Einführung in die Programmiersprache „BASIC" (Teil 3)

Maximilian Herberger

Die heutige Folge setzt die Diskussion der Rechengenauigkeit mit zwei weiteren Beispielsprogrammen fort. Dabei werden die in der folgenden Übersicht zusammengestellten neuen BASIC-Elemente eingeführt und erläutert.

Übersicht 4: Erläuterte Elemente der Programmiersprache BASIC

DATA
END
GOSUB
IF . . THEN
INPUT
INT(X)
READ
RETURN

Außerdem wird das zweite Beispielsprogramm zum Anlaß genommen, wesentliche Fragestellungen der „Inkassoprogramm"-Entscheidung des BGH zu erörtern (vgl. IuR 1986, S. 18–23).

2.3.2 Das Programm PROG1.BAS

2.3.2.1 Mathematische Problemstellung

Im Mittelpunkt des Programms (vgl. Zeile 40) steht die folgende Formel:

y = 1/(1-(x-1)/x)

Vereinfacht man die rechte Seite, so sieht man, daß diese Formel eine umständlichere Schreibweise für y = x ist. Die Vereinfachungsschritte werden im folgenden angegeben, nicht weil sie mathematisch interessant wären, sondern um eine Vorstellung davon zu vermitteln, wie die Formelschreibweise in BASIC mit der ansonsten gewohnten zusammenhängt.

Übersicht 5

Schritt 1	Schritt 2	Schritt 3
y=1/(1-(x-1)/x)	in BASIC? (s. Übung 1)	in BASIC? (s. Übung 1)
$y = \frac{1}{1-\frac{x-1}{x}}$	$= \frac{1}{\frac{x-x+1}{x}}$	$= \frac{1}{\frac{1}{x}} = x$

Das Zeichen „/" steht in BASIC für die Division. Die Klammern sind notwendig, um komplexe Zähler und Nenner zu kennzeichnen, da der Bruchstrich nicht wie in der zweiten oben wiedergegebenen Schreibweise zur Verfügung steht.

Alle drei Formeln sind mathematisch vollständig gleichwertig und müssen aus diesem Grunde zu demselben Ergebnis führen.

Übung 1

Schreiben Sie die entsprechenden BASIC-Befehle für die Formeln in Schritt 2 und Schritt 3 in Übersicht 5.

(Auflösung unter 2.3.2.5.)

2.3.2.2 Listing des Beispielsprogramms 2: PROG1.BAS

```
10 REM Programm PROG1.BAS
20 REM Stand 16.4.1986
30 INPUT "Geben Sie bitte eine Zahl ein ! ";ZAHL
40 WERT = 1/(1-(ZAHL-1)/ZAHL)
50 PRINT "Sie haben eingegeben: ";ZAHL
60 PRINT "Der vom Programm errechnete Wert, der aus mathematischen"
70 PRINT "Gruenden mit der eingegebenen Zahl identisch sein muesste"
80 PRINT "betraegt: ";WERT
90 DIFFERENZ = ZAHL-WERT
100 PRINT "Die Ungenauigkeit fuer die von Ihnen gewaehlte Zahl ";ZAHL
110 PRINT "betraegt im Rahmen der Formel y = 1/(1-(x-1)/x)"
120 PRINT DIFFERENZ
130 PRINT "auf zwei Nachkommastellen gerundet: " USING "# # # #.# #";DIFFE-
    RENZ
140 PRINT "auf sechs Nachkommastellen gerundet: " USING
    "# # # #.# # # # # #";DIFFERENZ
150 END
```

2.3.2.3 Kommentar zum Listing von PROG1.BAS

Im folgenden werden nur die Zeilen erläutert, die Neues gegenüber PROG0.BAS bringen (vgl. IuR 4/1986).

Zeile 30

30 INPUT "Geben Sie bitte eine Zahl ein!" ,ZAHL

Der Befehl INPUT läßt den Text, der zwischen den Anführungszeichen steht, auf dem Bildschirm erscheinen und hält dann im Programmablauf an, da eine Eingabe erwartet wird. Gibt der Benutzer eine Zahl ein und betätigt die Taste ‹RETURN›, so wird die eingegebene Zahl als Wert der Variable „ZAHL" gespeichert, da diese Variable in Zeile 30 am Ende genannt ist.

(Vgl. bei Friederici z.B. Zeilen 70, 80, 90, 100, 120 u.ö.; IuR 2/86, S. 91)

Zeile 40

40 WERT = 1/(1-(ZAHL-1)/ZAHL)

Diese Zeile entspricht der oben besprochenen Formel mit dem einzigen Unterschied, daß andere Variablennamen gewählt wurden: „WERT" steht für y, und „ZAHL" für x. Die Auswahl möglichst „sprechender"

Variablenbezeichnungen ist ein wichtiger Beitrag zur Lesbarkeit von Programmen. Im Idealfall können Programme sich auf diese Weise (zusammen mit geeigneten Kommentaren) selbst dokumentieren.

Zeile 40 ließe sich aus den eben angedeuteten Gründen in mathematisch äquivalenter Weise folgendermaßen schreiben:

40 WERT=ZAHL

Diese Zeile sollte also bewirken, daß der Variablen „WERT" die eingegebene Zahl als Wert zugewiesen wird.

