONT fortzufahren ist:

```
Wieviele Exemplare sind entnommen worden? w
?TM ERROR in 2480
READY
>CONT
Wieviele Exemplare sind entnommen worden?
```

Denn das Kommando CONT bewirkt ja, daß mit dem Programm an der unterbrochenen Stelle fortgefahren wird. Das Wahre ist das allerdings auch nicht. Wir müssen einen anderen Weg finden, das Programm so narrensicher wie möglich zu machen.

Im Idealfall sollte alles – auch der allerletzte Blödsinn – eingegeben werden können, und das System sollte darauf in irgendeiner Weise sinnvoll reagieren. Allerdings erfordert dieses «alles» einen ungemein hohen Programmieraufwand, der nicht

unbedingt immer gerechtfertigt ist. Wir wollen zunächst einmal nur die Zahleneingabe sicherer gestalten.

Das wiederum ist gar nicht so schwer, wie es scheinen mag. Wir haben weiter oben bereits einmal untersucht, wie der Rechner reagiert, wenn in eine *Zeichenkettenvariable* formal eine Zahl eingegeben wird. Sehen wir uns das noch einmal an:

```
>NEW
>10 INPUT "Wuerden Sie bitte eine Zahl eingeben";A$
>20 PRINT "Sie haben ";A$;" eingegeben."
>30 GOTO 10
>RUN
Wuerden Sie bitte eine Zahl eingeben? 123.45
Sie haben 123.45 eingegeben.
Wuerden Sie bitte eine Zahl eingeben? alpha
Sie haben alpha eingegeben.
Wuerden Sie bitte eine Zahl eingeben?
```

Und so weiter. Wir können jetzt alles mögliche eingeben, ohne daß der Rechner gleich «aussteigt». (Um genau zu sein, *fast* alles: Erinnern Sie sich bitte an das Experiment, das sie bezüglich der Eingabe verschiedener Zeichen gemacht haben.) Leider können wir die so übernommenen Zahlen nicht ohne weiteres weiterverwenden. Es sind für den Rechner nämlich gar keine Zahlen, sondern nichts weiter als irgendwelche Zeichenketten. Die Eingabe 123.45 in unserem Beispiel hat das gleiche Ergebnis für die Variable A$ wie die Zuordnung

```
A$ = "123.45"
```

Um unser Problem zu lösen, müssen wir dem Computer also irgendwie verständlich machen, daß wir nicht die *Zeichenkette*, sondern den *Zahlenwert* 123.45 gemeint haben. Dazu benötigen wir eine Funktion, die uns den *numerischen Wert* einer Zeichenkette liefern kann. «Wert» heißt auf englisch «value», und so finden wir im Wortschatz der meisten BASIC-Versionen denn auch die Funktion

VAL(X$)

Sie ermittelt den numerischen Wert des in Klammern angegebenen alphanumerischen Arguments. Sehen wir uns das etwas näher an. Dazu ergänzen wir unser obiges Testprogramm um

```
>25 PRINT "Es hat den numerischen Wert";VAL(A$)
```

Experimentieren wir damit ein bißchen

```
>RUN
Wuerden Sie bitte eine Zahl eingeben? 123.45
Sie haben 123.45 eingegeben.
Es hat den numerischen Wert 123.45
Wuerden Sie bitte eine Zahl eingeben? -123
Sie haben -123 eingegeben.
Es hat den numerischen Wert -123
Wuerden Sie bitte eine Zahl eingeben? 543E21
Sie haben 543E21 eingegeben.
Es hat den numerischen Wert 5.43E+23
```

Im letzten Versuch wird deutlich, daß wir es in den beiden Bestätigungsausdrucken tatsächlich mit verschiedenen Größenarten zu tun haben. In A$ steht Zeichen für Zeichen die von uns eingegebene «Zahl». VAL(X$) in der folgenden Zeile dagegen hat den Wert den Regeln von BASIC entsprechend in wissenschaftliche Notation umgewandelt, mit einer Stelle vor dem «Komma» und angepaßter Zehnerpotenz nach E, wobei sogar noch ein Vorzeichen eingefügt worden ist.

Wir können damit offensichtlich die unmittelbare Eingabe einer Zahl durch die Eingabe einer Zeichenkette mit nachfolgender Umwandlung in einen numerischen Wert mit Hilfe der VAL-Funktion ersetzen. Nur – lohnt sich der Aufwand eigentlich? Gewinnen wir wirklich die gewünschte Sicherheit gegen Tippfehler? Was geschieht, wenn wir irgendwelchen nichtnumerischen Unsinn eingeben?

```
Wuerden Sie bitte eine Zahl eingeben? Unsinn
Sie haben Unsinn eingegeben.
Es hat den numerischen Wert 0
Wuerden Sie bitte eine Zahl eingeben?
```

Zunächst einmal: Die Arbeit läuft ohne jede Fehlermeldung

weiter. Das ist genau das, was wir brauchen. Zum zweiten: Die Eingabe «Unsinn» hat nicht ihrerseits wieder einen unsinnigen Zahlenwert hervorgebracht, sondern den (logisch zu erwartenden) Wert Null. Wir können das bei Bedarf ausnutzen.

So bleibt eigentlich nur noch zu untersuchen, was geschieht, wenn wir einen Zahlenwert außerhalb der gültigen Grenzen angeben:

```
Wuerden Sie bitte eine Zahl eingeben? 123E123
Sie haben 123E123 eingegeben.
Es hat den numerischen Wert
?OV ERROR IN 25
READY
```

Das ist weniger schön. «OV» steht für «overflow» (Überlauf) und gibt an, daß bei der Umrechnung der Eingabe 123E123 in Zeile 25 der zulässige Zahlenbereich überschritten worden ist. In unserem Fall ist dabei sogar die Rechnung abgebrochen worden.

Es gibt allerdings auch BASIC-Versionen, die nur die Tatsache des Überlaufs melden (möglicherweise ein paarmal hintereinander) und anschließend mit dem falschen Wert weiterarbeiten. So oder so, die Situation ist unangenehm, und es ist nicht ganz einfach, den Fehler schon *vor* der Umwandlung durch VAL zu erkennen. (Es geht zu machen, erfordert aber beträchtlichen Aufwand.) Um die Sache nicht unnötig zu verkomplizieren, wollen wir hier annehmen, daß sich die Eingaben im gehörigen Rahmen halten.

Wir haben uns damit ein ganz wichtiges Werkzeug erarbeitet, mit dem jedes beliebige Programm mit wenig Aufwand benutzerfreundlicher gestaltet werden kann. Benutzen Sie es! Schreiben Sie – von kurzlebigen Tests einmal abgesehen – kein Programm, das mit dem INPUT-Befehl unmittelbar einer numerischen Variablen einen Wert zuweisen soll. Übernehmen Sie *immer* eine Zeichenkette und wandeln Sie anschließend in einen numerischen Wert um. Der geringe Mehraufwand wird durch die erhöhte Bedienungssicherheit mehr als aufgewogen.

3.13 Das Datumsproblem: LEFT$(X$), MID$(X$), RIGHT$(X$)

Damit können wir uns dem letzten noch offenstehenden Problem unserer Datenbeschreibung zuwenden: Wie steht es mit dem Datum? Wir haben dazu bereits einige Vorüberlegungen angestellt, die darin gipfelten, daß wir das Datum *im Rechner selbst* als Zahl der Form JJMMTT festhalten wollten. Dem 27. März 1984 z.B. würde die Darstellung 840327 entsprechen und dem 8. Dezember 1993 die Zahl 931208.

Diese Form eignet sich besonders gut, rasche Datumsvergleiche vorzunehmen, wie wir sie z.B. für einen Teilauszug aus dem Lagerbestand benötigen.

Nur hat sie den Nachteil, daß wir von einem «normal» denkenden Menschen eine derart verrückte Datumsangabe beim besten Willen nicht verlangen können. Wir müssen einen anderen Weg suchen, wobei uns Zeichenketten möglicherweise behilflich sein können.

Vereinbaren wir zunächst einmal, was als gültige Datumseingabe angesehen werden soll. Wenn wir der üblichen Schreibweise folgen, dann wären beispielsweise folgende Angaben zulässig:

1.1.85

03.11.1983

27.05.97

wohingegen wir zumindest fürs erste die ausdrückliche Angabe des Monatsnamens der Form

19. April 1989

ausschließen wollen. Es ist zwar nicht allzu schwierig, auch diese Form auszuwerten, doch wollen wir uns hier zunächst einmal mit dem einfacheren Fall beschäftigen, der ja folgende Zahlen erbringen soll:

1.1.85 – 850101

03.11.1983 – 831103

27.05.97 – 970527

Eines ist zunächst einmal sicher: Wir können eine Eingabe dieser Form nicht unmittelbar in eine irgendwie geartete Zahl umformen. Das nicht nur wegen der höchst verschiedenen Zahlenangaben, die für dasselbe Datum möglich sind, sondern ganz einfach auch deswegen, weil unsere VAL-Funktion da nicht mitmacht. Das zeigt sich sofort, wenn wir unser letztes Testprogramm heranziehen:

```
>RUN
Wuerden Sie bitte eine Zahl eingeben? 12.3.1983
Sie haben 12.3.1983 eingegeben.
Es hat den numerischen Wert 12.3
```

VAL wandelt so viel von der Zeichenkette in eine Zahl um, wie einer gültigen Schreibweise entspricht. Alles Nachfolgende wird ignoriert, und wenn gar nichts gültig ist, dann wird lediglich Null zurückgeliefert, wie wir oben gesehen haben. Das liegt ganz einfach am Umformungsmechanismus:

Der Rechner nimmt zunächst an, die Zahl sei Null. Dann sieht er sich das erste, d.h. ganz links stehende Zeichen der Kette an. Entspricht es einer gültigen Zahlendarstellung, dann wird es passend erfaßt und das nächste Zeichen betrachtet. Ist das zur gegebenen Darstellung gehörig, wird es umgewandelt und so weiter, bis ein Zeichen entdeckt wird, das nicht in die (einmal angefangene) Zahlenschreibweise paßt. An dieser Stelle wird die Umwandlung abgebrochen und der bis dahin erhaltene Wert dem Programm als numerische Entsprechung der untersuchten Zeichenkette übergeben.

Vollziehen wir das an unserem Beispiel nach:

Schritt	untersucht	ergibt	fertig?
1	12.3.1983	1	nein
2	12.3.1983	12	nein
3	12.3.1983	12.	nein
4	12.3.1983	12.3	nein
5	12.3.1983	12.3	ja

So wichtig es ist, zu wissen, wie VAL eine Zeichenkette bearbeitet, so wenig hilft es uns in unserem Problem weiter. Wir wissen zwar, was *nicht* geht – was geht denn dann eigentlich?

Wir wissen bis jetzt dreierlei:

1. Die Eingabe ist eine Zeichenkette der Form

   Tag.Monat.Jahr

2. Wir wollen daraus eine numerische Angabe der Form

   JJMMTT

   erzeugen.
3. Wir können *passende* Zeichenketten in Zahlen umwandeln.

Sehen wir den dritten Punkt einmal positiv an, dann heißt das, daß wir die drei Einzelbestandteile Tag, Monat und Jahr der Eingabe, jeweils für sich alleine, ohne weiteres in Zahlen verwandeln können. Die Zeichenkette

«21.05.87»

läßt sich beispielsweise aufspalten in die Angaben:

«21»,

«05» und

«87»

Diese wiederum ergeben die Zahlenwerte

«21» → 21

«05» → 5

«87» → 87

aus denen die interne Datumsangabe ganz leicht zusammengesetzt werden kann:

$$87 * 10000 + 5 * 100 + 21 = 870521$$

Das steht und fällt mit der Möglichkeit, die einzelnen Teilangaben aus der Eingabe zu isolieren, sie in Tag, Monat und Jahr zu zerlegen. Sehen wir uns die Sache daraufhin einmal an. Die einzelnen Teile der Zeichenkette

21.05.87

lassen sich ja durch die Trennungspunkte eindeutig voneinander unterscheiden. Wir brauchen also nur die Möglichkeit, die Lage dieser Punkte aufzufinden und die dadurch markierten Teilketten zu isolieren. Die dabei nötige Vorgehensweise läßt sich etwa so beschreiben.

Datum D ermitteln:

I ← Lage des ersten Punkts,
J ← Lage des zweiten Punkts,
T$ ← linke Teilkette bis I − 1 einschließlich,
M$ ← mittlere Teilkette von I + 1 bis J − 1,
J$ ← rechte Teilkette ab J + 1,
D ← 10000 * Wert(J$) + 100 * Wert(M$) + Wert(T$).
Ende.

Die Lage der Punkte ließe sich beispielsweise wie folgt herausbekommen:

Lage der Punkte:
  I ← 0, J ← 0, K ← 1,
  Bis K > Länge der Kette:
    Wenn an Stelle K ein Punkt ist, dann:
      Wenn I = 0 ist, dann:
        I ← K.
      sonst:
        J ← K.
      K eine Stelle weitersetzen.
  Ende (K > Länge der Kette).
Ende.

In einer BASIC-Version mit «Northstar»-Zeichenkettendarstellung könnten wir das Problem unmittelbar lösen. Wir brauchen die Strukturbeschreibungen nur Eins-zu-Eins zu übertragen, was z.B. folgendes Programm ergeben würde (die untersuchte Zeichenkette möge dazu in D$ stehen):

```
10 I = 0
20 J = 0
30 K = 1
40 IF K>LEN(D$) THEN 120
50 IF D$(K)<>"." THEN 100
60 IF I<>0 THEN 90
70 I = K
80 GOTO 100
90 J = K
100 K = K+1
110 GOTO 40
120 T$ = D$(1,I-1)
130 M$ = D$(I+1,J-1)
140 J$ = D$(J+1,LEN(D$))
150 D = 10000*VAL(J$)+100*VAL(M$)+VAL(T$)
```

Das ist zwar recht umständlich und läßt sich durch geschicktes Umgruppieren und Zusammenfassen ziemlich vereinfachen, doch uns soll es zunächst genügen. Überzeugen Sie sich jedoch davon, daß es funktioniert, indem sie das Programm anhand eines Beispiels mit Hilfe von Papier und Bleistift Schritt für Schritt nachvollziehen!

In «Microsoft»-Versionen sieht die Sache dagegen schon komplizierter aus. Da hier eine Zeichenkette bereits für sich als Einheit gilt, kann man auf die einzelnen Teilketten nicht einfach durch Angabe ihrer Lage zugreifen. Andererseits nutzen die schönsten Zeichenketten nicht viel, wenn man nicht auch mit Ausschnitten aus ihnen arbeiten kann. Kann man nun nicht unmittelbar auf diese Ausschnitte zugreifen, so muß man irgendein Werkzeug besitzen, das die gewünschten Teilketten liefert. Als solches Werkzeug hat man drei Funktionen geschaffen, mit denen man – ganz im Sinne unserer Strukturbeschreibung – aus einer als Argument vorgegebenen Zeichenkette das linke oder rechte Ende oder ein Teilstück aus der Mitte erhält. Die englischen Bezeichnungen dafür lauten «left» (links), «right» (rechts) und «mid» (inmitten), und so hat man die Funktionen denn auch benannt:

LEFT$(...) – linkes Ende

RIGHT$(...) – rechtes Ende

MID$(...) – Mittelstück

Beachten Sie das Dollarzeichen ($) im Namen dieser Funktionen: Es zeigt an, daß das Ergebnis keine Zahl, sondern eine Zeichenkette (ein «string») ist. Das erleichtert dem Rechner seine Aufgabe. Wir werden später noch einmal darauf eingehen.

Die Pünktchen zwischen den Klammern müssen noch durch die passenden Argumente ersetzt werden. Hier braucht man zunächst einmal die Angabe, auf welche Zeichenkette sich die Funktion beziehen soll. In der Regel ist das eine Zeichenkettenvariable, in unserem Fall D$, was zunächst einmal ergibt:

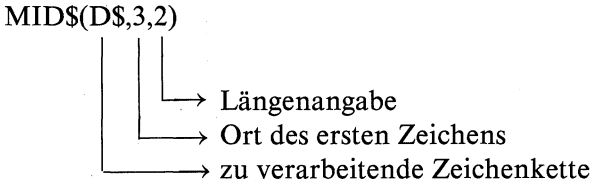
LEFT$(D$ ...) – linkes Ende von D$

RIGHT$(D$ ...) – rechtes Ende von D$

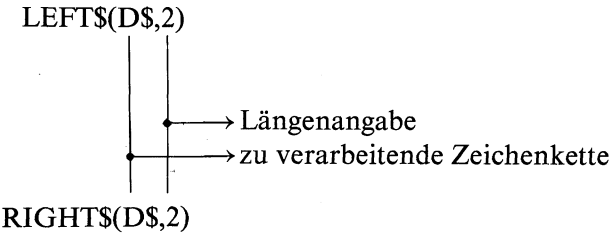
MID$(D$ ...) – Mittelstück von D$

Besonders im letzten Fall, der MID$-Funktion, wird deutlich,

daß das nicht ausreicht. Wo ist die Mitte anzusiedeln? Man gibt hier also noch zwei weitere Werte an: die Platznummer des ersten Zeichens, das erfaßt werden soll (das entspricht dem Index im «HP/Northstar»-Format) und dann – abweichend von unserer bisherigen Praxis – in der Regel noch die *Länge,* welche die Teilkette haben soll. (Einige Versionen benutzen allerdings auch die Lage des letzten Zeichens im Ausschnitt – Sie müssen dazu im Zweifelsfall Ihr Handbuch zu Rate ziehen.) Um in «21.05.87» die Monatsangabe zu erhalten, brauchen wir demnach eine Teilkette ab dem dritten Zeichen, die zwei Zeichen umfassen soll. Das würde so aussehen:



Bei der Isolierung von Teilketten am rechten oder linken Ende liegt der Anfangs- bzw. Endpunkt bereits fest. Es genügt also, wenn wir nur die benötigte Länge im Argument angeben. In unserem Beispiel wären das jeweils zwei Zeichen, was folgende Schreibweise ergibt:



Diese Funktionen können wir unmittelbar in unser Programmbeispiel übertragen:

```
10 I = 0
20 J = 0
30 K = 1
40 IF K>LEN(D$) THEN 120
50 IF MID$(D$,K,1)<>"." THEN 100
60 IF I<>0 THEN 90
70 I = K
80 GOTO 100
90 J = K
100 K = K+1
110 GOTO 40
120 T$ = LEFT$(D$,I-1)
130 M$ = MID$(D$,I+1,J-I-1)
140 J$ = RIGHT$(D$,LEN(D$)-J-1)
150 D = 10000*VAL(J$)+100*VAL(M$)+VAL(T$)
```

Auch hier die Aufgabe: Arbeiten Sie das einmal von Hand durch! Machen Sie sich aber die Mühe, die *Funktionen* tatsächlich zu berechnen und nicht einfach nur den Wert hinzuschreiben, «von dem man ja weiß, daß er da steht». So lernen Sie ihre Wirkungsweise am schnellsten kennen.

Vereinfachung der Datumsermittlung: INSTR, INDEX, Konkatenation

Wir können das Programm noch um einiges vereinfachen, wenn wir eine Funktion zu Hilfe nehmen, die sich allerdings leider selbst in ausgebauten BASIC-Versionen nicht immer findet. Da solche Aufgaben (wie in unserem Beispiel das Aufsuchen der Punkte) relativ oft vorkommen, hat man eine Funktion geschaffen, welche die Position einer Teilkette in einer anderen Kette ermittelt und so ein paar Programmzeilen einsparen hilft. Die Bezeichnung dieser Funktion ist wie bei allen späteren Zusätzen nicht ganz einheitlich; man findet im wesentlichen zwei Benennungen:

INSTR und INDEX

Das erste ist die Abkürzung für «in string» – «(Ort) in der Zeichenkette», das andere weist unmittelbar darauf hin, daß die Platznummer (index) in der Kette ermittelt werden soll. Als Argument brauchen diese Funktionen offensichtlich mindestens die Angabe der Zeichenkette, in der gesucht werden soll, und die Angabe der Kette, nach der gesucht wird. Nennen wir die durchsuchte Kette X$ und die zu suchende Y$, dann erhalten die Funktionen die Form

INSTR(X$,Y$) bzw. INDEX(X$,Y$)

was sich ungefähr so lesen läßt:

«Suche in der Zeichenkette X$ nach der Kette Y$ und liefere den Anfang des (ersten) Auftauchens von Y$ in X$ zurück!»

In unserem Fall sieht das so aus:

```
>PRINT INSTR("12.05.87",".")
 3
```

Der Punkt tritt in der Datumsangabe zum erstenmal an Platz 3 auf, und dieser Wert wird zurückgeliefert. Wir können natürlich auch etwas anderes suchen lassen, z.B.

```
>PRINT INSTR("12.05.87","5.8")
 5
```

d.h., die Teilkette «5.8» beginnt am fünften Platz der Kette «12.05.87».

Und was geschieht, wenn die gesuchte Kette im Original nicht enthalten ist? Probieren wir es aus:

```
>PRINT INSTR("12.05.87","ABC")
 0
```

In allen solchen Fällen erhalten wir eine Null als Ergebnis.

Den ersten Punkt der Kette finden wir ganz einfach. Doch wie kommen wir an den zweiten? Derartige Probleme treten in der Praxis öfter auf. Um ihnen gerecht zu werden, muß man noch ein

weiteres Argument mit in die Funktion aufnehmen. Das könnte entweder eine *Wiederholungsangabe* sein («Suche das *zweite* Auftreten von Y$ in X$!») oder es ist eine Angabe der *Anfangsposition,* ab welcher gesucht werden soll («Suche in X$ *ab Platz 4* nach Y$!»). Die erste Form würde es gestatten, rasch verschiedene Vorkommen einer Teilkette in einer anderen aufzusuchen, die andere dagegen ist etwas universeller anwendbar, da sie es gestattet, bestimmte Bereiche der Originalkette von vornherein auszublenden. Man verwendet deshalb in der Regel ein besonderes *Anfangsargument,* wenn nicht vom ersten Zeichen ab gesucht werden soll. Dieses Argument stellt man in den meisten Fassungen an die erste Stelle in der Klammer:

```
INSTR (K,X$,Y$)
```

was besagt:

«Suche in der Zeichenkette X$ ab Platz Nummer K nach dem ersten Auftreten der Teilkette Y$ und liefere deren Anfangsplatznummer zurück!»

In unserem Beispiel erhalten wir damit die Lage des zweiten Punkts durch

```
>PRINT INSTR(4,"12.05.87",".")
 6
```

Den neuen Suchanfang ermitteln wir aus dem Ort des ersten Auftretens, womit wir in unserem Programm die Zeilen

```
10 I = 0
20 J = 0
30 K = 1
40 IF K>LEN(D$) THEN 120
50 IF MID$(D$,K,1)<>"." THEN 100
60 IF I<>0 THEN 90
70 I = K
80 GOTO 100
90 J = K
100 K = K+1
110 GOTO 40
```

ersetzen können durch:

```
10 I = INSTR(D$,".")
20 J = INSTR(I+1,D$,".")
```

Das spart neun Programmzeilen und viel Rechenzeit ein.

Aber auch dann, wenn keine derartige INSTR-Funktion zur Verfügung steht, können wir das Programm vereinfachen. Wer sagt denn, daß wir unbedingt erst nach einem Punkt suchen müssen, bevor wir die betreffende Teilkette isolieren können? Das müßte man doch in einem Aufwasch erledigen können, wenn man die untersuchten Zeichen bis zum Auftreten eines Punkts zu einer neuen Kette zusammenfügen würde.

Diese Möglichkeit, zwei Teilketten zu einer größeren zusammenzufügen, bezeichnet man in der Regel als *Konkatenation*. Man findet sie allerdings nur in «DEC/Microsoft»-Versionen, da dazu unbedingt die Auffassung der Zeichenkette als einer Einheit gehört, die gerade so viel Raum im Speicher einnimmt, wie gültige Zeichen in ihr enthalten sind. Es ist kaum möglich, ohne großen Aufwand zu treiben, zwei Felder so aneinander zu hängen, daß die nicht mit gültigen Zeichen belegten Plätze ausgeschieden werden (sonst hätte das Ergebnis schlichtweg ein Loch, und was das bedeutet, haben wir uns bereits angesehen).

Die Konkatenation läßt sich als «Addition» der beiden Teilketten zu einem gemeinsamen Ganzen ansehen. Man verwendet in vielen Mikrocomputer-BASIC-Versionen daher den Operator (das «Rechenzeichen») «+» zu diesem Zweck. (In anderen BASIC-Versionen findet man als Konkatenationsoperatoren auch das Und-Zeichen «&» bzw. das Komma.) So erhalten wir z.B. aus den Ketten «Hans» und «Meier» durch Konkatenation:

```
>PRINT "Hans" + "Meier"
HansMeier
```

was allerdings noch nicht ganz das Wahre ist. Wir müssen noch einen Leerschritt einfügen, also:

```
>PRINT "Hans" + " " + "Meier"
Hans Meier
```

(Die vergessenen Leerschritte sind ein häufig vorkommendes Problem bei der Übernahme fremder Programme. Man kann sich einiges an Korrekturarbeit sparen, wenn man an solchen Stellen beim Abtippen etwas mitdenkt und den Originaltext aufmerksam auf Leerschritte in Zeichenketten durchsucht.)

Die Lösung unseres Problems der Zerlegung der Datumszeichenkette in Tag, Monat und Jahr läßt sich damit so beschreiben:

```
Datum D ermitteln:
   K ← 1, I ← 1
   Bis K > Länge(D$):
      Wenn MID$(D$,K,1) < > «.», dann:
         D$(I) ← D$(I) + MID$(DS,K,1),
      Sonst:
         I ← I +1. / * nächste Teilkette aktivieren * /
      K ← K+1.
   Schleifenende (K > Länge(D$)),
   Wenn D$(3) länger als 2 Zeichen, dann:
     D$(3) ← RIGHT$(D$(3),2).
  D ← 10000 * VAL(D$(3)) + 100 * VAL(D$(2)) + VAL(D$(1)).
Ende.
```

Wir tasten hier die Originalkette D$ wie vorher von links nach rechts ab. Ist kein Punkt erreicht, dann fügen wir das vorgefundene Zeichen der gerade aktivierten Teilkette an. Anders als oben haben wir für diese Teilketten ein Feld D$ verwendet, in dem die Tagesangaben in D$(1), die Monatsangaben in D$(2) und die Jahresangaben in D$(3) stehen. Die Variable I zählt den zugehörigen Feldindex durch. Mit ihrer Hilfe können wir einfach zwischen den verschiedenen Teilketten umschalten. Hätten wir die ursprünglich verwendeten Variablen T$, M$ und J$ verwendet, so wären zusätzliche Bedingungsabfragen notwendig geworden, durch welche sich das Programm unnötig verkompliziert hätte.

(Probieren Sie zur Übung einmal selbst aus, wie die Lösung mit T$, M$ und J$ aussehen müßte.) Außerdem testen wir, ob als Jahresangabe wirklich nur zwei Stellen eingegeben wurden, wie wir es zur Umformung benötigen. Wenn nicht, dann verwenden wir einfach die beiden letzten Ziffern dieser Angabe.

Wir können das wieder «vom Blatt weg» auf ein BASIC-Programm übertragen:

```
10 K = 1 : I = 1 : D$(1) = "" : D$(2) = "" : D$(3) = ""
20 IF MID$(D$,K,1)<>"." THEN D$(I)=D$(I)+MID$(D$,K,1)
   : GOTO 40
30 I = I+1
40 K = K+1
50 IF K<=LEN(D$) THEN 30
60 IF LEN(D$(3))>2 THEN D$(3)=RIGHT$(D$(3),2)
70 D = 10000*VAL(D$(3))+100*VAL(D$(2))+VAL(D$(1))
```

Beachten Sie die Initialisierung der Feldelemente von D$ in Zeile 10! Ohne diese Maßnahme würden in wiederholten Durchläufen die neuen Daten immer an die alten angehängt, so daß das Programm nur beim erstenmal ein richtiges Ergebnis liefern könnte.

3.14 Steueranweisungen für Fortgeschrittene

FOR-NEXT-Schleifen

Betrachten wir das letzte Programmbeispiel noch einmal auf seine Struktur hin. Der Hauptteil läuft in einer Schleife ab, die von Zeile 20 bis Zeile 40 reicht. Diese Schleife hat gegenüber den meisten anderen, die wir bis jetzt verwendet hatten, eine ganz besondere Eigenschaft: Eine *Laufvariable* (K) wird von einem bestimmten *Anfangswert* (1) bis zu einem bestimmten *Endwert* (Länge von D$) um einen bestimmten *Betrag* (1) in jedem Schleifendurchgang weitergezählt. Bis diese Laufvariable ihren Endwert erreicht hat, wird die in der Schleife angegebene Aktion (hier also Zeile 20 und 30) durchgeführt. Ist der Wert überschritten, so

wird die Schleife verlassen und mit dem Rest des Programms weitergemacht.

Solche Schleifenstrukturen kommen so häufig vor, daß man ihnen in BASIC einen eigenen Formalismus gegeben hat, den man sinngemäß so formuliert:

«FÜR die Dauer, in der die Variable K mit dem Anfangswert a BIS ZU einem Endwert e in jedem Schleifendurchgang um die SCHRITTWEITE s heraufgezählt ist, wird die im folgenden angegebene Operation durchgeführt.»

Das läßt sich stenogrammartig so zusammenfassen:

FÜR K = a BIS ZU e SCHRITTWEITE s

worauf die in der Schleife auszuführenden Aktionen folgen. Übersetzt man das wörtlich ins Englische, so erhält man die BASIC-Formulierung:

FOR K = a TO e STEP s

Allerdings genügt das noch nicht. Es muß noch das *Schleifenende* bezeichnet werden, was ganz einfach dadurch geschieht, daß man befiehlt:

«Den NÄCHSTEN Durchgang der zur Zählvariablen K gehörenden Schleife durchführen.»

Verkürzt und übersetzt ergibt das den Befehl

NEXT K

Sehen wir uns das einmal an einem einfachen Beispiel an: Wir zählen die Variable K in einer derartigen FOR-NEXT-Schleife durch und lassen uns den zugehörigen Wert auf dem Bildschirm anzeigen.

```
10 FOR K=1 TO 100 STEP 1
20 PRINT K
30 NEXT K
```

Lassen wir das laufen, so rauschen die Zahlen wie der Blitz über den Schirm – der richtige Genuß am Zählen will sich dabei nicht so richtig einstellen. Wir bräuchten eine Möglichkeit, die Zählerei zu verlangsamen, d.h., nach jedem Zählschritt eine kleine Pause einzulegen.

Das ist bei einigem Nachdenken nicht schwierig zu erreichen. Tatsache ist ja, daß der Computer zum Abarbeiten jedes Befehls etwas Zeit braucht. Es geht zwar ziemlich schnell vonstatten, aber ein bißchen Zeit ist immer nötig. Läßt man den Rechner daher irgendeine Operation, die keine Verwirrung anrichten kann, oft genug wiederholen, so summieren sich die dafür benötigten Ausführungszeiten ganz schön. Fügt man das in das Programm ein, so scheint der Computer zu pausieren.

Eine solche Operation ist beispielsweise das Heraufzählen des Werts einer Variablen. Und dazu wiederum bietet sich eine FOR-NEXT-Schleife fast von selbst an. Wir brauchen nicht einmal irgendwelche Berechnungen zwischen FOR ... und NEXT anzugeben. Eine «leere» Schleife tut es auch. Fügen wir doch so etwas einfach einmal in unser Programm ein:

```
>25 FOR L=1 TO 100 STEP 1 : NEXT L
```

Das paßt, wie man sieht, in eine einzige Programmzeile, und es hat seine Wirkung, wie sich bei RUN zeigt: Man kann die Zählarbeit auf dem Schirm jetzt relativ bequem verfolgen. Die Dauer der Verzögerung hängt allerdings stark von dem eingesetzten Computer und von der BASIC-Version ab. Versuchen Sie, für diese Verzögerungsschleife auf ihrem Gerät einen Endwert so zu finden, daß die Arbeit gerade angenehm zu verfolgen, nicht zu schnell und nicht zu langsam ist. Diesen Wert sollten Sie sich merken. Sie werden ihn relativ häufig benötigen.

Man hätte so etwas natürlich auch durch eine IF-THEN-Schleife realisieren können, und wir sollten uns das zum Vergleich auch ansehen.

```
>25 M=1
>26 M = M+1 : IF M<100 THEN 26
```

Das belegt eine Zeile mehr. Der eigentliche Unterschied (bei den meisten BASIC-Versionen) zeigt sich aber, wenn man das Programm startet: Es läuft beträchtlich langsamer als die FOR-NEXT-Schleife ab. Wir haben das auf einem unserer Testcomputer einmal gemessen. Das Programm

```
10 PRINT "START"
20 N = 0
30 N = N+1
40 IF N<10000 THEN 30
50 PRINT "STOPP"
```

brauchte für die Schleife zwischen START und STOPP 135 Sekunden, während die Form

```
10 PRINT "START"
20 FOR N=0 TO 10000 STEP 1
30 NEXT N
40 PRINT "STOPP"
```

ganze 14 Sekunden, also etwa ein Zehntel der Zeit benötigte! Wenn es also nicht ausnahmsweise auf möglichst lange Verzögerung ankommt (und in der Regel ist ja das gerade Gegenteil der Fall – das Programm soll so schnell wie möglich laufen), dann sollte man derartige FOR-NEXT-Schleifen unbedingt anderen Lösungen vorziehen. Sie sparen nicht nur Platz im Programmspeicher ein, sie beschleunigen darüber hinaus die Programmabarbeitung.

Man kann im übrigen die Angabe STEP 1 auch *entfallen* lassen. BASIC nimmt in diesem Fall an, daß es die Laufvariable in jedem Durchlauf um Eins weiterzählen soll. Auch ist man nicht auf positive Schrittweiten angewiesen. Man kann auch *rückwärts* zählen, indem man einen negativen STEP-Wert angibt. (Natürlich muß dann der Anfangswert a der Laufvariablen größer als der Endwert e sein.)

Unser kleines Testprogramm läßt sich damit noch verkürzen zu:

```
10 FOR K=1 TO 100
20 PRINT K
25 FOR L=1 TO 100 : NEXT L
30 NEXT K
```

Das Beispiel zeigt noch mehr: FOR-NEXT-Schleifen lassen sich ineinander *schachteln* – unter zwei Voraussetzungen:

a) Für jede Schleife muß eine eigene Laufvariable verwendet werden.

b) Jede in eine andere Schleife eingeschachtelte Schleife muß ganz beendet sein, bevor ein neuer Durchlauf der äußeren Schleife befohlen wird.

```
10 FOR N=0 TO 123
20 FOR M=0 TO 234
30 NEXT N
40 NEXT M
```

führt zu der Fehlermeldung

```
?NF ERROR IN 40
```

NF heißt dabei «NEXT without FOR» (NEXT ohne FOR) und besagt, daß der Computer zu dem M in Zeile 40 keinen passenden Schleifenanfang FOR M ... finden konnte. Er hat das Vorhandensein von Zeile 20 mit Abarbeiten des ersten Neudurchgangs der N-Schleife (d.h. ab dem ersten Abarbeiten von Zeile 30) vergessen.

Die mögliche *Schachtelungstiefe* von FOR-NEXT-Schleifen unterscheidet sich von BASIC-Version zu BASIC-Version, ja oft sogar von Computer zu Computer. Sehen Sie dazu in Ihrem Handbuch nach.

Außerdem unterscheiden sich die verschiedenen BASIC-Versionen darin, was geschieht, wenn die Laufvariable bereits *bei Eintritt in die Schleife* ihren Endwert erreicht hat. Lassen Sie dazu einmal folgendes Testprogramm laufen:

```
>NEW
>10 PRINT "Start"
>20 A = 1 : B = 1
>30 FOR I=A TO B
>40 PRINT "Schleife"
>50 NEXT I
>60 PRINT "Stopp"
```

Hier hat I bereits seinen Endwert erreicht, wenn Zeile 30 abgearbeitet worden ist. (Sie können sogar ins Extrem gehen und für B einen *niedrigeren* Wert angeben, etwa

```
>20 A = 1 : B = 0
```

Das bringt dasselbe Ergebnis.) Wenn Sie hier folgenden Ausdruck erhalten

```
>RUN
Start
Schleife
Stopp
```

dann besitzen Sie eine der Versionen, die eine FOR-NEXT-Schleife *mindestens einmal* abarbeiten. Diese Versionen sind in der Mehrzahl, da sich so das Schleifenende leichter auffinden läßt. Im Falle von

```
>RUN
Start
Stopp
```

arbeitet Ihre BASIC-Version eine FOR-NEXT-Schleife *nur dann ab, wenn die Schleifenbedingung erfüllt ist.*

Dieses unterschiedliche Verhalten von FOR-NEXT-Schleifen kann bei der Übernahme von fremden Programmen Schwierigkeiten bereiten. Analysieren Sie daher sorgfältig, ob ein solches von anderen übernommenes Programm auch das Schleifenverhalten annimmt, das Ihre BASIC-Version aufweist. Wenn nicht – das Problem ist einfach zu lösen:

Besitzen Sie eine Version, die eine FOR-NEXT-Schleife mindestens einmal abarbeitet, und darf das nach dem zu übernehmenden Programm nicht geschehen, so fügen Sie unmittelbar vor Eintritt in die Schleife einen Test ein, der zum Schleifenende verzweigt, wenn die Schleifenbedingung nicht erfüllt ist. Im anderen Fall müssen Sie den Test so auslegen, daß die Schleifenbedingung *auf alle Fälle* für mindestens einen Durchgang erfüllt ist.

Das geschieht am einfachsten, wenn man die Laufvariable in einem derartigen Fall auf Werte setzt, die einen solchen Schleifendurchgang ermöglichen. Achten Sie aber darauf, daß dadurch keine unerwünschten Nebenwirkungen in anderen Programmteilen entstehen (sogenannte Seiteneffekte)! Im Zweifelsfall empfiehlt es sich, das Programm geeignet umzuschreiben.

Ersetzen wir die IF-THEN-Schleifenkonstruktion in unserem Datumsprogramm durch eine FOR-NEXT-Schleife, dann nimmt es schließlich folgende Gestalt an:

```
10 I = 1 : D$(1) = "" : D$(2) = "" : D$(3) = ""
20 FOR K=1 TO LEN(D$)
30 IF MID$(D$,K,1)<>"." THEN D$(I)=D$(I)+MID$(D$,K,1)
40 I = I+1
50 NEXT K
60 IF LEN(D$(3))>2 THEN D$(3)=RIGHT$(D$(3),2)
70 D = 10000*VAL(D$(3))+100*VAL(D$(2))+D$(1)
```

IF-THEN-ELSE-Strukturen

Wir können noch etwas aus dem Beispiel entnehmen. Sehen wir uns an, wie die Strukturbeschreibung

Wenn an der Stelle K kein Punkt steht, dann:

$$D\$(I) \leftarrow D\$(I) + MID\$(D\$,K,1).$$

Sonst:

$$I \leftarrow I + 1.$$

in ein Programm umgesetzt worden ist. Wir haben hier eine Fallunterscheidung in einer allgemeineren Form als sonst üblich vorliegen, in einer logischen Struktur nämlich, die üblicherweise (im Englischen) durch die Befehlsfolge

IF (Bedingung) THEN
(1. Alternative, THEN-Teil)
ELSE
(2. Alternative, ELSE-Teil)

ausgedrückt wird (siehe Bild 9a). Leider lassen nur die ausgebauteren BASIC-Versionen derartige IF-THEN-ELSE-Konstrukte zu. In der Regel folgt der THEN-Teil in derselben Befehlszeile, in welcher der IF-Befehl steht, und der ELSE-Teil beginnt automatisch mit der nächsten Programmzeile. Bei umfangreicheren THEN- und ELSE-Teilen muß man diese als zusammenhängende Befehlsblöcke auf die eigentliche Entscheidung folgen lassen, wie es Bild 9b zeigt.

Ist die Bedingung erfüllt, so verzweigt das Programm zur (hier symbolisch mit XXX) bezeichneten Anfangszeile des THEN-Blocks. Dieser wird abgearbeitet und dann mit den restlichen Programmbefehlen (ab Zeile YYY) fortgefahren.

Ist die Bedingung nicht erfüllt, so springt das Programm sofort zu der Zeile, die auf den IF-Befehl folgt. Dort beginnt der ELSE-Teil, der durch einen Sprungbefehl GOTO YYY abgeschlossen wird. Damit kann das Programm den nachfolgenden THEN-Teil umgehen.