Übung 2

Schreiben Sie ein BASIC-Programm, in dem eine Zahl eingegeben wird und das dann den Ausdruck 1/(1-(ZAHL-1)/ZAHL) beginnend mit (ZAHL-1) schrittweise (mit Anzeige der Zwischenergebnisse) auswertet.

Zeile 90

90 DIFFERENZ=ZAHL-WERT

Da in Zeile 40 die eingegebene Zahl der Variablen „WERT" zugewiesen wird, müßte die Differenz zwischen ZAHL und WERT, wenn absolute Rechengenauigkeit bestünde, gleich 0 sein.

Zeilen 50–80 und 100–140

Diese Zeilen erläutern mit dem bereits bekannten PRINT-Befehl auf dem Bildschirm den Programmablauf und die Ergebnisse. Ein derartiger Programmteil ist ebenfalls eine Form, in der das Programm sich selber darstellt, allerdings nicht im Quellcode für dessen Leser, sondern auf dem Bildschirm für den Benutzer. Diese äußere Gestaltung des Programmablaufs ist ein wichtiger Aspekt der „Benutzerfreundlichkeit" und der „Bedienerführung". In der urheberrechtlichen Debatte sollte (wie bereits angedeutet, vgl. 2.3.1.3) berücksichtigt werden, daß die Struktur einer solchen „Benutzeroberfläche" mit den zugehörigen Texten (und gegebenenfalls graphischen Mustern) wie ein Handbuch Schutz verdienen kann. Im amerikanischen Recht hat man dafür die gelungenen Bezeichnungen „visual copyrights" bzw. „protection of the look and feel" geprägt (vgl. dazu James E. Bransfield, in: Byte, April 1986, S. 14ff.).

Zeile 150

150 END

Das END-Statement beendet die Durchführung eines Programms. Es unterscheidet sich von dem bereits behandelten STOP-Statement (vgl. 2.3.1.3) u.a. dadurch, daß im Laufe des Programms eröffnete Dateien geschlossen werden. (Näheres dazu bei der Erläuterung der Dateibehandlung.)

2.3.2.4 Diskussion der Ergebnisse von PROG1.BAS

In Übersicht 6 ist zu sehen, wie ein Ablauf des Programms PROG1.BAS bei Eingabe der Zahl 8 aussieht.

Übersicht 6

```
Sie haben eingegeben: 8
Der vom Programm errechnete Wert, der aus mathematischen
Gruenden mit der eingegebenen Zahl identisch sein muesste
betraegt: 8
Die Ungenauigkeit fuer die von Ihnen gewaehlte Zahl 8 betraegt
im Rahmen der Formel y=1/(1-(x-1)/x)
0
auf zwei Nachkommastellen gerundet: 0.00
auf sechs Nachkommastellen gerundet: 0.000000
```

Dieses Ergebnis entspricht dem mathematisch zu Erwartenden. Das ändert sich aber, wie Übersicht 7 zeigt, wenn der Benutzer nicht 8 (wie im Falle von Übersicht 6), sondern z.B. 88 eingibt.

Übersicht 7

```
Sie haben eingegeben: 88
Der vom Programm errechnete Wert, der aus mathematischen
Gruenden mit der eingegebenen Zahl identisch sein muesste
betraegt: 88.00008
Die Ungenauigkeit fuer die von Ihnen gewaehlte Zahl 88 betra-
egt im Rahmen der Formel y=1/(1-(x-1)/x)
-8.392334E-05
auf zwei Nachkommastellen gerundet: -0.00
auf sechs Nachkommastellen gerundet: -0.000084
```

Hier taucht in der fünften Nachkommastelle der Differenz, die 0 sein müßte, eine Rechenungenauigkeit auf. Diese Differenz wächst, je größer die eingegebene Zahl ist. Bei 888 etwa beläuft sie sich bereits auf -0.012268. Wenn es sich bei den Zahlen um Geldbeträge handeln würde, wäre man damit bereits im Pfennigbereich.

Woran liegt es nun, daß die beschriebene Ungenauigkeit auftritt? Es gibt dafür zwei Gründe.

Erstens wird der Bruch (x-1)/x gerundet. Das führt zu Rundungsungenauigkeiten (vgl. zur Erläuterung 2.3.4.1).

Zweitens kommt es innerhalb der Subtraktion (1-(x-1)/x) zu einem sogenannten „cancellation error" (Abbruchfehler). Dieser Fehler entsteht, wenn man zwei annähernd gleich große Zahlen voneinander subtrahiert. (x-1)/x hat nun aber immer einen nahe bei 1 liegenden Wert. Wird dieser innerhalb der Formel (1-(x-1)/x) von 1 subtrahiert, so ist die Fehlerbedingung für den „cancellation error" gegeben. Dies erklärt übrigens auch, warum die Ungenauigkeit mit steigendem x zunimmt (vgl. die Ergebnisse für 88 und 888). Im Falle von x=888 ist der Wert für (x-1)/x näher bei 1 als im Falle von x=88.