Dieser Umstand macht BASIC-Programme manchmal recht schwer lesbar. Es empfiehlt sich, bei der Analyse von BASIC-Programmen nach IF-THEN-Verzweigungen ausdrücklich auf solche IF-THEN-ELSE-Blockmöglichkeiten zu achten und die verschiedenen Alternativen irgendwie farbig hervorzuheben. Ein wichtiger Anhaltspunkt sind dabei die verschiedenen Sprungangaben, die den Beginn des THEN-Teils und der Programmfortsetzung (die oft mit diesem identisch ist!) bezeichnen. Beim Entwurf eines Programms sollten die beiden Alternativblöcke der Übersichtlichkeit halber so klein wie möglich gehalten werden. Sind die auszuführenden Arbeiten sehr umfangreich (so daß man sie z.B. nicht mehr auf einer Seite überblicken kann), dann ist es in der Regel besser, Unterprogramme für diesen Zweck einzusetzen. Das Programm ist dann wesentlich leichter zu verstehen und zu warten.

Nicht selten kommt es jedoch vor, daß der THEN-Teil einer solchen bedingten Verzweigung mit ein oder zwei Befehlen recht kurz ist. Das ist in unserem Beispielprogramm der Fall. Hier

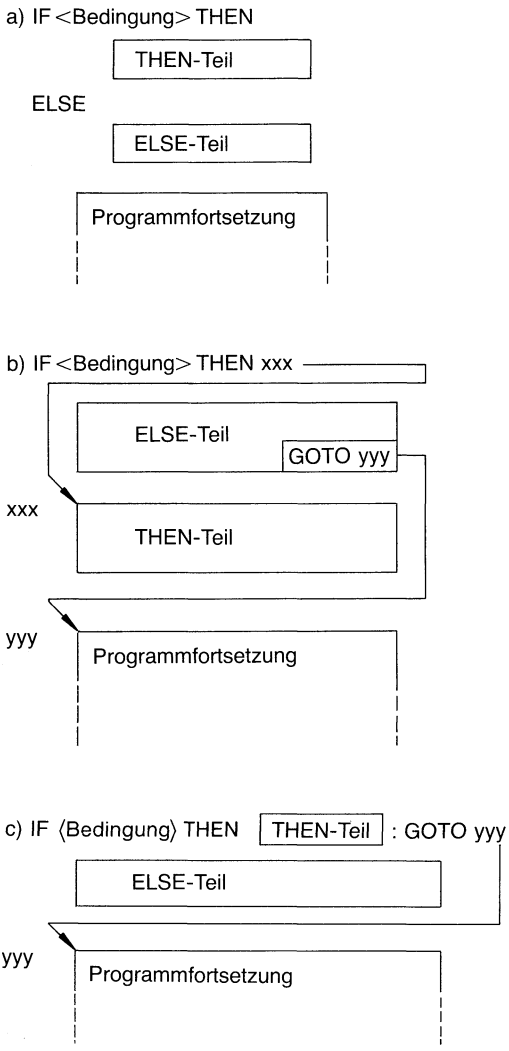


Bild 9 IF-THEN-ELSE-Strukturen

haben wir den THEN-Teil mit in der IF-Befehlszeile untergebracht, wodurch wir zu einer «natürlicheren» Befehlsform gekommen sind, wie sie Bild 10c zeigt. Man darf allerdings nicht vergessen, den THEN-Teil in diesem Fall durch einen Sprungbefehl GOTO YYY um den nachfolgenden ELSE-Teil herum zur Programmfortsetzung abzuschließen. Allerdings ist das nicht in allen BASIC-Versionen möglich. Der folgende Versuch kann darüber Auskunft geben:

```
>NEW
>10 INPUT "Bedingung";A$
>20 IF A$="0" THEN PRINT "Anfang" : GOTO 40
>30 PRINT "Mitte"
>40 PRINT "Ende"
>50 GOTO 10
```

Lassen wir das auf einer der moderneren Versionen laufen, dann erhalten wir in der Regel:

```
>RUN
Bedingung? 0
Anfang
Ende
Bedingung? 1
Mitte
Ende
```

Das entspricht genau dem erwarteten Verhalten. Es kann aber auch sein, daß Sie etwas anderes herausbekommen haben, nämlich:

```
>RUN
Bedingung? 0
Anfang
Mitte
Ende
Bedingung? 1
Mitte
Ende
```

In diesem Fall haben Sie eine der (mittlerweile seltenen) BASIC-Versionen, die in der IF-Befehlszeile nur einen einzigen Befehl abarbeiten (und den Rest als eine Art Kommentar anse-

hen). In diesem Fall müssen Sie IF-THEN-ELSE-Zweige immer in der Form von Bild 9b nachbilden. Sollten Sie sogar eine Syntax-Fehlermeldung bekommen haben, so studieren Sie bitte Ihr Handbuch.

Mehrfachentscheidungen: ON ... GOTO, ON ... GOSUB

Mit IF-THEN-ELSE-Konstruktionen können wir immer nur zwischen zwei Alternativen wählen. Sind mehr als zwei Fälle möglich, dann muß man IF-THEN-*Ketten* verwenden. Wir hatten ein Beispiel dazu bei der Besprechung der Menütechnik betrachtet, wo u.a. stand:

```
100 GOSUB 500 : REM Menueauswahl
110 IF A<>1 THEN GOTO 150 : REM naechste Moeglichkeit
120 GOSUB 1000 : REM Lagerfuehrung
130 GOSUB 3500 : REM Nachbestellmeldung
140 GOTO 100 : REM Schleifenanfang
150 IF A<>2 THEN GOTO 180 : REM naechste Moeglichkeit
160 GOSUB 5000 : REM Lageruebersicht
170 GOTO 100 : REM Schleifenanfang
180 IF A<>3 THEN GOTO 100 : REM Schleifenanfang
190 GOTO 10000 : REM Programm beenden
```

Das ist recht umständlich und wegen des Zerstückelns der Entscheidungskette auf den ganzen zugehörigen Programmbereich auch ziemlich unübersichtlich. Und es wird immer unübersichtlicher, je mehr Entscheidungsmöglichkeiten anstehen. In einem solchen Fall wäre es ganz praktisch, wenn man eine übersichtliche Abkürzung der IF-THEN-Kette zur Verfügung hätte.

Es gibt verschiedene Möglichkeiten zur Lösung dieses Problems, die alle darauf hinauslaufen, *in Abhängigkeit von einer Entscheidungsvariablen* (in unserem Fall ist das A) eine bestimmte Aktion auszuführen. Im einfachsten Fall gibt man die Zeilennummern, zu denen je nach Wert der Entscheidungsvariablen gesprungen werden soll, in geordneter Folge an und befiehlt dann sinngemäß:

«JE NACH dem Wert der Entscheidungsvariablen n GEHE ZU der Zeile, die als n-tes Argument folgt.»

Formal könnte das so aussehen

JE NACH n GEHE ZU $x_1, x_2, \ldots, x_{max}$

wobei x_1 die Zeilennummer ist, zu der bei $n = 1$ gesprungen werden soll, x_2 entsprechend bei $n = 2$ und x_{max} schließlich die Zeilennummer, die beim größtmöglichen Wert von n an der Reihe ist. Voraussetzung ist, daß die Zeilennummern *lückenlos* angegeben werden, also für jeden Wert von 1 bis max eine Nummer vorhanden ist, selbst wenn der eine oder andere Wert von n gar nicht vorkommen kann. In diesem Fall muß man eben einen *Hilfswert* (englisch: «dummy value») als Zielzeile angeben.

Wenn n z.B. die Werte 1, 3 und 4 annehmen kann, für die zu den Zeilen 1000, 2000 bzw. 3000 gesprungen werden soll, dann brauchen wir noch irgendeinen Füller für den Wert $n = 2$, der zwar normalerweise nicht vorkommt, dessen Platz in der Liste der Zeilennummern aber zur Sprungentscheidung mitgezählt werden muß. Man kann z.B. eine Zeile 32700 (also ganz am Ende des Programms) vorsehen, die die *Fehlermeldung*

```
32700 PRINT "Ungültiger Entscheidungswert"
```

trägt und befiehlt dann:

```
JE NACH N GEHE ZU 1000, 32700, 2000, 3000
```

Diese Möglichkeit der Mehrfachverzweigung wird in ausgebauten BASIC-Versionen mit dem Befehlspaar

ON ... GOTO

geboten, die genau den Sinn von

JE NACH ... GEHE ZU

hat. Unser Beispiel würde dann so aussehen:

```
ON N GOTO 1000, 32700, 2000, 3000
```

Das ist beträchtlich kürzer als die zugehörige IF-THEN-Kette. Auf das Menübeispiel angewendet ergäbe das folgenden Programmausschnitt:

```
100 GOSUB 500 : REM Menueauswahl
110 IF A<1 THEN GOTO 100 : REM Eingabefehler
120 IF A>3 THEN GOTO 100 : REM (Grenzen ueberschritten)
130 ON A GOTO 140,180,210
140 REM *** A=1: Lagerfuehrung ***
150 GOSUB 1000 : REM Lagerfuehrung
160 GOSUB 3500 : REM Nachbestellmeldung
170 GOTO 100 : REM Schleifenanfang
180 REM *** A=2: Lageruebersicht ***
190 GOSUB 5000 : REM Lageruebersicht
200 GOTO 100 : REM Schleifenanfang
210 REM *** A=3: Programm beenden ***
220 GOTO 10000 : REM Programm beenden
```

Beachten Sie, daß wir hier bereits *vor* der Verzweigungsentscheidung prüfen, ob die Eingabe für A in den zulässigen Grenzen liegt. Das ist zum einen notwendig, damit die ON-Anweisung kein falsches Argument erhält, zum anderen ist es aber auch viel logischer als im Fall der IF-THEN-Kette, wo diese Entscheidung erst ganz am Schluß getroffen wurde. Es ist ferner wichtig, daß die ON-Anweisung immer *Argumente größer als Null* erwartet. Bereits der Fall A = 0 ist in den meisten BASIC-Versionen ausgeschlossen und führt zu einer Fehlermeldung.

ON-Verzweigungen sind auch zu Unterprogrammen hin möglich, in welchem Fall der Befehl die Form

ON ... GOSUB

erhält. Allerdings müssen dann *alle* angesprungenen Programme als Unterprogramm abgefaßt sein, d.h. mit RETURN abschließen. In unserem Menübeispiel ist das durchaus anwendbar:

```
100 GOSUB 500 : REM Menueauswahl
110 IF A<1 THEN GOTO 100 : REM Eingabefehler
120 IF A>3 THEN GOTO 100 : REM (Grenzen ueberschritten)
130 ON A GOSUB 140,5000,10000
135 GOTO 100 : REM Schleife schließen
140 REM *** A=1: Lagerfuehrung ***
```

```
150 GOSUB 1000 : REM Lagerfuehrung
160 GOSUB 3500 : REM Nachbestellmeldung
170 RETURN
 . . .

5000 REM *** Lageruebersicht ***
 . . .
10000 REM *** Programm beenden ***
 . . .
```

Der Vorteil dieser Lösung liegt darin, daß die Menüschleife eine völlig in sich abgeschlossene Einheit geworden ist. Die Sprünge zum Menüanfang

```
GOTO 100
```

in den aufgerufenen Programmen sind durch einfaches RETURN ersetzt worden. Das führt den Rechner zu Zeile 135 zurück, von wo aus die Schleife geschlossen wird. Allerdings wird die ON-Anweisung selbst undurchsichtiger als im ersten Fall, da die Sprungziele nicht mehr sofort zu entschlüsseln sind. Man muß darum hier der Dokumentation des Programms noch mehr Aufmerksamkeit als ohnehin schon notwendig zuwenden. Immerhin – das Programm wird insgesamt kompakter und ist bei vielen Entscheidungsmöglichkeiten auf alle Fälle leichter zu verfolgen als eine umfangreiche IF-THEN-Kette.

3.15 Planung der Teilprogramme

Damit haben wir einen Großteil des zum weiteren Fortschritt notwendigen Werkzeugs erarbeitet. Von Feinarbeiten abgesehen, verfügen wir über alle Mittel, unser Problem einer einfachen Lagerverwaltung in BASIC zu lösen, zumindest soweit es die im inneren Speicher des Geräts stehenden Daten anbetrifft. Es ist Zeit, die bisherigen Ansätze zu systematisieren.

Sehen wir uns an, wie weit wir mit der Planung bis jetzt gekommen sind. Wir haben die Daten festgelegt, mit denen

gearbeitet werden soll, wobei hier in erster Linie die Variablenzuordnung wichtig ist:

D(F) – Datum der letzten Änderung eines Eintrags
D1 – Untere Datumsgrenze beim Ausdruck von Lagerauszügen
D2 – Obere Datumsgrenze beim Ausdruck von Lagerauszügen
D$ – Zeichenkette zur Datumsübernahme
D$(1) – Tagesinformation im übernommenen Datum
D$(2) – Monatsinformation im übernommenen Datum
D$(3) – Jahresinformation im übernommenen Datum
F – Nummer des gerade betrachteten Feldelements
F1 – Maximalzahl der behandelten Lagerartikel
M(F) – Menge auf Lager
M0 – Menge, die gerade entnommen oder hinzugefügt wurde
M1(F) – Mindestmenge auf Lager
N$(F) – Namen der gelagerten Artikel
W1(F) – Einzelwert der verschiedenen Artikel
W – Gesamtwert der verschiedenen Artikel
W0 – Gesamtwert des Lagerbestands

Außerdem hatten wir uns intensiv über die Steuerung des geplanten Programmablaufs mit Hilfe der Menütechnik unterhalten und dabei folgende Grobstruktur aufgestellt:

```
Lagerverwaltung:
    Programm einleiten,
    Solange Programm nicht beendet:
        Menü «Lagerverwaltung» anbieten,
        Auswahl anfordern,
        Wenn 1 ausgewählt, dann:
            Lagerführung:
                Menü «Lagerführung» anbieten,
                Auswahl anfordern,
```

```
                Wenn 1 ausgewählt, dann:
                    Artikel neu aufnehmen.
                Sonst:
                Wenn 2 ausgewählt, dann:
                    Artikel ändern.
                Sonst:
                Wenn 3 ausgewählt, dann:
                    Bestand verändern:
                        Menü «Bestand verändern»,
                        Auswahl anfordern,
                        Wenn 1 ausgewählt, dann:
                            Exemplare entnehmen.
                        Sonst:
                        Wenn 2 ausgewählt, dann:
                            Exemplare einfügen.
                    Ende (Bestand verändern).
            Ende (Lagerführung),
            Nachbestellmeldung.
        Sonst:
        Wenn 2 ausgewählt, dann:
            Lagerübersicht.
        Sonst:
        Wenn 3 ausgewählt, dann:
            Programm beenden.
Ende.
```

Wir müssen uns also insbesondere damit beschäftigen, was in den folgenden Unterprogrammen geschehen soll:

- Programm einleiten
- Artikel neu aufnehmen
- Artikel ändern
- Exemplare entnehmen
- Exemplare einfügen
- Nachbestellmeldung
- Lagerübersicht
- Programm beenden

Beschreibungshilfsmittel: Die Programmierhierarchie

Um den Wald vor lauter Bäumen nicht aus den Augen zu verlieren, empfiehlt es sich, die Abhängigkeit der verschiedenen Programmteile voneinander in einer *Programmhierarchie* wie in Bild 10 grafisch darzustellen. Im Gegensatz zu den Darstellungsmethoden, die wir bis jetzt kennengelernt haben, ist hier der eigentliche Programm*ablauf* nebensächlich. In der Programmhierarchie kommt es auf die gegenseitige Abhängigkeit der einzelnen Teilprogramme oder, wie man auch sagt, *Programmoduln* an. Diese Moduln sind in Kästen dargestellt und die gegenseitige Abhängigkeit durch Verbindungen zwischen den Kästen wiedergegeben. Dabei wird eine deutliche Hierarchie sichtbar, die sich in verschiedenen *Ebenen* in der Darstellung äußert (daher der Name dieser Beschreibungsform). Die weiter oben stehenden Moduln machen von den weiter unten stehenden Gebrauch. So benutzt beispielsweise der Modul «Bestand verändern» die Untermoduln «Exemplare entnehmen» und «Exemplare einfügen». Dagegen wird der Modul «Nachbestellübersicht» von den Moduln «Lagerführung» und «Lagerübersicht» gemeinsam verwendet. Demgegenüber sind die Moduln «Programm einleiten» und «Programm beenden» relativ unabhängig.

Ein anderer Unterschied, etwa zu den verschiedenen Formen der Strukturbeschreibung, wird deutlich, wenn man die Programmhierarchie genauer betrachtet: Jeder Modul kommt nur einmal vor, und er wird von genauso vielen übergeordneten Moduln verwendet, wie Verbindungslinien in ihn hineinführen. Damit kann diese Beschreibung Auskunft darüber geben, welche Beziehungen ein beliebiges Unterprogramm zu den übrigen Programmteilen hat. Diese Möglichkeit bieten die Strukturbeschreibungen nicht, in denen es auf die Verkettung im Programm*ablauf* ankommt, was bewirkt, daß ein solcher Programmodul u.U. recht häufig in der Beschreibung auftaucht. Dagegen besitzt in einem gut strukturierten Programm vom Ablauf her gesehen jeder

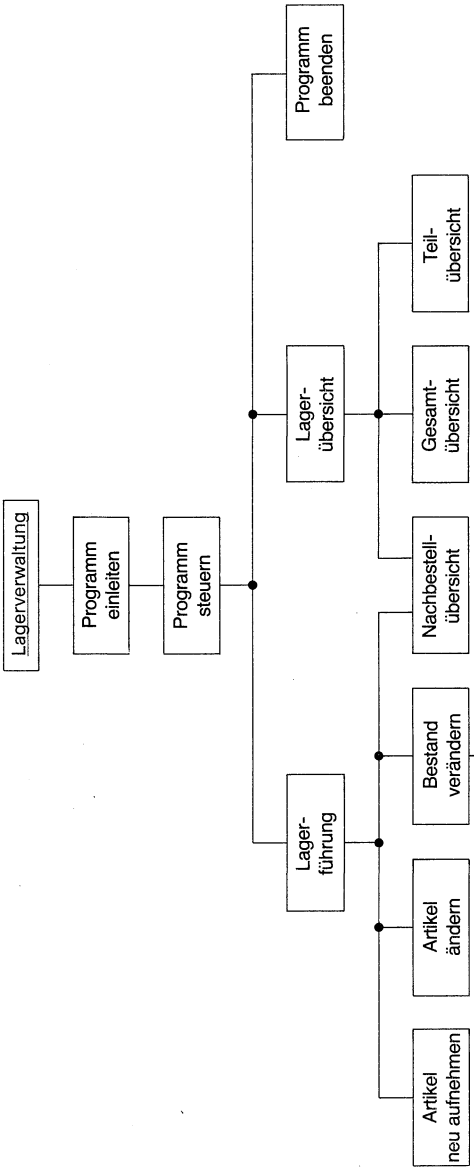


Bild 10 Hierarchiediagramm des Lagerverwaltungsprogramms

Modul nur einen einzigen Eingang und einen einzigen Ausgang.

Eine Hierarchiebeschreibung hat demzufolge statischen Charakter: Sie bezeichnet die Abhängigkeit der Programmteile, wie sie beispielsweise im Programmausdruck (im sogenannten Listing) vorliegen. Insofern ist sie bei der Planung und Wartung sehr wertvoll: Man erkennt Seiteneffekte bei Änderungen relativ einfach, und man kann doppelten Programmieraufwand vermeiden, indem man z.B. gleichartige Moduln aus dem ursprünglichen Entwurf zu einem zusammenfaßt.

Demgegenüber sind Strukturbeschreibung und Struktogramm von dynamischem Charakter: Sie stellen die Struktur der Programmteile bei der Abarbeitung des Programms dar und geben so seine logische Gliederung wieder. Man kann mit ihrer Hilfe verfolgen, wie die Eingabedaten im Verlauf der Abarbeitung in die Ausgabedaten umgeformt werden. Das macht ihre zentrale Stellung bei der Programmplanung aus.

Das Flußdiagramm wiederum gibt noch dynamischere Zusammenhänge wieder. Es beschreibt die Abfolge der einzelnen Programmteile wesentlich genauer als die Strukturbeschreibung, bietet dafür aber auch weniger Übersicht. Sein Wert liegt besonders darin, die gegenseitige Verkettung der einzelnen Operationen verfolgen und verfeinern zu können. Das Schwergewicht seiner sinnvollen Nutzung liegt in der Detailplanung, wohingegen die (ausschließliche) Verwendung von Flußdiagrammen auf höheren Beschreibungsebenen leicht zu unstrukturierten und schwer wartbaren Programmen führt.

Wie schon öfter betont, schließen sich die verschiedenen Beschreibungsformen nicht gegenseitig aus, sondern können – sinnvoll eingesetzt – einander beträchtlich unterstützen. Jede von ihnen erfaßt einen besonderen Gesichtspunkt des Programms, den die anderen nicht oder nur ungenügend wiedergeben können. Dabei sind die hier vorgestellten Beschreibungsformen bei weitem nicht erschöpfend. Es gibt noch sehr viele andere Ansätze, die entweder die hier verwendeten nur leicht abwandeln,

manchmal aber durchaus neue Zusammenhänge deutlich machen können. Wir können in dieser Einführung natürlich nicht näher darauf eingehen, doch ist es wichtig zu wissen, daß es so etwas gibt und daß man sich bei besonderen Entwurfsproblemen durchaus auch um andere Beschreibungsformen kümmern sollte.

Welche der verschiedenen Möglichkeiten man letztlich in den Mittelpunkt der eigenen Programmiertätigkeit stellt, ist in erster Linie eine Frage der persönlichen Arbeits- und Ausdrucksgewohnheiten. Der eine mag lieber mit Flußdiagrammen arbeiten, weil sie seiner Vorstellungstätigkeit besser entgegenkommen, der andere verwendet sehr viele Struktogramme, und wir werden uns – in erster Linie weil sie leichter zu schreiben und zu drucken sind – vor allem auf Strukturbeschreibungen stützen.

Speicherplatzaufteilung FRE, INT, DIM und CLEAR

Am Anfang der Arbeit steht die Einleitung des Programms. Hier müssen die wichtigsten Variablen dimensioniert und initialisiert sowie der letzte Lagerbestand (genauer seine Beschreibung) in den Speicher des Rechners geladen werden. Halten wir das schon einmal fest:

Programm einleiten:
    Variablen dimensionieren und initialisieren,
    Lagerbestand laden.
Ende.

Zu dimensionieren sind auf alle Fälle die Variablen, welche später die einzelnen Artikelbeschreibungen aufnehmen sollen, also:

D(F) – Datum der letzten Änderung eines Eintrags
M(F) – Menge auf Lager
M1(F) – Mindestmenge auf Lager
N$(F) – Namen der gelagerten Artikel
W1(F) – Einzelwert der verschiedenen Artikel

Das «F» in diesen Feldvariablen ist der laufende Index, wie er in den Rechnungen verwendet wird. Als solcher ist er hier nicht verwertbar. Wir müssen diese Felder auf die maximale Größe dimensionieren, die unser Lagerbestand umfassen soll. Die maximale Größe wiederum richtet sich nach den Systemgegebenheiten, in erster Linie nach dem Umfang des für die Daten noch zur Verfügung stehenden Speicherbereichs. Theoretisch wären in der hier eingesetzten BASIC-Version Felder bis zu 32 768 Elementen je Felddimension möglich. Theoretisch – das heißt, der Rechner kann Indexangaben im Bereich von 0 bis 32 767 (in unserer Version) verarbeiten. Dazu kommt noch, daß in unserer BASIC-Version – immer theoretisch – bis zu 255 Dimensionen je Feld verarbeitet werden können. Nun rechnen wir einmal: Eine Zahl belegt, den Platz für die Verwaltung nicht gerechnet, in unserem Fall 5 Bytes im Speicher. («Byte», um es zu wiederholen, ist die Einheit, in welcher die Speichergröße gemessen wird.) Dann benötigt ein Feld mit 255 Dimensionen zu je 32 768 Elementen einen Speicherplatz von mindestens $255 * 32\,768 * 5 = 41\,779\,200$ Bytes. So viel Speicherplatz finden Sie höchstens bei Großrechnern vor. Und selbst wenn wir je Feld nur eine Dimension benötigen, sind das mindestens $32768 * 5 = 163\,840$ Bytes, wenn wir die Möglichkeiten voll ausschöpfen wollten. Das ist bei weitem mehr als die vier- bis sechzigtausend Bytes, die in einem Mikrocomputer heutzutage für Programm *und* Variable bereitstehen.

Auf der anderen Seite sollten wir die Felder natürlich nicht zu klein halten, wollen wir das Programm auch nur einigermaßen praktisch verwendbar machen. Das heißt, wir kommen mit der Ersatzdimensionierung auf 10 Feldelemente, die der Rechner automatisch vornimmt, nicht sehr viel weiter. Wir müssen einen Weg finden, den Speicherplatz so intensiv wie möglich zu nutzen.

Dazu kommen wir nicht umhin, etwas zu rechnen. Zunächst einmal sollten Sie wissen, wieviel Platz ein Feldelement im Speicher belegt. Sie können das in Ihrem Handbuch nachschlagen

oder auch mit Hilfe der FRE-Funktion herausfinden. Wir haben das oben bei der Besprechung von Feldern ja bereits beschrieben. In unserem Fall sind das je Element eines numerischen Felds 5 Bytes, wozu noch eine gewisse Anzahl von Bytes kommt, die zur Verwaltung benötigt werden. Da diese je Feld nur einmal benötigt werden, fallen sie im Gesamtbedarf nicht so sehr ins Gewicht – unterschlagen wir sie einfach.

Weiter brauchen wir für jeden Namen 15 Zeichen Speicherplatz in einem alphanumerischen Feld. Jedes alphanumerische Zeichen belegt nun genau ein Byte im Speicher, so daß hier die Rechnung einfach ist: 15 Bytes je Feldelement. Allerdings kommt für jedes alphanumerische Feldelement noch besondere Verwaltungsinformation hinzu. In unserer BASIC-Version sind das 3 Bytes, so daß wir je Namen 18 Bytes ansetzen müssen.

Insgesamt benötigen wir für einen Eintrag in die Lagerdatei, wie sich aus unserer Variablenliste ergibt, vier numerische und ein alphanumerisches Feld. Das belegt im Speicher $4 * 5 + 18 = 38$ Bytes je Eintrag.

Diese 38 Bytes sind unsere Grundlage. Rechnen wir nun für die anderen Variablen und den sonstigen Bedarf des Rechners bei seiner Arbeit großzügig 1000 Bytes ein (man könnte es auch genau berechnen, aber hier soll ein Überschlag genügen), dann können wir die Feldvariablen in unserem Programm *dynamisch dimensionieren*. «Dynamisch» heißt hier, wir machen die Feldgröße vom tatsächlich freien Speicherplatz abhängig, anstatt sie von vornherein starr im Programm festzulegen. Das gibt uns die Garantie, daß der jeweils verwendete Rechner wirklich voll für unseren Zweck ausgenutzt werden kann. Die Größe des verfügbaren Speichers erhalten wir wiederum ganz einfach, wenn wir uns der FRE-Funktion bedienen. Bezeichnen wir die maximale Feldgröße mit

F1 – Maximalzahl der behandelten Lagerartikel

dann erhalten wir ihren aktuellen Wert dadurch, daß wir am Programmanfang folgende Rechnung durchführen:

```
F1 = (FRE(0)-1000)/38
```

FRE(0) liefert uns die Anzahl von noch verfügbaren Bytes im Speicher, wovon wir die «Sicherheitsreserve» von 1000 Bytes abziehen und das Ganze durch die Anzahl der von jedem Eintrag benötigten Bytes teilen. Das ergibt – beinahe – die Zahl der maximal möglichen Einträge in unserem System. Das «beinahe» deshalb, weil wir auf diese Art und Weise mit hoher Wahrscheinlichkeit eine gebrochene Zahl erhalten. Für die Dimensionierungsangabe sind aber nur positive ganze Zahlen zugelassen. Wir müssen also aus diesem Ergebnis irgendwie noch eine ganze Zahl machen – die noch dazu nicht größer als die gebrochene Zahl sein darf.

Nun – diese Möglichkeit gibt es in BASIC. Es steht uns hier wieder eine Funktion zur Verfügung, die zu einem Zahlenargument dessen ganzzahligen Anteil (das ist der vor dem «Komma» stehende Teil) zurückliefert. Eine ganze Zahl wird im Englischen als «integer» bezeichnet, wodurch die Funktion die Form

INT(X)

hat. «X» ist hier irgendein Ausdruck, der einen numerischen Wert liefert, also eine Zahl, eine Variable oder eine Rechenformel. Die Wirkungsweise ist ganz einfach:

```
>PRINT INT(123.456)
 123
>PRINT INT(0.123)
 0
>PRINT INT(13E29)
 1.3E+30
```

INT sieht sich das Argument an, unterschlägt alles, was nach dem Komma (d.h. dem Dezimalpunkt) steht und liefert den Rest zurück. Dabei spielt die Größe der Zahl (sofern sie im erlaubten Bereich liegt) keine Rolle. Man muß allerdings bei negativen Zahlen etwas vorsichtiger sein. Viele BASIC-Versionen verhalten sich hier unregelmäßig:

```
>PRINT INT(-123.456)
-124
>PRINT INT(-0.123)
-1
>PRINT INT(-123)
-123
```

Wenn die negative Zahl einen gebrochenen Anteil hat, dann wird sie zur nächstnegativeren ganzen Zahl hin abgerundet. Steht im Argument des INT-Befehls dagegen von vornherein eine negative ganze Zahl, dann bleibt ihr Wert voll erhalten. Das hängt mit der Art und Weise zusammen, in welcher der Rechner intern mit den Zahlen umgeht und soll uns nicht weiter berühren. Nur – wissen sollte man es, zumindest wenn das System plötzlich irgendwelche merkwürdigen Rechenfehler macht.

In unserem Fall können wir von vornherein ein positives Argument annehmen und die benötigte Feldobergrenze wie folgt ermitteln:

```
F1 = INT((FRE(0)-1000)/38)
```

Die Dimensionierung der Variablen unserer Lagerdatei erfolgt dann mit diesem F1 als Parameter:

```
DIM D(F1), M(F1), M1(F1), N$(F1), W1(F1)
```

Wir nutzen hierbei die Tatsache aus, daß der DIM-Befehl mehrere durch Komma getrennte Argumente zuläßt. Für jedes dieser Argumente allerdings muß in Klammern die Anzahl der Feldelemente angegeben werden, die für sie im Speicher reserviert werden sollen.

Es gibt bei manchen BASIC-Versionen allerdings eine Schwierigkeit bei der Verarbeitung von Zeichenketten, die den verwendeten Speicherplatz betrifft (das gilt in erster Linie für «Microsoft»-Versionen). Machen Sie einmal folgenden Versuch:

```
>NEW
>10 FOR I=1 TO 25
>20 A$ = A$+"1234567890"
>30 PRINT I
>40 NEXT I
```

Das verlängert in jedem Schleifendurchlauf die Kette in A$ um zehn Zeichen. Der nachfolgende Ausdruck der Laufvariablen I soll als Bestätigung für Zeile 20 dienen: Wenn er ausbleiben sollte, ist wahrscheinlich ein Fehler in Zeile 20 aufgetreten. Man kann ja nie wissen. Jedenfalls sehen wir daran, ob das Programm auch arbeitet. Eine Rückmeldung vom Programm ist immer sinnvoller als die Frage zwischen RUN und READY vor dem toten Bildschirm: «Tut er jetzt was, oder tut er rein gar nichts?».

Es kann nun sein, daß Sie folgendes Ergebnis bekommen haben:

```
>RUN
1
2
3
4
?OS ERROR IN 20
```

Das heißt, viermal wurde die Schleife erfolgreich durchlaufen, und beim Versuch, die Länge der Zeichenkette auf 50 aufzustokken, gab es einen «OS»-Fehler. Das kann zweierlei bedeuten. Entweder reicht der Speicherplatz, den eine Zeichenkettenvariable zur Verfügung stellt, nicht für 50 Zeichen aus, oder es steht für Zeichenketten überhaupt nur beschränkter Speicherplatz zur Verfügung. Nun – das läßt sich herausbekommen. Anstatt eine einzige Zeichenkette mit 100 Zeichen vollzustopfen, können wir es ja auch einmal mit zehn Zeichenketten zu je zehn Zeichen versuchen. (Zur Vorsicht werden wir sogar zehn verschiedene Zeichenkombinationen verwenden.)

```
>NEW
>10 A$ = "1234567890"
>15 PRINT 1
>20 B$ = "2345678901"
>25 PRINT 2
>30 C$ = "3456789012"
>35 PRINT 3
>40 D$ = "4567890123"
>45 PRINT 4
>50 E$ = "5678901234"
>55 PRINT 5
>60 F$ = "6789012345"
```

```
>65 PRINT 6
>70 G$ = "7890123456"
>75 PRINT 7
>80 H$ = "8901234567"
>85 PRINT 8
>90 I$ = "9012345678"
>95 PRINT 9
>100 J$ = "0123456789"
>105 PRINT 10
>RUN
 1
 2
 3
 4
?OS ERROR IN 50
```

Es liegt also offenbar nicht am Speicherplatz für eine einzige Variable. So ist es auch, wie uns das Handbuch belehrt, nach dem «OS» für «out of string space» (kein Speicherplatz für Zeichenketten mehr übrig) bedeutet. Zur Erklärung wird hinzugefügt, daß die betreffende («Microsoft»-)BASIC-Version von sich aus nur 50 Bytes im Speicher für Zeichenketten bereitstellt. Und diese werden sowohl für die zu speichernden Zeichen als auch von den zugehörigen Verwaltungsdaten belegt.

Wie kommt das? Vergleichen wir die Situation einmal mit der von «Northstar»-Versionen, bei denen dieser Fehler in der Regel nicht auftritt. Dort reservieren wir ausdrücklich durch einen Dimensionierungsbefehl den für die betreffende Zeichenkette notwendigen Speicherplatz. Der Rechner weiß also immer, mit wievielen Bytes er es zu tun hat und kann den reservierten Platz fest im Speicher anlegen.

Anders bei «Microsoft»-Versionen. Hier weiß der Rechner über die Länge der einzelnen Zeichenkette in keiner Weise Bescheid. Anstatt nun hinzugehen und eine bestimmte Höchstlänge für die Zeichenketten vom System aus festzulegen (und so bei vielen kurzen Zeichenketten sehr viel wertvollen Speicherplatz zu verschwenden), wird der von diesen belegte Platz *dynamisch* im Speicher verwaltet. D.h., man belegt immer gerade so viele Bytes im Speicher, wie für die festzuhaltende Information notwendig ist.

Wird die Kette nun länger gemacht, so muß der gesamte folgende von den Variablen belegte Speicherplatz verschoben werden, um die dazukommenden Zeichen unterzubringen. Und umgekehrt gibt man beim Verkürzen der Zeichenkette den Platz im Speicher durch Verschieben der nachfolgenden Daten wieder frei. Daten im Speicher zu verschieben kostet aber Zeit, und zwar um so mehr, je mehr zu verschieben sind. Das kann die Abarbeitung des Programms beträchtlich verlangsamen.

Also hat man einen anderen Weg gewählt: Anstatt die Zeichenketten wie sie kommen zwischen die anderen Variablenwerte zu packen, reserviert man einen eigenen Speicherbereich für sie, den sogenannten «string space». Das sind bei der betrachteten Version zunächst einmal 50 Bytes, was für einfache Anwendungen durchaus ausreicht. Man braucht jedenfalls nur noch die paar Dutzend oder paar Hundert Bytes im Speicherraum für Zeichenketten hin und her zu schieben und nicht mehr die ganzen numerischen Variablen mit. Das spart je nach Anwendung sehr viel Zeit ein.

Das einigermaßen anspruchsvolle Arbeiten mit Zeichenketten wäre jedoch ein Ding der Unmöglichkeit, würde dieser Speicherbereich ein für allemal festliegen. Man hat deshalb dafür gesorgt, daß man ihn auf eine geeignete Länge *initialisieren* kann. Dieses Initialisieren geschieht durch den Befehl

```
CLEAR(X)
```

durch den alle Variablen im Speicher gelöscht (Englisch: cleared) werden und dieser gleichzeitig neu aufgeteilt wird. Diese Neuaufteilung erfolgt so, daß das System X Bytes für Zeichenketten reserviert und den Rest numerischen Variablen zur Verfügung stellt. (Ausgebautere Versionen gestatten ein weiteres Argument, mit dem die Speichergröße überhaupt verändert werden kann. Wir wollen hier allerdings nicht darauf eingehen. Schlagen Sie bei Bedarf dazu in Ihrem Handbuch nach.)

Mit diesem CLEAR-Befehl sind wir aus dem Schneider. Man kann ausdrücklich berechnen, wieviel Speicherraum zur Verfü-

gung steht. In den meisten Fällen genügt es aber, wenn man einfach «auf Verdacht» hin genügend Platz reserviert. Machen wir das so:

```
>NEW
>5 CLEAR 500 : REM - Das reserviert 500 Bytes.
>10 FOR I=1 TO 25
>20 A$ = A$+"1234567890"
>30 PRINT I
>40 NEXT I
>RUN
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
READY
```

Wenn Sie dieses Ergebnis von vornherein bekommen haben, dann besitzen Sie eine Version, welche entweder sehr viel «Schiebereien» im Speicher durchführt (das merkt man an der relativ geringen Geschwindigkeit bei umfangreichen Operationen mit Zeichenketten) oder die mehr Steuerungs- und Verwaltungsaufwand treibt und im wesentlichen ohne Verschiebungen auskommt. Diese Versionen sind aber immer noch relativ selten. Sie stellen auch mehr Ansprüche an den verfügbaren Speicherplatz.

Wir können jetzt im übrigen noch einen Versuch machen. Die Variableninhalte sind ja erhalten geblieben, d.h., wir können

```
>PRINT A$
```

befehlen und erhalten dann die 250 dort festgehaltenen Ziffern ausgedruckt (deren Angabe wir uns aus Platzgründen sparen wollen). Was geschieht nun, wenn wir noch einmal zehn Zeichen anhängen?

```
>A$=A$+"1234567890"
?LS ERROR
```

Noch ein Fehler! Diesmal ist in der Tat der für die einzelne Variable maximal verfügbare Platz überschritten worden: «LS» heißt «long string» – (zu) lange Zeichenkette. Allerdings ist diese Fehlermeldung nicht darauf zurückzuführen, daß kein Speicherplatz mehr zur Verfügung steht. Der Grund liegt in der Art, wie die Zeichenketten verwaltet werden: Mit jeder Zeichenkette wird eine Angabe über ihre Länge mitgespeichert, die gerade ein Byte im Speicher belegt. In einem Byte kann man nun gerade eine ganze Zahl von 0 bis 255 festhalten, mehr nicht. In unserem Versuch haben wir diesen Längenzähler schlichtweg überfordert, worauf das System mit einer Fehlermeldung reagierte.

Dieser Fehler kann unangenehm werden, wenn im Verlauf der Verarbeitung von Zeichenketten ein Zwischenergebnis einmal die «magische Grenze» von 255 Zeichen überschreitet. Das führt in den meisten Versionen unweigerlich zum Programmabbruch. Achten Sie bei der Anlage der Daten für ein Programm auch darauf.

In «HP/Northstar»-Versionen hat man dieses Problem im übrigen nicht. Hier können Zeichenketten so lang gemacht werden, wie es der Speicher nur zuläßt. Allerdings muß man den Platz dafür fest reservieren, auch wenn nur einmal im Rechnungsverlauf ein einziges Zwischenergebnis ein einziges Mal astronomische Länge erreicht. Und das hat wieder seine Nachteile für sich.

Bleiben wir bei unserer vergleichsweise bescheidenen Version, in welcher der Speicherbereich für Zeichenketten passend reserviert werden muß. Wir müssen bei der Einteilung allerdings darauf achten, daß unser Problem noch mehr Zeichenketten als nur das N$-Feld umfaßt. Das ist z.B. die skalare Zeichenkette D$ für das Datum, mit Eingabereserve sollten wir hier 25 Zeichen ansetzen, dann das Feld D$ mit den drei Elementen für Tag, Monat und Jahr. Hier sollten wir auch Fehleingaben berücksichtigen und für sie mindestens ebenso viel wie für die Datumseingabe, d.h. je 25 Zeichen berücksichtigen. Weiter benötigen wir eine allgemeine Zeichenkette für Benutzereingaben, die maximal eine ganze Zeile

aufnehmen kann, in der Regel also 80 Zeichen. Das sind 180 Zeichen plus Verwaltungsinformation, die mindestens benötigt werden. Geben wir noch ein gutes Stück zu und reservieren 300 Bytes für diesen Zweck, dann sieht die Initialisierung so aus:

```
F1 = (INT(FRE(0)-1000)/38)
CLEAR (F1*18+300)
DIM D(F1), D$(3), M(F1), M1(F1), N$(F1), W1(F1)
```

Zur Verwirklichung des Moduls «Programm einleiten» sind nur noch die geeigneten Zeilennummern festzulegen, und es sind Maßnahmen zum Laden der Lagerdatei in den Speicher zu treffen. Dieses Laden aber ist ein Problem für sich, dem wir uns erst ein Stück weiter hinten im Zusammenhang mit dem Abspeichern der Daten zuwenden werden. Nehmen wir für das folgende einfach an, die Daten würden bereits richtig im Speicher stehen.

Artikel neu aufnehmen: Zum Vorgehen bei der Programmerstellung

Bevor wir mit der Lagerdatei überhaupt arbeiten können, muß erst einmal etwas in ihr gespeichert sein. Das bedeutet, daß die allererste Arbeit, die wir zu tun haben, die Neuaufnahme von Artikeln in die Datei sein muß. Da die gesamte spätere Arbeit davon abhängt, daß diese Einträge richtig aufgenommen worden sind, müssen wir uns noch einmal genau überlegen, wie man diese Eingabe so sicher wie möglich gestalten kann.

Doch zunächst einmal: Was ist zu tun? Das Programm muß von der Tastatur die Werte für die Einträge

D(F) – Datum der letzten Änderung eines Eintrags
M(F) – Menge auf Lager
M1(F) – Mindestmenge auf Lager
N$(F) – Namen der gelagerten Artikel
W1(F) – Einzelwert der verschiedenen Artikel

übernehmen, auf ihre Richtigkeit und ihre Folgerichtigkeit hin

überprüfen und falls in Ordnung, in die Datei eingliedern. Dazu ist es nötig, einen freien Eintrag in der Datei aufzufinden und den zugehörigen Index F bereitzustellen. Ferner muß gemeldet werden, wenn die Datei voll ist, und eine Unterrichtung des Benutzers über den gerade vorliegenden Belegungsgrad wäre durchaus angebracht. Diese Prozesse müssen so oft wiederholt werden, bis der Benutzer ausdrücklich das Ende der Neueingabe von Artikeln signalisiert oder bis die Datei endgültig voll ist.

Wir haben also einen system- und einen benutzerorientierten Problembereich zu lösen:

- systemorientierte Aufgaben:
  - freien Eintrag auffinden und Index F bereitstellen
  - Eintrag neu in die Datei einfügen
  - Dateibelegung ermitteln und anzeigen
  - vollständige Auffüllung der Datei erkennen und melden
- benutzerorientierte Aufgaben:
  - Eintragsdaten vom Benutzer anfordern
  - Eintragsdaten prüfen
  - Eintragsdaten bestätigen lassen
  - Ende der Neueingabe von Artikeln vom Benutzer erfragen

Eine derartige Aufgabensammlung sollte am Anfang eines jeden Programmierabschnitts stehen, falls er nicht relativ trivial ist, wie sie im letzten Abschnitt betrachtete Initialisierung, wo man sofort zur Strukturbeschreibung übergehen kann. Sie gibt einen ersten Hinweis darauf, wie das betreffende Problem lösbar ist, ob sich unter Umständen eine weitere Aufteilung in Unteraufgaben empfiehlt oder ob die Aufgabenstellung sich mit anderen Bereichen überschneidet, daß man sie besser zusammenlegt.

Das genaue Vorgehen bei dieser Aufgabensammlung hängt natürlich vom Geschmack und der Erfahrung des jeweiligen Programmierers ab und auch davon, wie kompliziert das betrachtete Problem ist. Doch hat es sich allgemein als hilfreich erwiesen,

zunächst einmal die Aufgabe schriftlich zu *formulieren*. Das zwingt bereits zu einem ersten Durchdenken und legt logische Fehler rascher bloß, als es eine reine Kopfarbeit oder gar ein zufälliges Vorgehen tun. Außerdem erhält man so ganz natürlich wichtiges Dokumentationsmaterial.

Dieser Aufgabenformulierung sollte sich eine gliedernde Zusammenfassung der voraussichtlich zu lösenden Unteraufgaben anschließen. Das verbessert die Übersicht und läßt sich oft unmittelbar in eine Strukturbeschreibung übertragen. Man kann aus dieser Übersicht oft auch bereits erkennen, ob zuarbeitende Programmteile irgendwelche wichtigen Voraussetzungen nicht erfüllen oder umgekehrt im Rückgriff von anderen Programmteilen her untersuchen, ob alle Voraussetzungen für ihre Arbeit hier geschaffen werden. Insofern ist diese erste Grobgliederung ein wichtiges Werkzeug beim Programmieren. Und – selbstverständlich – wird auch dadurch die Dokumentation bereichert, ohne daß irgendwelche Zusatzarbeit notwendig ist.

In unserem Fall spiegelt die Grundgliederung in system- und benutzerorientierte Aufgaben zwar wichtige Arbeiten in diesem Programmodul wieder, doch können wir das nicht unmittelbar in eine Strukturbeschreibung übertragen. Wir müssen uns also zusätzlich Gedanken über den Ablauf, das Ineinandergreifen dieser Programmfunktionen machen.

Als erstes muß das Programm die Zahl der belegten Einträge ermitteln, herausfinden, ob die Datei bereits voll ist, und das Ergebnis dieser Untersuchungen an den Benutzer melden. Ist noch Platz, dann muß ein freier Eintrag gefunden und die Indexvariable F geeignet gesetzt werden, was in der Regel im selben Arbeitsgang erfolgt. Ist das geschehen, dann müssen vom Benutzer die für den Eintrag benötigten Daten angefordert werden, was man am sinnvollsten in der Reihenfolge

- Artikelbezeichnung
- gegenwärtige Lagermenge
- Mindestlagermenge und
- Einzelpreis

vornimmt. Diese Eingaben müssen schrittweise überprüft werden. Ist ein offensichtlicher Fehler aufgetreten, so sollte das umgehend gemeldet und vom Benutzer eine Neueingabe angefordert werden. Ist alles formal richtig übernommen, so empfiehlt es sich, das Ganze geordnet zurückzumelden und vom Benutzer eine ausdrückliche Bestätigung zu verlangen. Wird diese nicht gegeben, dann wiederholt sich der Eingabeprozeß geeignet, bis diese Bestätigung eingegangen oder der Befehl zum Abbruch der Operation erteilt worden ist. Erst dann, wenn alles in Ordnung ist, darf der Eintrag in der Datei selbst verändert (und gezählt) werden. Ihm muß die Anfrage folgen, ob weitere Artikel aufzunehmen sind. Falls ja, wird der gesamte Prozeß wiederholt, andernfalls kann der Programmteil beendet werden.

Und das Datum? Es wäre höchst umständlich und fehlerträchtig, müßte das jedesmal von neuem eingegeben werden. Außerdem benötigen wir die Datumsangabe auch in anderen Programmteilen, so daß es sich empfiehlt, den Modul zur Programmeinleitung zu ergänzen:

Programm einleiten:
  Variable dimensionieren und initialisieren,
  Datum übernehmen und in D0 ablegen,
  Lagerbestand laden.
Ende.

Das erweitert außerdem unsere Variablenliste um:

D0 – Gültiges Tagesdatum

Wir können noch etwas vermerken: Die Änderung der Eingabe ist eine Operation, die nicht nur bei der Neueingabe von Artikeln, sondern auch bei Änderungen der Einträge benötigt wird. Das heißt, wir müssen sie als eigenen Programmodul auslagern. Wenn wir das geschickt machen, dann können wir dort auch die Neueingabe von Eintragsdaten erledigen, da das im Grund nichts anderes als die Änderung eines leeren Eintrags ist.

Formulieren wir das alles als Strukturbeschreibung:

```
Artikel neu aufnehmen:
   Bis Operation beendet:
      Zahl der belegten Einträge ermitteln,
      Wenn Datei voll, dann:
         «Datei ist voll»,
         Operation beendet.
      Sonst:
       Zahl der Einträge melden,
         Freien Eintrag aufsuchen,
         Eintragsdaten übernehmen,
         Wenn kein Abbruch, dann:
            Eintrag in die Datei schreiben,
            Eintrag zählen.
         Anfrage: «Noch ein Artikel?»,
         Wenn kein weiterer Artikel, dann:
            Operation beendet.
   Schleifenende (Operation beendet).
Ende.
```

Von der Übernahme der Eintragsdaten haben wir bereits festgestellt, daß sie in ähnlicher Form auch von einem anderen Programmteil benötigt wird und es daher sinnvoll ist, sie als eigenen Modul zu formulieren. Sehen wir uns auch die anderen Operationen daraufhin an, ob sie vielleicht auch woanders, so oder ähnlich, gebraucht werden.

Da muß zunächst die Zahl der belegten Einträge ermittelt werden. Das ist eine Aufgabe, die wir ziemlich sicher auch bei der Ausgabe der Lagerübersicht benötigen. Das Aufsuchen eines freien Eintrags in der Datei scheint zunächst eine Operation zu sein, die nur hier gebraucht wird. Doch betrachten wir das näher: Was heißt das, einen freien Eintrag aufzusuchen? Nun – ein freier Eintrag ist offensichtlich doch ein nicht belegter Eintrag. Das aber verkettet diese Operation mit der Ermittlung der Zahl der belegten Einträge, so daß beides auf derselben Grundarbeit

beruht: Jeder Eintrag der Datei muß daraufhin untersucht werden, ob er frei oder belegt ist. Also wäre auch das als eigener Modul zu fassen. Schließlich haben wir da die Operation «Eintrag in die Datei schreiben» und «Eintrag zählen», die im Prinzip auch von dem Modul zur Eintragsänderung benötigt werden. Das sind relativ kurze Operationen, bei denen im wesentlichen nur Variablenwerte übertragen werden. Es empfiehlt sich, diese Operationen getrennt in den jeweiligen Moduln zu behandeln. Das verbessert die Übersicht etwas, da die Operationen kurz sind und man keinem zusätzlichen GOSUB-Aufruf folgen muß.

Wir sollten allerdings auch nicht zu voreilig sein. Wie ist das nun eigentlich mit der Zahl der belegten Einträge? Ist es sinnvoll, jedesmal die ganze Datei zu durchforsten, nur um herauszufinden, *wie viele* – und nicht *welche* – Einträge belegt sind? Dazu genügt es doch, eine eigene Variable zu definieren, die Bestandteil der Datei ist (also auch extern mitgespeichert werden muß) und die eben diese Eintragszahl angibt. Wir hatten ja ohnehin die Operation «Eintrag zählen» vorgesehen. Dann schrumpft die ganze Operation «Zahl der belegten Einträge ermitteln» auf eine einfache Variablenabfrage zusammen – und das kann man nun wirklich mit in jedem Unterprogramm unterbringen. Wir müssen nur eine passende Variable für die Zahl der belegten Einträge festlegen – nennen wir sie doch einfach «B»:

B – Zahl der belegten Einträge in der Datei.

Und wie sieht es dann mit der Ermittlung eines freien Eintrags aus? Der eben vorgeschobene Grund, das separat zu formulieren, ist entfallen. Allerdings ist anzunehmen, daß es sich dabei um eine relativ komplizierte Operation handelt. So etwas behandelt man der Übersicht halber immer besser für sich. In welchem Maße es sich dabei um eine völlig selbständige Arbeit handelt, sei noch dahingestellt. Dazu müssen wir festlegen, woran man eigentlich einen freien Eintrag erkennt. Wir können hier aber bereits eine Vermutung aussprechen:

Um einen freien Eintrag in der Datei aufzufinden, muß diese

nach einem passenden Merkmal hin durchsucht werden. Eine ähnliche Operation brauchen wir aber auch an einigen anderen Stellen, dann nämlich, wenn zum Ändern oder Ausdrucken ein bestimmter Eintrag gebraucht wird. Das heißt, wir müssen ohnehin einen Modul «Datei durchsuchen» aufstellen. Es fragt sich, ob man das Suchen nach einem freien Eintrag nicht als Sonderfall dort mit unterbringen sollte.

Wir sehen: Die optimale Strukturierung eines Programms ist durchaus keine einfache Tätigkeit. Es gibt viele Dinge dabei zu bedenken, und viele Faktoren gehen letztlich in die gefundene Programmstruktur ein. Dazu gehört, wie gerade festgestellt, auch die Art und Weise, in der die Daten dargestellt werden. Eine geschickte Wahl der Datenstruktur kann die Programmstruktur und/oder die Abarbeitungsgeschwindigkeit des Programms ziemlich verbessern. Allerdings muß man hier auch wieder abwägen, ob man nun ein einfach aufgebautes oder ein sehr schnelles Programm haben will. Oft zahlt man für die Geschwindigkeit mit komplexeren Daten- und Programmstrukturen. Wir werden uns hier an möglichst einfachen Strukturen orientieren, und es sei jetzt schon gesagt, daß unsere Lösung bei weitem nicht optimal sein wird. Allerdings ist das Auffinden eine dem jeweiligen Problem optimal angepaßten Datenstruktur bereits eine Wissenschaft für sich und verlangt Kenntnisse, die den Rahmen dieser einfachen Einführung bei weitem überschreiten würden. Wichtig ist eigentlich nur, daß man sich nicht von den ersten besten (meist schlechtesten) Einfällen leiten läßt, sondern versucht, möglichst viele Alternativen gegeneinander abzuwägen, seien sie selbst erfunden oder von anderen übernommen. Und es ist wichtig, sich nicht zu früh für die eine oder die andere Struktur festzulegen. Man sollte das erst tun, wenn alle wesentlichen Programmstrukturen (im Prinzip) von oben nach unten untersucht worden sind.

Eines können wir allerdings bereits auf dieser Stufe festhalten, und das betrifft die benötigten Variablen. Neben der Zahl der belegten Einträge brauchen wir noch eine Reihe weiterer Varia-

blen. Wir haben nämlich ausdrücklich festgelegt, daß ein Eintrag erst dann in der Datei geändert werden darf, wenn er vollständig richtig übernommen ist. Diese Eingaben aber müssen so lange bis sie stimmen, auch irgendwo festgehalten werden, wozu weitere Variable benötigt werden. Verwenden wir dazu doch einfach die skalaren Gegenstücke zu den Feldvariablen für die Datei:

M – Menge auf Lager
M1 – Mindestmenge auf Lager
N$ – Namen der gelagerten Artikel
W1 – Einzelwert der verschiedenen Artikel

Artikeleintrag ändern

Kaum ein Eintrag in unserer Lagerdatei wird auf ewige Zeiten so dort stehen bleiben, wie er beim erstenmal aufgenommen wurde. Da ändert sich z.B. der Einzelpreis des Artikels. Oder es muß die am Lager zu haltende Mindestmenge verändert werden. Oder der Artikel läuft aus und ist folglich aus der Lagerdatei zu streichen.

Alles das erfordert die Möglichkeit, jeden beliebigen Eintrag in die Lagerdatei ändern zu können. Das heißt, der betreffende Artikel muß (anhand seines Namens) in der Datei aufgesucht, die betreffenden Eintragsdaten müssen geändert und der korrigierte Eintrag schließlich in die Datei zurückgeschrieben werden. Dazu kommt die Mitteilung darüber, daß ein bestimmter Eintrag – etwa aufgrund eines Tippfehlers – nicht aufgefunden werden konnte. Und es muß möglich sein, einen Eintrag nicht nur zu ändern, sondern ganz aus der Datei zu löschen, was wiederum erfordert, daß die Angabe über die Zahl der belegten Einträge verändert werden muß. Weiter darf ein Artikel nur dann gelöscht werden, wenn dies ausdrücklich verlangt worden ist. Wie bei der Neuaufnahme schließlich sollte dieser Programmteil nur verlassen werden, wenn der ausdrückliche Befehl dazu gegeben worden ist.

Wir haben also folgende Probleme zu lösen:

- systemorientierte Aufgaben:
  - Auffinden eines Eintrags in der Datei aufgrund seines Namens
  - Meldung, falls der Eintrag unauffindbar ist
  - Löschen des gegebenen Eintrags aus der Datei
- benutzerorientierte Aufgaben:
  - Zu ändernde Eintragsdaten vom Benutzer anfordern
  - Eintragsdaten prüfen
  - Eintragsdaten bestätigen lassen
  - Löschbestätigung ausdrücklich einholen
  - Ende der Änderung von Artikeln vom Benutzer erfragen

Diese Arbeiten laufen in einer ähnlichen Schleife ab wie bei der Neuaufnahme von Artikeln. Hauptunterschied ist die Art und Weise, wie der Eintrag bereitgestellt wird. Bei einer Neuaufnahme genügt es, einen freien Eintrag aufzusuchen und die vom Benutzer übernommenen Daten dort hineinzukopieren. Hier dagegen muß ein bereits vorhandener Eintrag aufgesucht und die dort stehende Information in die Hilfsvariablen zur Datenübernahme kopiert werden. Falls ein Löschen des Eintrags gefordert wird, verläuft die übrige Arbeit ganz entsprechend: Die vorgefundenen Daten werden dem Benutzer angezeigt, der sie den Bedürfnissen gemäß ändert und diese Änderung bestätigt, worauf die neue Information in die Datei an dieselbe Stelle zurückgeschrieben werden. Die Zahl der Einträge ändert sich dadurch nicht.

Anders ist das beim Löschen des Eintrags. Hier muß die Information so abgeändert werden, daß sie einer nicht belegten Aufzeichnung entspricht. Wir müssen uns also jetzt darüber einig werden, was man unter einem freien Eintrag verstehen soll. Nun – gehen wir den Weg des geringsten Widerstands: Wenn wir die Datei ganz neu definieren, dann sind offensichtlich alle Zeichenketten N$(F) leer und alle numerischen Variablen in den Dateifeldern auf Null gesetzt. Die Information Null in den numerischen Feldern ist in unserer Anwendung durchaus sinnvoll,

sowohl die gelagerte Menge, als auch die Mindestmenge, als auch der Einzelpreis können den Wert Null annehmen. Dagegen muß jeder Artikel einen eindeutigen Namen besitzen. Hier setzen wir an:

Ein Eintrag in der Datei gilt genau dann als nicht belegt (gelöscht), wenn er ein leeres Namensfeld besitzt.

(Wir haben dabei angenommen, daß unsere BASIC-Version die Variablen in dieser Art und Weise automatisch initialisiert. Falls das in Ihrer Version nicht zutrifft, so müssen Sie das ausdrücklich beim Einstieg in das Programm, noch vor dem Laden der Dateiinformation vom externen Speicher vornehmen – beispielsweise mit Hilfe einer FOR-NEXT-Schleife. Überprüfen Sie Ihre Version daraufhin und nehmen Sie bei Bedarf die entsprechende Ergänzung im Modul «Programm einleiten» vor.)

Damit aber müssen wir bei der Übernahme der geänderten Eintragsinformation auch prüfen, ob der Name versehentlich gelöscht wurde. Falls ja, dann sollten wir umgehend eine Löschbestätigung anfordern. Wird diese verweigert, dann muß der Name auf seinen alten Wert zurückgesetzt werden.

Alles das läßt sich in folgender Strukturbeschreibung zusammenfassen:

```
Artikel ändern:
    Bis Operation beendet:
        Artikelbezeichnung anfordern,
        Eintrag aufsuchen,
        Wenn Eintrag nicht gefunden, dann:
            «Eintrag nicht gefunden».
        Sonst:
            Eintragsdaten in Hilfsvariable kopieren,
            Neue Eintragsdaten übernehmen,
            Wenn kein Abbruch, dann:
                Eintrag in die Datei schreiben.
            Anfrage: «Noch ein Artikel?»,
```

Wenn kein weiterer Artikel, dann:
Operation beendet.
Schleifenende (Operation beendet).
Ende.

Davon brauchen wir das Anfordern der Artikelbezeichnung noch öfter. Es kommt auf die Komplexität an, mit der diese Aufgabe erledigt werden soll. Wenn ein einfacher Eingabebefehl der Form

```
INPUT "Artikel";A$
```

genügt, dann ist es sinnvoll, dies in den jeweiligen Moduln selbst zu erledigen. Wollen wir aber eine umfassende Korrektur und Prüfung der Eingabe vornehmen, dann empfiehlt sich die Abfassung als eigenes Unterprogramm.

Um das entscheiden zu können, müssen wir uns über die Darstellung der Namen genauer klar werden. Es genügt nicht, nur die maximale Länge festzulegen. Zumindest sollten Eingabefehler, die auf verschiedener Groß- und Kleinschreibung beruhen, ausgeschieden werden. Mit anderen Worten

MESSER

Messer

messer

sollte alles auf denselben Eintrag in der Datei zurückgreifen. Am besten wandeln wir dazu die Artikelbezeichnung bei jeder Eingabe in Großbuchstaben um. Diese Umwandlung wiederum ist allerdings etwas, was wir als ein eigenständigen Programmmodul anlegen sollten. Dann erhält die Übernahme der Artikelbezeichnung in den verschiedenen Moduln die Form

```
INPUT "Artikel";A$
GOSUB xxxx : REM - A$ in Grossschreibung umwandeln
```

Eintrag in der Datei aufsuchen: Etwas über Parameter

Damit haben wir bereits genügend Information zusammen, um den Modul zum Aufsuchen eines Eintrags in der Datei zu skizzieren. Die anderen Moduln können hier keine wesentlichen Anforderungsänderungen mehr bringen. Andererseits sparen wir uns viele Diskussionen, wenn wir hier bereits etwas Greifbares zur Hand haben – obwohl es sich um einen Modul handelt, der eigentlich bei ganz strengem «Top-Down»-Entwurf viel später an der Reihe wäre.

Die Aufgabe ist klar: Wir müssen die Datei von Anfang bis Ende durchsuchen, bis der gesuchte Eintrag gefunden oder das Dateiende erreicht ist. Den zu suchenden Eintragsnamen erhalten wir vom aufrufenden Programmteil übermittelt, und das Ergebnis muß als Indexvariable F zurückgeliefert werden, wenn der Eintrag gefunden worden ist. Wenn nicht, dann setzen wir diesen Index einfach auf einen Wert, der sonst nicht vorkommen kann – mit Sicherheit ist das eine negative Zahl, wählen wir also -1 für diesen Fall.

Das können wir sofort als Strukturbeschreibung formulieren:

```
Eintrag aufsuchen:
    Indexvariable F auf Dateianfang –1 setzen,
    Bis Eintrag gefunden oder Dateiende:
        Nächsten Eintrag aufsuchen:
            Indexvariable F weitersetzen,
            Wenn F > F1, dann:
                Dateiende.
        Wenn nicht Dateiende, dann:
            Eintragsname N$(F) mit Vorgabe A$
                vergleichen,
            Wenn Namen gleich, dann:
                Eintrag gefunden.
  Schleifenende (Eintrag gefunden oder Dateiende).
```

```
Wenn Dateiende, dann:
    F = -1. /* Dateiendeflagge setzen */
Ende.
```

Das Wichtige daran ist die Verbindung mit den aufrufenden Moduln: Diese *übergeben* dem Modul in der Zeichenkette A$ den zu suchenden Namen (so hatten wir das im vorigen Abschnitt festgelegt). Das Ergebnis der Suche wiederum wird als Wert der Indexvariablen F *zurückgeliefert.* Diese beiden Variablen A$ und F stellen gewissermaßen den Kontakt zwischen dem Modul zur Eintragssuche und dem aufrufenden Modul her. Sie spielen damit über ihre normale Funktion hinaus im Aufruf *dieses* Moduls eine besondere Rolle als sogenannte *Parameter,* worunter wir hier ganz allgemein Werte verstehen wollen, die ein bestimmter selbständiger Programmteil von außen vorgegeben erhält, bzw. zur Verwendung durch andere Programmeinheiten nach außen abgibt.

In BASIC unterscheiden sich solche Parameter formal nicht von allen übrigen Variablen. Sie werden einmal passend definiert und bleiben dann bis zum Programmende erhalten. Das bedeutet, daß ein Eingangsparameter, der Wert also, der diesem zur Bearbeitung übergeben wird, außerhalb dieses Moduls eine ganz andere Rolle spielen kann als innerhalb. Solange der Fall so einfach ist wie in unserer obenstehenden Betrachtung, dann macht das in der Regel nichts aus. Anders aber ist es, wenn ein Modul mit dem Wert weiterarbeiten muß, den die Parametervariable beim Aufruf eines anderen Moduls erhalten hat. Verändert dieser andere Modul absichtlich oder unabsichtlich diesen Wert, dann ist guter Rat teuer. Man muß hier in BASIC zusätzliche Maßnahmen treffen, um einen solchen Effekt auszuschließen, wozu zwei Wege denkbar sind:

- Der aufrufende Programmteil speichert den kritischen Wert vor der Parameterübergabe in einer anderen Variablen und ruft ihn nach Rückkehr des Programms von dort wieder ab.
- Der aufgerufene Programmteil verwendet nur solche Varia-

blen, die in keinem anderen Programmteil benötigt werden. In diesem Fall muß der aufrufende Programmteil die Parameter erst in diese Variablen übertragen und das Ergebnis bei Bedarf von dort zurückholen.

Welche der beiden Methoden man einsetzt, hängt stark von den Gegebenheiten des Programms ab. Wird der Wert z.B. nur in einem aufrufenden Programm benötigt, dann ist die erste Möglichkeit sinnvoll. Kommen solche Situationen aber häufiger vor, dann müssen für das Unterprogramm eigene Variable definiert werden, was sehr viel Speicherplatz auffressen kann. Man muß hier in der Regel Kompromisse schließen und beide Methoden miteinander verbinden.

Das Problem liegt ganz allgemein daran, daß in BASIC unter einer Variablenbezeichnung auch immer nur ein einziger Wert stehen kann. Wird diese Variable im Rahmen der Abarbeitung auch von einem anderen Programm benötigt, dann kann es zum Konflikt kommen, in dem die Variable vorübergehend zwei oder noch mehr Werte gleichzeitig festhalten müßte. Da das unmöglich ist, muß man eben die im Augenblick nicht notwendigen Werte woanders unterbringen und nach Abarbeiten des betreffenden Programmteils wieder in ihren alten Wert zurückversetzen. Viele der anderen höheren Programmiersprachen – wie beispielsweise Pascal – machen das bei Bedarf automatisch. In BASIC dagegen ist die Belegung der Variablen sehr sorgfältig auszuarbeiten und zur ständigen Bezugnahme bereitzuhalten.

Besonders kritisch wird die Situation in den Fällen, in denen eine Variable, die ein Programmteil benötigt, in einem eingeschachtelten Programmteil *unabsichtlich* verändert wird. Man spricht hier von sogenannten *Seiteneffekten*. So etwas kommt häufig bei FOR-NEXT-Schleifen vor, wo man sich traditionsgemäß der Laufvariablen I, J und K bedient. Die Situation ist in größeren Programmen nicht selten, daß ein Programm aus einer FOR-NEXT-Schleife heraus ein ganz harmloses Unterprogramm aufruft, das wieder ein anderes Unterprogramm benötigt, in dem

derselbe Laufparameter in einer FOR-NEXT-Schleife verwendet wird. Und beim Programmentwurf hat alles doch so problemlos ausgesehen! Diese Situation tritt besonders oft bei nachträglichen Programmänderungen ein. Man kann sie aber bei einer sorgfältigen Programmdokumentation relativ einfach in den Griff bekommen. Ansonsten können die verrücktesten Effekte auftreten, die scheinbar auf einen Fehler an einer ganz anderen Stelle im Programm hinweisen. Es kostet manchmal viel Mühe, so etwas ohne ausführliche Programmdokumentation, nur anhand eines «Listings», herauszufinden.

Bestand verändern: Etwas zur Dialoggestaltung

Dieser Programmodul unterscheidet sich von dem Modul zur Änderung eines Artikeleintrags nur dadurch, daß lediglich ein Bestandteil – die Lagermenge – geändert werden darf. Das ist eine Vorsichtsmaßnahme, durch die versehentliches Ändern anderer Eintragsteile verhindert wird. Außerdem wird der Dialog für diese häufigste Arbeit des Systems wesentlich verkürzt und ist damit nicht so nervenaufreibend.

Man könnte das Verändern des Bestands ganz einfach dadurch vornehmen, daß der betreffende Benutzer den jeweils neuesten Stand angibt, was den alten Eintrag in der Datei überschreibt. Allerdings ist das etwas praxisfremd. Was geschieht, ist ja, daß von einem Artikel Exemplare aus dem Lager entnommen oder solche hinzugefügt werden. Mehr nicht. Also sollten wir auch nicht mehr für die Bedienung verlangen. Einzige Angaben müssen sein:

- der Name des zu verändernden Artikels,
- die Tatsache, ob etwas hinzugefügt oder entnommen worden ist und
- die Menge, die entnommen oder hinzugefügt wurde.

Das alles sollte in einem Dialog ablaufen, der sich etwa wie folgt gestaltet:

```
Welcher Artikel wurde veraendert? Hammer

Lagerbestand, Hammer: 17
Wurde etwas entnommen? Ja
Wie viele Exemplare sind entnommen worden
(Keine Aenderung: 0)? 3
3 Exemplare entnommen.
Neuer Lagerbestand, Hammer: 14
Ist das in Ordnung? Nein

Lagerbestand, Hammer: 17
Wurde etwas entnommen? Ja
Wie viele Exemplare sind entnommen worden
(Keine Aenderung: 0)? 2
2 Exemplare entnommen.
Neuer Lagerbestand, Hammer: 15
Ist das in Ordnung? Ja

Wurde noch ein Artikel veraendert? Jq
Bitte antworten Sie mit "Ja" oder "Nein".
Wurde noch ein Artikel veraendert? Ja

Welcher Artikel wurde veraendert? Zange

Lagerbestand, Zange: 3
Wurde etwas entnommen (Ja oder Nein)? Nein
Wie viele Exemplare sind hinzugefuegt worden
(Keine Aenderung: 0)? zehn
Geben Sie bitte die Aenderung als Zahl an.
Wie viele Exemplare sind hinzugefuegt worden
(Keine Aenderung: 0)? 10
10 Exemplare hinzugefuegt.
Neuer Lagerbestand, Zange: 13
Ist das in Ordnung? Ja

Wurde noch ein Artikel veraendert? Ja

Welcher Artikel wurde veraendert? Meisel
Meisel nicht gefunden. Bitte Namen neu eingeben.
Welcher Artikel wurde geaendert? Meissel

Lagerbestand, Meissel: 13
Wurde etwas entnommen (Ja oder Nein)? Nein
Wie viele Exemplare sind hinzugefuegt worden
(Keine Aenderung: 0)? 0
Nichts veraendert.

Wurde noch ein Artikel veraendert?
```

Wir haben also so viele Sicherungen gegen Fehlbedienung wie gerade sinnvoll eingefügt. Und noch eine Änderung ist zu vermerken: Der Dialog gestaltet sich besser, wenn man hier auf das Menü

```
1 - Exemplare hinzufuegen
2 - Exemplare entnehmen

Welche Operation soll durchgefuehrt werden?
```

verzichtet und statt dessen eine direkte Abfrage vorsieht, wie sie oben gezeigt ist. Das wiederum läßt sich im selben Programmodul erledigen, womit in der Programmhierarchie von Bild 11 eine Ebene («Exemplare entnehmen», «Exemplare einfügen») entfallen kann.

Wichtig ist in einem solchen Dialog zweierlei: Zum einen soll der Benutzer möglichst «natürlich» durch die Systemanforderungen geleitet werden. Und zum anderen müssen die Fehlermeldungen so gehalten sein, daß die Korrektur sofort ersichtlich ist. So genügt es nicht (wie man es oft findet), bloß «Eingabefehler» zu melden – man muß schon noch mit angeben, *was* falsch gemacht wurde, oder besser noch, wie es *richtig* zu machen ist. Genau das haben wir oben versucht. Ein Dialog ist erst dann in Ordnung, wenn der Benutzer sich auf seine eigene Arbeit konzentrieren kann und nicht ständig überlegen muß, was der Computer eigentlich gerade will. Optimal ist es, wenn jemand das Programm zum erstenmal einschaltet und ohne irgendwo nachschlagen zu müssen, völlig problemlos zu Rande kommt. Allerdings wird man nicht immer den gewaltigen Aufwand treiben können, der im Idealfall dafür nötig ist. Es gilt dann Kompromisse zu finden, die wenigstens die häufigsten Schwierigkeiten umschiffen helfen.

Die Strukturbeschreibung dieses Programmoduls läßt sich nach dem Modell des vorigen Abschnitts etwa folgendermaßen formulieren:

```
Bestand verändern:
    Bis Operation beendet:
        Bis Artikel gefunden:
            Artikelbezeichnung anfordern,
            Eintrag aufsuchen,
            Wenn Eintrag nicht gefunden, dann:
                Korrektur anfordern.
        Schleifenende (Artikel gefunden),
        Bis Eintrag in Ordnung:
            Lagermenge in Hilfsvariable kopieren,
            Lagermenge melden,
            Anfrage, ob Exemplare zu entnehmen sind,
            Wenn «Ja», dann:
                Anzahl übernehmen,
                Lagermenge - entnommene Anzahl.
            Sonst:
                Anzahl übernehmen,
                Lagermenge + hinzugefügte Anzahl.
            Wenn Änderung < > 0, dann:
                Neue Lagermenge melden,
                Richtigkeit bestätigen lassen,
                Wenn Richtigkeit bestätigt, dann:
                    Lagermenge in die Datei schreiben,
                    Eintrag in Ordnung
                Sonst:
                    Eintrag nicht in Ordnung.
                Sonst:
                    «Nichts verändert.»,
                    Eintrag in Ordnung. /* Abbruch */
            Schleifenende (Eintrag in Ordnung).
            Anfrage: «Noch ein Artikel?»,
                Wenn kein weiterer Artikel, dann:
                    Operation beendet.
    Schleifenende (Operation beendet).
Ende.
```

Das ist bereits recht viel, und doch ist der Dialog noch nicht ganz optimal. Versuchen Sie herauszufinden, was noch zu verbessern wäre und wie man das im Programm unterbringt. Sehen Sie sich auch die *Form* des Dialogs an. Versuchen Sie Alternativen zu finden und entwickeln Sie dafür eine eigene Strukturbeschreibung. Sie werden in der Praxis finden, daß Bedienungsdialoge oft durch kleine Änderungen im Ansatz viel günstiger zu formulieren sind. Versuchen Sie bei praktischen Projekten auf alle Fälle, nach einiger Zeit Probebetrieb auch die Erfahrungen der Benutzer mit den Systemdialogen in die endgültige Fassung mit einfließen zu lassen. Die Dialoge sind nun einmal Bestandteil des Programms und als solche eigentlich nie «fertig».

Lagerübersicht und Nachbestellübersicht: AND, OR, LPRINT

Die Grundaufgabe dieser Programmteile ist dieselbe: Die Datei wird Eintrag für Eintrag gelesen und die dort vorgefundene Information – eventuell unter Berücksichtigung besonderer Bedingungen – in einer Tabelle ausgedruckt. Dies wiederum umfaßt im wesentlichen Aufgaben, die wir bereits im ersten Kapitel behandelt haben. Das Neue ist eigentlich nur, woher die Daten kommen. Am einfachsten ist das beim Gesamtausdruck des Lagerbestands. Hier wird alles ausgedruckt, so wie wir es Eintrag für Eintrag in der Datei vorfinden. Das hat zwar den Nachteil, daß der Ausdruck völlig ungeordnet ist, aber für unseren Zweck mag es genügen. In der Praxis *sortiert* man die Datei nach passenden Gesichtspunkten. Sortieren aber gehört wie die verschiedenen Suchmethoden und Datenorganisationen zu den «höheren» Programmiertechniken. Ihre Behandlung würde den Rahmen dieser Einführung sprengen.

Die beiden Alternativen, die wir vorgesehen haben, Ausdruck eines Lagerauszugs zwischen Datumsgrenzen und Ausgabe einer Nachbestellübersicht, unterscheiden sich nur darin, wie die auszudruckenden Einträge ausgewählt werden. Am einfachsten sieht

es mit der Nachbestellübersicht aus, wo wir zwei Felder im jeweiligen Eintrag untersuchen: die vorliegende Lagermenge und die Mindestmenge, die gelagert werden soll. Wird die tatsächlich vorhandene Menge kleiner als diese Untergrenze, dann wird der Eintrag ausgedruckt. Sonst wird er übersprungen.

Etwas komplizierter ist es mit der Lagerübersicht zwischen Datumsgrenzen. Hier müssen wir zwei Bedingungen untersuchen. Ein Eintrag soll nämlich genau dann gedruckt werden, wenn der in seinem Datumsfeld gespeicherte Wert größer oder gleich der unteren Datumsgrenze D1 *und* zugleich kleiner oder gleich der oberen Datumsgrenze D2 ist. Oder anders herum ausgedrückt: Er wird *nicht* gedruckt, wenn die Untergrenze D1 unterschritten *oder* wenn die Obergrenze D2 überschritten worden ist. Das läßt sich z.B. durch zwei IF-THEN-Anweisungen erledigen:

```
IF D(F)<D1 THEN xxxx : REM Nicht drucken
IF D(F)>D2 THEN xxxx
(Eintrag ausdrucken)
```

«xxxx» ist hier der Programmteil, in dem der nächste Eintrag übernommen wird. In diesem einfachen Fall wird noch ausreichend deutlich, was geschehen soll. Schwieriger wird es, wenn noch mehr Bedingungen berücksichtigt werden sollen. Man kommt so rasch zu ziemlich unübersichtlichen Entscheidungsketten, in denen jeweils ein Stückchen der Gesamtbedingung nachgeprüft und je nach Ergebnis irgendwohin gesprungen wird. Warum soll man die Entscheidung nicht am Stück erledigen können? In unserem Fall wäre z.B. zu prüfen

wenn *D(F)< D1 ODER D(F)> D2,* dann nicht drucken

Dieses ODER faßt die beiden Argumente der Einzelanweisungen von oben zu einem sogenannten *logischen Ausdruck* zusammen, der immer dann zutrifft, wenn mindestens auch eines seiner beiden Teilargumente zutrifft. Man sagt, er sei *wahr,* wenn mindestens eines seiner Teilargumente wahr ist, ansonsten sei er

falsch. Bezeichnet man die beiden Teilargumente mit A und B, so kann man das tabellarisch so wiedergeben:

A	B	A ODER B
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Man bezeichnet diese ODER als *Logische Operation* (oder auch als logische Verknüpfung). Die meisten BASIC-Versionen bieten eine solche Form des ODER zum Einsatz in logischen Entscheidungen an. Es hat die Form eines *Operators* ähnlich den arithmetischen Operatoren +, −, * usw. und steht *zwischen* seinen Argumenten A und B:

A OR B

Wir hätten die Entscheidungsbedingung für den Ausdruck eines Eintrags natürlich auch anders herum formulieren können

wenn D(F)>=D1 UND D(F)<=D2, dann (drucke Eintrag)

was die Absicht sogar deutlicher zum Ausdruck bringt. Der logische Ausdruck

D(F)>=D1 UND D(F)<=D2

ist nun genau dann wahr, wenn *beide* Operanden wahr sind. In jedem anderen Fall ist er offensichtlich falsch. Tabellarisch sieht das so aus:

A	B	A UND B
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Diesem UND entspricht in BASIC der Operator AND, mit

dessen Hilfe sich die Druckentscheidung auch so formulieren ließe:

```
IF D(F)>=D1 AND D(F)<=D2 THEN GOSUB xxxx
```

wobei xxxx in diesem Falle das Unterprogramm zum Ausdrucken eines Eintrags wäre.

Diese logischen Operationen gehören mittlerweile zum Standard der meisten BASIC-Versionen. Falls sie fehlen, so kann man sie u.U. auch durch passende Rechenoperationen nachbilden. Man nutzt dabei folgende Zusammenhänge aus:

1 + 1 = 2	1 * 1 = 1
1 + 0 = 1	1 * 0 = 0
0 + 1 = 1	0 * 1 = 0
0 + 0 = 0	0 * 0 = 0

Ersetzen wir hier die Null durch «falsch» und alle Werte, die ungleich Null sind, durch «wahr», dann erhalten wir:

wahr	+ wahr	= wahr	wahr	* wahr	= wahr
wahr	+ falsch	= wahr	wahr	* falsch	= falsch
falsch	+ wahr	= wahr	falsch	* wahr	= falsch
falsch	+ falsch	= falsch	falsch	* falsch	= falsch

Vergleichen wir das Ergebnis mit den Tabellen, dann stellt sich die « + »-Operation als logisches ODER und die Verknüpfung mit « * » als logisches UND heraus. Unter der Voraussetzung, daß die Vergleichsoperationen derartige Werte liefern, also Null, wenn der Vergleich nicht zutrifft,und einen Wert ungleich Null, wenn er zutrifft – unter *dieser* Bedingung kann man die ODER-Verknüpfung durch die Addition und die UND-Verknüpfung durch die Multiplikation ersetzen. Falls Ihre BASIC-Version also kein OR bzw. AND kennt, so experimentieren Sie ein bißchen, ob Sie es eventuell durch Addition und Multiplikation ersetzen können.

Und andersherum: Wenn sie in einem fremden Programm auf derartige «verrückte» Additionen und Multiplikationen stoßen, dann denken Sie daran, daß sich dahinter ziemlich sicher die logischen Operationen OR bzw. AND verbergen.