2.3.2.5 Einige Anmerkungen aus der juristischen Perspektive

Die oben (vgl. 2.3.1.5) erwähnten Gedanken treffen auch hier zu. Es gibt programmiertechnische Möglichkeiten, den Fehler zu vermeiden. Die einfachste Lösung wäre es, die beiden als Antwort auf Übung 1 gewonnenen Formeln zu verwenden. Diese sind:

a. y=1/((x-x+1)/x)
b. y=1/(1/x)

(Es geht dabei, um das nochmals zu wiederholen, nicht um die sinnvollste Lösung eines Problems. Praktisch würde man selbstverständlich den einfachsten Ausdruck wählen, nämlich y = x. Die Formeln dienen hier nur zur Demonstration der Bedingungen, unter denen eine bestimmte Art von Rechenungenauigkeit auftritt. Gleichzeitig eignen sie sich damit als Test für Rechengenauigkeit.)

Sowohl mit der Formel a. als auch mit der Formel b. verschwindet die Rechenungenauigkeit. Das liegt vor allem daran, daß die den „cancellation error" bewirkende Subtraktion nicht mehr vorhanden ist. Derartige Fehlerbedingungen zu vermeiden, liegt also unter bestimmten Umständen durchaus im Bereich des Möglichen, so daß sich der von einem Programmierer einzuhaltende Sorgfaltsmaßstab danach richten kann.

Die oben (2.3.1.5) ebenfalls angesprochene Idee, über Hilfsprogramme die Ungenauigkeitsbedingung zu vermeiden bzw. die Genauigkeit zu erhöhen, bietet gleichfalls einen Ausweg. Man könnte beispielsweise den Weg wählen, nur mit Brüchen zu rechnen (vgl. dazu das interessante Programm FRACT.BAS von Alan L. Zeichick, in: portable 100/200/600, vol. 3 no. 8 (April 1986), S. 45–50). Auf diese Weise könnten die Rundungsfehler umgangen werden.

Für die Bestimmung des juristischen Sorgfaltsmaßstabs ergeben sich aus diesen Beobachtungen einige Folgerungen. Abstrakt läßt sich auf Grund der Kenntnisse über die Struktur einer Programmiersprache und die rechnerinterne Zahlenbehandlung eine Liste möglicher Fehlerbedingungen aufstellen. Ob diese Liste vollständig ist, ist nicht feststellbar, weswegen immer wieder mit unerwarteten Fehlern zu rechnen ist. Das juristische Prinzip kann also nicht lauten „Vermeide alle möglichen Fehler", sondern nur „Vermeide die bekannten Fehlerbedingungen, soweit das möglich ist". Dadurch wird eine konkrete Betrachtungsweise erzwungen. Denn ob eine Fehlerbedingung umgangen werden kann, entscheidet sich im Rahmen des zu lösenden Problems auf der Grundlage der bekannten fehlerumgehenden Algorithmen. Es ist daher nur scheinbar ein Paradox, wenn man feststellt, daß ein (abstrakt) fehlerhaftes Programm bei konkreter Beurteilung als „fehlerfrei" einzustufen ist, weil eine Fehlervermeidung nach dem gegebenen Kenntnisstand nicht möglich ist. Als juristisch geboten könnte man dann allenfalls noch Hinweispflichten annehmen.

2.3.3 Das Programm PROG2.BAS

2.3.3.1 Mathematische Problemstellung

Wenn man mit mehr Nachkommastellen rechnet, als im Endergebnis erscheinen dürfen, stellt sich das Problem der Rundung. Das folgende Beispiel, das dem Programm PROG2.BAS zugrundeliegt, kann das veranschaulichen (s. Übersicht 8).

Man sieht, daß beide Rechenwege (im Rahmen der Vorgabe, ein zweistelliges Ergebnis zu erzielen) zu unterschiedlichen Ergebnissen führen. Mathematisch gibt es keinen Grund, einem von beiden den Vorzug zu geben. Das richtige Ergebnis lautet 5.165. Sowohl 5.16 (Ergebnis bei Methode 2) als auch 5.17 (Ergebnis bei Methode 1) sind von dem richtigen Ergebnis gleich weit entfernt. Anders als bei PROG0.BAS und PROG1.BAS geht es damit hier nicht um ein Problem der Rechengenauigkeit in dem Sinne, daß ein vom Programm ausgegebenes Resultat von dem mathematisch richtigen Wert abweicht. Denn für zwei Nachkommastellen ist bei Summanden mit drei Nachkommastellen gar kein eindeutig „richtiges" Ergebnis definiert. Man muß also normativ entscheiden, welcher Methode man den Vorzug gibt. Derartige „normative Rundungsregeln" bedürfen einer zusätzlichen Rechtfertigung, die etwa auf einer statistischen Annahme der Art beruhen könnte, daß sich Auf- und Abrundungen in großen Zahlenmengen ausgleichen.

Übersicht 8

Summand	Methode 1 Rechnung mit drei Nachkommastellen (Rundung des Ergebnisses auf zwei Stellen)	Methode 2 Rechnung mit zwei Nachkommastellen (Rundung jedes Summanden auf zwei Stellen vor der jeweiligen Addition)
1.011	1.011	1.01
1.022	+1.022	+1.02
1.033	+1.033	+1.03
1.044	+1.044	+1.04
1.055	+1.055	+1.06
	5.165	5.16
	gerundet: 5.17	

2.3.3.2 Listing des Beispielprogramms 3: PROG2.BAS

```
10 REM PROG2.BAS
20 DATA 1.011,1.022,1.033,1.044,1.055
30 REM Einlesen der Daten und Aufsummieren
40 FOR I=1 TO 5          'Beginn einer Schleife mit fuenf Schritten
50     READ X            'Einlesen eines Werts fuer die Variable x aus der DATA-Zeile
60     SX=SX+X           'Aufsummieren mit der eingelesenen Zahl
70     PRINT X,          'Ausdrucken der Zahlen mit drei Nachkommastellen
80     GOSUB 190         'Runden von X auf zwei Nachkommastellen
90     PRINT X1          'Ausdrucken der auf zwei Nachkommastellen gerundeten Zahlen
100    SX1=SX1+X1        'Aufsummieren mit der gerundeten Zahl
110 NEXT I
120 REM Ausdrucken der Summen
130 PRINT "----------"
140 PRINT SX,
150 PRINT SX1
160 PRINT USING "# #.# #";SX;
170 PRINT TAB(15) USING "# #.# #";SX1
180 STOP
190 REM Unterroutine zum Runden
200 XT=X*1000
210 Y1=XT/10
220 X1=INT(Y1)
230 DIFF=Y1-X1
240 B=10*DIFF
250 IF B>=5 THEN X1=X1+1
260 X1=X1/100
270 RETURN
```

2.3.3.3 Kommentar zum Listing von PROG2.BAS

Das Programm PROG2.BAS führt die unter 2.3.3.1 beschriebenen Berechnungen durch. Dabei erläutert sich das Hauptprogramm (Zeile 30–180) durch die darin enthaltenen Kommentare weitgehend selbst. Die folgenden Erläuterungen beziehen sich deshalb nur auf neue Elemente von BASIC und die Unterroutine zum Runden (Zeile 190–260).

Zeile 20

20 DATA 1.011,1.022,1.033,1.044,1.055

Diese Zeile stellt fünf Werte zur Verfügung, die im weiteren Verlauf des Programms eingelesen werden können. Dazu dient der READ-Befehl (vgl. Zeile 50).

(Vgl. bei Friederici Zeile 5010, 5040, 5060 u.ö.; IuR 2/86, S. 90.)

Zeile 50

50 READ X 'Einlesen eines Werts fuer die Variable x aus der DATA-Zeile

Bei jedem Durchgang durch diese Zeile wird der nächstfolgende Wert aus der DATA-Zeile gelesen und der Variablen X zugewiesen. (X erhält also beim erstenmal den Wert 1.011, beim zweitenmal den Wert 1.022 usw.) Man muß darauf achten, nicht öfter auf eine DATA-Zeile zuzugreifen, als Werte darin enthalten sind, da sonst eine Fehlermeldung auftritt. Die Leseschleife (Zeile 40 und 110) läuft deshalb hier nur über fünf Schritte.

(Vgl. bei Friederici Zeile 500; IuR 2/86, S. 91.)

Zeile 80

80 GOSUB 190 'Runden von X auf zwei Nachkommastellen

Mit GOSUB verzweigt das Programm in ein Unterprogramm, das in Zeile 190 beginnt. Das Unterprogramm hat hier die Aufgabe, die Zahl X auf zwei Nachkommastellen zu runden.

Unterprogramme sind besonders dann zweckmäßig, wenn dieselbe Aufgabe mehrmals in einem Programm zu erfüllen ist. Für das Runden ist diese Situation gegeben, wenn mehr als einmal in einem Programm Zahlen zu runden sind. In diesem Falle muß man nicht jeweils die Rundungsroutine aufführen; es genügt stattdessen, jeweils zum Runden in das entsprechende Unterprogramm zu springen. Unterprogramme können auf diese Weise ein Beitrag zur guten Strukturierung von Programmen sein.

Zeilen 190–270

Die Unterroutine zum Runden wird im folgenden in zwei Schritten erläutert: Zuerst werden die neuen Befehle und eine Funktion erklärt (2.3.3.3.1), anschließend wird beschrieben, wie man einen solchen Algorithmus schrittweise entwickeln kann (2.3.3.3.2). Durch die Darstellung des Algorithmus soll das Algorithmus-Konzept deutlich werden. Das ist im juristischen Kontext von Bedeutung, weil dort der Begriff „Algorithmus" verwandt wird, um etwas dem Urheberrechtsschutz Entzogenes zu kennzeichnen (vgl. BGH IuR 1986, S. 21). Das führt zu der Frage, ob damit das Algorithmus-Konzept der Informatik in Bezug genommen wird (vgl. dazu unten 2.3.3.5.2).

Die Unterroutine zum Runden in PROG2.BAS ist mit Hinblick auf den einführenden Charakter bewußt einfach gehalten. Es gibt eine Vielzahl anderer Möglichkeiten, Rundungsoperationen zu programmieren. (Vgl. z.B. Bernard H. Robinson, Structuring Basic Output With Field Statements, in: Programmer's Journal, vol. 4 no. 2 (March/April 1986), S. 43–47, 47.)

2.3.3.3.1 Die neuen Befehle in der Unterroutine

IF ... THEN (in Zeile 250)

Ein „IF ... THEN"-Befehl verknüpft eine Bedingung in der Weise mit einer Folge, daß der Folgeteil ausgeführt wird, wenn die Bedingung wahr ist. Im anderen Fall (Bedingung falsch) bleibt die Bedingung wirkungslos, d.h. die Konsequenz wird nicht realisiert. In Zeile 250 wird X1 um 1 erhöht, wenn der Wert der Variablen B größer oder gleich 5 ist.