Wir haben noch ein Problem zu lösen: Wohin mit dem Ausdruck? Eigentlich hat dieser nur dann einen Sinn, wenn die Daten auf Papier festgehalten werden. Nun erlaubt bei den meisten Mikrocomputer-BASIC-Versionen der einfache PRINT-Befehl lediglich die Darstellung der Information auf dem *Bildschirm* des Geräts. Wir aber wollen sie auf einem *Drucker* wiedergegeben wissen. Was tun?

Zwei Lösungen sind verbreitet. Die eine schiebt die Verantwortung auf das *System,* also den Computer selbst, ab, der irgendwie die Möglichkeit bieten soll, die Ausgabe vom Bildschirm auf den Drucker umzuleiten oder auf beiden gleichzeitig zu vollziehen. Das findet man besonders bei sogenannten *Allzweckcomputern,* die kein fest eingebautes BASIC besitzen, sondern das Interpreterprogramm erst in den Arbeitsspeicher laden müssen. In diesem Fall nimmt der BASIC-Interpreter an, daß der Computer die Ein- und Ausgabe von Daten vornehmen kann, und bedient sich dann einfach dieser Möglichkeiten.

Die andere Lösung ist typisch für die ganze Anlage von BASIC: Man definiert einfach einen eigenen Befehl. Dieser unterscheidet sich – wie alle nicht standardgemäßen Erweiterungen – von BASIC-Hersteller zu BASIC-Hersteller. Am häufigsten findet man die Formulierung von Microsoft, die sinngemäß besagt:

«Drucke das nachfolgende Argument auf dem *Zeilendrucker* aus.»

Stören Sie sich nicht an den «Zeilen» beim Drucker – dieser Begriff stammt noch aus der Zeit, als man zur Bedienung des Rechners kein Bildschirmgerät, sondern einen Fernschreiber benutzte, der die Texte langsam, Zeichen für Zeichen, wiedergab und folglich mit «Zeichendrucker» benannt wurde. Tabellen und andere Texte, die unabhängig vom Computer benötigt wurden,

druckte man auf einem schnellen Drucker aus, der getrennt davon angeschlossen war und oft ganze Zeilen auf einmal ausgeben konnte. Diese Unterscheidung hat sich bis heute gehalten, wenn auch der «Zeilendrucker» in einem Mikrocomputersystem manchmal nur eine elektrische Schreibmaschine ist.

Wie dem auch sei – ein Zeilendrucker ist im Englischen ein «line printer», und davon leitet sich die Befehlsbezeichnung ab.

```
LPRINT
```

Dieser Befehl verwendet genau dieselben Argumente wie der PRINT-Befehl. Der einzige Unterschied ist, daß der auszugebende Text nicht auf dem Bildschirm (oder Bedienungsfernschreiber), sondern auf einem zusätzlich angeschlossenen Drukker ausgegeben wird. Auch das hängt natürlich vom verwendeten Computersystem ab – man wird einen solchen Befehl dann kaum finden, wenn der betrachtete Computer keinen Druckeranschluß bietet.

Es gibt noch zwei andere Methoden, die wir hier der Vollständigkeit halber kurz erwähnen wollen. Die eine findet man in einer «CBASIC» genannten BASIC-Version für den kommerziellen Einsatz, die im Mikrocomputerbereich weite Verbreitung gefunden hat. Hier gibt es einen Befehl

```
LPRINTER
```

der jedoch kein Argument entgegennimmt. Seine Aufgabe ist es lediglich, das System so *umzuschalten,* daß alle nachfolgenden PRINT-Ausgaben nicht über die Benutzerkonsole – also Bildschirmgerät oder Bedienungsfernschreiber – laufen, sondern an den Drucker des Systems weitergegeben werden. Dieser Befehl wird begleitet von einem Rückschaltbefehl namens

```
CONSOLE
```

durch den alle im Programm nachfolgenden PRINT-Befehle wieder an den Bediener ausgegeben werden.

Die andere Methode verwendet keinen besonderen Befehl für den Ausdruck an den Drucker, sondern benutzt ein *zusätzliches Argument* zum PRINT-Befehl, das dem Rechner mitteilt, an welches Gerät er das eigentliche Druckargument ausgeben soll. Der Befehl hat hier meist die Form

```
PRINT #x,y
```

wobei «#» die Abkürzung für «Nummer» ist und «x» die Nummer des Geräts angibt, zu dem der Text «y» übertragen werden soll. Diese Nummer ist entweder vom System oder vom Benutzer irgendwo festgelegt worden. Wenn beispielsweise die Konsole die Nummer 0 und der Drucker die Nummer 1 trägt, dann wird durch

```
PRINT #0,"Konsolenausgabe"
PRINT #1,"Druckerausgabe"
```

auf dem Bildschirm das Wort «Konsolenausgabe» und auf dem Drucker das Wort «Druckerausgabe» wiedergegeben. Diese Gerätenummern nennt man auch «logische Einheiten», sie können auch Dateien auf dem Massenspeichergerät bezeichnen – wir werden gleich noch einmal darauf zurückkommen.

Ein letzter Hinweis noch, bevor wir uns formalen Gesichtspunkten zuwenden. Im Gegensatz zur Bedienerkonsole ist der Drucker des Geräts nicht ständig betriebsbereit. Man muß ihn einschalten, Papier einlegen und sonstige Justierungsarbeiten vornehmen, bevor man ihn benutzen kann. Das bedeutet für uns, daß wir im Programm vor *jedem* Druckkomplex eine Meldung an den Benutzer ausgeben müssen, die z.B. so aussieht:

```
Bitte schalten Sie den Drucker an und legen Sie Papier ein.
Ist er betriebsbereit?
```

worauf der Rechner auf eine «Ja»-Antwort wartet:

```
Bitte schalten Sie den Drucker an und legen Sie Papier ein.
Ist er betriebsbereit? Nein
Ist er betriebsbereit? Ja
(Drucker läuft)
```

Wird so etwas öfter benötigt, formuliert man es am besten als Unterprogramm für sich. Der Rechner bleibt dann in diesem Unterprogramm, bis er den Drucker freigegeben bekommen hat. Der zugehörige GOSUB-Befehl wirkt im Programm wie ein Schalter, den der Benutzer bedienen muß, damit die Arbeit fortgesetzt werden kann.

Beim Ausdrucken selbst muß beachtet werden, daß die Seitengrenzen nicht überschrieben werden und jedes neue Blatt eine Kopfzeile erhält. Das wird am einfachsten erreicht, wenn wir jede Druckzeile in einer besonderen Variablen zählen und immer dann, wenn die maximale Seitenlänge erreicht ist, ein Signal an den Drucker ausgeben, daß er eine neue Seite anfangen soll. Es liegt dann am Gerät, ob bei Endlospapier einfach bis zum nächsten Blatt vorgeschoben oder bei Einzelbögen der Bediener benachrichtigt wird, ein neues Blatt einzulegen. Bezeichnen wir diese Variable mit

Z – Zahl an den Drucker ausgegebenen Zeilen

Für den Ausdruck sehen wir folgende Reihenfolge der Dateiinformationen in den Einträgen vor:

Artikelname – Anzahl – Mindestanzahl – Einzelwert – Gesamtwert

Dabei soll der Gesamtwert der verschiedenen Artikel aufaddiert und am Schluß des Ausdrucks angegeben werden.

Wir haben also folgende Aufgaben zu erfüllen:

- Menüanfrage: normaler Ausdruck, Ausdruck in Datumsgrenzen, Nachbestellübersicht
- Kopfzeile drucken
- Seitensteuerung des Druckers durchführen
- Lagerdatei Eintrag für Eintrag lesen
- Entscheidung über Ausdruck eines Eintrags treffen
- Eintrag als Tabellenzeile ausdrucken
- Gesamtwert der Einzelartikel berechnen
- Gesamtwert der ausgedruckten Liste berechnen

Dazu brauchen wir insgesamt vier Programmoduln:

1. Eine allgemeine Ausdrucksteuerung (Lagerübersicht)
2. Einen Modul zum Ausdruck zwischen Datumsgrenzen
3. Einen Modul für die Nachbestellübersicht
4. Einen Modul für den eigentlichen Druck der Einträge

Diese lassen sich z.B. so strukturieren:

```
Lagerübersicht:
   F, W0 und Z auf 0 initialisieren,
   Menü anbieten,
   Auswahl übernehmen,
   Wenn Gesamtausdruck gewählt, dann:
       Untere Datumsgrenze auf Minimum setzen,
       Obere Datumsgrenze auf Maximum setzen,
       Lagerauszug drucken.
    Sonst, wenn Ausdruck in Datumsgrenzen, dann:
       Untere Datumsgrenze übernehmen,
       Obere Datumsgrenze übernehmen,
       Lagerauszug drucken.
    Sonst, wenn Nachbestellübersicht, dann:
       Nachbestellübersicht drucken.
Ende.

Lagerauszug drucken:
    «Drucker einschalten»,
    Bis Dateiende:
        Dateiindex F weiterzählen,
        Wenn Dateiindex F>F1, dann:
            Dateiende.
        Sonst:
            Wenn D(F)>=D1 und D(F)<=D2, dann:
                Eintrag ausdrucken.
    Schleifenende (Dateiende),
    Gesamtwert W0 ausdrucken.
Ende.
```

```
Nachbestellübersicht drucken:
    «Drucker einschalten»,
    Bis Dateiende:
        Dateiindex F weiterzählen,
        Wenn Dateiindex F>F1, dann:
            Dateiende.
        Sonst:
            Wenn M(F)<M1(F), dann:
                Eintrag ausdrucken.
    Schleifenende (Dateiende).
Ende.

Eintrag ausdrucken:
    Wenn Z=0, dann:
        Kopfzeile ausdrucken,
        Z ← Z+1.
    Sonst:
        N$(F),M(F),M1(F),W1(F) drucken,
        M(F) * W1(F) drucken,
        W0 ← W0+M(F) * W(F),
        neue Zeile anfahren,
        Z ← Z+1.
    Wenn Z>Maximum, dann:
        Neue Seite anfordern,
        Z ← 0.
Ende.
```

Beachten Sie, daß wir die Grundvariablen Dateiindex F, Gesamtwert W0 und Zeilennummer Z bereits im Hauptmodul dieses Abschnitts initialisieren. Das hätte auch auf der nachfolgenden Ebene geschehen können, doch müßten wir diese Operation dann *zweimal* ausführen. Dadurch, daß wir sie in der Hierarchie so weit wie möglich nach oben geschoben haben, ersparen wir uns Programmieraufwand. Es empfiehlt sich generell, ein Programm auf solche Operationen hin durchzuforsten, die sinnvoll «ausgeklammert» werden können. Das hilft Programmspeicherplatz und Rechenzeit einzusparen.

Beachten Sie weiter die Art und Weise, wie wir den Gesamtausdruck der Lagerdatei behandeln. Anstatt hier – wie ursprünglich geplant – einen eigenen Modul vorzusehen, ist es sinnvoller, das als Sonderfall des Ausdrucks zwischen Datumsgrenzen zu behandeln. Man setzt einfach die Untergrenze auf den kleinstmöglichen Wert, also

$$D1 = 0$$

und die Obergrenze auf den größtmöglichen Wert, d.h.

$$D2 = 999999$$

Das erfordert zwar bei jedem Eintrag ein – eigentlich überflüssiges – Überprüfen der Grenzen, was etwas Zeit kostet. Doch auf die Rechenzeit kommt es uns hier wirklich nicht an: Selbst der schnellste Drucker ist um Größenordnungen langsamer als unser BASIC-Rechner. Warten müssen wir also auf alle Fälle.

So erspart man sich bei vernünftigen Anforderungen an das Programm mitunter sehr viel Programm- und Arbeitsaufwand.

3.16 Probleme der Datenspeicherung

Der Modul «Programm beenden» beinhaltet als einzige Aufgabe das Abspeichern der Datei auf einen externen Massenspeicher, in der Regel also auf Band oder Diskette. Das Angebot von BASIC-Versionen mit Massenspeichermöglichkeiten reicht im Mikrocomputersektor nun aber von Ausführungen, die sehr komplexe Dateien auf Disketten aufbauen und verwalten können über solche, die relativ einfache Datenspeicherungsoperationen auf Tonbandkassetten beherrschen, bis hin zu solchen Versionen, die zwar Programme, aber keine Daten extern festhalten können. Jede dieser Möglichkeiten erfordert besondere Befehle, und jeder dieser Befehle wurde vom jeweiligen Hersteller jeweils neu definiert. Die Situation ähnelt dem Turmbau zu Babel – keiner versteht den anderen mehr. Dennoch gibt es ein paar Grundkon-

zepte, die sich in den verschiedenen Lösungsmöglichkeiten widerspiegeln. Kennt man diese Grundkonzepte, so ist es nicht mehr ganz so schwer, sich zurechtzufinden. Wir wollen hier zwei Ansätze behandeln:

- Speicherung von numerischen Feldern auf Tonbandkassetten
- Speicherung komplexer Dateien auf Disketten

Die verschiedenen Möglichkeiten zum Umgang mit Massenspeichern sind ziemlich kompliziert und umfangreich. Wir werden uns hier nur auf die allerwichtigsten Grundlagen beschränken, eine erste Orientierungshilfe bieten – mehr nicht. Sie werden bei genauem Studium Ihrer Version feststellen, daß – besonders bei ausgebauteren Systemen – sehr viele Feinheiten und Tricks möglich sind, die den Umgang mit den gespeicherten Daten beträchtlich erleichtern.

*Numerische Felder abspeichern und laden: CSAVE * und CLOAD **

Das Problem besteht hier darin, die in den Feldern D(F), M(F), M1(F), N$(F) und W1(F) festgehaltenen Daten auf Band abzuspeichern und bei Bedarf wieder in den Rechner zurückzuholen. In den einfachsten BASIC-Versionen von Microsoft gibt es dazu die Befehle

CSAVE * «X» (Cassette SAVE Array X – Feld X auf Kassette speichern)

und

CLOAD * «X» (Cassette LOAD ARRAY X – Feld X von Kassette laden)

«X» bezeichnet hier den Namen des abzuspeichernden bzw. in den Rechner zu ladenden Felds (ohne Indexangaben!). Um die Datenaufzeichnung aus dem Feld D festzuhalten, müßten wir beispielsweise befehlen:

```
>CSAVE* "D"
```

und um die Lagermenge in das Feld M zurückzuholen, ist der Befehl

```
>CLOAD* "M"
```

notwendig.

Der Umgang mit diesen Befehlen ist im Grunde unkompliziert:

- Kassette in den Rekorder einlegen
- Band auf die richtige Stelle setzen
- Befehl CSAVE * bzw. CLOAD * mit den richtigen Argumenten angeben (noch kein RETURN drücken!)
- Tonband einschalten
- Befehl durch RETURN aktivieren
- Band laufen lassen, bis sich der Computer zurückmeldet
- Band abschalten

Der Rechner wird bei CLOAD * angeben, ob der Ladevorgang erfolgreich war oder ob ein Ladefehler aufgetreten ist. Sollte ein Ladefehler aufgetreten sein, dann müssen Sie den Vorgang wiederholen. Ansonsten kann weitergearbeitet werden.

Schwierigkeiten gibt es bei diesen Befehlen aus zwei Gründen:

- Bei einigen BASIC-Versionen kann man den Befehl CLOAD * zwar im Programm angeben, nach dem Laden der Felddaten kehrt der Rechner jedoch nicht in das Programm zurück, sondern meldet sich mit

  ```
  READY
  >
  ```

  von außerhalb des Programms wieder. Nach CSAVE *dagegen arbeitet der Rechner in der Regel wie nach jeder anderen Anweisung auch im Programm weiter.
- CSAVE * und CLOAD * verarbeiten nur *numerische* Felder. Ein unmittelbares Abspeichern und Laden von Zeichenkettenfeldern ist nicht vorgesehen.

Wenn Ihre BASIC-Version nach jedem CLOAD * aus dem Programm aussteigt, müssen Sie eine Möglichkeit finden, wieder an die unterbrochene Programmstelle zurückzukommen, ohne die Daten zu verlieren. Meist wird der Fortsetzungsbefehl

CONT

dazu genügen. Das ist zwar ein ziemlicher Umstand, jedoch immer noch besser als gar nichts. Wenn CONT nicht verfängt, dann müssen Sie untersuchen, ob die Variablen beim Einstieg in das Programm (RUN) erhalten bleiben oder ob sie gelöscht werden. Das läßt sich mit folgendem Test herausbekommen:

```
>NEW
>10 PRINT A
```

Das ganze Programm besteht hier darin, den Wert der Variablen A auszudrucken. Es gibt dabei zwei Möglichkeiten, entweder:

```
>A = 123
>RUN
 123
```

dann behält Ihre BASIC-Version die alten Variablenwerte beim Neustart bei, oder:

```
>A = 123
>RUN
 0
```

dann löscht Ihre Version die Variablen bei jedem Neueinstieg mit RUN.

Da es nun nicht sehr sinnvoll ist, die Möglichkeit zum Laden der Werte eines Felds vom Band vorzusehen, aber keinen Weg zu bieten, mit diesen Werten auch zu arbeiten, muß in diesem Fall ein weiterer Befehl vorhanden sein, der den Einstieg in das Programm ohne Löschen der Variablenwerte ermöglicht. In einigen Versionen ist das der Befehl

EXEC oder XEQ

der für «execute» – abarbeiten – steht und ansonsten wie RUN wirkt.

Unabhängig davon müssen Sie bei Versionen, die keinen Wiedereinstieg an die beim CLOAD * -Vorgang unterbrochene Stelle gestatten, die Fehler am Beginn der Arbeit von Hand laden und dann erst das Programm starten. Damit fällt im ersten Programmmodul der Teil «Datei laden» fort.

Zeichen, Zeichensatz und Zeichenkodes: ASC(X$), CHR$(X), INPUT-Feinheiten

Die zweite Einschränkung allerdings, ausschließlich mit numerischen Feldern arbeiten zu können, wirft unser ganzes Konzept über den Haufen. Wir müssen offensichtlich doch unsere Forderung aufgeben, den Artikel mit Namen zu bezeichnen und uns auf die fehlerträchtige Kennzeichnung durch Artikelnummern zurückziehen. Oder etwa nicht?

Nun – nicht unbedingt. Wenn es eine Möglichkeit gäbe, die alphanumerischen Zeichen – gleich welcher Art – so in Zahlenwerte umzuwandeln, daß sie daraus wieder unverändert gewonnen werden können, dann könnte man ja diese Zahlenwerte anstelle der alphanumerischen Zeichen speichern. Das macht die Arbeit zwar ein gutes Stück umständlicher und langsamer, aber wenn es möglich ist, dann sollten wir es zumindest einmal ausprobieren.

Das erfordert allerdings, daß wir uns etwas intensiver ansehen, wie so ein alphanumerisches Zeichen eigentlich für den Computer aussieht. Genauer gesagt müssen wir uns nicht nur ein solches Zeichen ansehen, sondern alle alphanumerischen Zeichen, die der Computer versteht, und das umfaßt Buchstaben, Ziffern, Satzzeichen und eine Reihe von Hilfszeichen, die eine bestimmte Funktion auslösen, wie es z.B. die Anwahl einer neuen Zeile darstellt.

Alle diese Zeichen sind zu einem sogenannten *Zeichensatz* zusammengefaßt. Ein solcher Zeichensatz ist jedoch kein kunter-

bunter Haufen von alphanumerischen Zeichen, sondern im Gegenteil streng gegliedert. Wie im Alphabet das «A» vor dem «B» und das «Z» ganz zum Schluß kommt und man daher die Buchstaben von 1 bis 25 durchnumerieren kann, so gibt es auch in jedem Mikrocomputer-Zeichensatz eine strenge Abfolge der in ihm enthaltenen Buchstaben, Ziffern, Satz- und Sonderzeichen. Man spricht deshalb auch hier oft von dem *Alphabet,* das der Computer beherrscht.

Jedes Zeichen in einem solchen Alphabet besitzt seine eigene unverwechselbare «Platznummer». Und genau diese Nummer ist es, mit welcher der Computer beim Hantieren mit einem bestimmten Zeichen arbeitet. Für ihn ist die Zeichenkette

A - R - B - E - I - T

nichts weiter als eine Aufeinanderfolge von sechs Zahlen, welche die Stelle dieser Zeichen im Alphabet wiedergeben. So kennt der Computer in der Regel von der ganzen «ARBEIT» beispielweise nur die Zahlenfolge

65 - 82 - 66 - 69 - 73 - 84

Erst wenn die Zeichenkette an einen Drucker oder ein Bildschirmgerät ausgegeben werden soll, werden die Zahlenwerte in die ihnen entsprechenden Buchstabenbilder umgewandelt. Und umgekehrt werden die über die Tastatur empfangenen Zeichen im Eingabevorgang verschlüsselt. d.h. in die zugeordneten Zahlen umgewandelt, die dann vom Rechner weiterverarbeitet werden können.

Mit anderen Worten: Jedes Zeichen aus dem Zeichensatz, den der betreffende Computer beherrscht, ist intern nach einem bestimmten *Zeichenkode* verschlüsselt. Dieser Kode ist gerade durch die Aufeinanderfolge der betreffenden Zeichen im Alphabet festgelegt. Dabei hat man sich auf verschiedene Zeichenkodes geeinigt, von denen die meisten Mikrocomputer den international genormten «ISO-7-bit»-Zeichenkode verwenden, in der Regel in seiner amerikanischen Variante, dem «ASCII-Kode». (ASCII

ist die Abkürzung von «American Standard Code for Information Interchange», spricht sich Ass-ki und bedeutet etwa «Amerikanischer Standardkode zum Informationsaustausch».) Der ASCII-Kode umfaßt 128 Zeichen, die von 0 bis 127 durchnumeriert sind. Der größte Teil davon sind sogenannte «druckende» Zeichen, die man auf Papier oder dem Bildschirm wiedergeben kann. Daneben gibt es eine Reihe von *Steuerzeichen,* die sogenannten «control codes», die jeweils eine bestimmte Operation des Rechners veranlassen. Ein solches Steuerzeichen im ASCII-Kode ist beispielsweise das uns schon länger vertraute «CONTROL-C», das dort übrigens die Nummer 3 trägt.

Es gibt also eine enge Beziehung zwischen Zahlen und den Zeichen des jeweils verwendeten Alphabets. Wenn es aber eine solche enge Beziehung gibt, dann sollte es auch die Möglichkeit geben, derartige Zeichen *mit Absicht* in die ihnen entsprechenden Zahlen umzuformen und umgekehrt. Das ist notwendig, da man mit den Zeichenkodes selbst nicht rechnen kann. Der Versuch der Subtraktion

```
>PRINT "1"-"1"
```

ergibt beispielsweise nicht etwa den Zeichenkode 0, sondern die Fehlermeldung

```
?TM ERROR
```

Wollen wir wirklich mit den Zeichenkodes selbst rechnen, dann müssen wir sie erst geeignet in *Zahlenwerte* umformen. Und umgekehrt, wollen wir zu einer Zahl – z.B. zu 65 – das entsprechende Druckzeichen haben, dann kommen wir mit

```
>PRINT 65
 65
```

oder

```
>PRINT "65"
65
```

nicht weiter. Das erste ergibt die *Zahl* fünfundsechzig und das andere die *Zeichenkette* sechs – fünf. Was wir hier eigentlich befehlen wollten, war ja

«Drucke das Zeichen Nummer 65 aus!»

Nun heißt «Zeichen» auf englisch «character», so daß sich ein solches Argument im PRINT-Befehl als

CHARACTER(65)

wiedergeben ließe – allerdings nur beinahe. Es gibt zwar in den meisten BASIC-Versionen, die Zeichenketten verarbeiten können, eine Funktion, die zum angegebenen Zeichenkode das zugehörige Zeichen liefert, doch ist «CHARACTER» dafür viel zu lang. Man kürzt daher ab und schreibt

CHR(65)

was jedoch immer noch nicht ganz stimmt. So, wie die Funktionsangabe jetzt abgefaßt ist, nimmt BASIC an, daß ein numerischer Wert zurückgeliefert wird. Wir wollen aber einen *alphanumerischen* Wert, einen «String», haben und müssen das für den BASIC-Rechner geeignet kennzeichnen, indem wir der Funktionsbezeichnung ein Dollarzeichen nachstellen:

CHR$(65)

Das liest sich im Englischen dann beispielsweise als «character-string of 65». Bauen wir das in einen PRINT-Befehl ein, dann erhalten wir

```
>PRINT CHR$(65)
A
```

was besagt, daß das 65. Zeichen im verwendeten Zeichenkode ein «A» ist. Das läßt sich ausbauen. Mit Hilfe der CHR$-Funktion können wir das Wichtigste über den im betreffenden Rechner verwendeten Zeichensatz erfahren, ohne das Handbuch bemühen zu müssen. Wir müssen dazu nur der Reihe nach PRINT

CHR$(0), PRINT CHR$(1) usw. befehlen und uns ansehen, was dabei passiert. Und um Tipperei zu sparen, formulieren wir das als Programm:

```
>NEW
>10 FOR N=0 TO 127
>20 PRINT N,CHR$(N),
>30 NEXT N
```

Lassen Sie das einmal laufen. Sie erhalten dann – schön in Spalten angeordnet – den Zeichenkode und daneben das ihm entsprechende (druckende) Zeichen. Bei einigen Kodes passiert etwas, z.B. wird eine neue Zeile angelaufen oder das Bild verschwindet vom Schirm u.ä. Andere Kodes scheinen bei manchen Rechnern überhaupt keine Wirkung zu haben. Das sind dann in der Regel Steuerzeichen, die normalerweise nicht alle ausgenutzt werden.

So weit ist das ganz nett, nur geht das alles viel zu schnell. Es ist kaum möglich, sicher zu sagen, welches Ereignis durch welchen Kode ausgelöst worden ist. Fügen wir daher noch eine Verzögerungsschleife in das Programm ein, und zwar so, daß erst der Zeichenkode (N) ausgedruckt wird, der Rechner dann etwas wartet, damit diese Zahl sicher gelesen werden kann und er anschließend erst die zugehörige Aktion (PRINT CHR$(N)) ausführt.

```
>20 PRINT N,
>30 FOR M=1 TO 100 : NEXT M
>40 PRINT CHR$(N)
>50 NEXT N
```

Sehen Sie sich das ein paarmal an, daß Sie ungefähr damit vertraut werden, was bei welchem Zeichen geschieht. Sie brauchen es allerdings auf keinen Fall auswendig zu lernen – für so etwas ist Ihr Handbuch da, und außerdem finden Sie anschließend an unseren Kurs eine Übersichtstabelle, in welcher die Zeichen des ASCII-Kodes aufgeführt sind.

Von einem Steuerzeichen allerdings sollten Sie sich den Kode merken, da er am häufigsten gebraucht wird, derjenige nämlich,

der den Bildschirm löscht. Es handelt sich im ASCII-Kode in der Regel um das Steuerzeichen «Form Feed» – Formularvorschub (das kommt von der Druckeransteuerung her, wo durch diesen Befehl eine neue Seite angefordert wird). Dieses mit FF abgekürzte Zeichen besitzt den Kode 12, d.h., bei vielen Computern wird durch

```
PRINT CHR$(12);
```

der Schirm gelöscht.

Beachten Sie den Strichpunkt: In den meisten Fällen wird beim Löschen des Schirms der Kursor in die linke obere Ecke zurückgeführt. Der Strichpunkt gewährleistet, daß er da auch stehenbleibt. Das ist in all den Fällen notwendig, in denen Sie den Schirm nicht voll ausnutzen, also mit der ersten und nicht mit der zweiten Zeile anfangen wollen. Wird das nicht unbedingt so gebraucht, dann kann der Strichpunkt auch entfallen.

Allerdings reagieren nicht alle Rechner auf das FF-Zeichen mit dem Löschen des Bildschirms. Insbesondere, wenn der Bildschirm nicht im Gerät enthalten, sondern als *Videoterminal* getrennt davon aufgestellt ist, benötigt man für diese Funktion auch im ASCII-Kode entweder ein anderes Steuerzeichen oder eine ganze Folge solcher Zeichen.

Im letzteren Fall wird der Zeichenfolge zur Unterscheidung von normalen Druckzeichen ein besonderes Steuerzeichen vorangesetzt, das sogenannte «Escape»-Zeichen («Flucht»-Zeichen), das anzeigt, daß die folgende Zeichenkette nicht zum normalen ASCII-Zeichensatz gehört, ihn sozusagen verlassen, «geflohen», hat. Dieses mit ESC abgekürzte Zeichen hat den Kode 27, was sich spätestens dann zu merken lohnt, wenn fremde Programme übernommen werden, die intensiv die Fähigkeiten des jeweiligen Bildschirmgeräts ausnutzen. Da allerdings die nachfolgenden Zeichen keineswegs standardisiert sind, bleibt nur zu hoffen, daß in einem solchen Fall das Programm ausreichend dokumentiert ist. Sonst dürfte es sehr schwierig sein, herauszufinden, was das Programm eigentlich macht.

Nicht alle Mikrocomputer verwenden den ASCII-Kode. Einige haben eigene Kodedefinitionen, manche benutzen nur einen Teil der zur Verfügung stehenden Zeichen (z.B. nur die Großbuchstaben), und wieder andere haben den Kode um besondere Grafiksymbole erweitert, mit denen sich Punkte oder Zeichenelemente auf dem Bildschirm darstellen lassen. Bevor Sie im Handbuch zu diesem Punkt nachschlagen, sollten Sie versuchsweise selbst ansehen, was es da gibt. Erweitern Sie einfach die äußere FOR-NEXT-Schleife auf

```
>10 FOR N=1 TO 255
```

Dadurch erhalten Sie die den Zeichenkodes 0 bis 255 entsprechenden Darstellungen auf dem Schirm. In einigen Fällen befinden sich unter den Zeichen von 128 bis 255 die Grafiksymbole, die der Rechner beherrscht. Manchmal wird auch nur der Zeichensatz von 0 bis 127 wiederholt, oder es tut sich gar nichts. Auf alle Fälle sollten Sie nach diesem Versuch in Ihrem Handbuch nachsehen, was dort dazu geschrieben steht.

Wenn man auf diese Weise eine Zahl in ein Zeichen umwandeln kann, gibt es dann auch den umgekehrten Weg, ein Zeichen in eine Zahl umzuformen? Aber ja doch: In den meisten BASIC-Versionen, die mit ASCII-Zeichen arbeiten, lautet dieser Befehl sinngemäß:

«Wandle das Zeichen X$ in die zugehörige ASCII-Kodeziffer um!»

Kurz heißt das

ASCII(X$)

oder noch häufiger anzutreffen

ASC(X$)

Beachten Sie hierbei, daß ASC zwar ein Zeichen als Argument übernimmt, daß es aber einen *numerischen* Wert liefert. Aus diesem Grund darf ihm nach den BASIC-Regeln *kein* Dollarzeichen

nachgestellt werden. Andernfalls erhält man eine Fehlermeldung. Mit ASC allein läßt sich aber arbeiten:

```
>PRINT ASC("A")
 65
```

Sie sollten sich auch mit dieser Funktion vertraut machen. Das kann beispielsweise so geschehen, daß in einer einfachen Schleife eine Eingabe über die Tastatur angefordert und der zugehörige ASCII-Kodewert ausgedruckt wird.

```
>NEW
>10 INPUT "Taste";A$
>20 PRINT ASC(A$)
>30 GOTO 10
>RUN
Taste? 1
49
Taste? A
65
Taste? +
43
Taste? "
?FC ERROR IN 20
```

Das geht offensichtlich nicht immer glatt. In diesem Falle hier wird durch «FC» ein Fehler im Zusammenhang mit «function call», mit dem Aufruf einer BASIC-Funktion in Zeile 20 gemeldet. Die gemeinte Funktion ist ASC, und da die Zeile formal stimmt und auch in den ersten Schritten richtig abgearbeitet worden ist, kann der Fehler nur am Wert des Arguments selbst liegen. Das Argument ist A$ – folglich stimmt irgend etwas mit unserem INPUT-Befehl nicht. Sehen wir uns doch den Inhalt der Variablen A$ vor der Weiterverarbeitung durch ASC erst einmal an. Und damit wir genau erkennen können, was da eingegeben wurde, schließen wir es zwischen zwei deutlich sichtbare Begrenzer ein, z.B. zwischen zwei Ausrufezeichen:

```
>15 PRINT "!";A$;"!",
>RUN
Taste? 1
!1!          49
```

```
Taste? A
!A!          65
Taste? "
!!
?FC ERROR IN 20
```

Das ist eine wichtige Erkenntnis zum Verhalten des Mustergeräts: Ein einfaches Anführungszeichen wird bei Eingabe über den INPUT-Befehl ignoriert. Da Anführungszeichen zum Einschließen anderer Zeichen dienen, liegt es nahe, sich das in Zusammenhang mit dem INPUT-Befehl einmal anzusehen:

```
>RUN
Taste? "1"
!1!          49
```

So weit ist das Ergebnis nicht verschieden von dem einfachen Drücken der 1-Taste. Und wenn wir einen Leerschritt versuchen?

```
Taste? " "
! !          32
```

Dem steht das Verhalten gegenüber, wenn wir *nur* einen Leerschritt eingeben:

```
Taste?
!!
?FC ERROR IN 20
```

Wir können im Mustergerät also mit Hilfe der Anführungszeichen die Begrenzung bei der INPUT-Eingabe teilweise überwinden. Das ist wichtig zu wissen – doch es ist nur das Verhalten des Mustergeräts. Ihr Computer wird sich möglicherweise ganz anders benehmen. Nehmen Sie sich hier die Zeit, auch damit vertraut zu werden. Probieren Sie alle Tasten durch. Notieren Sie die Ergebnisse. Versuchen Sie dann, den Grund für Abweichungen vom Erwarteten experimentell herauszufinden, so wie wir es oben getan haben. Sie müssen Ihr Gerät aus dem Eff-Eff kennen, wenn Sie es wirklich ausnutzen wollen. Außerdem wird das Umsteigen auf andere Geräte und BASIC-Versionen, ja auf an-

dere Sprachen erleichtert, wenn man von vornherein geübt hat, vor Schwierigkeiten nicht zu kapitulieren, sondern sie bewußt aufzusuchen, kennenzulernen und, wenn es sein muß, zu umgehen.

Mit unserem Testprogramm können wir im übrigen noch eine Frage im Zusammenhang mit der ASC-Funktion beantworten: Was geschieht, wenn wir nicht nur ein Zeichen, sondern eine längere Zeichenkette als Argument verwenden?

```
>RUN
Taste? ABCDE
!ABCDE!      65
Taste? EFGHI
!EFGHI!      69
Taste? A
!A!          65
Taste? E
!E!          69
```

Die ASC-Funktion übernimmt also nicht nur einzelne Zeichen, sondern auch ganze Zeichenketten, verarbeitet davon aber immer nur das erste, ganz links stehende Zeichen. Dieses Verhalten kann manchmal Arbeit einsparen helfen.

Umwandlung von Zeichenketten in numerische Felder

Kehren wir zu unserem Problem zurück. Wir suchen einen Weg, Zeichenketten in numerische Felder umzuwandeln und umgekehrt. Mit CHR$ und ASC haben wir die wichtigsten Werkzeuge bereits zur Hand. Im Prinzip könnten wir bereits jetzt die Zeichenkette N$ in das Feld N(F) umwandeln. Dazu müssen wir nur die Zeichenkette von links nach rechts abtasten, die vorgefundenen Zeichen in ihre numerischen Kodewerte umwandeln und in den zugehörigen Feldelementen von N speichern. Legen wir maximal 15 Zeichen lange Ketten zugrunde, dann würde das beispielsweise so aussehen:

```
>NEW
>10 DIM N(15)
>20 INPUT "Kette";N$
>30 FOR F=1 TO LEN(N$)
>40 N(F) = ASC(MID$(N$,F,1))
>50 PRINT MID$(N$,F,1);N(F);
>60 NEXT F
```

Das funktioniert so: In Zeile 10 dimensionieren wir das Feld N auf die zu erwartende Maximallänge der Kette. Deren Wert wird durch den INPUT-Befehl von Zeile 20 bestimmt und in der FOR-NEXT-Schleife von Zeile 30 bis Zeile 60 in das Feld N umgewandelt. Zeile 30 legt die Variable F als Laufvariable fest. Ihr Wert dient in Zeile 40 als Feldindex für N und als Platzangabe in der Zeichenauswahlfunktion MID$ für die Zeichenkette N$. Aus diesem Grund darf F nicht über die Kette hinauslaufen, was in Zeile 20 durch die Endfestlegung LEN(N$) erreicht wird. Die eigentliche Umwandlung erfolgt in Zeile 40, wo mit MID$(N$,F,1) ein Zeichen aus der Kette ausgewählt, mit ASC in den Zeichenkode umgeformt und dann dem Feldelement N(F) zugewiesen wird. Zeile 50 protokolliert diese Umwandlung, indem zuerst das aus der Kette ausgewählte Zeichen und unmittelbar darauf der dem Feldelement zugewiesene Kode ausgedruckt wird.

Das funktioniert einwandfrei:

```
>RUN
Kette? ABC
A 65 B 66 C 67
```

Und solange nur eine oder ein paar wenige kurze Ketten dieser Art umzuwandeln sind, ist das durchaus annehmbar. Unser Fall liegt aber etwas anders. Wir müssen sehr viele solche Ketten im Speicher festhalten. Das bedeutet, daß wir mit dem Platz haushalten müssen. Und das wiederum wird durch den Umstand erschwert, daß ein Zeichen der Kette im Speicher nur ein Byte belegt, eine Zahl – in unserer Version – aber 5 Bytes verlangt. Andere Versionen kommen in der Regel mit 4 Bytes je numeri-

schem Feldelement aus. Doch viel ist auch damit nicht eingespart. Wir brauchen bei numerischer Speicherung in unserer Testversion für einen Artikelnamen etwa voraussichtlich fünfmal so viel Platz wie bei alphanumerischer Speicherung. Man muß allerdings den Verwaltungsbedarf noch mit einrechnen. Sehen wir uns diesen Platzbedarf doch anhand unseres Beispiels an. Dazu brauchen wir uns nur den jeweils belegten Speicherplatz ausdrucken zu lassen:

```
>5 A = FRE(0)
>15 PRINT FRE(0)-A : A = FRE(0)
>25 PRINT FRE(0)-A
```

Den Protokollausdruck von Zeile 50 brauchen wir nicht mehr:

```
>50
```

Unser Programm sieht damit so aus:

```
>LIST
 5 A = FRE(0)
 10 DIM N(15)
 15 PRINT FRE(0)-A : A = FRE(0)
 20 INPUT "Kette";N$
 25 PRINT FRE(0)-A
 30 FOR F=1 TO LEN(N$)
 40 N(F) = ASC(MID$(N$,F,1))
 60 NEXT F
```

In Zeile 5 wird der noch verfügbare Speicher gemessen und als Bezug in A festgehalten. Zeile 15 druckt den von Feld N belegten Speicherplatz aus und legt danach in A einen neuen Bezug fest. In Zeile 25 schließlich wird der von der Zeichenkette N$ benötigte Speicherbedarf gemeldet. Das alles bringt folgendes Ergebnis:

```
>RUN
 87
Kette? 123456789012345
 22
```

Das fünfzehn Zahlen umfassende Feld N belegt also mit 87 Bytes rund viermal soviel Platz im Speicher wie die fünfzehn Zeichen lange Kette N$, die nur 22 Bytes benötigt. Bei 200 und mehr Artikeln ist das natürlich unhaltbar. Wir müssen einen Weg finden, möglichst viel Platz einzusparen.

Der Schlüssel dazu liegt in den Zahlen selbst. Eine ASCII-Kodezahl kann 128 Werte von 0 bis 127 annehmen. Das ist wesentlich weniger, als man in einer numerischen Variablen ohne Genauigkeitsverlust festhalten kann. Man müßte dort also bei geschicktem Vorgehen mehr als nur ein Zeichen unterbringen können.

Das wiederum ist gar nicht so schwierig. Wenn ein Zeichen 128 verschiedene Werte annehmen kann, dann kommen sich zwei Zeichen nicht mehr in die Quere, wenn man den Wert des einen mit 128 multipliziert und dann den des zweiten dazuaddiert. Denn in diesem Fall belegt das erste Zeichen die Werte

$$0 * 128 = 0$$

$$1 * 128 = 128$$

$$2 * 128 = 256$$

und so weiter bis

$$127 * 128 = 16\,256$$

Die Zwischenräume werden von dem nächsten Zeichen ausgefüllt. Das läßt sich weiterführen, wenn man das Ergebnis der Zeichenkodekombination wieder mit 128 multipliziert und den Kode eines weiteren ASCII-Zeichens addiert. (Das ist genau dasselbe wie das, was bei unserer Zahlenschreibweise im Dezimalsystem geschieht, nur daß in unserem Fall der Abstand von «Stelle» zu «Stelle» nicht 10, sondern 128 Zeichen umfaßt.)