INT(X)

Bei INT(X) handelt es sich um eine von BASIC zur Verfügung gestellte Funktion. Innerhalb einer Programmiersprache kann man sich eine Funktion als abgeschlossenes kleines Programm vorstellen, das mit der Funktionsbezeichnung aufgerufen wird und eine bestimmte Aufgabe erfüllt: „Funktion" ist „eine Prozedur, die in einer festgelegten Weise einen Wert liefert, der von einer oder mehreren Eingangsvariablen abhängt" (vgl. IBM, BASIC-Reference, BASIC 3.0, 1984, Glossary-8). Die Funktion INT(X) liefert den ganzzahligen Anteil einer Zahl als Ergebnis. INT(101.5) erbringt demnach das Ergebnis 101, INT(105.9) das Ergebnis 105 usw. Dieses Ergebnis kann man einer Variablen zuweisen: E = INT(105.9) bewirkt, daß E den Wert 105 bekommt.

RETURN (in Zeile 270)

Jede mit GOSUB angesprungene Unterroutine muß mit RETURN abgeschlossen werden. Dieses RETURN bewirkt die Rückkehr in das Hauptprogramm. Dort wird dann der auf das GOSUB folgende Befehl ausgeführt, d.h. im vorliegenden Fall der PRINT-Befehl in Zeile 90.

2.3.3.3.2 Der Algorithmus der Unterroutine

Ein Verfahren, mit dem man Zahlen mit drei Nachkommastellen auf zwei Nachkommastellen runden kann, läßt sich umgangssprachlich folgendermaßen beschreiben:

1. Schritt:
Betrachte die dritte Nachkommastelle der zu rundenden Zahl.
2. Schritt:
Wenn die dritte Nachkommastelle größer oder gleich 5 ist, dann erhöhe die zweite Nachkommastelle um 1.
3. Schritt:
Reduziere die so entstehende Zahl auf zwei Nachkommastellen.

Diese Beschreibung weißt schon ein wesentliches Merkmal eines Algorithmus auf: Das zu lösende Problem wird in einer endlichen Anzahl von Schritten gelöst. Bei näherer Betrachtung zeigt sich aber, daß die einzelnen Schritte noch nicht genau genug definiert

sind. „Betrachte die dritte Nachkommastelle der zu rundenden Zahl" ist als Anweisung nur mit Hilfe von Hintergrundwissen über den Aufbau einer Zahl ausführbar. Der Algorithmus muß also noch verfeinert werden. Man könnte beispielsweise den ersten Schritt folgendermaßen weiterentwickeln.

(Zum besseren Verständnis ist die Wirkungsweise für die Zahlen 1.011 und 1.055 jeweils in Klammern miterläutert. Die arithmetische Schreibweise ist die in BASIC übliche.)

1. Schritt
1.1 Multipliziere die zu rundende Zahl mit 1000.
(1.011*1000 = 1011)
(1.055*1000 = 1055)
1.2 Teile die so entstehende Zahl durch 10.
(1011/10 = 101.1)
(1055/10 = 105.5)
1.3 Nimm den ganzzahligen Anteil der so entstehenden Zahl.
(Ganzzahliger Anteil von 101.1 = 101)
(Ganzzahliger Anteil von 105.5 = 105)
1.4 Subtrahiere die so entstehende Zahl von der in Schritt 1.2 erhaltenen Zahl.
(101.1-101 = 0.1)
(105.5-105 = 0.5)
1.5 Multipliziere das Ergebnis aus Schritt 1.4 mit 10. Diese Zahl ist die dritte Nachkommastelle.
(0.1*10 = 1)
(0.5*10 = 5)

Diese Verfeinerung des ersten Schritts hat dazu geführt, daß an die Stelle der globalen Anweisung „Betrachte die dritte Nachkommastelle" präzise mathematische Anweisungen getreten sind, die Schritt für Schritt automatisch ausgeführt werden können und zu dem gewünschten Ergebnis führen.

Gemessen an den gerade genannten Kriterien bedarf auch Schritt 2 in der Grobformulierung des Algorithmus einer Verfeinerung, da dort von der „zweiten Nachkommastelle" die Rede ist. Man könnte meinen, auch diese Stelle müßte wie die dritte Nachkommastelle isoliert werden, damit eine präzise Behandlung möglich wird. Das ist jedoch so nicht notwendig, weil es einen einfacheren Weg gibt, wenn man das Ergebnis des Schrittes 1.3 aus der eben entwickelten Verfeinerung von Schritt 1 betrachtet:

Addiert man 1 zu 105 (in dem Beispiel, in dem aufzurunden ist), so entsteht 106. Man muß nun nur noch 106 durch 100 teilen, um 1.06 zu erhalten. Und das ist der gesuchte Wert mit zwei Nachkommastellen. Hiermit ist gleichzeitig auch Schritt 3 präzisiert.