Das geht so lange, wie der betreffende Variablenwert mit 128 multipliziert werden kann, ohne daß ein Genauigkeitsverlust eintritt. Wie weit es reicht, d.h. wieviele Zeichen in einer Variablen sicher festgehalten werden können, läßt sich berechnen. Wir können aber auch ein kleines Experiment dazu machen:

```
>NEW
>10 N = 0
>20 FOR I=1 TO 10
>30 N = N*128+127
>40 PRINT I,N
>50 NEXT I
>RUN
 1             127
 2             16383
 3             2097151
 4             268435455
 5             3.43597384E+10
 6             4.39804651E+12
 7             5.62949954E+14
 8             7.2057594E+16
 9             9.22337204E+18
 10            1.19059162E+21
```

Bis zum fünften Durchgang nehmen die Dezimalstellen von N beständig zu. Ab dem fünften Durchgang aber geht der Rechner zum wissenschaftlichen Format über. Die Zahl der Dezimalstellen bleibt bei maximal neun stehen. Der Wert wird gerundet und seine Größe durch den Zehnerexponenten nach E angegeben. Wir haben das alles ja bereits besprochen. Wichtig ist für uns hier, daß die Zahlen bis einschließlich des vierten Durchgangs bis zur letzten Stelle genau sind. Erst im fünften Durchgang geht der Rundung etwas an Genauigkeit verloren. Dieses Runden bedeutet aber auch, daß das in diesem Durchgang eingearbeitete Zeichen nicht mehr zurückgewonnen werden kann. Denn von seinem Kode fehlen ja einige Stellen in der endgültigen Form der Zahl.

Das wirft die Frage danach auf, wie man die so gespeicherten Zeichen wiedergewinnen kann. Nun – wenn sich die Zahl N aus den Kodes der Zeichen A und B so zusammensetzt

$$N = 128 * A + B$$

dann erhalten wir durch Teilen mit 128

$$N/128 = A + B/128$$

Nun hat B aber als ASCII-Kode immer einen Wert zwischen 0

und 127, d.h., der Wert von B/128 ist immer gebrochen. Symbolisch läßt sich das so schreiben:

$$B/128 = 0.bbbbbb$$

wobei «bbbbbb» gerade die Zahl darstellt, die 1/128 des Zeichenkodes von B ausmacht. Beim Teilen von N durch 128 steht vor dem Komma demnach der Kode für A, hinter dem Komma dagegen der Kode von B, geteilt durch 128.

$$N/128 = aaa.bbbbbb$$

Entfernen wir den ganzzahligen Anteil, dann bleibt B/128 übrig:

$$N/128\text{-}INT(N/128) = 0.bbbbbb$$

Und um B zu erhalten, müssen wir das Ganze wieder mit 128 multiplizieren:

$$(N/128\text{-}INT(N/128)) * 128 = N\text{-}128 * INT(N/128) = B$$

Kompliziert? Nun – sehen wir uns das einfach an einem Beispiel an. Schreiben wir uns ein kleines Testprogramm:

```
>NEW
>10 INPUT "A";A$
>20 INPUT "B";B$
>30 A = ASC(A$) : B = ASC(B$)
>40 PRINT "A =";A,"B =";B
>50 N = 128*A+B
>60 PRINT "N =";N
>70 N1 = N/128
>80 PRINT "N/128 =";N1
>90 PRINT "INT(N/128) =";INT(N1)
>100 N2 = INT(N1) : N3 = N-128*N2
>110 PRINT "A(neu) =";N2
>120 PRINT "B(neu) =";N3
>130 PRINT "A$ = ";CHR$(N2),"B$ =";CHR$(N3)
>140 GOTO 10
```

Hier vollziehen wir den ganzen Rechengang der Erklärung von oben nach und lassen die jeweiligen Zwischenergebnisse ausdrucken. Sehen wir uns an, was dabei geschieht:

```
>RUN
A? A
B? B
A = 65          B = 66
N = 8386
N/128 = 65.515625
INT(N/128) = 65
A(neu) = 65
B(neu) = 66
A$ = A          B$ = B
```

Es kommt am Schluß also genau das wieder heraus, was wir am Anfang eingegeben haben. Experimentieren Sie jetzt selbst ein bißchen damit. Suchen Sie sich drei, vier Zeichenpaare heraus, ermitteln Sie ihre Zeichenkodes und rechnen Sie erst einmal von Hand durch, was geschehen wird. Dann geben Sie die Werte in das Programm ein und vergleichen den Ausdruck mit Ihrer Vorhersage.

Das Verfahren läßt sich ohne weiteres auf drei und mehr Zeichen erweitern. Setzt sich N z.B. so aus den Zeichenkodes A, B und C zusammen:

$$N = (A * 128 + B) * 128 + C$$

dann erhalten wir als Ergebnis des ersten Schritts

$$INT(N/128) = A * 128 + B$$

$$N\text{-}128 * INT(N/128) = C$$

In einem zweiten Schritt erhält man dann aus $A * 128 + B$ gerade wie oben die Kodewerte von A und B.

Erweitern Sie selbst unser Testprogramm auf diesen Fall und sehen Sie sich ein paar Beispiele dazu an. Dann machen Sie dasselbe für vier Zeichen und wenn es sein muß auch für fünf, je nachdem, wann in ihrer Version der Rechner auf Fließkommaformat, d.h. auf die «wissenschaftliche» Zahlenschreibweise umgeschaltet hat. Sie werden feststellen, daß in diesem Fall die Zeichen nicht wieder richtig zurückgewonnen werden können.

Das genügt, um möglichst viel Zeichen in einem numerischen Feld unterzubekommen. Wir wissen, wieviel Zeichen in einer

Zahl gespeichert werden können (vier in unserem Fall), und wir wissen, daß die Kette maximal 15 Zeichen lang werden darf. Damit brauchen wir ein Feld mit vier Elementen, um einen Artikelnamen festhalten zu können. Dimensionieren wir also schon einmal:

```
>NEW
>10 DIM N(4)
```

Die Umwandlung der Zeichenkette in die Feldelemente muß nun so geschehen, daß die Zeichenkette selbst in Gruppen zu je vier Zeichen zerlegt und die Zeichen in diesen Gruppen nacheinander wie oben besprochen in einen numerischen Wert überführt werden. Das Problem ist noch, wie wir diese Zerlegung am besten durchführen.

Sehen wir uns die Zuordnung von Zeichen und Feldelementen einmal näher an:

Zeichen-Nr.	kommt in Feldelement-Nr.
1	1
2	1
3	1
4	1
5	2
6	2
7	2
8	2
9	3
10	3
11	3
12	3
13	4
14	4
15	4

Oder anders herum gesehen:

Feldelement-Nr.	erhält	Zeichen-Nr.
1		1+0, 1+1, 1+2, 1+3
2		5+0, 5+1, 5+2, 5+3
3		9+0, 9+1, 9+2, 9+3
4		13+0, 13+1, 13+2

Der Zusammenhang wird noch deutlicher, wenn wir mit Null zu zählen beginnen:

Feldelement-Nr.	erhält	Zeichen-Nr.
0+1		0*4+1, 0*4+2, 0*4+3, 0*4+4
1+1		1*4+1, 1*4+2, 1*4+3, 1*4+4
2+1		2*4+1, 2*4+2, 2*4+3, 2*4+4
3+1		3*4+1, 3*4+2, 3*4+3

Formulieren wir das allgemeiner, dann ergibt sich:

Feldelement-Nr.	erhält	Zeichen-Nr.
K+1		K*4+1, K*4+2, K*4+3, K*4+4

Wenn wir also zwei FOR-NEXT-Schleifen ineinander so verschachteln, daß die Variable K der äußeren Schleife von 0 bis 3 läuft und die Variable L der inneren Schleife ebenfalls von 1 bis 4 läuft, dann erhalten wir die numerische Entsprechung der Zeichenkette durch die Formel

```
N(K+1) = N(K+1)*128 + ASC(MID$(N$,K*4+L,1))
```

Die Laufvariable K der äußeren Schleife wählt dabei den Feldindex und den umzuwandelnden Viererabschnitt aus der Zei-

chenkette aus. Die Laufvariable L der inneren Schleife bestimmt, welches Zeichen aus diesem Abschnitt gerade einzurechnen ist. Wir müssen nur darauf achten, daß das Feldelement N(K + 1) vor Einstieg in die innere Schleife den Wert 0 hat. Andernfalls würde unser Ergebnis falsch werden.

Formulieren wir diesen Programmabschnitt schon einmal.

```
100 FOR K=0 TO 3
110 K1 = 4*K : K2 = K+1
120 N(K2) = 0
130 FOR L=1 TO 4
140 N(K2) = N(K2)*128+ASC(MID$(N$,K1+L,1))
150 NEXT L
160 NEXT K
```

Beachten Sie die beiden Hilfsvariablen K1 und K2, die wir in Zeile 110 eingeführt haben. Sie dienen uns dazu, aus der inneren Schleife alle unnötigen Rechenarbeiten auszuklammern. Jede derartige Rechnung nämlich kostet einiges an Abarbeitungszeit, und da die Umwandlung Zeichenkette in Zahlenfeld ohnehin schon viel Zeit kostet, sollten wir hier sparen, soviel es geht. In diesem Fall werden die Werte 4 * K und K + 1 nur einmal je Feld berechnet, statt wie in unserer Originalformel vier- bzw. achtmal. Sie sollten das, wo es *sinnvoll* ist, immer durchführen. Bedenken Sie aber auch, daß dieses Ausklammern von Teilrechnungen, besonders wenn es über weite Strecken hin geschieht, das Programm schwerer verständlich macht. Denn man sieht an den betreffenden Stellen ja nicht mehr, was genau geschieht. Sinnvoll ist das Ausklammern also überall da, wo Abarbeitungszeit eingespart werden *muß*. Ansonsten sollten Sie das Programm doch so lesbar wie möglich abfassen.

Wir müssen bei dieser Umwandlung allerdings zweierlei beachten:

In der Schleife werden *sechzehn* Elemente bearbeitet, denn K nimmt vier Werte an, und L nimmt vier Werte an. Das bedeutet, daß die ursprünglich maximal fünfzehn Zeichen lange Kette auf sechzehn Zeichen erweitert werden muß, soll in der ASC-Funk-

tion mangels Argument beim sechzehnten Durchgang keine Fehlermeldung erfolgen. Fügen wir zu diesem Zweck vor Einstieg in die Umwandlungsschleifen einfach rechts so viele Leerzeichen wie nötig an.

```
>90 IF LEN(N$)<16 THEN N$=N$+" " : GOTO 90
```

Das ist eine Minischleife, die so lange durchlaufen wird, wie die Kette N$ noch nicht 16 Zeichen umfaßt.

Weiter müssen wir darauf achten, daß in der umzuwandelnden Kette wirklich – unserer ursprünglichen Definition der Artikelnamen gemäß – nur 15 gültige Zeichen sind. Das sechzehnte Zeichen muß immer ein Leerschritt sein. Andernfalls finden wir den betreffenden Artikel in der Datei nicht mehr wieder.

D.h., wir müssen vor Beginn der Umwandlung die Maximallänge 15 von N$ garantieren. Das kann durch

```
>80 IF LEN(N$)>15 THEN N$=LEFT$(N$,15)
```

geschehen. Hier wird jede Vorgabe mit mehr als 15 Zeichen – die bei Eingabe über die Tastatur ja nicht auszuschließen ist – auf die Maximallänge gekürzt, wobei die zuerst eingegebenen Werte gültig sind.

Wir sollten uns das Umwandlungsergebnis ansehen. Dazu fügen wir einfach eine weitere FOR-NEXT-Schleife an:

```
>170 FOR K=1 TO 4
>180 PRINT N(K);
>190 NEXT K
```

Schließlich und endlich müssen wir uns davon überzeugen, daß auch wirklich das gespeichert wurde, was als N$ vorgegeben war. Dazu wandeln wir das Feld wieder zurück in eine Zeichenkette und lassen uns diese zwischen Ausrufezeichen als Begrenzern ausdrucken.

Dabei können wir die Überlegungen von oben fast unmittelbar

übertragen. Wir müssen lediglich darauf achten, daß das Feld jetzt *von hinten* entschlüsselt wird. Denn in jedem Teilstück erhalten wir zuerst das am weitesten rechts stehende Zeichen, dann das nächste links davon usw. Wir müssen die Schleife mit der Positionsangabe K also rückwärts (STEP −1) laufen lassen:

```
>200 N$ = "!" : REM - rechte Grenze setzen
>210 FOR K=4 TO 1 STEP -1
>220 FOR L=1 TO 4 : REM - Das dient nur als Zähler
>230 N1 = INT(N(K)/128)
>240 N$ = CHR$(N(K)-128*N1)+N$
>250 N(K) = N1
>260 NEXT L
>270 NEXT K
>280 N$ = "!"+N$ : REM - linke Grenze setzen
>290 PRINT : PRINT N$ : REM - N$ in neuer Zeile drucken
```

Fügen wir zum Experimentieren noch eine Anforderung von N$ über die Tastatur ein und machen aus dem Ganzen eine Schleife:

```
>20 INPUT "Kette";N$
>300 GOTO 20
```

Dann können wir den Umwandlungsvorgang überprüfen:

```
>RUN
Kette? A
 136843296   67637280   67637280   67637280
!A               !
Kette? ABC
 137404832   67637280   67637280   67637280
!ABC             !
Kette? ABCDEFGHIJKLMNOPQRSTUVWXYZ
 137404868   145859528   154314188   162768800
!ABCDEFGHIJKLMNO !
Kette? " "
 67637280   67637280   67637280   67637280
!                !
```

Der vorletzte Versuch machte deutlich, wie die Beschränkung der gültigen Zeichen der Kette auf 15 Stück arbeitet. Die übrigen Versuche zeigen außer den im Feld N festgehaltenen Kodes, daß tatsächlich immer 16 Zeichen verarbeitet werden.

Dieses Verfahren läßt sich natürlich auch anwenden, wenn der betreffende Rechner nur 3 Zeichen je Zahl aufnehmen kann. In diesem Fall muß die Vier in den Schleifen- und Indexangaben durch eine Drei ersetzt werden. Und es ist darauf zu achten, daß die Länge der Zeichenkette stimmt. Bei drei Zeichen je Zahl geht das genau mit fünf Zahlen auf: 3 * 5 = 15.

Ein letzter Versuch sei Ihnen selbst überlassen: Ermitteln Sie den Platzbedarf dieser Umwandlung und vergleichen Sie das mit dem Fall, in dem jedes Zeichen in genau eine Zahl überführt wird. Bedenken Sie dabei auch den vergrößerten Bedarf für das Programm selbst und versuchen Sie daraus zu ermitteln, ab wieviel im Speicher festzuhaltenden Zeichenketten sich der Mehraufwand lohnt. Erweitern Sie den Versuch außerdem auf die Umwandlung eines Zeichenkettenfelds, sagen wir mit einer Dimensionierung N$(5). Am sinnvollsten definiert man hier für N ein zweidimensionales Feld. Sehen Sie sich auch hier das Verhalten anhand von zwei, drei Beispielen an. Und schließlich und endlich sollten Sie sich überlegen, wie man mit diesen Werkzeugen unser Problem der externen Datenspeicherung durch CSAVE * und CLOAD * lösen kann. Denn darum ging es ja eigentlich.

Vom Umgang mit Diskettendateien

Die Speicherbefehle CSAVE * und CLOAD * sind zwar in einfachen Fällen unkompliziert zu handhaben, aber es hat sich gezeigt, daß auch nur ein bißchen mehr Anspruch an die Speichermöglichkeiten hier gleich ein recht umständliches Programmieren erfordert. Ausgebautere BASIC-Versionen – vor allem diejenigen, die zum Betrieb mit Diskettenspeichern («Floppy Disks») ausgelegt sind –, nehmen einem in dieser Hinsicht einiges an Arbeit ab und bieten darüber hinaus Arbeitsmöglichkeiten, die bei Datenspeicherung auf Band prinzipiell unmöglich sind. Man erkauft diesen Luxus allerdings durch komplexere Befehle, deren richtiger Einsatz einiges an Übung und Vertrautheit mit dem jeweiligen Computersystem erfordert. Darüber hinaus unter-

scheiden sich die von den verschiedenen Herstellern angebotenen BASIC-Versionen beim Diskettenbetrieb beträchtlich voneinander, so daß es sehr schwer ist, etwas Allgemeinverbindliches darüber auszusagen. Aus diesem Grund werden wir uns hier recht allgemein halten müssen. Wir wollen uns die wichtigsten Punkte ansehen, die es beim Umgang mit Diskettendateien zu beachten gibt – gerade so viel, daß die diesbezügliche Darstellung im Handbuch verständlicher und vertrauter wird. Falls Sie eine BASIC-Version mit Diskettenspeichermöglichkeiten besitzen, dann lesen Sie am besten für jeden hier angesprochenen Punkt gleich in Ihrem Handbuch nach, was dort an konkreten Hinweisen zu finden ist.

Zunächst: Was unterscheidet eigentlich eine Diskette von einem Tonband? Nun – die Antwort darauf kann Bände füllen. Wir wollen hier nicht auf die Einzelheiten eingehen – dazu gibt es andere Bücher. Fassen wir uns deshalb etwas genauer: Was unterscheidet eigentlich von BASIC aus gesehen die Datenspeicherung auf Disketten von der auf Band?

Da ist als erstes das Fassungsvermögen für Programme und Daten. Diese *Speicherkapazität* ist bei Disketten in der Regel beträchtlich größer als bei üblichen Tonbandkassetten. Wichtiger aber ist, daß man auf die auf Disketten festgehaltene Information sehr viel schneller *zugreifen* kann als auf die auf einem Band.

Das liegt schon am Prinzip. Auf einem Band werden die Daten schön hintereinander festgehalten. Wenn Sie dort am Anfang eines der verschiedenen Weltraumspiele festgehalten haben, dann (Gott behüte!) unser Lagerverwaltungsprogramm und darauf wiederum die eigentliche Lagerdatei, dann müssen Sie, um beispielsweise die Lagerdatei aufzufrischen, erst das Spielprogramm und dann das Programm zur Lagerverwaltung durchgehen, bevor Sie an das Abspeichern der Daten gehen können. Und wenn die alten Daten sicherheitshalber erhalten bleiben sollen, dann müssen Sie noch weiter umspulen. Selbst im schnellen Vorlauf dauert das seine Zeit.

Bei Disketten ist das anders. Hier sind die Daten auf der

Scheibe in konzentrischen Ringen, den *Aufzeichnungsspuren,* gespeichert. Jede solche Spur kann innerhalb von Sekundenbruchteilen angewählt werden, ohne irgendwelche davorstehende Information lesen zu müssen. Der Zugriff auf bestimmte Datenstücke wird weiter dadurch erleichtert, daß die Spuren in sogenannte *Sektoren* unterteilt sind. Ein solcher Sektor kann in der Regel 128 oder 256 Bytes festhalten. Diese 128 oder 256 Bytes sind die Grundeinheit, in der die Daten von und zu der Diskette übertragen werden. Und um einen bestimmten Sektor auf einer Spur zu erreichen, muß man auch nicht lange warten. Da sich eine Diskette 300- oder 360mal in der Minute um sich selbst dreht, also fünf- oder sechsmal in der Sekunde, dauert es auch hier höchstens ein Fünftel oder ein Sechstel einer Sekunde, bis die gewünschten Daten gefunden sind. Und – wohlgemerkt – man kann auf jede beliebige Aufzeichnungsspur der Diskette in weniger als einer Sekunde zugreifen.

Nun brauchen Sie sich als BASIC-Programmierer(in) um Spuren oder Sektoren auf der Diskette überhaupt keine Gedanken zu machen. Das erledigt der Rechner schon von alleine. Es ist nur wichtig, ungefähr zu wissen, was da vor sich geht, um die Möglichkeiten auch ausschöpfen zu können. Wie aber benutzt der Rechner diese Möglichkeiten?

Das ist eine Frage der Organisation der Aufzeichnungen auf der Diskette. Die vielen Daten sind in Form von logisch zusammengehörigen Einheiten, den *Dateien* (englisch: files) auf der Diskette zusammengefaßt. Jede Datei besitzt einen bestimmten unverwechselbaren Namen, und dieser – jetzt wird es wichtig –, dieser Name muß von Ihnen selbst bestimmt werden. Mit anderen Worten: Wenn Sie auf der Diskette irgendwelche Programme oder Daten abspeichern oder von dort zurückholen möchten, dann müssen Sie den *Dateinamen* («Filenamen») kennen, unter dem diese stehen sollen bzw. zu finden sind. Der Rechner führt auf jeder Diskette ein besonderes *Verzeichnis* (englisch: directory), in dem festgehalten ist, in welchen Spuren und Sektoren die aufgezeichneten Daten stehen und wohin neue Aufzeichnungen

kommen können. Alles, was man zunächst zum Umgang mit Disketten benötigt, ist dieser Dateiname.

Dateinamen können die verschiedenste Form haben. In der Regel umfassen sie sechs bis acht Buchstaben als eigentlichen Namen und – davon meist durch einen Punkt abgetrennt – eine Gruppenbezeichnung aus drei oder mehr Buchstaben. In manchen Fällen ist man bei der Namensgestaltung ziemlich frei, manchmal muß man sich an strenge Vorschriften halten – die eine oder andere Regel gibt es aber immer, die zu beachten ist. Schlagen Sie in Ihrem Handbuch nach, was für Möglichkeiten Ihre BASIC-Version bietet.

Nun empfiehlt es sich aber nicht, in einem Programm den Dateinamen, so wie er sich auf der Diskette vorfindet, zu verwenden. Das hat mehrere Gründe. Zum einen ist es ziemlich platz- und rechenzeitaufwendig, bei jedem Diskettenzugriff den langen Namen anzugeben und verarbeiten zu müssen. Dann – und das ist wichtiger – kann es sein, daß man ein Programm universell auslegen möchte. Die Daten unserer Lagerdatei können auf der einen Diskette einen anderen Namen als auf einer anderen haben. Z.B. können wir unser Verwaltungsprogramm für die verschiedensten Läger verwenden. Und damit da nichts durcheinanderkommt, erhält die eine Datei z.B. den Namensvorspann WE (für Waren-Eingangs-Lager) und die andere den Namen AL (für Auslieferungs-Lager). Das muß dem Programm natürlich irgendwie bekannt gemacht werden, und um nicht bei jeder Umstellung Dutzende von Dateinamen im Programm ändern zu müssen (und dann doch einen zu vergessen), faßt man das alles an einer einzigen Stelle zusammen.

Nun müssen wir uns aber auf die jeweilige Datei des öfteren im Programm beziehen können. Das läßt sich erreichen, wenn wir für den Gebrauch im Programm selbst einen *Kurznamen* einführen, meist sogar nur eine *Dateinummer*. An einer Stelle allerdings, irgendwo am Programmanfang, müssen wir dem Rechner mitteilen, welche Datei auf der Diskette mit der betreffenden Kurzbezeichnung angetroffen wird. Man bezeichnet das als das *Eröffnen*

der Datei, d.h., man macht sie durch einen besonderen Befehl dem Rechner zugänglich. Diese Anweisung besagt sinngemäß:

«ERÖFFNE die Datei mit dem Namen ⟨Dateiname⟩
ALS ⟨interner Name⟩.»

abgekürzt:

ERÖFFNE ⟨Dateiname⟩ ALS ⟨interner Name⟩

oder auf Englisch:

OPEN ⟨Dateiname⟩ AS ⟨interner Name⟩

«⟨Dateiname⟩» und «⟨interner Name⟩» müssen durch die in der betreffenden BASIC-Version gültigen Bezeichnungen ersetzt werden.

Oft braucht man noch mehr Information zum Eröffnen einer Datei. Z.B. will der Rechner wissen, ob die Aufzeichnungen in der Datei *gelesen* (wiedergewonnen) oder *geschrieben* (abgespeichert) werden sollen. Das dient außer der Organisation des Diskettenzugriffs auch als Sicherheitsmaßnahme: In eine Datei, die ausdrücklich nur zum Lesen eröffnet ist, darf nichts mehr geschrieben werden, was ein unabsichtliches Löschen oder Verändern von Aufzeichnungen in der Datei verhindert.

Man erweitert also die Eröffnungsanweisung um diese *Modusangabe* Lesen oder Schreiben:

ERÖFFNE ⟨Dateiname⟩ ZUM ⟨Modus⟩
ALS ⟨interner Name⟩

In Englisch lautet das:

OPEN ⟨Dateiname⟩ FOR ⟨Modus⟩ AS
⟨interner Name⟩

Der interne Name wird oft als Nummer angegeben, da Zahlen in BASIC einfacher gehandhabt werden können als Zeichenketten. (Dabei ist meist ein bestimmter Bereich – z.B. von 0 bis 7 – vorgeschrieben.) Nummern werden im Englischen in der Regel

durch ein besonderes Symbol, ein *Doppelkreuz* (#) gekennzeichnet. In diesem speziellen Fall bekommt die Eröffnungsanweisung die Form:

OPEN ⟨Dateiname⟩ FOR ⟨Modus⟩ AS #⟨Nummer⟩

Es gibt aber auch Kurzformen für die Eröffnungsanweisung. Sie sind meist leicht zu erkennen, wenn man weiß, welche Bestandteile überhaupt gebraucht werden. Z.B. verwendet man in BASIC-80 von Microsoft folgende Form:

OPEN «⟨Modus⟩»,⟨Nummer⟩,«⟨Dateiname⟩»

Es ist leicht zu erkennen, was gemeint ist. Der Modus wird meist als Abkürzung angegeben, zum Lesen z.B. mit «R» (von «read») oder mit «I» (von «input») und zum Schreiben z.B. mit «W» (von «write») oder mit «O» (von «output»). Der interne Name ist in der Regel eine Nummer oder eine deutlich erkennbare Kurzbezeichnung. Im Zweifelsfall muß man nachsehen, was für eine Angabe in den Anweisungen zum Lesen aus der oder Schreiben in die Datei steht. Und der Dateiname schließlich ist meist an seiner Länge erkennbar. Dieses Grundformat wird oft durch allerhand Ergänzungen erweitert, die man aber mit etwas Praxis (und Kenntnis des eigenen Systems) relativ einfach entschlüsseln kann.

Das Eröffnen einer Datei hat nebenbei bemerkt noch eine wichtige Organisationsfunktion. Der Rechner überträgt den zugehörigen Ausschnitt aus dem Diskettenverzeichnis in den internen Speicher des Rechners. Dort steht er für Bezüge zur Verfügung, wird beim Schreiben von Daten passend ergänzt und muß am Ende des Programms aber – falls die Datei verändert wurde – wieder auf die Diskette zurückgeschrieben werden. Vergißt man nämlich, das Verzeichnis auch auf der Diskette zu ergänzen, dann kann der Rechner beim nächsten Versuch, die betreffende Datei zu eröffnen, auch nur die alte Information aus dem Verzeichnis herausholen – und damit sind die vorher neu aufgezeichneten Daten praktisch verloren. Der Rechner weiß nicht, wo

sie stehen, ja, er hat sogar keine Ahnung, daß sie überhaupt da sind. Denn das Verzeichnis ist für ihn die einzige Bezugsquelle für die auf der Diskette festgehaltene Information.

Diesen Vorgang des Zurückschreibens der Verzeichnisinformation für die betreffende Datei auf die Diskette bezeichnet man als *Schließen* der Datei. Das geschieht durch eine Anweisung, die befiehlt:

«SCHLIESSE die durch den ⟨internen Namen⟩ gekennzeichnete Datei!»

kurz:

SCHLIESSE ⟨interner Name⟩

und auf Englisch:

CLOSE ⟨interner Name⟩

Dabei hat der interne Name meist die Form einer Nummer, so daß man oft die Sonderform

CLOSE #⟨Nummer⟩

oder auch nur

CLOSE ⟨Nummer⟩

findet.

Durch Eröffnen und Schließen wird eine Datei dem Rechner im Programm zur Verfügung gestellt bzw. wieder weggenommen. Damit aber ist die Information auf der Diskette noch in keiner Weise erreicht. Man muß die dort festgehaltenen Daten wenigstens noch lesen bzw. neue Information dorthin schreiben können. In der Grundform ist das nichts Neues für uns. Viele BASIC-Versionen lesen etwas aus der Datei mit Hilfe einer INPUT-Anweisung und schreiben Daten mit Hilfe einer PRINT-Anweisung in eine Datei ein. Man muß nur noch angeben, welche Datei man meint, und dazu wiederum dient der interne Dateiname. Um einer Variablen X den in der Datei ⟨interner Name⟩ festgehaltenen Wert zuzuweisen, befiehlt man in der Regel:

INPUT ⟨interner Name⟩,X

bzw. in einigen Versionen auch

READ ⟨interner Name⟩,X

Und zum Ausschreiben des Werts einer Variablen X in die Datei ⟨interner Name⟩ verwendet man die Anweisung:

PRINT ⟨interner Name⟩,X

Dabei macht es in der Regel keinen Unterschied, ob es sich um eine numerische oder eine Zeichenkettenvariable handelt. (Man darf nur das Dollarzeichen bei Zeichenkettenvariablen nicht vergessen.)

In den meisten BASIC-Versionen, welche die Dateien mit Nummern bezeichnen, sind die Datenzugriffe im Unterschied zu Konsolenoperationen (also z.B. Lesen von der Tastatur und Schreiben zum Bildschirm) leicht durch das als Nummernzeichen verwendete Doppelkreuz erkennbar. Man hat so z.B. die Anweisungen

INPUT #⟨Nummer⟩,X

und

PRINT #⟨Nummer⟩,X

zur Verfügung. (Im BASIC-80 von Microsoft ist das Doppelkreuz Bestandteil des Befehlsworts selbst, so daß hier zwischen dem Befehlsnamen und dem Doppelkreuz kein Leerschritt stehen darf:

INPUT# ⟨Nummer⟩,X

und

PRINT# ⟨Nummer⟩,X

Das macht die Befehle eindeutiger von Konsolenoperationen unterscheidbar.)

Bei der Besprechung der Ausgabe auf einen Drucker haben wir darüber hinaus einen verallgemeinerten Fall kennengelernt. Dort werden den betreffenden Ein- und Ausgabe*geräten* ebenfalls Nummern zugeordnet. Man spricht hier von *logischen* Einheiten (weil sie im logischen Fortgang des Programms eine Rolle spielen). Diesen logischen Einheiten werden durch besondere Vereinbarungen wie z.B. bei unserer Dateieröffnung *physische Einheiten* zugewiesen, d.h. diejenigen Geräte oder Dateien, über die die Ein- oder Ausgabeoperationen tatsächlich (eben physisch) abgewickelt werden. Man unterscheidet vom Programm her dann nicht mehr zwischen den verschiedenen Geräten. Es ist einem PRINT-Befehl hier prinzipiell egal, ob er etwas an die Konsole, an einen Drucker oder an eine Datei weitergibt. Er kennt nur ein numeriertes Datenziel, und dorthin überträgt er seine Daten. Alles andere, die eigentliche Datenausgabe auf die zugeordnete physische Einheit, ist dann Aufgabe des Computers selbst. Richtig genutzt, kann man damit beträchtlich flexibel programmieren.

Wichtig ist noch die Reihenfolge, in welcher die Daten aus einer Datei ausgelesen bzw. in sie eingeschrieben werden. Beim Auslesen der Daten mit Hilfe des INPUT-Befehls beginnt man in der Regel ganz am Dateianfang mit der ersten dort stehenden Information. Sie wird der angegebenen Variablen zugewiesen, und der Rechner merkt sich die Stelle, von der die Daten gelesen wurden. Beim nächsten INPUT-Befehl sind die darauffolgenden Daten an der Reihe und so weiter bis zum Dateiende. Dieses Verhalten entspricht im wesentlichen dem beim READ-Befehl zum Auslesen der im Programm in DATA-Anweisungen festgehaltenen Informationen.

Entsprechendes geschieht beim Schreiben in die Datei. Wenn mit dem OPEN-Befehl eine Datei zum Schreibzugriff eröffnet wird, erzeugt der Rechner sie in der Regel ganz neu auf der Diskette. Bereits unter diesem Namen vorhandene Dateien werden dabei gelöscht. Mit anderen Worten: Die durch die Anweisung

OPEN ⟨Dateiname⟩ FOR ⟨Output⟩ AS #⟨Nummer⟩

eröffnete Datei ist zunächst *leer,* sie trägt keinerlei Information. Erst mit der ersten PRINT-Anweisung für diese Datei wird etwas in sie hineingeschrieben. Und die Daten der nachfolgenden PRINT-Anweisungen folgen einander lückenlos in der Datei, wobei es wieder keine Rolle spielt, ob es sich um numerische oder alphanumerische Daten handelt.

In diesem Zusammenhang ist in einigen BASIC-Versionen noch auf etwas anderes zu achten. Der INPUT-Befehl behandelt die Diskettendaten oft so, als ob sie Zeichen für Zeichen von der Eingabetastatur kommen würden. D.h., es wird nicht zwischen numerischen und alphanumerischen Daten unterschieden. Alles Aufgezeichnete ist in der Datei lückenlos aneinandergefügt. Und alles ist im ASCII-Kode festgehalten. Der Befehl

```
PRINT #1,"HALLO",123,"Wie gehts?"
```

wird in diesen Fällen als

```
HALLO123Wie gehts?
```

gespeichert. Das macht Schwierigkeiten beim Auslesen der Daten. Was soll beispielsweise bei

```
INPUT #1,A$
```

in die Variable A$ übernommen werden? Ohne besondere Maßnahmen wird BASIC versuchen, hier so viel wie möglich einzulesen, also A$ den Wert «HALLO 123 Wie gehts» zu geben. Dem läßt sich begegnen, wenn man die aufgezeichneten Daten passend *begrenzt.* Als solches Begrenzerzeichen dient in der Regel das Komma. D.h., befiehlt man

```
PRINT #1,"HALLO",",",123,",","Wie gehts?"
```

dann erhält man als Dateiaufzeichnung

```
HALLO,123,Wie gehts?
```

womit der Variablen A$ im Befehl

```
INPUT #1,A$
```

dann eindeutig der Wert «HALLO» zugewiesen wird.

Damit wollen wir es mit unserer Diskettendateibesprechung genug sein lassen. Es gäbe zwar noch sehr viel mehr zu diesem Thema zu sagen, doch gehört das in den Bereich gehobeneren BASIC-Programmierens und würde den Rahmen dieser Einführung hier sprengen. Nur soviel als Hinweis:

Man kann in vielen BASIC-Versionen auch die Tatsache ausnutzen, daß man *unmittelbar* (englisch: random) auf die jeweiligen Daten zugreifen kann. Man muß nur die betreffende Sektornummer kennen. Liegt nun die *Länge einer Aufzeichnung* in die Datei fest und ist bekannt, die *wievielte* Aufzeichnung gebraucht wird, dann kann der Rechner daraus leicht die zugehörige Sektornummer(n) berechnen und gezielt diese Daten lesen bzw. dorthin in die Datei schreiben. Dieser sogenannte *unmittelbare Zugriff* (englisch: random access) beschleunigt nicht nur die Datenverarbeitung, sondern gestattet auch das Schreiben in eine Datei, ohne daß sie zuvor gelöscht werden muß. Es muß allerdings beim Eröffnen der Datei angegeben werden, ob ein unmittelbarer Zugriff gewünscht wird. Der Schreibbefehl wird bei unmittelbarem Zugriff zumeist durch das Befehlswort

PUT

und der Lesezugriff durch

GET

bewirkt. Es gibt aber auch Sonderformen von READ, INPUT oder PRINT für diesen Zweck. Genaueres müssen Sie – wie zum Umgang mit Diskettendateien überhaupt – Ihrem Handbuch und weiterführender Literatur entnehmen.

3.17 Hilfsprogramme: Parameterauswahl, Beziehung zwischen Groß- und Kleinbuchstaben im ASCII-Kode

Damit bleiben im wesentlichen nur noch zwei Moduln festzulegen: die Anforderung der Artikelbezeichnung und das Einholen einer Ja/Nein-Bestätigung für die Eingabe oder Änderung von Dateieinträgen. Beides sind im Grunde ganz einfache Operationen, die im Kern jeweils eine Eingabeanforderung umfassen. Das ließe sich durch einen einfachen INPUT-Befehl bereits in den Moduln erledigen, in denen die Operation benötigt wird, wenn da nicht noch ein paar Zusatzaufgaben wären.

Bei der Übernahme der Artikelbezeichnung z.B. müssen wir darauf achten, daß die Länge des Artikelnamens stimmt, und daß die Angaben in Großbuchstaben vorliegen. Die *maximale* Länge des Namens haben wir bereits auf 15 Zeichen festgelegt. Wir müssen jedoch noch entscheiden, ob *jeder* Name 15 Zeichen lang sein soll. Das würde bereits bei der Eingabe ein Auffüllen mit Leerzeichen erfordern, falls ein kürzerer Wert eingegeben wurde. Erforderlich ist das in den Fällen, in denen das Feld, das den Namen festhalten soll, aus Elementen mit fester Länge besteht, also im Fall von «HP/Northstar»-BASIC-Versionen, oder wenn der Name in einem numerischen Feld festgehalten werden muß. Ist dies allerdings nicht notwendig, dann können wir Suchzeit und Speicherplatz einsparen, wenn wir die Namen unterschiedlich lang machen. Außerdem läßt sich so das Druckbild beim Melden eines Eintrags leichter gestalten. Da wir hier eine «DEC/Microsoft»-BASIC-Version voraussetzen, die Felder mit Zeichenketten als Grundeinheit beherrscht, bleiben wir bei variablen Namenslängen.

Die Ja/Nein-Bestätigung erfordert die Untersuchung der Eingabe auf «Ja» oder «Nein» hin. Um auch hier die Arbeit so einfach und narrensicher wie möglich zu gestalten, sollte die Eingabe vor der Untersuchung in Großschreibung umgewandelt werden.

Wir sollten ferner den Programmodul so auslegen, daß nur aufgrund einer eindeutigen Entscheidung für «Ja» oder für «Nein» zum aufrufenden Programmteil zurückgekehrt wird. Bei jeder anderen Antwort sollte bereits hier eine Neueingabe etwa mit folgender Meldung

Wie bitte (Ja oder Nein)?

angefordert werden.

Wir müssen uns demnach noch Gedanken darüber machen, wie man am sinnvollsten die Eingabe in Großschreibung umwandelt.

Diese Moduln sind allgemein verwendbar, wenn man sie passend auslegt. D.h., wir können sie als *Hilfsprogramme* (englisch: utilities) in einer eigenen Bibliothek zusammenfassen und je nach Bedarf und Möglichkeiten in andere Programme einbauen. Dazu programmiert man sie am besten mit möglichst hohen Zeichennummern, was gewährleisten soll, daß sie mit anderen Programmeinheiten nicht in Konflikt geraten.

Reservieren Sie sich unter den in Ihrer BASIC-Version möglichen Zeilennummern einen solchen Bereich. Wenn z.B. die höchstmögliche Zeilennummer 32 767 beträgt, dann dürfte es sinnvoll sein, den Bereich ab Zeile 20 000 für Hilfsprogramme zu reservieren. Haben Sie nur einen Bereich bis 9999 zur Verfügung, dann wählen Sie als Untergrenze für Ihre Hilfsprogramme sinnvollerweise die Zeilennummer 8000 oder 9000, je nachdem, wie lang Ihre Anwendungsprogramme werden und wie viele Hilfsprogramme Sie vorsehen wollen. Natürlich müssen Sie beim Anlegen einer Hilfsprogrammbibliothek darauf achten, daß sich die Programme dort nicht selbst wieder überschneiden. Am besten führt man über den jeweils belegten Platz genau Buch und schließt jedes neue Hilfsprogramm an das Vorhergehende an.

Da haben wir zunächst die Anforderung einer Artikelbezeichnung. Wir können das Problem etwas verallgemeinern, indem wir das Programm so auslegen, daß die Maximallänge der betreffenden Zeichenkette als Parameter mit übergeben werden muß und

daß man zwischen Auffüllen und Nichtauffüllen mit Leerzeichen wählen kann. Das Programm selbst gibt als Parameter die übernommene Zeichenkette zurück.

Für diese Parameter legen wir am besten eigene Variable fest, die wir möglichst so auswählen sollten, daß sie sich nicht mit anderen überschneiden können. Das ist zwar, wie wir bereits besprochen haben, in BASIC fast unmöglich, doch können wir versuchen, nur solche Variable festzulegen, die sehr selten für andere Zwecke gebraucht werden. Da wir zur Variablenbezeichnung nur Großbuchstaben, höchstens gefolgt von einer Ziffer, zur Verfügung haben, sollten wir für die Hilfsprogramme einen bestimmten Bereich reservieren:

Die für Hilfsprogramme notwendigen Parameter sind immer mit einer Ziffer im Bereich von 6 bis 9 zu versehen.