Der Algorithmus kann auf diese Weise in verfeinerter Form folgendermaßen fortgesetzt werden:

2. Wenn die Zahl aus Schritt 1.5 größer oder gleich 5 ist, dann addiere 1 zu der Zahl aus Schritt 1.3.
(Keine Aktion im Falle der Zahl 1, da 1 kleiner 5 ist. Es bleibt bei 101.)
(Im Falle der Zahl 5 folgende Aktion: 105 + 1 = 106.)
3. Dividiere die Zahl aus Schritt 2 durch 100. Das Ergebnis ist die gesuchte, unter Berücksichtigung der dritten Nachkommastelle auf zwei Nachkommastellen gerundete Zahl.
(101/100 = 1.01)
(106/100 = 1.06)

Umgangssprachliche Formulierungen von Algorithmen sind notwendigerweise umständlicher als Formelschreibweisen und möglicherweise auch mehrdeutig. Man kann aber trotzdem jetzt schon sehen, daß die Unterroutine der Zeilen 190 — 270 genau den eben beschriebenen Algorithmus ausführt. Um das deutlich zu machen, sind in Übersicht 9 zu den einzelnen Zeilen Kommentare hinzugefügt, die auf den umgangssprachlich gefaßten Algorithmus verweisen.

Übersicht 9: Unterroutine zum Runden in PROG2.BAS und Rundungsalgorithmus

Algorithmus in BASIC	Algorithmus in nichtcomputersprachlicher Form
200 XT=X*1000	'Vgl. 1.1
210 Y1=XT/10	'Vgl. 1.2
220 X1=INT(Y1)	'Vgl. 1.3
230 DIFF=Y1-X1	'Vgl. 1.4
240 B=10*DIFF	'Vgl. 1.5
250 IF B>=5 THEN X1=X1+1	'Vgl. 2.
260 X1=X1/100	'Vgl. 3.

Der BGH beschreibt den Übergang vom in nichtcomputersprachlicher Form ausgedrückten Algorithmus zum in einer Computersprache gefaßten Algorithmus folgendermaßen:

„In der dritten Phase erfolgt die eigentliche Kodierung des Programms. In ihr wird der Programmablaufplan nunmehr in eine dem Computer verständliche Befehlsfolge umgewandelt. Diese Kodierung wird in der Regel zunächst unabhängig von der Maschinensprache des zur Verfügung stehenden Computers in einer Programmiersprache vorgenommen; das Ergebnis ist das für den Fachmann lesbare sogenannte Primär- oder Quellenprogramm" (IuR 1986, S. 20).

Eine urheberrechtlich bedeutsame Frage ist die, ob beim Übergang vom „sprachlichen" zum „computerspachlichen" Algorithmus noch eine individuelle schöpferische Leistung möglich ist. Der BGH hält es für denkbar, „daß im Einzelfall ein nicht hinreichend konkretisierter Programmablaufplan noch genügend Raum für eine individuelle Auswahl und Einteilung bei der Kodierung läßt" (IuR 1986, S. 21). Was damit gemeint ist, wird deutlich, wenn man die obige „Grobfassung" des Algorithmus mit der „Verfeinerung" vergleicht. Zwischen der „Grobfassung" und der „Kodierung" in BASIC liegen noch verschiedene Entwicklungsschritte. Bei der „Verfeinerung" hingegen ist ein Ausarbeitungsgrad erreicht, der die Kodierung „eins zu eins" vorzeichnet.

Das Verständnis des Ablaufs der Unterroutine wird erleichtert, wenn man die Ergebnisse der jeweiligen Variablenzuweisungen verfolgt (vgl. dazu Übersicht 10). Derartige tabellarische Übersichten sind generell ein geeignetes Instrument, um einen Programmablauf besser zu verstehen. Man kann sich bei der Programmentwicklung diese Informationen auch jeweils mit PRINT-Befehlen auf dem Bildschirm ausgeben lassen.

Übersicht 10: Die Ergebnisse der Variablenzuweisungen in der Unterroutine von PROG2.BAS

Wert von X bei Beginn der Unterroutine:	1.011	1.022	1.033	1.0441.055
Wert von XT:	1011	1022	1033	10441055
Wert von Y1:	101.1	101.2	101.3	101.41015
Wert von X1:	101	102	103	104105
Wert von DIFF:	0.1	0.2	0.3	0.40.5
Wert von B:	1	2	3	45
Wert von X1:	1.01	1.02	1.03	1.041.06

Bei der Entwicklung des in der Unterroutine verwendeten Lösungsverfahrens sind die wesentlichen Bestimmungsstücke eines problemlösenden Algorithmus deutlich geworden, die hier in Anlehnung an Knuth (mit Hinweisen zu PROG2.BAS) zusammengefaßt werden:

— Ein Algorithmus terminiert nach einer endlichen Anzahl von Schritten („finiteness").
  In PROG2.BAS sind das die in den Zeilen 200 — 260 enthaltenen Schritte, nach denen die Unterroutine verlassen wird.
— Die einzelnen Schritte eines Problemlösungsalgorithmus müssen so präzise beschrieben sein, daß bei ihrer Ausführung keine Zweifelsfragen auftauchen („definiteness").
  Wird der Algorithmus wie in PROG2.BAS in einer Programmiersprache ausgedrückt, so sichert deren präzise Syntax und Semantik die Eindeutigkeit des Algorithmus. Die Zeilen 200-260 sind eindeutig auf bestimmte Operationen bezogen.
— Der Algorithmus verarbeitet bestimmte Eingaben („input").
  In der Unterroutine von PROG2.BAS ist die Variable X der Input aus dem Hauptprogramm.
— Der Algorithmus gibt ein Ergebnis aus („output").
  Die Variable X1 ist der Output der Unterroutine von PROG2.BAS, der an das Hauptprogramm zurückgegeben wird.
— Das Ergebnis des Algorithmus ist die Lösung des Problems („effectiveness").
  Das Unterprogramm von PROG2.BAS löst das Problem, Zahlen mit drei Nachkommastellen auf zwei Nachkommastellen zu runden.