Dieser Bereich sollte tunlichst von den anderen Programmteilen nicht verwendet werden. Für sie bleiben als zweiter Variablenteil die Ziffern 0 bis 5 übrig, was für die meisten Probleme ausreichen dürfte. (Wenn Sie mehr Möglichkeiten zur Variablenbildung zur Verfügung haben, dann sollten Sie diese so gut ausnutzen, wie nur möglich. Auf alle Fälle empfiehlt es sich aber, einen besonderen Bereich von Variablennamen ausdrücklich für Hilfsprogramme zu reservieren. Das erleichtert die Programmierarbeit beträchtlich, insbesondere was die Fehlersuche anbetrifft.)

Bei den ersten Hilfsprogrammen hat man natürlich noch die meiste Freiheit zur Variablenbenennung. Wählen wir für unseren Fall folgende Werte:

E6$ – Zeichenkette, in der die Eingabe steht,
L6 – Maximallänge der Eingabe,
A6 – Auffüllflagge; sie möge den Wert Null haben, wenn nicht aufgefüllt werden soll, ansonsten irgendeinen Wert ungleich Null.

Damit müssen folgende Aufgaben erledigt werden:

```
Artikelbezeichnung anfordern:
   Eingabe von der Tastatur in E6$ übernehmen,
   Wenn E6$ länger als L6 ist, dann:
      E6$ auf L6 Zeichen kürzen.
   E6$ in Großschreibung umformen,
   Wenn A6<>0, dann:
      Bis Länge(E6$)=L6:
         Leerschritt rechts an E6$ anfügen.
      Schleifenende (Länge(E6$)=L6).
Ende.
```

Da dieses Hilfsprogramm von den verschiedensten Programmteilen her aufgerufen werden soll, verzichten wir auf die ausdrückliche Ausgabe einer Meldung. Eine solche muß vom aufrufenden Programmteil her bereitgestellt sein, z.B. in folgender Form:

```
PRINT «Welchen Artikel aufsuchen»;
GOSUB (Artikelbezeichnung anfordern)
```

Vergessen Sie dabei den Strichpunkt nach dem PRINT-Argument nicht!

Das Einholen einer Ja/Nein-Bestätigung ist noch einfacher. Hier brauchen keine Parameter von außen vorgegeben werden, dagegen muß man das Ergebnis «Ja» oder «Nein» zurückliefern, was am besten in einer numerischen Variablen erfolgt. Legen wir dazu fest:

J6 – Ja/Nein-Flagge; enthält Null bei einer «Nein»-Antwort, einen Wert ungleich Null bei einer «Ja»-Antwort.

Außerdem muß die Antwort selbst übernommen werden. Verwenden wir dazu die Zeichenkette:

J6$ – Übernahme einer Ja/Nein-Antwort

Dann sind folgende Aufgaben zu erledigen:

```
Bestätigung einholen:
   J6 = 1, / * «Ja»-Antwort annehmen * /
   Bis Antwort «JA» oder «NEIN»:
      Ja/Nein-Entscheidung in J6$ anfordern,
      J6$ in Großschreibung umwandeln,
      Wenn J6$ = «NEIN», dann:
         J6 = 0.
      Sonst, wenn J6$ < > «JA», dann:
         «Wie bitte».
   Schleifenende (Antwort «JA» oder «NEIN»).
Ende.
```

Beachten sie, daß wir hier die Variable J6 bereits beim Einstieg in den Modul auf den Wert für «JA» initialisieren (wir hätten auch «Nein» annehmen können). Dadurch sparen wir uns einen Testschritt ein, denn sonst hätten wir formulieren müssen:

```
      Wenn J6$ = «NEIN», dann:
         J6 = 0.
      Sonst, wenn J6$ = « JA», dann:
         J6 = 1.
      Sonst:
        «Wie bitte».
```

Wir tragen damit der Tatsache Rechnung, daß BASIC in der Regel keine ausdrückliche SONST-Entscheidung zuläßt. Dieses Vorgehen empfiehlt sich in allen Fällen, in denen durch die Entscheidung ein bestimmter Wert für eine Variable festgelegt werden muß. Man nimmt einfach einen Fall von vornherein an und testet dann nur noch, ob vielleicht nicht doch der andere Fall vorliegt und der Variablenwert geändert werden muß. Dabei empfiehlt es sich, jeweils den wahrscheinlichsten Wert von vornherein anzunehmen, da dadurch etwas Rechenzeit eingespart werden kann.

Bleibt noch das Problem der Umwandlung einer Zeichenkette

in Großschreibung zu lösen. Liegen die Zeichen im ASCII-Kode vor, dann ist das allerdings nicht schwierig. Dort belegen nämlich die Großbuchstaben die Kodewerte

$$A = 65, B = 66, \ldots, Z = 90$$

und die Kleinbuchstaben die Werte

$$a = 97, b = 98, \ldots, z = 122$$

In beiden Fällen folgen die Buchstaben lückenlos aufeinander in alphabetischer Reihenfolge. Stellt man die beiden Alphabete nebeneinander, dann ergibt sich so

Buchstabe	Wert	Buchstabe	Wert	Differenz
A	65	a	97	32
B	66	b	98	32
C	67	c	99	32
...		...		...
X	88	x	120	32
Y	89	y	121	32
Z	90	z	122	32

In allen Fällen unterscheidet sich der ASCII-Kodewert eines Großbuchstabens von einem eines Kleinbuchstabens um die Differenz 32. D.h. um einen Kleinbuchstaben in einen Großbuchstaben umzuwandeln, braucht man nur von seinem Kodewert die Differenz 32 abzuziehen. Das wiederum macht mit Hilfe von ASC und CHR$ keine Probleme. Man erhält einen Großbuchstaben G$ aus einem Kleinbuchstaben K$ ganz einfach durch

```
G$ = CHR$(ASC(K$)-32)
```

Wir müssen nur noch entscheiden, ob ein Zeichen einen Kleinbuchstaben darstellt oder nicht. Hier kommt uns eine Besonderheit von BASIC sehr zupaß: Man kann nämlich Zeichenketten unmittelbar nach ihrem Kodewert vergleichen. Ein kleiner Test macht das deutlich:

```
>NEW
>10 INPUT "Wie lautet das erste Zeichen";A$
>20 INPUT "Womit soll es verglichen werden";B$
>30 IF A$<B$ THEN PRINT "Das erste Zeichen ist kleiner."
>40 IF A$>B$ THEN PRINT "Das zweite Zeichen ist kleiner."
>50 IF A$=B$ THEN PRINT "Beide Zeichen sind gleich."
>60 GOTO 10
>RUN
Wie lautet das erste Zeichen? A
Womit soll es verglichen werden? B
Das erste Zeichen ist kleiner.
Wie lautet das erste Zeichen? a
Womit soll es verglichen werden? A
Das zweite Zeichen ist kleiner.
Wie lautet das erste Zeichen? A
Womit soll es verglichen werden? A
Beide Zeichen sind gleich.
```

Man sieht: Der Vergleich richtet sich nach dem *Kodewert* der zu vergleichenden Zeichen. Auf diese Weise können auch Ziffern mit Buchstaben, ja sogar Satzzeichen miteinander verglichen werden. Man kann noch mehr: Ganze Zeichenketten können miteinander verglichen werden. Z.B. so:

```
Wie lautet das erste Zeichen? ABCD
Womit soll es verglichen werden? BCDE
Das erste Zeichen ist kleiner.
```

Wir wollen hier nicht weiter darauf eingehen. Experimentieren Sie aber ein wenig damit. Versuchen Sie vor allem, herauszubekommen, unter welchen Bedingungen eine Zeichenkette für kleiner als eine andere eingestuft wird. Untersuchen Sie dazu verschiedene Zeichenkombinationen und verschieden lange Ketten. Notieren Sie sich das Vergleichsergebnis für jeden Schritt, bis klar ist, was geschieht. Dann erst sollten Sie in Ihrem Handbuch nachschlagen.

Für unser Problem der Umwandlung einer Zeichenkette in Großschreibung genügt es jedenfalls, einzelne Zeichenwerte miteinander vergleichen zu können. Dazu müssen wir jedes einzelne Zeichen der als Parameter übergebenen Originalkette untersu-

chen. Ist es ein Kleinbuchstabe, dann wird es in einen Großbuchstaben umgewandelt, ansonsten bleibt es unverändert.

Diese Umwandlung stellt uns allerdings vor ein anderes Problem. Wir können in «DEC/Microsoft»-Versionen einem Ausschnitt aus einer Zeichenkette nicht unmittelbar einen Wert zuweisen:

```
>MID$(N$,3,1) = "A"
?SN ERROR
```

Der Fehler ist hier darin zu suchen, daß MID$ eine *Funktion* ist, die zwar die verlangte Teilkette zurückliefert, dies aber als völlig selbständigen Wert. D.h., im Gegensatz zu «HP/Northstar»-Versionen wird nicht der Platz in der betreffenden Zeichenkette bezeichnet, sondern der dort vorgefundene Wert an eine *andere* Stelle im Speicher kopiert. Ein Zurückkopieren dieses Werts in die Originalkette ist in fast allen Versionen nicht vorgesehen. (Es gibt ein paar Ausnahmen. Wenn die Syntax-Fehlermeldung ausgeblieben ist, sollten Sie sich einmal N$ ansehen und eventuell noch ein wenig damit experimentieren.)

Aus dieser Situation gibt es nur einen Ausweg: Wir müssen die zurückzugebende Zeichenkette ganz neu konstruieren. D.h., wir müssen Sie Zeichen für Zeichen aus den in der Originalkette untersuchten Werten mittels Konkatenation aufbauen.

Damit müssen wir der universellen Anwendbarkeit dieses Programms wegen drei Variable festlegen. Nennen wir die eingehende Zeichenkette K6$ (da sie Kleinbuchstaben enthalten kann), das Ergebnis G6$ (da diese Kette nur Großbuchstaben enthält), und schließlich den Zähler der Zeichenposition Z6:

G6$ – Ergebnis bei der Großbuchstabenumwandlung
K6$ – Ein/Ausgabeparameter bei der Großbuchstabenumwandlung
Z6 – Zeichenposition bei der Großbuchstabenumwandlung

Damit können wir folgende Struktur skizzieren:

Großbuchstabenumwandlung:
    G6$ löschen,
    Wenn K6$ nicht leer ist, dann:
        Für Z6 = 1 bis Länge (K6$):
            Wenn Zeichen (Z6) > = «a» oder < = «z», dann:
                G6$ ← G6$ + Großbuchstabe(Zeichen(Z6)).
            Sonst:
                G6$ ← G6$ + Zeichen(Z6).
        Schleifenende (Z6 > Länge(K6$)).
Ende.

Beachtenswert ist hier, daß der übergebene Parameter K6$ erst daraufhin geprüft wird, daß er nicht leer ist. Sonst können Fehlermeldungen auftreten. Dieser ausführliche Test kann dann entfallen, wenn Ihre BASIC-Version eine Schleife *nicht* mindestens einmal abarbeitet. In diesem Fall führt der Computer diesen Test automatisch durch, da dann die Länge von K$ gleich Null und somit die Bedingung

```
FOR Z6=1 TO LEN(K6$)
```

nicht erfüllt ist.

3.18 Umsetzung in BASIC

Übersicht

Damit sind alle wichtigen Moduln für unser Lagerverwaltungsprogramm entworfen. Sie finden die Strukturbeschreibungen in Anhang 1 noch einmal zusammengefaßt. Bild 11 zeigt die endgültige Programmhierarchie, die sich aus diesem Ansatz ergibt.

Wir haben gegenüber dem Ansatz von Bild 10 ein paar Änderungen vorgenommen, durch die die Struktur des Programms deutlicher hervortritt. So haben wir an der Spitze einen allgemeinen Steuerungsmodul vorgesehen, der die gesamte Systemarbeit

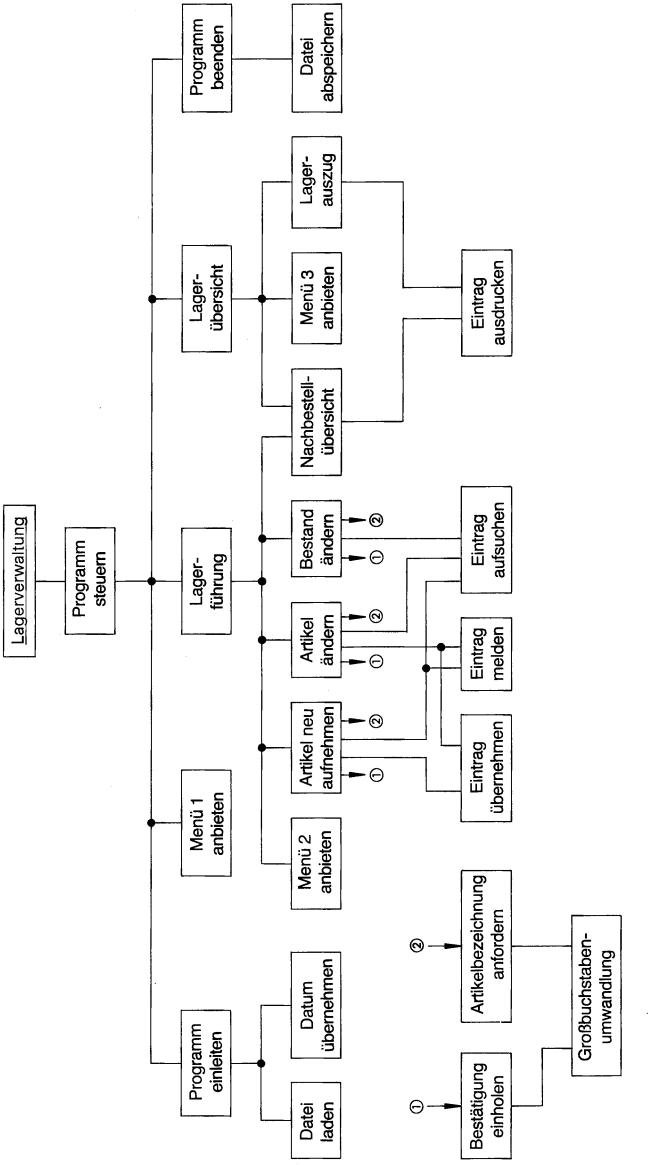


Bild 11 Endgültiges Hierarchiediagramm des Lagerverwaltungsprogramms

koordinieren soll. Insbesondere ruft dieser Modul ausdrücklich die Programmeinleitung als Unterprogramm auf, so daß das Laden der Datei (wenn möglich) und das Übernehmen des gültigen Tagesdatums als eigene in sich abgeschlossene Einheit abgefaßt werden können. Das macht das Programm übersichtlicher und leichter zu warten.

Des weiteren haben wir die verschiedenen Menüs jeweils als Modul für sich aus der übergeordneten Einheit ausgegliedert. Das ist nicht unbedingt notwendig, doch sind so die auszudruckenden Texte leichter von den sonstigen Befehlen im zugeordneten Modul zu unterscheiden, was außerdem das Experimentieren mit verschiedenen Texten erleichtert.

Beachten Sie schließlich, daß wir die beiden Moduln «Artikelbezeichnung anfordern» und «Bestätigung einholen» vom übrigen Diagramm etwas abgesetzt haben und durch Nummern auf sie Bezug nehmen. Durch diese Maßnahme erspart man sich eine Reihe von Überkreuzungen der Verbindungen zwischen den Moduln, was die Übersichtlichkeit beträchtlich steigert. Bei umfangreichen Programmen wird man so z.B. auf andere Blätter verweisen, auf denen die Darstellung fortgesetzt wird. Wichtig ist, die Darstellung so zu halten, daß sie möglichst leicht erfaßt werden kann.

Die für unsere Lagerverwaltung notwendigen Programmmoduln sind demnach in fünf Ebenen gegliedert. Ganz an der Spitze der Hierarchie steht die Programmsteuerung, gefolgt von den Moduln «Programm einleiten», Menü 1 anbieten», «Lagerführung», «Lagerübersicht» und «Programm beenden». Die dritte Ebene umfaßt die von diesen unmittelbar aufgerufenen Moduln «Datei laden» bis «Datei abspeichern». In der vierten Ebene befinden sich die Moduln «Bestätigung einholen», «Artikelbezeichnung anfordern», die ihrerseits auch den Modul fünfter Ebene «Klein/Groß-Umwandlung» zurückgreifen. Die andern vier Moduln der vierten Ebene, die sich mit der unmittelbaren Handhabung der Einträge beschäftigen, sind bereits in sich abgeschlossen, d.h., sie brauchen auf keine tieferere Ebene zurückgreifen.

In diesem Zusammenhang ist ein Punkt beachtenswert: Nach unserer bisherigen Planung hätten wir in der vierten Ebene von den Moduln zur Eintragshandhabung nur «Eintrag übernehmen», «Eintrag aufsuchen» und «Eintrag ausdrucken» vorliegen. Ein Durchdenken der so entstandenen Struktur, das Erfassen des Programms im Gesamtzusammenhang, förderte dann ein Problem zutage, das bis dahin übersehen wurde, die Tatsache nämlich, daß das Ausdrucken eines Eintrags im Rahmen der Nachbestellübersicht und des Lagerauszugs sich normalerweise von dem Melden des Eintrags auf dem Bildschirm im Verlauf der Neuaufnahme oder Änderung eines Artikels unterscheidet. Die Aufgaben sind verschieden: Was man beim Ausdrucken der Übersichten sinnvollerweise in einer Zeile anordnet, um die Tabellenstruktur zu bewahren, geschieht hier besser mit passenden Meldungen *untereinander,* was z.B. so aussehen kann:

```
Artikel: Hammer
am Lager: 123
Mindestmenge: 30
Einzelpreis (DM): 7.80
Gesamtwert (DM): 959.40
Zuletzt veraendert am: 12.07.1983
```

Die zugehörige Strukturbeschreibung ist bereits in Anhang 1 aufgenommen worden.

Dieser Fall ist ein Beispiel für einen Vorgang, der in der Praxis bei umfangreichen Problemen häufig auftritt. Die verschiedenen Strukturierungs- und Entwurfswerkzeuge ergänzen einander nicht nur, sie wirken auch aufeinander zurück. Oft werden logische Fehler, Widersprüche oder Auslassungen im Entwurf erst sichtbar, wenn man sich den Ansatz unter einem anderen Blickwinkel ansieht. Wir mußten ja auch unseren ursprünglichen Hierarchieentwurf aufgrund von Überlegungen bei der Erstellung der Strukturbeschreibungen ändern. Es gibt – außer bei trivialen Problemen – keinen reibungslosen Fortgang in einem Schritt auf das Ziel zu. Man muß immer wieder die Ergebnisse eines Schritts an denen der anderen messen, muß manchmal

sogar wieder bis fast an den Anfang zurück, weil sich irgendwo herausstellt, daß eine winzige Kleinigkeit übersehen wurde, die aber die ganze Arbeit in die Sackgasse getrieben hat. Das ist unangenehm, sicher, aber es ist besser, man entdeckt das vor Kodieren des jeweiligen Problems, als später ratlos vor einem Programm zu sitzen, das partout nicht das machen will, was es eigentlich doch ohne Probleme machen sollte. Auch darin, im kritischen und selbstkritischen Durchdenken aller Arbeitsschritte – ohne sich dabei in alle möglichen Seitenzweige zu verzetteln – liegt das Schöpferische beim Problemlösen mit Hilfe von Computern. Dieser Prozeß wird um so einfacher, je besser man die vorhandenen Werkzeuge beherrscht und je mehr Werkzeuge man für besondere Probleme zur Verfügung hat.

Im übrigen wurden durch die Einarbeitung dieser Änderung in die Strukturbeschreibungen auch dort ein paar Details verändert. Z.B. haben wir je eine Bestätigungsschleife in die Moduln «Artikel neu aufnehmen» und «Artikel ändern» eingefügt. Das war vorher sozusagen zwischen den Zeilen mit in der Übernahme eines neuen Eintrags enthalten. Durch diese Strukturänderung tritt die Bedeutung der Rückmeldung deutlich hervor. Die Eintragsübernahme selbst kann sich dann auf das einfache Anfordern von neuen Werten beschränken, wie es in Anhang 1 dargestellt ist.

Umformung in Großschreibung: Zur Notwendigkeit der Programmoptimierung

Damit können wir uns an die Kodierung als BASIC-Programm machen. Wir beginnen dazu mit den Moduln auf der untersten Ebene, formulieren sie – eventuell unter Zuhilfenahme eines Flußdiagramms – in BASIC-Anweisungen um, testen sie möglichst sorgfältig und gehen auf dieser Basis dann zu den Moduln der nächsthöheren Ebene über, wo sich das Spiel fortsetzt, bis das ganze Programm nach bestem Wissen und Gewissen einwandfrei fertiggestellt ist.

Beginnen wir ganz unten, also mit der:

Großbuchstabenumwandlung:
  G6$ löschen,
  Wenn K6$ nicht leer ist, dann:
    Für Z6 = 1 bis Länge (K6$):
      Wenn Zeichen (K6$,Z6) > = «a» und < = «z», dann:
        G6$ ← G6$ + Großbuchstabe
        (Zeichen (K6$,Z6)).
      Sonst:
        G6$ ← G6$ + Zeichen(K6$,Z6).
    Schleifenende (Z6 > Länge(K6$)).
Ende.

Das läßt sich (fast) unmittelbar in ein BASIC-Programm übersetzen. Da es sich um unser erstes Hilfsprogramm handelt, wählen wir als Anfang Zeile Nummer 20 000. Dort ist als erstes Titel und Beschreibung unterzubringen:

```
>NEW
>20000 REM *** Grossbuchstabenumwandlung ***
>20001 REM
>20002 REM - Wandelt die Zeichenkette K6$ in die entsprechende
>20003 REM - Zeichenkette G6$ um, die nur Grossbuchstaben ent-
>20004 REM - haelt.
>20005 REM - Variable:
>20006 REM   G6$ - umgeformte Zeichenkette
>20007 REM   K6$ - uebergebene Zeichenkette
>20008 REM   Z6  - zaehlt die bearbeiteten Zeichen
>20009 REM
```

Übersetzen wir jetzt die Strukturbeschreibung wörtlich in BASIC, dann erhalten wir:

```
>20010 G6$ = ""
>20020 IF K6$="" THEN RETURN
```

Das initialisiert die Zeichenkette für das Ergebnis und sorgt dafür, daß keine weitere Arbeit durchgeführt wird, wenn die als Parameter übergebene Zeichenkette leer ist. Dem folgt die eigentliche Übersetzungsschleife, die folgende Gestalt erhält.

```
>20030 FOR Z6=1 TO LEN(K6$)
>20040 IF MID$(K6$,Z6,1)>="a" AND MID$(K6$,Z6,1)<="z"
       THEN G6$ = G6$+CHR$(ASC(MID$(K6$,Z6,1))-32) :
       GOTO 20060
>20050 G6$ = G6$+MID$(K6$,Z6,1)
>20060 NEXT Z6
>20070 RETURN
```

Das sollte eigentlich funktionieren. Aber so richtig gefallen will es nicht. Es ist ein Musterbeispiel an Umständlichkeit. So steht da z.B. viermal der Kode MID$(K6$,Z6,1). Das geht sowohl an den Speicherplatz als auch an die Abarbeitungszeit. Außerdem ist dadurch Zeile 20 040 mit 103 Zeichen für manche BASIC-Versionen zu lang geraten. Warum sollen wir hier keine Hilfsvariable einführen, z.B. Z6$? Damit würde das Programm folgende Form erhalten:

```
>20030 FOR Z6=1 TO LEN(K6$)
>20040 Z6$ = MID$(K6$,Z6,1)
>20050 IF Z6$>="a" AND Z6$<="z" THEN
       G6$ = G6$+CHR$(ASC(Z6$)-32) : GOTO 20070
>20060 G6$ = G6$+Z6$
>20070 NEXT Z6
>20080 RETURN
```

Das macht die Sache schon übersichtlicher. Man kann sie aber noch eleganter gestalten, wenn man die Entscheidung in Zeile 20 050 umkehrt. Wenn wir nämlich hier untersuchen, ob *kein* Kleinbuchstabe vorliegt, dann schaffen wir uns die Möglichkeit, die Umwandlung auf einer Zeile für sich zu formulieren. Die THEN-Bedingung schrumpft dabei auf einen einfachen Sprung um diese Umwandlung herum zusammen, die Umwandlung selbst ist als ELSE-Bedingung programmiert. Damit ergäbe sich folgendes Bild für das Programm:

```
>20000 REM *** Grossbuchstabenumwandlung ***
>20001 REM
>20002 REM - Wandelt die Zeichenkette K6$ in die entsprechende
>20003 REM - Zeichenkette G6$ um, die nur Grossbuchstaben ent-
>20004 REM - haelt.
```

```
>20005 REM - Variable:
>20006 REM   G6$ - umgeformte Zeichenkette
>20007 REM   K6$ - uebergebene Zeichenkette
>20008 REM   Z6  - zaehlt die bearbeiteten Zeichen
>20009 REM   Z6$ - gerade bearbeitetes Zeichen
>20010 REM
>20020 G6$ = ""
>20030 IF K6$="" THEN RETURN
>20040 FOR Z6=1 TO LEN(K6$)
>20050 Z6$ = MID$(K6$,Z6,1)
>20060 IF Z6$<"a" OR Z6$>"z" THEN 20080
>20070 Z6$ = CHR$(ASC(Z6$)-32)
>20080 G6$ = G6$+Z6$
>20090 NEXT Z6
>20100 RETURN
```

Wir brauchen hier zwar ein paar Zeilen mehr, doch tritt die ausgeführte Aktion jetzt viel deutlicher hervor. Beachten Sie außerdem, daß wir den Vorspann um die Variable Z6$ ergänzt haben. Dieser Vorspann ist für das Verständnis des Programms und für seine Einbindung in andere Programme unerläßlich. Er sollte prinzipiell immer enthalten:

- den Programmtitel,
- eine Kurzbeschreibung der ausgeführten Operation,
- die verwendeten Variablen und ihre Bedeutung.

Wird das sorgfältig ausgeführt, dann ist die halbe Dokumentation bereits geschaffen. Beachten Sie weiterhin, daß wir im Vorspann die Zeilen unmittelbar aufeinanderfolgend numeriert haben. Das spart Zeilennummern, hat aber den Nachteil, daß eventuell notwendige Ergänzungen nicht mehr eingebaut werden, ohne den ganzen Vorspann umzuschreiben. Sie sollten sich über die Funktion völlig im klaren sein, bevor Sie den endgültigen Vorspann abfassen. Am besten entwirft man den Text auf einem separaten Blatt Papier und fügt ihn erst dann ein, wenn das eigentliche Programm fertiggestellt ist.

Das «Feilen» am Programm, das wir oben durchgeführt haben, bezeichnet man als *Optimieren*. Um genau zu sein: Es ist nur eine von vielen möglichen Formen der Programmoptimierung.

In der Regel arbeitet man auf drei Ziele hin:

- gut lesbare Programme,
- wenig Speicherbedarf für den Programmtext und
- rasche Programmabarbeitung.

Bis zu einem gewissen Grad treffen sich diese Aufgabenstellungen. Ein gut lesbares Programm benötigt oft – da es klar durchgearbeitet sein muß – relativ wenig Speicher und wird nicht allzu langsam abgearbeitet. Wenn es jedoch darauf ankommt, ein Programm möglichst schnell zu machen oder möglichst wenig Speicher zu belegen, dann geht das auf Kosten der anderen beiden Punkte. Man muß hier immer abwägen, bis zu welchem Grad man sinnvoll gehen kann und darf. Als Faustregel sollte allerdings gelten, daß man sich zunächst bemüht, gut lesbare, sorgfältig durchstrukturierte Programme zu erstellen und dann – in dem dadurch abgesteckten Rahmen – den Programmkode selbst möglichst optimal gestaltet. Bedenken Sie immer, daß zu einem guten Programm immer auch die leichte Wartungsfähigkeit gehört und das um so mehr, je komplexer es ist.

Programmoptimierung, Flußdiagramm und Strukturbeschreibung

Wir haben die Optimierung oben unmittelbar anhand des Programmkodes durchgeführt. In einfachen Fällen wie diesem hier ist das machbar. Doch geschieht es leicht, daß man im endgültigen Programm den Wald vor lauter Bäumen nicht mehr sieht – selbst wenn man sich ein gutes Gespür für Optimierungsmöglichkeiten entwickelt hat. Es empfiehlt sich daher, andere Werkzeuge mit heranzuziehen, die den nötigen Überblick über die betrachtete Lösung verschaffen und sich mit ihrer Hilfe Hinweise auf Optimierungsmöglichkeiten schon dann zu verschaffen, wenn noch keine einzige Kodezeile vorliegt. Eines der dabei möglichen Werkzeuge ist das Flußdiagramm.

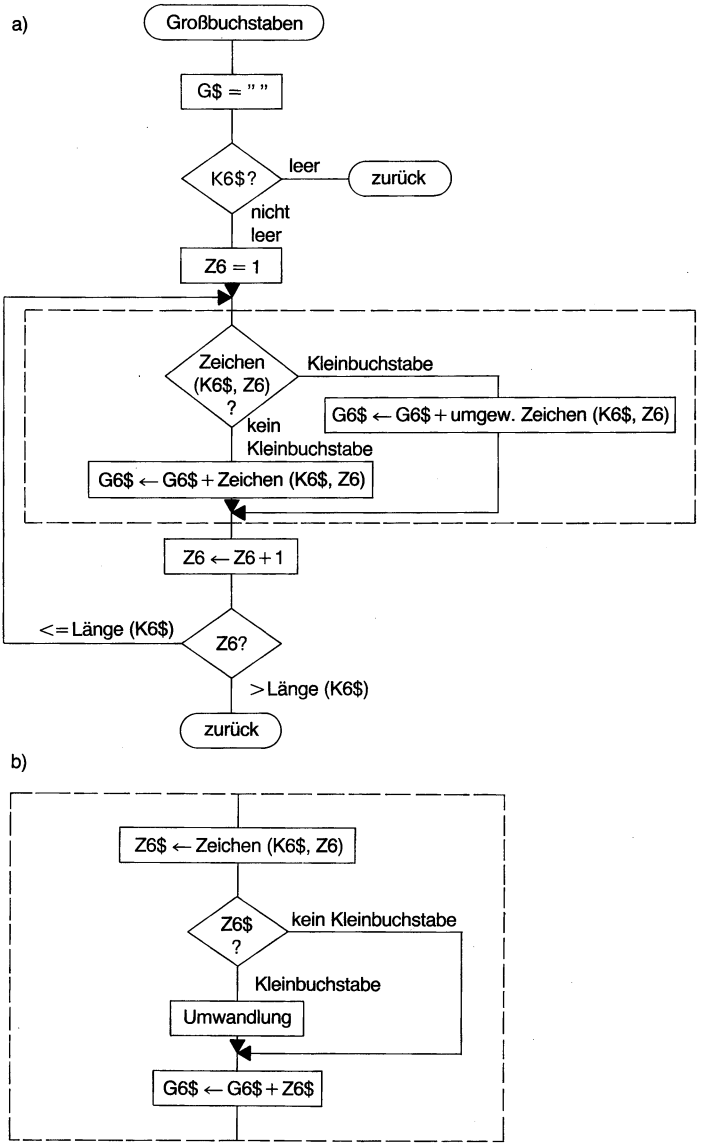


Bild 12 Flußdiagramm zur Großbuchstabenumwandlung

Wir haben in Bild 12 das Flußdiagramm zur Umwandlung in Großschreibung aufgezeichnet. Dabei stellt Teil a) die Lösung dar, wie sie sich aus der unmittelbaren Übertragung der Strukturbeschreibung ergibt. Man erkennt die Initialisierung von G6$, den Test von K6$ und schließlich die FOR-NEXT-Schleife, der die Umformung Zeichen für Zeichen stattfindet.

Die außerhalb der Schleife liegenden Aktionen folgen klar aufeinander. Hier ergibt sich — vom Flußdiagramm her — keine Notwendigkeit zum Eingriff. Anders in der Schleife selbst. Wir haben dort den Test des jeweils betrachteten Zeichens danach, ob ein Groß- oder ein Kleinbuchstabe vorliegt. Ist es ein Großbuchstabe, so wird er unmittelbar an G6$ angefügt. Andernfalls wird der umgewandelte Wert mit G6$ verbunden.

Zwei Punkte fallen in der Darstellung auf:

- Die Anbindung eines neuen Zeichens an G6$ ist in beiden Zweigen vorhanden, und
- die Aktionen sind recht komplex beschrieben.

Die erste Beobachtung wirft sofort die Frage auf, ob man die Operation, in der ein neues Zeichen an G6$ angebunden werden soll, nicht sinnvollerweise aus der Verzweigung herausnimmt. In den meisten Fällen läßt sich das machen. Wenn nicht, dann ist oft die Formulierung falsch oder ungenau. Man sollte dann die gesamte Logik dieses Programmteils sorgfältig überprüfen. In unserem Fall können wir die Zuweisung G6$ ← G6$+Zeichen(K6$,Z6) hinter die Verzweigung setzen, d.h. nach dem Punkt, in dem die beiden Zweige zusammengeführt sind, einfügen. Damit wird der eine Pfad leer, und im anderen bleibt nur die Umwandlungsoperation übrig. Eine solche Form entspricht besser den Möglichkeiten der IF-THEN-Anweisung in BASIC.

Bleibt noch die komplizierte Form «Zeichen (Z6,K$)», die *aufeinanderfolgend* immer wieder verwendet wird. Hier drängt sich ein Ausklammern dieses Prozesses und die Zuweisung an eine Hilfsvariable Z6$ schon rein grafisch auf. Diese Zuweisung erfolgt sinnvollerweise vor Eintritt in die Groß/Klein-Entscheidung.

Damit kommen wir für den Inhalt der FOR-NEXT-Schleife zu der Darstellung von Bild 12b, die dem oben von uns gefundenen Programmtext weitgehend entspricht. Die einzige Entscheidung, die beim Umsetzen in den BASIC-Kode noch vollzogen werden muß, betrifft die Frage, ob das untersuchte Zeichen daraufhin getestet werden soll, ob es ein Kleinbuchstabe oder ob es kein Kleinbuchstabe ist. Hier ist auch das kein Problem: Wir müssen den Test sinnvollerweise so auslegen, daß der THEN-Teil auf einen einfachen Sprung reduziert werden kann, d.h., wir müssen untersuchen, ob das Zeichen *kein* Kleinbuchstabe ist.

So kommen wir ganz natürlich zu der optimierten Lösung, die wir oben mit einiger Mühe erarbeiten mußten, indem wir das Flußdiagramm nach zwei Grundrichtungen hin untersucht haben:

1. Können Operationen sinnvoll aus verschiedenen Zweigen ausgeklammert werden?
2. Können IF-THEN-Entscheidungen so formuliert werden, daß der THEN-Teil möglichst wenig Aktionen erfordert?

So lohnt sich die Erstellung eines Flußdiagramms auch in vielen Fällen, in denen eine Strukturbeschreibung unmittelbar hätte übertragen werden können.

Es ist im übrigen interessant, sich diese Lösung als Strukturbeschreibung anzusehen:

```
Großbuchstabenumwandlung:
    G6$ löschen,
    Wenn K6$ nicht leer ist, dann:
        Für Z6=1 bis Länge(K6$):
            Z6$ = Zeichen(K6$,Z6),
            Wenn Z6$<«a» oder Z6$>«z», dann:
                Z6$ in Großschreibung umwandeln.
            G6$ ← G6$+Z6$.
        Schleifenende (Z6>Länge(K6$)).
    K6$ ← G6$.
Ende.
```

Wir haben hier den kritischen Teil, nämlich die Entscheidung in der Schleife, ohne weiteres entschärfen können, indem wir sie so umformulierten, daß der «Sonst»-Teil entfallen ist. Das läßt sich bei relativ einfachen Aufgaben wie dieser hier machen. In komplexeren Fällen empfiehlt sich allerdings eine ausführlichere Darstellung, da dadurch die innere Programmlogik deutlicher hervortritt. Es muß zwischen Strukturdiagramm und endgültigem Programmtext nicht unbedingt eine Eins-zu-Eins-Entsprechung vorliegen. Man sollte jedoch auch auf dieser Stufe bereits nach Optimierungen suchen, wobei die für Flußdiagramme angegebenen Fragestellungen sinngemäß lauten:

1. Kann eine in einer Einrückebene mehrfach auftretende Operation durch Ausklammern in eine äußere Ebene vereinfacht werden?
2. Können Entscheidungen so vereinfacht werden, daß der «Sonst»-Teil wegfällt?

Diese Vereinfachungen in der Strukturbeschreibung dürfen allerdings die Lesbarkeit nicht beeinträchtigen. Insbesondere gilt das für das Ausklammern von Operationen, die man sinnvollerweise oft auf einer späteren Stufe der Programmentwicklung – etwa auf Flußdiagrammebene – vornimmt. Falls dadurch der Bezug zwischen Flußdiagramm und Strukturbeschreibung schwieriger herzustellen ist, sollte eine Beschreibung dieser Änderung ausdrücklich mit in die Dokumentation mit aufgenommen werden.

Programmtest

Jeder Programmodul muß sorgfältig getestet werden. Das gilt auch dann, wenn er nur vier Zeilen umfaßt und «man doch sofort sieht, daß er richtig ist». Zumindest Tippfehler lassen sich nicht ausschließen, und auch die werden oft nur durch einen Testlauf entdeckt.

Testen kann man nun so oder so. Zum Beispiel kann es genü-

gen, wenn man sich nur eine Übersicht verschafft, wenn man sicherstellt, daß keine allzu groben Fehler vorhanden sind. Dann genügt es, sich «nach Gefühl» durchzutasten, d.h. das Programm abzuarbeiten und nachzusehen, ob die Ergebnisse mit dem übereinstimmen, was man haben wollte. So ein Vorgehen ist durchaus legitim, wenn die Alternativen übersichtlich sind oder wenn das Programm nicht auf Dauer entwickelt werden soll, d.h., nur ein- oder zweimal verwendet und dann gelöscht wird. Man gewinnt dadurch vor allem beim raschen Experimentieren mit verschiedenen Möglichkeiten ausreichende Sicherheit für die Richtigkeit des Programms.

Der Test verläuft in diesem Fall so, daß einige typische Daten oder Befehle vorgegeben werden und man an ihnen das erarbeitete Ergebnis prüft. In unserem Fall geht es um drei Fragen:

- Was geschieht, wenn K6$ beim Aufruf leer ist?
- Werden Kleinbuchstaben in Großschreibung umgewandelt?
- Bleiben andere Zeichen, insbesondere Großbuchstaben und Ziffern unberührt?

Um das nachzuprüfen, müssen wir ein kleines Testprogramm schreiben, das den Eingabeparameter in K6$ bereitstellt, das Unterprogramm zur Umwandlung in Großschreibung aufruft und dann die Ergebnisse zurückliefert.

Eine Grundregel beim Programmtest sollten Sie jedoch ganz strikt einhalten:

Testen Sie niemals ein Programm- oder Datenstück, ohne sich vorher eine Sicherheitskopie auf einem externen Speicher gemacht zu haben.

Das ist ungemein wichtig, denn es kann sein, daß irgendein versteckter Fehler vorhanden ist, der entweder Programm und/oder Daten verändert, oder der dazu zwingt, den Computer abzuschalten, weil er sich irgendwo unrettbar «aufgehängt» hat. Dann ist alles verloren, wenn Sie keine Aufzeichnung des zu testenden

Programmstücks in der Hinterhand hatten. Dazu kommt noch, daß das Testprogramm selbst oft recht umfangreich werden kann und es sehr aufwendig und fehlerintensiv ist, nach dem Testabschluß die Originalversion durch Löschen von Programmzeilen wiederherzustellen. Es gibt viele Gründe für Sicherungskopien. Machen Sie also jetzt eine Aufzeichnung auf Band oder Diskette, bevor Sie weiterarbeiten.

Wir können dann folgenden Test versuchen:

```
>100 K6$ = "" : REM - Leer-Reaktion testen
>110 CALL 20000 : REM - umwandeln
>120 PRINT "!";K6$;"!" : REM - Original veraendert?
>130 PRINT "!";G6$;"!" : REM - Ergebnis in Ordnung?
>140 K6$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijkl
     mnopqrstuvwxyz1234567890"
>150 CALL 20000 : REM - umwandeln
>160 PRINT K6$ : REM - Original veraendert?
>170 PRINT G6$ : REM - Ergebnis in Ordnung?
>180 END
```

Das Programm ist recht einfach. Zuerst untersuchen wir, was sich bei einer leeren Zeichenkette tut, dann lassen wir alle die Zeichen bearbeiten, die voraussichtlich in unseren Artikelnamen vorkommen werden. Übergebener Parameter und Ergebnis der Operation werden danach untereinander ausgedruckt. Auf diese Art und Weise lassen sich die Veränderungen mit einem Blick erkennen.

Sehen wir uns das einmal an:

```
>RUN
!!
!!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
```

Das ist genau das Ergebnis, das wir erwarten.

Oder etwa nicht? Die Frage ist, ob wir wirklich das Verhalten getestet haben, das untersucht werden soll. Bei der eigentlichen Umwandlung ist das in der Tat der Fall. Doch wie steht es mit dem

Verhalten, wenn eine leere Zeichenkette übergeben wurde? Anscheinend haben wir das getestet – doch es scheint halt nur so. Sehen wir unser Testprogramm noch einmal an, so stellt sich heraus, daß wir eigentlich gar keine Rücksicht auf einen vorgegebenen Inhalt von G6$ nehmen. In den meisten BASIC-Versionen nämlich werden die Zeichenketten beim Aufruf durch RUN erst einmal gelöscht. Und so testen wir im ersten Schritt eigentlich überhaupt nichts. Dem ist leicht abzuhelfen. Wir stellen einfach den Test an das Ende des Programms.

```
>180 K6$ = "" : REM - Leer-Reaktion testen
>190 CALL 20000 : REM - umwandeln
>200 PRINT "!";K6$;"!" : REM - Original veraendert?
>210 PRINT "!";G6$;"!" : REM - Ergebnis in Ordnung?
>220 END
```

Das ergibt dann:

```
>RUN
!!
!!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz1234567890
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
!!
!!
```

Trauen Sie Ihren Testergebnissen nicht ohne weiteres. Ein falsch angelegter Test kann unter Umständen das ganze Programm verderben, wenn dadurch ein Fehler verdeckt wird, der sich möglicherweise ganz woanders im System äußert. Diese Gefahr ist vor allem bei ungeplanten Tests gegeben, in denen «nach Gefühl» vorgegangen wird. Ganz umgehen läßt sich das nicht, doch ist in kritischen Situationen viel geholfen, wenn der jeweilige Programmtest und die Testergebnisse der Dokumentation beigegeben werden. Dann lassen sich übersehene Stellen von späterer Warte aus viel leichter entdecken und gegebenenfalls ausbügeln.

Bestätigung einholen

Wenden wir diese Kenntnisse auf einen weiteren Programmodul an, z.B. auf:

```
Bestätigung einholen:
    J6 ← 1, / * «Ja»-Antwort annehmen * /
    Bis Antwort «JA» oder «NEIN»:
        Ja/Nein-Entscheidung in J6$ anfordern,
        J6$ in Großschreibung umwandeln,
        Wenn J6$ = «NEIN», dann:
            J6 ← 0.
        Sonst, wenn J6$ < > «JA», dann:
            «Wie bitte».
    Schleifenende (Antwort «JA» oder «NEIN»).
Ende.
```

Wir müssen hier noch festlegen, wie die Ja/Nein-Entscheidung angefordert werden soll. Am sinnvollsten ist das, wenn das aufrufende Programm eine Anfrage ausgibt, z.B. durch

```
PRINT "Ist das richtig so";
```

und dann das Unterprogramm zur Bestätigungsübernahme aufruft. Hier fügen wir die allgemeine Meldung «(Ja oder Nein)» an und fordern dann die Entscheidung vom Benutzer an. Dieses «(Ja oder Nein)» wiederholt sich auch dann, wenn eine falsche Antwort gegeben wurde. Denn dann können wir melden:

```
Wie bitte (Ja oder Nein)?
```

Das bedeutet, daß wir die Meldung immer in zwei Teile aufspalten, eine spezifische Anforderung, die durch einen PRINT-Befehl ausgegeben wird (mit Abschluß durch Strichpunkt) und einen Teil, der mit dem INPUT-Befehl ausgegeben wird, d.h.

```
INPUT " (Ja oder Nein)";J6$
```

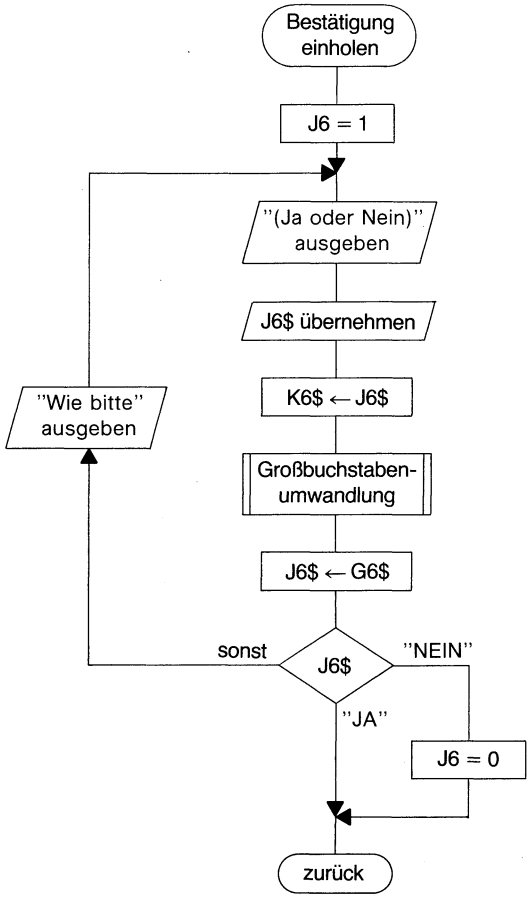


Bild 13 Flußdiagramm zur Bestätigungseinholung

(Beachten Sie hier den führenden Leerschritt im Meldungstext. Er bewirkt einen richtigen Anschluß der Meldung an die vorangehende Anforderung.)

Damit haben wir alle Information für ein Flußdiagramm zusammen, das in Bild 13 wiedergegeben ist. Dort ist ein Symbol neu aufgenommen, das wir bis jetzt nicht verwendet haben: ein Kasten mit Doppelstrichen an den senkrechten Seiten. Er bezeichnet den Aufruf des *Unterprogramms* mit dem angegebenen Titel, in unserem Fall also des Unterprogramms «Großschreibung».

Dieser Unterprogrammaufruf umfaßt im Grunde jedoch mehr als nur die im Kasten angegebene Aktion. Hier gehört auch die Parameterübergabe hinein, und mit ihr sollten wir uns noch einen Moment beschäftigen. Wir haben in dem Modul zur Umwandlung in Großschreibung festgelegt, daß die umzuwandelnde Zeichenkette als K6$ übergeben und die umgewandelte Form in G6$ zurückgeliefert wird. In unserem Entwurf für die Bestätigungsübernahme arbeiten wir dagegen mit der Zeichenkette J6$, wobei die Grundidee war, durch eine eigene Variable mögliche Überschneidungen auszuschließen.

Doch – ist das eigentlich notwendig? Im Grunde nicht, wie das Flußdiagramm ausweist. Wir führen ja nur folgende Operationen aus:

- Übernahme der Benutzerantwort,
- Umwandlung in Großschreibung,
- Test des Umwandlungsergebnisses.

Die übernommene Antwort wird für keine andere Aufgabe gebraucht, für die man sie irgendwie aufbewahren müßte. Dann brauchen wir aber auch J6$ und die Parameterzuweisungen K6$ ← J6$ bzw. J6$ ← G6$ nicht, sondern können die Benutzerantwort gleich in K6$ übernehmen und den Test auf «JA» oder «NEIN» mit G6$ ausführen. Das macht J6$ für andere Aufgaben frei.

Damit erhält das Unterprogramm die Form:

```
>20200 REM *** Ja/Nein-Bestaetigung einholen ***
>20201 REM
>20202 REM - Fordert vom Benutzer eine "Ja"- oder "Nein"-Antwort
>20203 REM - an. Kehrt erst zurück, wenn er sich fuer eines von
>20204 REM - beiden entschieden hat.
>20205 REM - Variable:
>20206 REM    J6 - Antwortflagge:
>20207 REM         "Nein" = 0 , "Ja" = 1
>20208 REM
>20210 J6 = 1 : REM - "Ja"-Antwort annehmen
>20220 INPUT " (Ja oder Nein)";K6$
>20230 GOSUB 20000 : REM - Antwort in Grossschreibung umwandeln
>20240 IF G6$="JA" THEN RETURN
>20250 IF G6$<>"NEIN" THEN PRINT "Wie bitte"; : GOTO 20220
>20260 J6 = 0 : REM - Antwort war "Nein"
>20270 RETURN
```

Beachten Sie, daß wir im Vorspann nur die von dem behandelten Unterprogramm selbst belegte Variable J6 angegeben haben. Die im Programmteil zur Großschreibungsumwandlung verwendeten und hier auftauchenden Variablen K6$ und G6$ werden nicht gesondert dargestellt. Das ist ein Kompromiß, da sonst der Bedarf für die Variablenlisten von Schachtelungsstufe zu Schachtelungsstufe nach außen immer mehr anwachsen würde. Die Alternative zu dieser Darstellungsform wäre, *sämtliche* verwendete Variablen am Programmanfang zusammenzufassen. Das widerspricht aber wieder der relativen Eigenständigkeit von Hilfsprogrammen, und für mehr als eine Variablenangabe reicht der Platz in der Regel nicht. Eine zusammenfassende Liste der im Programm eingesetzten Variablen und ihrer Funktion sollte aber auf alle Fälle in der Dokumentation enthalten sein. Für unseren Programmentwurf finden Sie die Variablen in Anhang 2 zusammengestellt.

Entwerfen Sie jetzt selbst ein kleines Testprogramm für dieses Unterprogramm und testen Sie es auf Herz und Nieren. Stellen Sie dann auch den BASIC-Kode für die Moduln «Artikelbezeichnung anfordern» und die vier Moduln zur Handhabung des Eintrags auf der vierten Stufe der Programmhierarchie von Bild 11 auf. Vergessen Sie dabei das Flußdiagramm nicht, überlegen Sie sich mögliche Optimierungen, und schließlich und endlich erstellen Sie ein paar Tests dazu und führen Sie diese aus.

3.19 Testen für Fortgeschrittene

Sehen wir uns zum Abschluß einen etwas anspruchsvolleren Modul an:

```
Artikel neu aufnehmen:
    Bis Operation beendet:
        Zahl der belegten Einträge ermitteln,
        Wenn Datei voll, dann:
            «Datei ist voll»,
            Operation beendet.
        Sonst:
            Zahl der Einträge melden,
            Freien Eintrag aufsuchen,
            Bis Bestätigung erteilt:
                Eintragsdaten übernehmen,
                Eintrag melden,
                Bestätigung einholen,
                Wenn Bestätigung erteilt, dann:
                    Eintrag in die Datei schreiben,
                    Eintrag zählen.
            Schleifenende (Bestätigung erteilt),
            Anfrage: «Noch ein Artikel?»,
            Wenn kein weiterer Artikel, dann:
                Operation beendet.
    Schleifenende (Operation beendet).
Ende.
```

Das beginnt mit dem Ermitteln der Zahl der belegten Einträge. Hier genügt es, die Zahl B der in der Datei belegten Einträge mit der Maximalzahl F1 der überhaupt möglichen Einträge zu vergleichen. Sind beide gleich, so wird die Operation mit einer Meldung abgeschlossen, ansonsten muß der nächste freie Eintrag aufgesucht, geändert, bestätigt und in die Datei geschrieben werden. Dieser Vorgang wiederholt sich, bis eine ausdrückliche Abbruchaufforderung erteilt wurde.

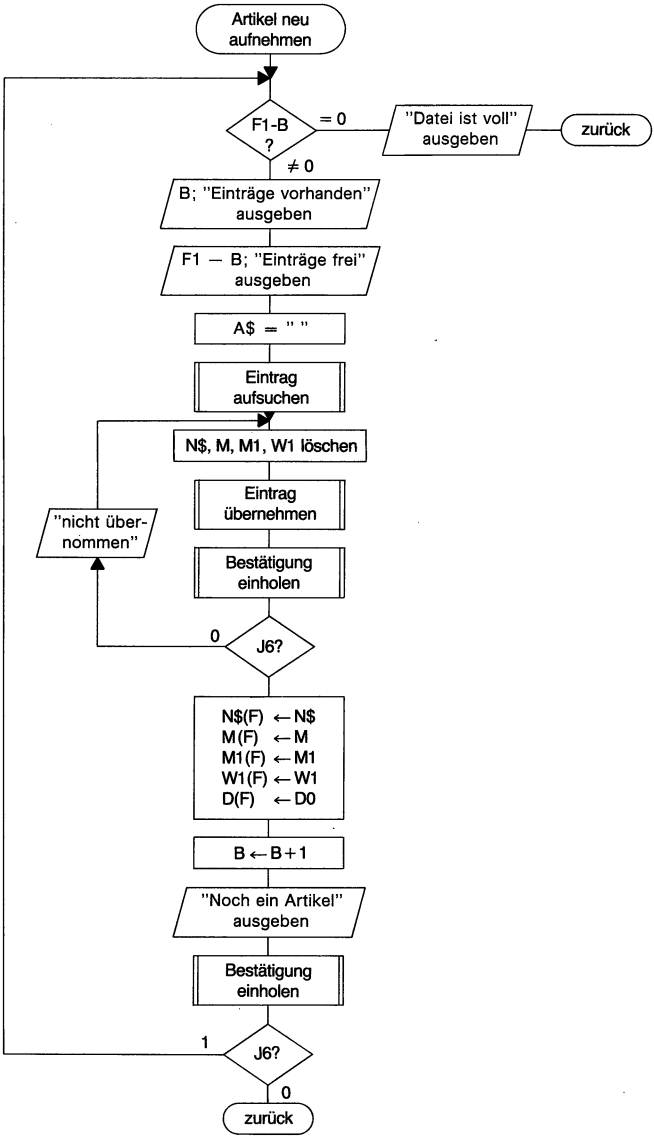


Bild 14 Flußdiagramm zur Neuübernahme von Artikeln

Bild 14 zeigt das Flußdiagramm zu diesem Unterprogramm, das sich wie folgt kodieren läßt:

```
>3000 REM *** Artikel neu aufnehmen ***
>3001 REM
>3002 REM - Untersucht, ob in der Datei noch ein Eintrag frei ist.
>3003 REM - Wenn ja, sucht einen freien Eintrag auf, uebernimmt
>3004 REM - die dafür notwendigen Daten und schreibt sie nach
>3005 REM - Bestaetigung in die Datei ein.
>3006 REM
>3007 REM - Variable:
>3008 REM   A$    - hier als leere Suchvariable benutzt
>3009 REM   B     - zur Zeit belegte Eintraege
>3010 REM   D(F)  - Aenderungs-(Anlage)-Datum des Eintrags
>3011 REM   DO    - gueltiges Tagesdatum
>3012 REM   F     - Nummer des bearbeiteten Eintrags
>3013 REM   F1    - Maximalzahl der verfuegbaren Eintraege
>3014 REM   N$    - Artikelname bei der Eintragsuebernahme
>3015 REM   N$(F) - Artikelname von Eintrag F
>3016 REM   M     - Menge auf Lager bei der Eintragsuebernahme
>3017 REM   M(F)  - in Eintrag F festgehaltene Lagermenge
>3018 REM   M1    - Mindestmenge im Lager, Eintragsuebernahme
>3019 REM   M1(F) - in Eintrag F festgehaltene Mindestlagermenge
>3020 REM   W1    - Einzelwert des Artikels, Eintragsuebernahme
>3021 REM   W1(F) - in Eintrag F festgehaltener Einzelwert
>3022 REM
>3030 PRINT CHR$(12); : REM - Schirm loeschen
>3040 PRINT "*** ARTIKEL NEU UEBERNEHMEN ***" : PRINT : PRINT
>3050 IF F1-B<>0 THEN 3080
>3060 PRINT "*** Datei ist voll! ***"
>3070 RETURN
>3080 PRINT B;"Eintraege sind belegt"
>3090 PRINT F1-B;"Eintraege sind noch frei" : PRINT
>3100 A$ = "" : GOSUB <freien Eintrag aufsuchen>
>3110 N$ = "" : M = 0 : M1 = 0 : W1 = 0
>3120 GOSUB <Eintrag übernehmen>
>3130 PRINT CHR$(12); : REM - Schirm loeschen
>3130 PRINT "Sie haben folgenden Eintrag vorgegeben:" : PRINT
>3140 GOSUB <Eintrag melden>
>3150 PRINT
>3160 PRINT "Ist das richtig so"; : GOSUB 20200 : REM - Bestaetigu
>3170 IF J6=0 THEN PRINT "Eintrag nicht uebernommen!" : GOTO 3110
>3180 N$(F) = N$ : M(F) = M : M1(F) = M1 : W1(F) = W1 : D(F) = DO
>3190 B = B+1
>3200 PRINT "Eintrag uebernommen." : PRINT
>3210 PRINT "Ist noch ein Artikel neu aufzunehmen";
>3200 GOSUB 20200 : REM - Ja/Nein-Antwort uebernehmen
>3210 IF J6=1 THEN 3030
>3220 RETURN
```

<freien Eintrag aufsuchen>, <Eintrag übernehmen> und <Eintrag melden> stehen für die Adressen der betreffenden Unterprogramme. Sie müssen hier die Zeilennummern eintragen, die Sie hierfür gewählt hatten.

Beachten Sie, daß wir in Zeile 3030 und Zeile 3130 den Schirm löschen. Das ist immer dann sinnvoll, wenn ein neuer Bedienungsabschnitt beginnt, der keinen unmittelbaren Bezug auf vorher eingegebene Daten nimmt, und bei dem der alte Schirminhalt nur verwirren würde. Wir haben hier die Funktion CHR$(12) gewählt, durch die das Steuerzeichen «Form Feed» an den Bildschirm ausgegeben wird. Wenn Ihr Gerät eine andere Ansteuerung benötigt, empfiehlt sich möglicherweise eine Formulierung als eigenes Hilfsprogramm «Schirm löschen».

Das Unterprogramm ist komplex genug, um es nicht mehr stichprobenartig «nach Gefühl» durchzutesten. Hier sollte man systematisch vorgehen, also eine *Teststrategie* ausarbeiten.

Eine gute und umfassende Teststrategie kann außerordentlich komplex sein. Genaue Regeln für Ihre Ausarbeitung anzugeben würde den Rahmen dieser Einführung bei weitem sprengen. Testen und Prüfen ist ein ganzes Forschungsgebiet für sich. Wir wollen hier nur *intuitiv* ein paar Punkte herausarbeiten, die man beachten sollte.

1. Arbeiten Sie einen schriftlichen Testplan aus

Ein Testplan gehört genau wie Fluß-, Struktur- und Hierarchiediagramme zur Dokumentation des Programms. Man kann – vor allem bei komplexen Programmen – nicht hundertprozentige Fehlerfreiheit garantieren. Ist das betreffende Programmstück aber sorgfältig getestet worden, und sind Testplan und Testergebnisse in der Dokumentation enthalten, so ist die Beseitigung von Fehlern während der Wartungsarbeiten wesentlich vereinfacht. Selbstverständlich muß der Testplan dann entsprechend erweitert und müssen die neuen Testergebnisse der Dokumentation beigegeben werden.

2. Erfassen Sie alle möglichen Programmparameter

Um es noch einmal zu wiederholen: Programmparameter sind alle die Werte, die in den Abarbeitungsgang eingehen und entweder das Ergebnis unmittelbar oder den Verlauf der Abarbeitung beeinflussen. Dazu gehören im wesentlichen drei Kategorien:

a) Aufrufparameter:

Sie werden dem Programm zur Abarbeitung übergeben. In unserem Programmbeispiel sind das die Parameter B, D0 und F1.

b) Verarbeitungsparameter:

Das sind «innere» Parameter, d.h. Variable, die nur im Innern des Programmstücks definiert und verwendet werden.

Diese Verarbeitungsparameter können in BASIC eine Quelle ständigen Ärgers sein, wenn ihre Verwendung nicht genau erfaßt ist. Wir sind ja bereits öfter auf dieses Problem eingegangen. BASIC kennt nur globale Variable, d.h., eine einmal verwendete Variable behält ihren jeweils letzten Wert im ganzen Programm bei. Besonders bei tiefer Verschachtelung von Unterprogrammen kommt es daher immer wieder zu sogenannten Seiteneffekten, bei denen ein eingeschachteltes Programmstück plötzlich einen inneren Parameter eines umfassenderen Teils verändert – obwohl diese Variable in beiden Teilen als ausschließlich innerer Parameter vereinbart war. Am besten folgen Sie hier dem von uns eingeschlagenen Weg, alle derartigen Werte in BASIC so eindeutig wie nur möglich zu halten, d.h. eine Variable nur zu einem einzigen Zweck zu verwenden. Das kostet zwar Speicherplatz, bietet aber zusätzliche Sicherheit – nicht zuletzt bei Programmerweiterungen.

In unserem Unterprogrammbeispiel sind A$, F, N$, M, M1 und W1 derartige Verarbeitungsparameter.

c) Rückgabeparameter:

Das sind Werte, die von einem Programmstück erarbeitet oder bewußt verändert und im übrigen Programm benötigt werden. Insbesondere werden beim Aufruf von Unterprogrammen oft Parameter an den aufrufenden Programmteil zurückgegeben. Man muß hier unterscheiden zwischen Werten, die ausdrücklich als Parameter zurückgegeben werden und solchen, die außerhalb des betrachteten Unterprogramms definiert sind und von ihm während seiner Abarbeitung verändert werden. In vielen Programmen muß man beide Formen einsetzen (das gilt allerdings vor allem für höhere Programmiersprachen als BASIC, die ein etwas anderes Variablen- und Unterprogrammkonzept besitzen). Es empfiehlt sich jedoch, möglichst alle Veränderungen im Hauptprogramm ausdrücklich als solche zu benennen, um Seiteneffekte so gering wie möglich zu halten.

Rückgabeparameter in unserem Beispiel sind die Variablen B, D(F), N$(F), M(F), M1(F) und W1(F). Von ihnen dient B sowohl als Aufrufparameter zur Bestimmung der Programmbelegung als auch als Rückgabeparameter, der den neuesten Belegungsstand dem übrigen System mitteilt.

3. Gliedern Sie die Parameter nach ihrem Einfluß auf das Programm

Die diversen Parameter verhalten sich bei der Abarbeitung eines Programms verschieden. Sie dienen unterschiedlichen Zwecken, und sie werden unter bestimmten Bedingungen verändert. Das muß systematisch erfaßt werden, um einen Testplan erarbeiten zu können. Die Verwendung einer Variablen im Programm läßt sich in drei Hauptgruppen gliedern:

a) Programmsteuerung,
b) Wertermittlung,
c) sonstige Funktion (z.B. eine standardisierte Meldung).

Daneben ist wichtig, welche Variable bei der Abarbeitung verändert wird und durch wen das geschieht. In BASIC-Programmen ist es darüber hinaus sinnvoll, die Zeile zu kennen, in der eine Variable auftaucht. Am besten faßt man das zu einer geeignet gestalteten Tabelle wie in Bild 15 zusammen. Dabei sollte diese Übersicht anhand einer durch LIST entstandenen Programmauslistung geschehen. Dann hat man bereits erste Kontrollmöglichkeiten:

- Werden nur die Variablen verändert, von denen das vorgesehen ist, und werden dabei alle zu ändernden Variablen erfaßt?
- Werden die Variablen da verändert, wo es vom logischen Programmfluß her notwendig ist?
- Werden alle Variablen an den richtigen Stellen zur Programmsteuerung eingesetzt?
- Werden alle Werte an den vorgesehenen Stellen und mit den richtigen Formeln ermittelt?
- Werden die sonstigen Operationen, insbesondere die Meldungen, an den richtigen Stellen ausgeführt?

An dieser Stelle lassen sich bereits sehr viele Flüchtigkeits- und Tippfehler ausbügeln. Alles, was Sie auf dieser Stufe erkannt haben, macht Ihnen beim eigentlichen Testlauf kein Kopfzerbrechen mehr. Der Aufwand für die Tabellenanfertigung lohnt sich durchaus.

(Nebenbei gesagt läßt sich eine derartige Tabelle gewinnbringend bei der Analyse fremder Programme verwenden, die man an den eigenen Rechner anpassen, verändern oder einfach nur verstehen möchte.)

4. Stellen Sie die Wirkung von steuernden Variablen zusammen

Das betrifft im Prinzip die verschiedenen Programmverzweigungen. Stellen Sie für jede in Betracht kommende Variable, die Einfluß auf die Programmabarbeitung hat, die möglichen Alter-

	wird verändert		Programm-steuerung	Wert-ermittlung	Sonstiges
	durch	in Zeile			
Aufrufparameter					
B			3050	3050, 3090	3080, 3090
D0					3180
F1			3050	3050, 3090	3090
Verarbeitungs-parameter					
A$	Definition	3100			
F	Unterprogramm	3100			
J6	Unterprogramm	3160, 3200	3170, 3210		
N$	Definition Unterprogramm	3110 3120			3130, 3180
M	Definition Unterprogramm	3110 3120			3130, 3180
M1	Definition Unterprogramm	3110 3120			3130, 3180
W1	Definition Unterprogramm	3110 3120			3130, 3180
Rückgabeparameter					
B	Zählen	3190			
D(F)	D0	3180			
N$(F)	N$	3180			
M(F)	M	3180			
M1(F)	M1	3180			
W1(F)	W1	3180			

Bild 15 Parameteraufstellung zur Neuübernahme von Artikeln

nativen in einer Tabelle ähnlich der in Bild 15 zusammen. Diese Tabelle dient als wesentliche Grundlage beim Ausarbeiten des Testprogramms. Führen Sie die Wirkungen so auf, wie Sie sie bei der *Programmabarbeitung* kontrollieren können. D.h., kopieren Sie nicht einfach das Flußdiagramm oder die Strukturbeschreibung, sondern nehmen Sie in die Tabelle die zu erwartenden Meldungen oder eine Kurzbeschreibung der vorgesehenen Reaktionen des Systems auf.

	F1-B	
= 0		≠ 0
"Datei ist voll" Abbruch		Übergang zur Neueingabe
= 0	J6	≠ 0
	Zeile 3170 "Ist das richtig so?"	
"Eintrag nicht übernommen" Neueingabe		Dateizuweisung Eintrag in B zählen "Eintrag übernommen"
	Zeile 3210 "Noch ein Artikel?"	
keine weitere Aktion zurück		Schirm löschen Neueingabe

Bild 16 Aufstellung der Steuerparameter im Unterprogramm zur Neuübernahme von Artikeln

Diese Übersicht kann sehr schwierig zu erstellen sein, wenn viele Alternativen gegeben sind. Es gibt komplexe Programme, die eine einfache tabellarische Zusammenfassung wie in Bild 16 nicht erlauben. In diesem Fall muß man zu weiteren Hilfsmitteln, zumeist grafischer Art, greifen. Wir können hier allerdings nicht näher darauf eingehen. Oft hilft es, wenn man das Programm weiter in leichter zu testende Einheiten untergliedert (wobei diese Gliederung nicht unbedingt mit der logischen Gliederung in Unterprogramme zusammenfallen muß). Diese Einheiten lassen sich für sich selbst austesten und anschließend geeignet in ihrem Zusammenwirken untersuchen. In komplexen Fällen ist es sogar sinnvoll, solche Untereinheiten beim Prüfen der übergeordneten Strukturen einfach auszublenden und durch passende Meldungen oder sonstige Ersatzoperationen zu ersetzen. Wie gesagt – das Thema ist ein Wissenschaftsgebiet für sich. Mehr als ein paar intuitive Anregungen können hier nicht gegeben werden. Das Wichtigste ist, daß Sie die Testprozedur so sorgfältig wie möglich planen.

Anhand dieser Aufstellungen kann man dann ein Testprogramm zusammenstellen. Stellen Sie dabei sicher, daß alle möglichen Varianten prüfbar sind. Geben Sie die Aufrufparameter geeignet vor. Erarbeiten Sie Datensätze, die ein bestimmtes Ergebnis bringen müssen – oder noch besser, lassen Sie diese Daten von jemand anderem erarbeiten. Das gewährleistet, daß Sie nicht die eingeflossenen Denkfehler durch unbewußtes Zurechtschneidern des Tests verschleiern anstatt sie offenzulegen.

Wenn anhand der zuvor gemachten Aufstellungen die zu erwartenden Ergebnisse nicht deutlich werden, arbeiten Sie ein ausführliches Musterprotokoll für den Test aus. (Im einfachsten Fall wird der Testlauf über einen Drucker mitprotokolliert.)

In vielen Fällen genügt es, wenn man sich von den Vorarbeiten zu einem Testprogramm «inspirieren» läßt. Auf alle Fälle allerdings muß auch das Testprogramm selbst sorgfältig getestet werden. Denn bei dem Ganzen ist zweierlei zu bedenken:

1. Eine absolute Fehlerfreiheit läßt sich nicht garantieren.
2. Der Fehler kann auch im Testprogramm sitzen!

Damit sind Sie am Zuge: Stellen Sie anhand der Tabellen ein Testprogramm für das angegebene Unterprogramm auf und testen Sie es sorgfältig aus. Nehmen Sie sich dann nach und nach alle anderen Moduln unseres Dateibeispiels vor, kodieren sie und testen sie gründlich durch. Wir haben alles besprochen, was Sie als Hilfe dazu brauchen werden. Und wenn Sie sich nicht mehr zurechtfinden sollten: Im Anhang finden Sie sowohl eine Zusammenstellung von BASIC-Befehlen als auch ein Verzeichnis für die wichtigsten Tips und Hinweise, die sich im Lauf des Buchs angesammelt haben. Eine «Auflösung» werden Sie allerdings nicht finden. Dazu bestehen zu viele Möglichkeiten, ein derartiges Programm zu kodieren. Und außerdem können Sie sich die Auflösung selbst schaffen: Das Programm ist dann in Ordnung, wenn es funktioniert. Widerstehen Sie nach Möglichkeit auch der Versuchung, sich vor dem Ausarbeiten der noch fehlenden Programmteile zu drücken. Nicht nur, daß sich vielleicht irgendwo sinnvolle

Anwendung für ein solches Programm in Ihrem eigenen Bereich finden läßt, etwas anderes ist viel wichtiger:
Sie lernen Programmieren nur durch eigenes Programmieren!

Ausblick

Das beschließt unsere Reise durch das BASIC-Reich. Wie zu Beginn gesagt, hatten wir bei weitem nicht vor, *alle* Eigenschaften dieser Sprache darzustellen. Wenn Sie ein Gefühl dafür bekommen haben, was man in BASIC kann, wie man ein Problem angeht und es sauber in ein BASIC-Programm umsetzt, dann ist der Zweck dieses Buchs voll erreicht. Sie sollten aber auf keinen Fall hier stehenbleiben. Versuchen Sie als nächsten Schritt, sich mit den Eigenschaften Ihrer BASIC-Version so intensiv wie möglich vertraut zu machen. Lernen Sie weiter die Unterschiede zwischen Ihrer Fassung und anderen verbreiteten Versionen kennen, damit Sie fremde Programme ohne größere Mühe verstehen und übernehmen können. Untersuchen Sie so viele fremde Programme wie möglich, um herauszufinden, wie andere an ein Problem herangehen, was man übernehmen kann, was man besser machen und was man auf keinen Fall übernehmen sollte. Studieren Sie auch Ihre eigenen Programme von Zeit zu Zeit daraufhin, wo mögliche Schwächen liegen und was zu verbessern wäre. Und – versuchen Sie, andere Programmiersprachen als BASIC kennenzulernen. Das erweitert den Blick für die Möglichkeiten. Sie werden feststellen, daß man mit BASIC sehr viel machen kann, daß man aber einige Probleme besser in einer anderen Sprache angeht. Ein Beispiel dafür wäre die Programmiersprache *Pascal,* die sich derzeit auf breiter Front im Mikrocomputerbereich durchsetzt. Wie gesagt, erweitern Sie Ihren Horizont und Ihre Fähigkeiten, soweit es nur geht. Selbst wenn Sie bis an Ihr Lebensende mit BASIC programmieren sollten – Ihre Arbeit wird um so besser werden, je mehr Sie darüber wissen, wie es andere machen und wie sie mit anderen Mitteln gemacht werden kann.

Anhang

1. Strukturbeschreibungen der Dateimoduln

```
Programm steuern:
    Variable dimensionieren und initialisieren,
    Programm einleiten,
    Bis 3 ausgewählt:
        Menü 1 anbieten,
        Auswahl übernehmen,
        Wenn 1 ausgewählt, dann:
            Lagerführung.
        Sonst, wenn 2 ausgewählt, dann:
            Lagerübersicht.
        Sonst, wenn 3 ausgewählt, dann:
            Programm beenden.
    Schleifenende (3 ausgewählt).
Ende.

Lagerführung:
    Bis Ende:
        Menü 2 anbieten,
        Auswahl übernehmen,
        Wenn 1 ausgewählt, dann:
            Artikel neu aufnehmen.
        Sonst, wenn 2 ausgewählt, dann:
            Artikel ändern.
        Sonst, wenn 3 ausgewählt, dann:
            Bestand ändern.
        Sonst, wenn 4 ausgewählt, dann:
            Ende.
    Schleifenende.
Ende.
```

```
Lagerübersicht:
   Menü 3 anbieten,
   Auswahl übernehmen,
   Wenn 1 ausgewählt, dann:
      Lagerauszug drucken.
   Sonst, wenn 2 ausgewählt, dann:
      Nachbestellübersicht drucken.
Ende.

Programm einleiten:
   Datum übernehmen und in D0 ablegen,
   Lagerbestand laden.
Ende.

Datum übernehmen:
   Bis Bestätigung erteilt:
      Datum in D$ anfordern,
      Datum in interne Darstellung umwandeln:
         (gesondert aufgeführt).
      Bestätigung einholen.
   Schleifenende (Bestätigung erteilt).
Ende.

Datum in interne Darstellung umwandeln:
   K ← 1, I ← 1,
   Bis K > Länge(D$):
      Wenn Zeichen(K) < > «.», dann:
         D$(I) ← D$(I) + Zeichen(K).
      Sonst:
         I ← I+1.
      K ← K+1.
   Schleifenende (K > Länge(D$)).
   Wenn D$(3) länger als 2 Zeichen, dann:
      D$(3) auf 2 niederwertige Zeichen kürzen.
   D ← 10 000 * VAL(D$(3)) + 100 * VAL(D$(2)) + VAL(D$(1)).
Ende.

Artikel neu aufnehmen:
   Bis Operation beendet:
      Zahl der belegten Einträge ermitteln,
      Wenn Datei voll, dann:
         «Datei ist voll»,
         Operation beendet.
      Sonst:
         Zahl der Einträge melden,
         Freien Eintrag aufsuchen,
```

```
            Bis Bestätigung erteilt:
                Eintragsdaten übernehmen,
                Eintrag melden,
                Bestätigung einholen,
                Wenn Bestätigung erteilt, dann:
                    Eintrag in die Datei schreiben,
                    Eintrag zählen.
            Schleifenende (Bestätigung erteilt),
            Anfrage: «Noch ein Artikel?»,
            Wenn kein weiterer Artikel, dann:
                Operation beendet.
    Schleifenende (Operation beendet).
Ende.

Artikel ändern:
    Bis Operation beendet:
        Artikelbezeichnung anfordern,
        Eintrag aufsuchen,
        Wenn Eintrag nicht gefunden, dann:
            «Eintrag nicht gefunden».
        Sonst:
            Eintragsdaten in Hilfsvariable kopieren,
            Bis Bestätigung erteilt:
                Neue Eintragsdaten übernehmen,
                Eintrag melden,
                Bestätigung einholen,
                Wenn Bestätigung erteilt, dann:
                    Eintrag in die Datei schreiben.
            Schleifenende (Bestätigung erteilt),
            Anfrage: «Noch ein Artikel?»,
            Wenn kein weiterer Artikel, dann:
                Operation beendet.
    Schleifenende (Operation beendet).
Ende.

Eintrag übernehmen:
    Artikelbezeichnung N$ übernehmen,
    Mindestmenge M1 übernehmen,
    Lagermenge M übernehmen,
    Einzelwert W1 übernehmen.
Ende.

Eintrag aufsuchen:
    Indexvariable F auf Dateianfang -1 setzen,
    Bis Eintrag gefunden oder Dateiende:
        Nächsten Eintrag aufsuchen:
```

```
            Indexvariable F weitersetzen,
            Wenn F>F1, dann:
                Dateiende.
        Wenn nicht Dateiende, dann:
            Eintragsname N$(F) mit Vorgabe A$
                vergleichen,
            Wenn Name gleich, dann:
                Eintrag gefunden.
    Schleifenende (Eintrag gefunden oder Dateiende).
    Wenn Dateiende, dann:
    F ← -1. / * Dateiendeflagge setzen * /
Ende.

Eintrag melden:
    Artikelbezeichnung N$ melden,
    Lagermenge M melden,
    Mindestmenge M1 melden,
    Einzelwert W1 melden,
    Gesamtwert M * W1 melden,
    Änderungsdatum D melden.
Ende.

Bestand verändern:
    Bis Operation beendet:
        Bis Artikel gefunden:
            Artikelbezeichnung anfordern,
            Eintrag aufsuchen,
            Wenn Eintrag nicht gefunden, dann:
                Korrektur anfordern.
        Schleifenende (Artikel gefunden),
        Bis Eintrag in Ordnung:
            Lagermenge in Hilfsvariable kopieren,
            Lagermenge melden,
            Anfrage, ob Exemplare zu entnehmen sind,
            Wenn «Ja», dann:
                Anzahl übernehmen,
                Lagermenge - entnommene Anzahl.
            Sonst:
                Anzahl übernehmen,
                Lagermenge + hinzugefügte Anzahl.
            Wenn Änderung <> 0, dann:
                Neue Lagermenge melden,
                Richtigkeit bestätigen lassen,
                Wenn Richtigkeit bestätigt, dann:
                    Lagermenge in die Datei schreiben,
                    Eintrag in Ordnung.
```

```
                Sonst:
                    Eintrag nicht in Ordnung.
            Sonst:
                «Nichts verändert.»,
                Eintrag in Ordnung. /* Abbruch */
        Schleifenende (Eintrag in Ordnung).
        Anfrage: «Noch ein Artikel?»,
        Wenn kein weiterer Artikel, dann:
        Operation beendet.
    Schleifenende (Operation beendet).
Ende.

Lagerübersicht:
    F, W0 und Z auf 0 initialisieren,
    Menü anbieten,
    Auswahl übernehmen,
    Wenn Gesamtausdruck gewählt, dann:
        Untere Datumsgrenze auf Minimum setzen,
        Obere Datumsgrenze auf Maximum setzen,
        Lagerauszug drucken.
    Sonst, wenn Ausdruck in Datumsgrenzen, dann:
        Untere Datumsgrenze übernehmen,
        Obere Datumsgrenze übernehmen,
        Lagerauszug drucken.
    Sonst, wenn Nachbestellübersicht, dann:
        Nachbestellübersicht drucken.
Ende.

Lagerauszug drucken:
    «Drucker einschalten»,
    Bis Dateiende:
        Dateiindex F weiterzählen,
        Wenn Dateiindex F>F1, dann:
            Dateiende.
        Sonst:
            Wenn D(F)>=D1 und D(F)<=D2, dann:
                Eintrag ausdrucken.
    Schleifenende (Dateiende),
    Gesamtwert W0 ausdrucken.
Ende.

Nachbestellübersicht drucken:
    «Drucker einschalten»,
    Bis Dateiende:
        Dateiindex F weiterzählen,
        Wenn Dateiindex F>F1, dann:
```

```
            Dateiende.
        Sonst:
            Wenn M(F)<M1(F), dann:
                Eintrag ausdrucken.
    Schleifenende (Dateiende).
Ende.


Eintrag ausdrucken:
    Wenn Z=0, dann:
        Kopfzeile ausdrucken,
        Z ← Z+1.
    Sonst:
        N$(F),M(F),M1(F),W1(F) drucken,
        M(F) * W1(F) drucken,
        W0 ← W0+M(F) * W(F),
        neue Zeile anfahren,
        Z ← Z+1.
    Wenn Z>Maximum, dann:
        Neue Seite anfordern,
        Z ← 0.
Ende.


Artikelbezeichnung anfordern:
    Eingabe von der Tastatur in E6$ übernehmen,
    Wenn E6$ länger als L6 ist, dann:
        E6$ auf L6 Zeichen kürzen.
    E6$ in Großschreibung umformen,
    Wenn A6<>0, dann:
        Bis Länge(E6$)=L6:
            Leerschritt rechts an E6$ anfügen.
        Schleifenende (Länge(E6$)= L6).
Ende.


Bestätigung einholen:
    J6 ← 1, / * «Ja»-Antwort annehmen * /
    Bis Antwort «JA» oder «NEIN»:
        Ja/Nein-Entscheidung in J6$ anfordern,
        J6$ in Großschreibung umwandeln,
        Wenn J6$=«NEIN», dann:
            J6 ← 0.
        Sonst, wenn J6$<>«JA», dann:
            «Wie bitte».
    Schleifenende (Antwort «JA» oder «NEIN»).
Ende.
```

```
Großbuchstabenumwandlung:
   G6$ löschen,
   Wenn K6$ nicht leer ist, dann:
      Für Z6=1 bis Länge(K6$):
         Wenn Zeichen(K6$,Z6)>=«a» und <=«z», dann:
            G6$ ← G6$+Großbuchstabe(Zeichen(K6$,Z6)).
         Sonst:
            G6$ ← G6$+Zeichen(K6$,Z6).
      Schleifenende (Z6>Länge(K6$)).
Ende.
```

2. Im Lagerdateiprogramm verwendete Variable

A6	–	Auffüllflagge, sie möge den Wert Null haben, wenn nicht aufgefüllt werden soll, ansonsten irgendeinen Wert ungleich Null.
A$	–	Allgemeine Zeichenkette zur Übernahme von Antworten
B	–	Zahl der belegten Einträge in der Datei
D(F)	–	Datum der letzten Änderung eines Eintrags
D0	–	Gültiges Tagesdatum
D1	–	Untere Datumsgrenze beim Ausdruck von Lagerauszügen
D2	–	Obere Datumsgrenze beim Ausdruck von Lagerauszügen
D$	–	Zeichenkette zur Datumsübernahme
D$(1)	–	Tagesinformation im übernommenen Datum
D$(2)	–	Monatsinformation im übernommenen Datum
D$(3)	–	Jahresinformation im übernommenen Datum
E6$	–	Zeichenkette, in der die Eingabe steht
F	–	Nummer des gerade betrachteten Feldelements
F1	–	Maximalzahl der verwendbaren Dateielemente
G6$	–	Ergebnis bei der Großbuchstabenumwandlung
J6	–	Ja/Nein-Flagge; enthält Null bei einer «Nein»-Antwort, einen Wert ungleich Null bei einer «Ja»-Antwort
K6$	–	Ein/Ausgabeparameter bei der Großbuchstabenumwandlung
L6	–	Maximallänge der Eingabe
M(F)	–	Menge auf Lager
M	–	Menge auf Lager während der Datenübernahme
M0	–	Menge, die gerade entnommen oder hinzugefügt wurde
M1(F)	–	Mindestmenge auf Lager
M1	–	Mindestmenge während der Datenübernahme
N$(F)	–	Namen der gelagerten Artikel
N$	–	Name des Artikels während der Datenübernahme
W1(F)	–	Einzelwert der verschiedenen Artikel
W1	–	Einzelwert während der Datenübernahme
W	–	Gesamtwert der verschiedenen Artikel
W0	–	Gesamtwert des Lagerbestands

Z – Zahl der an den Drucker ausgegebenen Zeilen
Z6 – Zeichenposition bei der Großbuchstabenumwandlung
Z6$ – bei der Großbuchstabenumwandlung untersuchtes Zeichen

3. Befehle, Anweisungen und Funktionen in BASIC

Dieses Verzeichnis stellt die wichtigsten Sprachbestandteile von BASIC zusammen. Es kann gleichzeitig als Stichwortverzeichnis verwendet werden. Die nach der Beschreibung in Klammern angegebenen Zahlen verweisen auf die Seiten, in denen nähere Erläuterungen zum jeweiligen Befehl und zum Umgang mit ihm zu finden sind.