(Vgl. zum Algorithmusbegriff Donald E. Knuth, Fundamental Algorithms, vol. 1, Reading 1973, S. 4-6.)

2.3.3.4 Diskussion der Ergebnisse von PROG2.BAS

Ein Programmlauf von PROG2.BAS sieht auf dem Bildschirm folgendermaßen aus:

Übersicht 11: Von PROG2.BAS erzeugter Bildschirmausdruck

1.011	1.01
1.022	1.02
1.033	1.03
1.044	1.04
1.055	1.06
5.165	5.16
5.17	5.16

Dieser Ausdruck stellt das dar, was oben (2.3.3.1) in mathematischer Hinsicht besprochen wurde und deswegen nicht mehr näher erläutert werden muß.

2.3.3.5 Einige Anmerkungen aus der juristischen Perspektive

2.3.3.5.1 Rundungsprobleme in der Praxis

Das Programm PROG2.BAS macht auf ein Rundungsproblem aufmerksam, das in einigen frühen Fällen von Computerkriminalität eine Rolle gespielt hat. Die Wahl des Rechenweges „Runden bei jedem Schritt" (und nicht erst beim Ergebnis) führt in diesem Beispiel — interpretiert man die Zahlen als Geldbeträge — zu einer um einen Pfennig niedrigeren Summe. Sachverhalte dieser Art brachten Programmierer auf den Gedanken, diesen Pfennig für sich zu verbuchen. Ein derartiger Vorgang kann sich auch in entfernteren Nachkommastellen abspielen und trotzdem in der Summierung zu erheblichen Beträgen führen. Juristisch ist die Beurteilung dieses Vorgangs gar nicht so einfach, wie man sich ebenfalls an PROG2.BAS klarmachen kann. Denn die Rundungsmethode, die zu der niedrigeren Summe führt, kann ja nicht von vornherein als unkorrekt eingestuft werden. Die spezifisch juristische Natur der Pfennigdifferenz ist also durchaus der Überlegung wert.

(Vgl. dazu Walter Jaburek/Gabriele Schmölzer, Computerkriminalität, Wien 1985, S. 58.)

Allgemein sollte man aus PROG2.BAS die Anregung entnehmen, die Rundungsverfahren von Programmen genauer unter die Lupe zu nehmen. Es gibt Programmpakete für Anwälte, die in dieser Hinsicht problematisch sind. Rechnet man bei diesen Problemfällen die Summen in ausgedruckten Listen nach, so findet man, daß manchmal die Summanden zu einer anderen Summe als angegeben führen. Das ist dann (wenn man sonstige Programmierfehler ausschließen kann) ein Zeichen dafür, daß Rundungsprobleme vorliegen. Derartigen Rundungsproblemen ist übrigens nicht immer leicht auf die Spur zu kommen. Im Rahmen einer Podiumsdiskussion auf der Winter-COMDEX in Los Angeles berichtete Marr Haack von einem Versicherungsfall seiner Gesellschaft, bei dem eine Bank nach sieben Jahren hatte feststellen müssen, daß ihr Programm einen entsprechenden Fehler bei der Zinsberechnung machte. Der Schadensumfang war beträchtlich.

(Vgl. zum Problem der Rechengenauigkeit noch das Programm BEWARE, das Probleme im Zusammenhang mit der Berechnung der Standardabweichung demonstriert. Es ist abgedruckt bei Tonia Cope, Computing Using BASIC. An Interactive Approach, New York 1981, S. 258f.)

2.3.3.5.2 Algorithmus, Programm und Urheberrecht

In seiner Entscheidung zum Urheberrechtsschutz von Programmen hat der BGH eine in der deutschen juristischen Literatur nahezu unbefragt wiederholte These übernommen: Computerprogramme sind unter

den vom BGH angegebenen Bedingungen geschützt, nicht aber „die in dem Computerprogramm berücksichtigte, sich auf einen vorgegebenen Rechner beziehende Rechenregel (der sogenannte Algorithmus)" (IuR 1986, S. 21). Diese Unterscheidung ist unter zwei Gesichtspunkten außerordentlich problematisch, sollte der BGH hier „Algorithmus" im Sinne der Informatik (wie eben dargestellt) gemeint haben.