Die Übersicht richtet sich im wesentlichen nach der derzeit wohl umfassendsten BASIC-Version für Mikrocomputer, dem BASIC-80 von Microsoft. Die Darstellung beginnt mit dem Befehlswort, gefolgt von den möglichen Argumenten in spitzen Klammern. Dem schließt sich eine Kurzbeschreibung des Befehls an, und der Abschluß wird – wo möglich – durch einen Hinweis auf die Stelle(n) im Buch gegeben, an denen mehr zu dem betrachteten Befehl zu finden ist.

1. Kommandos

Kommandos sind direkt abgearbeitete Befehle, die in der Regel nicht im Programmtext auftauchen dürfen.

AUTO ⟨Startzeile⟩, ⟨Abstand⟩
Erzeugt bei der Programmeingabe automatisch die Zeilennummern, die beginnend mit ⟨Startzeile⟩ im ⟨Abstand⟩ durchnumeriert werden.

CLEAR ⟨numerische Argumente⟩
Löscht den Variablenspeicher und teilt ihn neu auf. Die Bedeutung der numerischen Argumente wechselt von BASIC-Version zu BASIC-Version. Meist kann man die Speicherobergrenze für BASIC und den Speicheranteil für Zeichenketten bestimmen. Kann auch im Programm stehen.
(236)

CONT
Setzt die Programmabarbeitung an der unterbrochenen Stelle fort.
(126, 270)

EDIT ⟨Zeilennummer⟩
Erlaubt die Edierung von Programmzeilen über die primitiven Möglichkeiten des Löschens und Überschreibens ganzer Zeilen hinaus.

LIST ⟨von⟩ – ⟨bis⟩
Gibt das Programm von Zeile ⟨von⟩ bis Zeile ⟨bis⟩ an die Bedienerkonsole aus.
(40, 44)

LLIST ⟨von⟩ – ⟨bis⟩
Wie LIST, aber Ausgabe auf den Drucker des Systems.

LOAD ⟨Dateiname⟩
Lädt das Programm mit dem Titel ⟨Dateiname⟩ vom Massenspeicher (meist von Diskette) in den Rechner. Ein vorher vorhandenes Programm wird gelöscht.
(81)

MERGE ⟨Dateiname⟩
Wie LOAD, aber ohne Löschen des alten Programms. Erlaubt das Zusammenfügen von Teilprogrammen aus einer Bibliothek.

NEW
Löscht den Programmspeicher. Viele Versionen – jedoch nicht alle – löschen ebenfalls den Datenspeicher.
(38)

RENUM ⟨neu⟩,⟨alt⟩,⟨Abstand⟩
Numeriert die Programmzeilen ab Zeile ⟨alt⟩ beginnend mit Zeilennummer ⟨neu⟩ im ⟨Abstand⟩ um. Sprungziele im Programm werden automatisch angepaßt.

RUN ⟨Zeile⟩
Startet die Programmabarbeitung, wahlweise ab der angegebenen ⟨Zeile⟩.
(40, 270)

SAVE ⟨Dateiname⟩
Speichert das im Rechner stehende Programm auf einem externen Massenspeicher (meist auf Disketten) unter ⟨Dateiname⟩ ab.
(81)

TROFF
Schaltet die Trace-Option in ausgebauten BASIC-Versionen ab.
(162)

TRON
Schaltet die Trace-Option in ausgebauten BASIC-Versionen ein.
(162)

2. *Anweisungen*

Anweisungen sind BASIC-Befehle, die im Programmtext selbst stehen können. Die meisten von ihnen können aber auch direkt abgearbeitet werden.

CHAIN ⟨Dateiname⟩
Erlaubt in ausgebauten BASIC-Versionen das Laden und Abarbeiten des Programms ⟨Dateiname⟩ von einem anderen Programm aus. Meist werden alle oder ein ausgewählter Satz von Variablen an das neu geladene Programm übergeben.

COMMON ⟨Variablenliste⟩
Liste der Variablen, die bei CHAIN dem neu geladenen Programm übergeben werden sollen.

DEF FN ⟨Variable(Argumentliste) = Ausdruck⟩
Gestattet die Definition von häufig benutzten Funktionen im Programm. Durch Aufruf mittels FN⟨Variable(Argumente)⟩ wird die im ⟨Ausdruck⟩ festgelegte Rechnung ausgeführt.

DIM ⟨Feldvariable(Dimensionsangabe)⟩
Reserviert für die angegebenen ⟨Feldvariablen⟩ Speicherplatz gemäß der ⟨(Dimensionsangabe)⟩. Achtung bei Zeichenketten: Die verschiedenen BASIC-Versionen verhalten sich hier unterschiedlich!
(178, 187)

END
Hält das Programm an, schließt (in ausgebauten Versionen) alle Dateien und kehrt zur Kommandoebene zurück. Achtung: Einige Versionen gestatten eine END-Anweisung nur ganz am Programmende. Aller nachfolgende Programmtext wird hier ignoriert, auch wenn er mit GOTO angesprungen werden soll.

ERROR ⟨Fehlerkode⟩
Nur in ausgebauten Versionen: Erzeugt absichtlich einen Fehler, der durch eine besondere «ON ERROR»-Anweisung ausgewertet werden kann. Nützlich zum Programmtest und zur Handhabung von Ausnahmesituationen im Programm.

FN ⟨Variable (⟨Argumentliste⟩)⟩
Ruft die durch DEF FN definierte Funktion auf.

FOR ⟨Variable = Ausdruck1⟩ TO ⟨Ausdruck2⟩ STEP ⟨Ausdruck3⟩
Bildet mit NEXT zusammen Schleifen, in denen eine Laufvariable ⟨Variable⟩ von ⟨Ausdruck1⟩ bis ⟨Ausdruck2⟩ in jedem Durchgang um ⟨Ausdruck3⟩ weitergesetzt wird. Die Schleife bricht ab, wenn ⟨Ausdruck2⟩ überschritten ist. Der STEP-Teil kann entfallen, wodurch jeweils um 1 weitergezählt wird. Achtung: Einige Versionen arbeiten die Schleife nicht ab, wenn die Abbruchbedingung bei Aufruf bereits erfüllt ist.
(211)

GOSUB ⟨Zeilennummer⟩
Ruft das Unterprogramm ab ⟨Zeilennummer⟩ auf. Nach Rückkehr durch RETURN wird die auf GOSUB folgende Anweisung abgearbeitet.
(149)

GOTO ⟨Zeilennummer⟩
Bewirkt eine Programmverzweigung zur durch ⟨Zeilennummer⟩ angegebenen Programmstelle. Kann in IF-Anweisungen den THEN-Teil ersetzen.
(56, 73)

IF ⟨Bedingung⟩ THEN ⟨Anweisung⟩
Wenn die ⟨Bedingung⟩ erfüllt ist, dann wird die Anweisung im THEN-Teil abgearbeitet. Sonst wird mit der nächsten Zeile weitergemacht. Sonderregelung für Sprünge: Die Nummer der Zielzeile kann unmittelbar hinter THEN angegeben oder THEN durch GOTO ersetzt werden.
(60, 73, 212, 216)

IF ⟨Bedingung⟩ THEN ⟨Anweisung⟩ ELSE ⟨Anweisung⟩
Nur in ausgebauten BASIC-Versionen. Arbeitet wie IF-THEN, wenn die ⟨Bedingung⟩ erfüllt ist, sonst wird der ELSE-Teil abgearbeitet.
(216)

LET ⟨Variable = Ausdruck⟩
Weist der ⟨Variablen⟩ den Wert von ⟨Ausdruck⟩ zu. LET kann auch entfallen, wird dann aber langsamer abgearbeitet.
(32, 73)

NEXT ⟨Variable⟩
Bezeichnet das Ende der FOR-NEXT-Schleife mit der Laufvariablen ⟨Variable⟩.
(211)

ON ERROR GOTO ⟨Zeile⟩
Nur in ausgebauten BASIC-Versionen: Bewirkt bei Auftreten eines Fehlers automatisch den Aufruf eines besonderen Unterprogramms ab ⟨Zeile⟩. Dort kann der Fehlerkode untersucht und entsprechende Abhilfe eingeleitet werden.

ON ⟨Ausdruck⟩ GOSUB ⟨Zeilenliste⟩
⟨Ausdruck⟩ muß einen ganzzahligen Wert N größer als 0 annehmen. Bewirkt Aufruf des Unterprogramms, dessen Zeilennummer an N-ter Stelle in der ⟨Zeilenliste⟩ steht.
(222)

ON ⟨Ausdruck⟩ GOTO ⟨Zeilenliste⟩
Wie On-GOSUB, jedoch mit einfacher Programmverzweigung durch GOTO.
(221)

OUT ⟨Tor,Byte⟩
Gestattet die Ausgabe von ⟨Byte⟩ über das Datentor ⟨Tor⟩ des Computers. Verlangt genaue Kenntnis des jeweils eingesetzten Rechners.

POKE ⟨Adresse,Byte⟩
Gestattet die Ablage eines ⟨Bytes⟩ unmittelbar im Speicher des Computers in der durch ⟨Adresse⟩ angegebenen Stelle. Verlangt genaue Kenntnis des jeweils eingesetzten Rechners.

RANDOMIZE ⟨Ausdruck⟩
Setzt den Zufallszahlengenerator auf einen durch ⟨Ausdruck⟩ gegebenen Anfangswert. Wird in den verschiedenen BASIC-Versionen unterschiedlich gehandhabt.

REM
Leitet im Programm einen Kommentar ein, der bis zum Zeilenende reicht.
(154, 159)

RESTORE ⟨Zeile⟩
Arbeitet in Zusammenhang mit READ und DATA. Setzt den Lesezeiger auf den Anfang des Datensatzes bzw. auf ⟨Zeile⟩ zurück. Achtung: Viele BASIC-Versionen erlauben kein Argument bei RESTORE.

RESUME
Abschluß des Unterprogramms, das durch «ON ERROR GOTO» aufgerufen worden ist. Bewirkt die Wiederaufnahme der Arbeit an der *durch den Fehler* unterbrochenen Programmstelle (also nicht nach «ON ERROR GOTO»).

RESUME NEXT
Bewirkt die Wiederaufnahme der Arbeit mit der Anweisung, die auf die den Fehler auslösende Stelle folgt.

RESUME ⟨Zeile⟩
Bewirkt nach einem Fehler die Wiederaufnahme der Arbeit an der durch ⟨Zeile⟩ bezeichneten Stelle.

RETURN
Kehrt aus einem Unterprogramm zu der Anweisung zurück, die auf den Aufruf durch GOSUB folgt.
(150)

STOP
Unterbricht die Programmabarbeitung, gibt eine Abbruchmeldung («BREAK AT») aus und kehrt zur Kommandoebene zurück. Die Arbeit kann mit CONT wieder aufgenommen werden.
(127)

WEND
Nur in ausgebauten BASIC-Versionen: Bezeichnet das Ende einer WHILE-Schleife.

WHILE ⟨Bedingung⟩
Nur in ausgebauten BASIC-Versionen: Arbeitet die bis zum zugeordneten WEND stehenden Anweisungen ab, SOLANGE die ⟨Bedingung⟩ erfüllt ist. Ansonsten wird mit den Anweisungen nach WEND weitergemacht.

3. Ein-/Ausgabe-Anweisungen

Diese Anweisungen dienen zum Datenverkehr mit dem Programm.

CLOSE ⟨interner Dateiname⟩
Schließt die Datei mit dem angegebenen ⟨internen Dateinamen⟩. Ohne Argument in der Regel Schließen aller Dateien.
(298)

CONSOLE
In CBASIC Umleitung der nachfolgenden PRINT-Ausgaben über die Bedienerkonsole des Systems, bis LPRINTER befohlen wird.
(262)

DATA ⟨Datenliste⟩
Stellt im Programm die Daten zur Verfügung, die durch READ der Reihe nach in Variable übernommen werden können.
(47)

GET ⟨Variable⟩
Frägt die Tastatur nach einer Eingabe ab und weist eine solche der ⟨Variablen⟩ zu. Macht im Programm weiter, wenn keine Taste gedrückt worden ist. Kann oft zur Umgehung der Beschränkungen des INPUT-Befehls verwendet werden, da *alles* übernommen wird.

GET ⟨interner Dateiname⟩,⟨Aufz.-Nr.⟩,⟨Variable⟩
Liest im direkten Zugriff aus der durch den ⟨internen Dateinamen⟩ gegebenen Datei die Aufzeichnung Nummer ⟨Aufz.-Nr.⟩ in die ⟨Variable⟩. In einigen BASIC-Versionen entfällt ⟨Variable⟩, dort wird die Aufzeichnung in einen speziellen Datenpuffer gelesen.

INPUT «⟨Anforderungsmeldung⟩»;⟨Variable⟩
Gibt die ⟨Anforderungsmeldung⟩ an die Tastatur aus und übernimmt den von dort eingegebenen Wert in die ⟨Variable⟩.
(119, 131)

INPUT ⟨interner Dateiname⟩,⟨Variable⟩
Liest aus der durch den ⟨internen Dateinamen⟩ gegebenen Datei die nächste Aufzeichnung und weist ihren Wert der ⟨Variablen⟩ zu.
(299)

LINE INPUT «⟨Anforderungsmeldung⟩»; ⟨Zeichenkettenvariable⟩
oder
INPUT LINE ...
Arbeitet ähnlich INPUT, weist jedoch der ⟨Zeichenkettenvariablen⟩ die *gesamte* eingegebene Zeile zu, insbesondere also auch Kommas, Anführungszeichen und andere Trennungssymbole.

OPEN ⟨Dateiname⟩ FOR ⟨Modus⟩ AS ⟨interner Dateiname⟩
oder
OPEN ⟨Modus⟩, ⟨interner Dateiname⟩, ⟨Dateiname⟩
oder ähnlich
Öffnet die durch den ⟨internen Dateinamen⟩ angegebene Datei zum ⟨Modus⟩ (z.B. Lesen oder Schreiben) und weist ihr den ⟨internen Dateinamen⟩ zu. Jede Diskettendatei muß vor Lesen oder Schreiben geöffnet sein.
(296)

PRINT ⟨Argument⟩
Druckt das ⟨Argument⟩ auf der Konsole aus.
(20, 61, 68, 261)

PRINT USING ⟨Formatangaben⟩,⟨Argument⟩
Nur in ausgebauten Versionen: Gestattet die *Formatierung* des Ausdrucks, z.B. rechts- oder linksbündiges Drucken, Einhalten der Spalte für den Dezimalpunkt, Bestimmung, wo Ziffern und wo Buchstaben gedruckt werden u.ä. Wichtige Erleichterung beim Ausdrucken von Tabellen.

PRINT ⟨interner Dateiname⟩,⟨Argument⟩
Schreibt das ⟨Argument⟩ als nächste Aufzeichnung in die durch den ⟨internen Dateinamen⟩ gegebene Datei. Diese muß vorher ausdrücklich zum Schreiben geöffnet worden sein.
(263, 299)

LPRINT ⟨Argument⟩
LPRINT USING ⟨Formatangaben⟩,⟨Argument⟩
Wie PRINT und PRINT USING, aber Ausgabe auf dem Drucker des Systems.
(262)

LPRINTER
In CBASIC Umleitung der nachfolgenden PRINT-Ausgaben auf den Drucker des Systems, bis CONSOLE befohlen wird.
(262)

PUT ⟨interner Dateiname⟩,⟨Aufz.-Nr.⟩,⟨Variable⟩
Schreibt die ⟨Variable⟩ im unmittelbaren Zugriff in die Aufzeichnung Nummer ⟨Aufz.-Nr.⟩ der durch den ⟨internen Dateinamen⟩ angegebenen Datei. In manchen BASIC-Versionen ohne Angabe der ⟨Variablen⟩ wird der Inhalt eines besonderen Ausgabepuffers in die ausgewählte Aufzeichnung geschrieben.

READ ⟨Variable⟩
Liest den nächsten Wert aus der durch DATA-Anweisungen gegebenen Datenliste und weist ihn der ⟨Variablen⟩ zu.
(48, 50)

READ ⟨interner Dateiname⟩,⟨Variable⟩
Arbeitet wie INPUT ⟨interner Dateiname⟩,⟨Variable⟩.

WRITE ⟨interner Dateiname⟩,⟨Variable⟩
Arbeitet wie PRINT ⟨interner Dateiname⟩,⟨Variable⟩.

4. Numerische Funktionen

Diese Funktionen erarbeiten zu dem angegebenen (numerischen) Argumenten einen numerischen Wert, der dann weiterverwendet werden kann.

ABS(⟨Ausdruck⟩)	Absolutwert
ATN(⟨Ausdruck⟩)	Arkustangens
COS(⟨Ausdruck⟩)	Cosinus
EXP(⟨Ausdruck⟩)	e-Funktion
FRE(⟨Ausdruck⟩)	ermittelt freien Speicherraum, Argument nur formal notwendig und ohne weitere Bedeutung (179, 232)
INT(⟨Ausdruck⟩)	ganzzahliger (integer) Anteil des Werts von ⟨Ausdruck⟩ (232)
LOG(⟨Ausdruck⟩)	(natürlicher) Logarithmus
RND(⟨Ausdruck⟩)	liefert eine Zufallszahl, meist aus dem Bereich zwischen 0 und 1, das Argument wird in den verschiedenen BASIC-Versionen höchst unterschiedlich gehandhabt, zumeist nur Bestimmung der Art der Zufallszahlermittlung, manchmal auch Festlegung der Obergrenze der zurückgelieferten Zahl.
SGN(⟨Ausdruck⟩)	Liefert 1 für positive Zahlen, −1 für negative Zahlen und 0 für Null
SIN(⟨Ausdruck⟩)	Sinus
SQR(⟨Ausdruck⟩)	Quadratwurzel
TAN(⟨Ausdruck⟩)	Tangens

5. Zeichenkettenfunktionen

Diese Funktionen übernehmen entweder einen numerischen ⟨Ausdruck⟩ oder eine alphanumerische ⟨Kette⟩ als Argument. Sie liefern eine Zeichenkette als Ergebnis, wenn Sie mit einem Dollarzeichen ($) versehen sind, sonst wird ein numerisches Ergebnis erarbeitet.

ASC(⟨Kette⟩)	Liefert den ASCII-Kodewert des ersten Zeichens der Kette. (277)
CHR$(⟨Ausdruck⟩)	Liefert das Zeichen, das den ASCII-Kodewert ⟨Ausdruck⟩ besitzt (274)
INSTR(⟨Anfang⟩,⟨Kette1⟩,⟨Kette2⟩)	Liefert die Anfangsposition des ersten Auftretens von ⟨Kette2⟩ in ⟨Kette1⟩, wobei ab ⟨Anfang⟩ gesucht wird. Liefert 0 falls nichts zu finden ist. (205)
LEFT$(⟨Kette⟩, ⟨Länge⟩)	Liefert die linke Teilkette von ⟨Kette⟩ mit der Länge ⟨Länge⟩. (203)
LEN(⟨Kette⟩)	Liefert die Anzahl von Zeichen in ⟨Kette⟩. (191)
MID$(⟨Kette⟩,⟨Anfang⟩,⟨Länge⟩)	Liefert die Teilkette von ⟨Kette⟩ ab ⟨Anfang⟩ mit der Länge ⟨Länge⟩. (203)
RIGHT$(⟨Kette⟩,⟨Länge⟩)	Liefert die rechte Teilkette von ⟨Kette⟩ mit der Länge ⟨Länge⟩. (203)
SPACE$(⟨Anzahl⟩)	Liefert eine Zeichenkette aus ⟨Anzahl⟩ Leerschritten.
STR$(⟨Ausdruck⟩)	Formt den Wert des ⟨Ausdrucks⟩ in eine Zeichenkette um, die das Druckbild dieser Zahl darstellt.
STRING$(⟨Anzahl⟩,⟨Zeichen⟩)	Liefert eine Kette, die aus ⟨Anzahl⟩ Wiederholungen von ⟨Zeichen⟩ besteht.
VAL(⟨Kette⟩)	Liefert den numerischen Gegenwert der in ⟨Kette⟩ dargestellten Zahl. (195)

6. Sonstige Funktionen

EOF	Zeigt an, daß das Ende der zuletzt angesprochenen Datei erreicht ist. Der zurückgelieferte Wert für «Wahr» oder «Falsch» hängt von der betrachteten BASIC-Version ab.
ERL	Liefert die Zeilennummer des zuletzt aufgetretenen Fehlers.
ERR	Liefert den Kodewert des zuletzt aufgetretenen Fehlers.
INP(⟨Tor⟩)	Übernimmt ein Byte von dem angegebenen Datentor ⟨Tor⟩ des Rechners. Verlangt genaue Kenntnis des eingesetzten Computermodells.
PEEK(⟨Adresse⟩)	Liest ein Byte aus dem Rechnerspeicher an der angegebenen ⟨Adresse⟩ aus. Verlangt genaue Kenntnis des eingesetzten Computermodells.
POS(⟨X⟩)	Liefert die Position des Cursors oder Druckwagens der Benutzerkonsole. Das Argument ⟨X⟩ ist nur aus formalen Gründen notwendig und hat sonst keine Bedeutung.
LPOS(⟨X⟩)	Die gleiche Funktion für den Drucker des Systems.
SPC(⟨Anzahl⟩)	In PRINT-Befehlen zum Ausdruck von ⟨Anzahl⟩ Leerzeichen benutzt.
TAB(⟨Spalte⟩)	In PRINT-Befehlen zum Anfahren der angegebenen ⟨Spalte⟩ benutzt.
USR⟨N⟩(⟨Argument⟩)	Aufruf eines in Maschinensprache geschriebenen Unterprogramms. Verlangt genaue Kenntnis des eingesetzten Computermodells.

4. Der ASCII-Kode

dezimal	hexa-dezimal	oktal	binär		Bedeutung	CON-TROL
0	00	000	0000000	NUL	Null, keine Operation	
1	01	001	0000001	SOH	Start of Heading Vorspannanfang	A
2	02	002	0000010	STX	Start of Text Textanfang	B
3	03	003	0000011	ETX	End of Text Textende	C
4	04	004	0000100	EOT	End of Transmission Übertragungsende	D
5	05	005	0000101	ENQ	Enquiry Stationsanruf	E
6	06	006	0000110	ACK	Acknowledge Bestätigung	F
7	07	007	0000111	BEL	Bell Klingel	G
8	08	010	0001000	BS	Backspace Rückwärtsschritt	H
9	09	011	0001001	HT	Horizontal Tabulation	I
10	0A	012	0001010	LF	Line Feed Zeilenvorschub	J
11	0B	013	0001011	VT	Vertical Tabulation	K
12	0C	014	0001100	FF	Form Feed Formularvorschub Schirm löschen	L
13	0D	015	0001101	CR	Carriage Return Wagenrücklauf	M
14	0E	016	0001110	SO	Shift Out Umschalten	N
15	0F	017	0001111	SI	Shift In Zurückschalten	O
16	10	020	0010000	DLE	Data Link Escape Austritt aus der Datenverbindung	P
17	11	021	0010001	DC1	Device Control 1 Gerätesteuerung 1	Q
18	12	022	0010010	DC2	Device Control 2	R

dezimal	hexa-dezimal	oktal	binär		Bedeutung	CON-TROL
19	13	023	0010011	DC3	Device Control 3	S
20	14	024	0010100	DC4	Device Control 4	T
21	15	025	0010101	NAK	Negative Acknowledge Fehlermeldung	U
22	16	026	0010110	SYN	Synchronous Idle Synchronisierung	V
23	17	027	0010111	ETB	End of Transm. Block Datenblockende	W
24	18	030	0011000	CAN	Cancel ungültig	X
25	19	031	0011001	EM	End of Medium Datenträgerende	Y
26	1A	032	0011010	SUB	Substitute Character Zeichen ersetzen	Z
27	1B	033	0011011	ESC	Escape Rücksprung	
28	1C	034	0011100	FS	File Separator Filetrennung	
29	1D	035	0011101	GS	Group Separator Gruppentrennung	
30	1E	036	0011110	RS	Record Separator Untergruppentrennung	
31	1F	037	0011111	US	Unit Separator Einheitentrennung	
32	20	040	0100000	SP	Space Leerschritt	
33	21	041	0100001	!		
34	22	042	0100010	"		
35	23	043	0100011	#	Nummernzeichen	
36	24	044	0100100	$	(oder nationales Währungssymbol)	
37	25	045	0100101	%		
38	26	046	0100110	&		
39	27	047	0100111	'		
40	28	050	0101000	(		
41	29	051	0101001	)		
42	2A	052	0101010	*	(Stern gilt oft als Multiplikationszeichen)	

dezimal	hexa-dezimal	oktal	binär		Bedeutung
43	2B	053	0101011	+	
44	2C	054	0101100	,	
45	2D	055	0101101	–	
46	2E	056	0101110	.	
47	2F	057	0101111	/	(dient als Divisionszeichen)
48	30	060	0110000	0	
49	31	061	0110001	1	
50	32	062	0110010	2	
51	33	063	0110011	3	
52	34	064	0110100	4	
53	35	065	0110101	5	
54	36	066	0110110	6	
55	37	067	0110111	7	
56	38	070	0111000	8	
57	39	071	0111001	9	
58	3A	072	0111010	:	
59	3B	073	0111011	;	
60	3C	074	0111100	<	
61	3D	075	0111101	=	
62	3E	076	0111110	>	
63	3F	077	0111111	?	
64	40	100	1000000	@	(kaufmännisches «at»)
65	41	101	1000001	A	
66	42	102	1000010	B	
67	43	103	1000011	C	
68	44	104	1000100	D	
69	45	105	1000101	E	
70	46	106	1000110	F	
71	47	107	1000111	G	
72	48	110	1001000	H	
73	49	111	1001001	I	
74	4A	112	1001010	J	
75	4B	113	1001011	K	
76	4C	114	1001100	L	
77	4D	115	1001101	M	
78	4E	116	1001110	N	

dezimal	hexa-dezimal	oktal	binär		Bedeutung
79	4F	117	1001111	O	
80	50	120	1010000	P	
81	51	121	1010001	Q	
82	52	122	1010010	R	
83	53	123	1010011	S	
84	54	124	1010100	T	
85	55	125	1010101	U	
86	56	126	1010110	V	
87	57	127	1010111	W	
88	58	130	1011000	X	
89	59	131	1011001	Y	
90	5A	132	1011010	Z	
91	5B	133	1011011	[	(oder «Ä»)
92	5C	134	1011100	\	backslash (oder «Ö»)
93	5D	135	1011101	]	(oder «Ü»)
94	5E	136	1011110	^	(oder «↑»)
95	5F	137	1011111	_	(oder «→»)
96	60	140	1100000	`	
97	61	141	1100001	a	
98	62	142	1100010	b	
99	63	143	1100011	c	
100	64	144	1100100	d	
101	65	145	1100101	e	
102	66	146	1100110	f	
103	67	147	1100111	g	
104	68	150	1101000	h	
105	69	151	1101001	i	
106	6A	152	1101010	j	
107	6B	153	1101011	k	
108	6C	154	1101100	l	
109	6D	155	1101101	m	
110	6E	156	1101110	n	
111	6F	157	1101111	o	
112	70	160	1110000	p	
113	71	161	1110001	q	
114	72	162	1110010	r	

dezimal	hexa-dezimal	oktal	binär		Bedeutung
115	73	163	1110011	s	
116	74	164	1110100	t	
117	75	165	1110101	u	
118	76	166	1110110	v	
119	77	167	1110111	w	
120	78	170	1111000	x	
121	79	171	1111001	y	
122	7A	172	1111010	z	
123	7B	173	1111011	{	(oder «ä»)
124	7C	174	1111100	|	(oder «ö»)
125	7D	175	1111101	}	(oder «ü»)
126	7E	176	1111110	~	(oder «ß»)
127	7F	177	1111111	DEL	Delete, Rub Out Löschzeichen

5. Tips, Tricks und sonstige Hinweise

Ein Buch wie dieses hier ist eigentlich nicht völlig ausgeschöpft, wenn es nur in einem Zug gelesen und dann auf Nimmerwiedersehen ins Regal gestellt wird. Man sollte es nach dem ersten Verstehen „gegen den Strich" lesen. Das beinhaltet die Suche nach neuen Zusammenhängen, das Lesen auf bestimmte Fragen hin, wobei so manches — und sei es nur als Problemstellung — deutlich wird, was im ersten Durchgang einfach übersehen wurde. Und, natürlich, selbst eine in sich geschlossene Darstellung wie diese hier, die sich von Fragestellung zu Fragestellung tragen läßt, sollte auch zum Nachschlagen verwendet werden können.

Um das alles etwas zu erleichtern, sind die wichtigsten der angesprochenen Punkte hier in Stichwörtern nach Themenbereichen zusammengefaßt. Allerdings wurde kein „klassisches" Stichwortverzeichnis angestrebt. Die Aufschlüsselung des Inhalts ist nur recht grob. Es sollen die über den Text verstreuten zusammenhängenden Themen noch einmal als Gesamtes dargestellt werden – mehr nicht.

Sie machen den besten Gebrauch von dieser Zusammenstellung, wenn Sie sich davon ein wenig zum „Schmökern" animieren lassen. Lesen Sie nicht nur die jeweils angegebenen Seiten, sondern auch ein bißchen drumherum. Versuchen Sie, den hier gegebenen Zusammenhang neu – eben gegen den Strich – zu erfassen. Und wenn Ihnen eigene Gliederungsmöglichkeiten einfallen sollten, dann scheuen Sie sich nicht, diese hier (oder sonstwo) geeignet nachzutragen.

Befehle
Grundstruktur 22
direkte (Kommandos) 37
im Programm (Anweisungen) 37
Daten
Alphabet 272
Darstellungsform 115
Datenquelle 119
Datumsdarstellung 116
Felder 171
Zeichenkode 272
Zeichensatz 271
Daten verarbeiten
Datumsdarstellung 198
ganzzahligen Anteil berechnen 232
Klammern 30
logische Operationen 258
numerische Ausdrücke 23
Operatoren 23
Priorität von Operatoren 24, 29
Umwandlung in Großschreibung 308
Umwandlung von Zeichen in Zahlen 195
Vergleich von Zeichenketten 308
Zeichen in Zeichenkode umwandeln 277
Zeichenketten 186
Zeichenkode 272
Zeichenkode in Zeichen umwandeln 274
Datenausgabe
an einen Drucker 261
Befehl abschließen 22
Fertigzeichen 20
Leerzeile ausdrucken 61
Spalten 63, 66, 71
Text ausdrucken 70
Zeile unterdrücken 65
Dateneingabe
Anforderungsmeldungen 129
DATA und READ 45
Dialoggestaltung 253
Eingabeverhalten testen 121, 278
Grenzfälle 121

Menütechnik 139
über die Tastatur 119
Zeichenketten 134
Dokumentation
Allgemeines 112
Begleitdokumentation 77
Formulierung der Aufgabe 241
Komprimieren des Programms 76
Mehrfachbefehlszeilen 75
Programmierstil 79
Programmvorspann 318
REM (Anmerkungen im Programm) 152
REM (Nachteile) 159
Titelangaben im Programm 156
Variable 318, 330
Edieren des Programms 40, 43
externe Speicheroperationen
Ausgabe an eine Datei 299, 301
back-up copies 82
Dateien 294
Datenspeicherung auf Disketten 292
Datenspeicherung auf Tonband 267
Eingabe von einer Datei 298
Eröffnen von Dateien 295
Programm speichern 79
Schließen von Dateien 298
Sicherheitskopien 82
unmittelbarer Dateizugriff 302
Fehlermeldungen
Allgemeines 50
BS (Bad Subscript) 177
CN (continue) 128
FC (Function Call) 278
LS (Long String) 238
OD (out of data) 51
OS (Out of String space) 232
OV (overflow) 28
RG (RETURN ohne GOSUB) 166
TM (type mismatch) 132
Fehlersuche
Fehler mit Unterprogramm 161
FRE (Speicherplatzermittlung) 179
Programm gezielt anhalten 127
Programmabarbeitung fortsetzen 126, 128
Programmabbruch 125
Programmrückmeldungen 234
TRACE (Ablauf verfolgen) 162
– und Programmstruktur 151
Felder
Allgemeines 171
dimensionieren 177
dynamisch dimensionieren 231
Indizes 172
Platzreservierung im Speicher 174
sonstige Eigenschaften 184
Speicherbedarf 181
Speicherplatzbedarf 230
Zeichenketten, Northstar-Version 188
FOR-NEXT-Schleifen
Allgemeines 210
Schachtelung 214
Schleifenendbedingung 214
– zur Programmverzögerung 212
Programmbeschreibung
Flußdiagramme 81
Flußdiagrammsymbole 91
Form von Strukturbeschreibungen 97
Hierarchiediagramm 226
Programmoduln 226
Programmoptimierung 319
Schleifenstrukturen 95
statische und dynamische 228
Steuerbefehl GOTO 85
Struktogramm 94
Strukturbeschreibung 92, 226
Verhältnis der Möglichkeiten 323
Zuweisung 34
Programmiertechniken
Ablaufsteuerung des Programms 145
Bottom-up-Programmierung 102
Experimentieren 103
Hilfsprogrammbibliotheken 304
Lösungsansatz finden 102
Menütechnik 139, 222

Parameterübergabe 251
Programmoduln 226
Programmoptimierung 318
Regeln bei FOR-NEXT-Schleifen 214
Regeln bei Unterprogrammen 164
Top-Down-Kodierung 167
Top-Down-Programmierung 101
Unterprogramme 143
Verzögerungsschleifen 212
zum Vorgehen beim Programmieren 239
Programmtest
Allgemeines 323
Grundregel 324
– nach Gefühl 324
Parameter erfassen 335
Parameter gliedern 336
Testkontrolle 326
Testplan 334
Testprogramm erstellen 340
Wirkung von Steuervariablen 337
Programmvereinfachungen
Bedingte Sprünge 73
FOR-NEXT-Schleifen 213
LET entfällt 73
Mehrfachargumente 74
Mehrfachbefehle 75
Streichen von REM-Anweisungen 160
Programmvorbereitung
Datengruppen 114
Datenstruktur 168
Dialoggestaltung 253
Grundeinheiten eines Programms 106
Hilfsprogramme 304
Programm strukturieren 137
Speicherplatzaufteilung 229
Strukturieren des Programms 245
Teilprogramme planen 223
Überprüfen des Ansatzes 109, 111, 314
Unterprogrammparameter 251
zu verwendende Daten 113
Steueranweisungen
endlose Schleifen 123
FOR-NEXT-Schleifen 211
GOTO 56
IF ... THEN 60
IF ... THEN ... ELSE 216
On ... GOTO, ON ... GOSUB 220
Variable
alphanumerische 132
Felder 171
Matrizen 172
Namen in BASIC 33, 115, 182
numerische 30
Speicher löschen 236
Speicherbedarf 181, 231
Typ (Größenart) 132
Zahlen
Bereich 27
Dezimalpunkt 25
Form in BASIC 25
ganze 23
Stellenzahl 27
wissenschaftliches Format 26
Zeichenketten
Allgemeines 134
als Einheit betrachtet 187
als Feld betrachtet 187
– aus numerischen Feldern gewinnen 284
Dimensionierung 187
Länge 190
Northstar-Versionen 188
Position von Teilketten finden 205
Speicherbelegung 191
Speicherplatz initialisieren 236
Speicherplatzbedarf 235
Teilketten, Microsoft-Version 203
Teilketten, Northstar-Version 192
Umwandlung in numerische Felder 280
Vergleich von Zeichenketten 308
Zusammenfügen (Konkatenation) 208
Zuweisung an eine Teilkette 310
Zeilennummern
Allgemeines 37
– als Sprungziele 164

Bernd Pol

1946 in Freiburg im Breisgau geboren. Nach Volksschule und Lehre als Elektroniker Abitur über den zweiten Bildungsweg, danach u.a. Studium von Mathematik und Physik in Marburg. Kam 1963 das erste Mal mit Computern in Berührung, die ihn seitdem nie mehr so recht losgelassen haben. Die Beschäftigung mit Mikrocomputern seit 1977 führte zum eigentlichen Beruf: Seit 1978 in Stuttgart freiberuflich als Übersetzer und Fachjournalist für Elektronik tätig. Übersetzte, bearbeitete und verfaßte mehrere Bücher und Fachartikel zum Thema Mikrocomputer.

VOGEL-BUCHVERLAG WÜRZBURG

Kompetent für Technik · Elektronik · Management · Wirtschaft

Baumann, Rüdeger
Computerspiele und Knobeleien programmiert in BASIC

ISBN 3-8023-**0703**-8

Mit Eigeninitiative weg von der Spielkonserve: Der Leser wird zum aktiven und schöpferischen Umgang mit Computerspielen aufgerufen und angeleitet — aus der Spielidee entwickelt sich die Spielstrategie und hieraus das Programm. Das Programmieren des Computers selbst ist das Spiel; so lernt der Leser spielend das Programmieren.

Sacht, Hans Joachim
Von der passiven zur aktiven Computerei

ISBN 3-8023-**0665**-1

Jeder, der mit der aktiven Computerei beginnt, sollte vor dem Kauf eines Computers dieses Buch zu Rate ziehen: Hier wird auch das für jeden verständlich erklärt, was nicht in den Handbüchern der Hersteller steht. Er erfährt, wie man einen geeigneten Computer auswählt, ihn in Betrieb nimmt, mit ihm arbeitet und Programme in BASIC entwickelt.

ISBN 3-8023-0637-6

Ein Buch für den Praktiker und mehr als nur eine BASIC-Einführung: Anhand von zwei bis ins Detail ausgearbeiteten Fallstudien werden die Grundlagen des Programmierens verdeutlicht und die wichtigsten BASIC-Bestandteile eingehend und umfassend besprochen. Die zentrale Fragestellung lautet dabei: Wie ist ein Problem zu lösen? Warum ist das so formuliert? Und nicht zuletzt: Wie wendet man moderne Programmiertechniken mit BASIC an?

VOGEL-BUCHVERLAG
WÜRZBURG

Kompetent für Technik · Elektronik
Management · Wirtschaft