Den ersten Einwand kann man sich schon auf der Grundlage der simplen Unterroutine zum Runden in PROG2.BAS verdeutlichen. Denn dort entsprachen sich ja der umgangssprachlich formulierte Algorithmus in seiner endgültigen Fassung und das Unterprogramm „eins zu eins" (vgl. Übersicht 9). Ein Algorithmus kann also offensichtlich in verschiedenen Sprachen ausgedrückt werden. Die Wahl der Sprache richtet sich nach dem Adressaten: Soll ein Mensch den Algorithmus „mit Papier und Bleistift" ausführen, so empfiehlt sich eine natürlichsprachige Fassung. Soll ein Computer die Aufgabe durchführen, muß er in einer für ihn „verständlichen" Sprache „angesprochen" werden. Abgebildet wird aber in beiden sprachlichen Fassungen jeweils dasselbe Problemlösungs-„Programm". Ein Computerprogramm ist demnach nichts anderes als die Wiedergabe eines Problemlösungswegs in einer Computersprache.

(Knuth versteht „program" als „expression of a computational method in a computer language"; vgl. Fundamental Algorithms, vol. I, Reading 1973, S. 5. Auf dieser Grundlage ist es berechtigt, „Algorithmus" und „Programm" parallel zu behandeln. Vgl. z. B. die folgenden Definitionen: „algorithm" als „a series of instructions or procedural steps for the solution of a specific problem" und „program" als „a set of instructions composed for solving a given problem by computer"; Anthony Chandor (Hrsg.), The Penguin Dictionary of Computers, Middlesex 1983, S. 30, 323.)

Der Unterschied zwischen der natürlichsprachigen und der computersprachlichen Fassung eines Algorithmus reduziert sich also darauf, daß die natürliche Sprache Mehrdeutigkeiten und Vagheiten aufweisen kann, die beim Entwurf der künstlichen Computersprache ausgeschlossen wurden. Das ändert aber nichts daran, daß beide Sprachfassungen Abbildungen des gleichen Problemlösungsweges sind. Es ist nicht ersichtlich, mit welcher Rechtfertigung man eine dieser beiden äquivalenten Abbildungen vom Schutz ausschließen, die andere hingegen für schutzwürdig erklären kann. Denn der „geistige Gedankeninhalt", der „seinen Niederschlag und Ausdruck in der Gedankenformung und -führung des dargestellten Inhalts und/oder der besonders geistvollen Form und Art der Sammlung, Einteilung und Anordnung des dargebotenen Stoffs" findet (vgl. BGH IuR 1986, S. 21) realisiert sich bereits beim Entwurf des Algorithmus. Es hat den Anschein, daß die amerikanische Diskussion um die Patentierbarkeit „nicht-mathematischer Algorithmen" diesem Gedanken Rechnung trägt.

(Vgl. Henri W. Hanneman, The Patentability of Computer Software, Deventer 1985, S. 98 — 101, 98.)

Der zweite mögliche Einwand bezieht sich auf die Widerspruchsfreiheit des Urteils. Denn müßte man unterstellen, daß der BGH im Sinne der Informatik von „Algorithmus" spricht, würde sich das Urteil im strengen Sinne als widersprüchlich erweisen. Denn um die beiden ersten Phasen der Programmentstehung zu kennzeichnen, spricht der BGH von „genereller Problemlösung", die zu einem sprachlich gefaßten „Lösungsweg" führt, und von „Projektion der Problemlösung", die den „Lösungsweg" graphisch (z. B. mit Hilfe eines Fluß- oder Blockdiagramms) widergibt (IuR 1986, S. 20). Beide Stufen können, wenn die allgemeinen Bedingungen der Schutzwürdigkeit erfüllt sind, Urheberrechtsschutz beanspruchen (vgl. IuR 1986, S. 21 unter bb). Nun trifft aber auf eine „Problemlösung", die einen „Lösungsweg" beschreibt, die oben behandelte Definition eines Algorithmus zu, der — so der BGH — nicht schutzwürdig sein soll.

Als widerspruchsfrei aufrechtzuerhalten ist das Urteil damit nur, wenn man annimmt, daß der BGH den Begriff „Algorithmus" in einem abweichenden Sinne versteht. Dafür spricht immerhin der Zusatz „Rechenregel" (IuR 1986, S. 21). Dies schränkt aber die Tragweite der Feststellung, daß „Algorithmen" (im Sinne des BGH) nicht schützbar seien, erheblich ein und eröffnet den Weg für eine Diskussion der Frage, unter welchen Voraussetzungen „Algorithmen" (im Sinne der Informatik) Schutz beanspruchen können, gleichgültig in welcher Form sie zum Ausdruck gebracht worden sind.

Verschiedentlich wird die Unterscheidung von (nicht-schützbarem) Algorithmus und (prinzipiell schützbarem) Programm auch so erläutert, daß der Algorithmus lediglich die Lösungs-„Idee" (der „Inhalt") sei, das Programm hingegen die sprachliche Darstellung dieser Idee (die „Form"). Auf diese Weise wird die Regel vom nicht-schützbaren Algorithmus aber praktisch folgenlos. Denn Ideen dieser Art sind intersubjektiv außerhalb irgendeiner sprachlichen Fassung (sei sie nun in natürlicher oder symbolischer Sprache gehalten) gar nicht zugänglich. Das „Nicht-Schützbare" ist diesem Gedankengang nach für Außenstehende gar nicht faßbar, und der in einer Sprache zugängliche Algorithmus muß nach den angegebenen Kriterien auch auf Grundlage dieser Unterscheidung für schützbar gehalten werden.

(wird fortgesetzt)