So funktioniert der Turbo-Basic-Compiler

Soll ein Basic-Programm schneller werden, dann greift man einfach zum Basic-Compiler. Aber wie funktioniert ein solches Beschleunigungswunder?

Manche Anwendungen erfordern mehr Tempo, als reine Basic-Programme zu bieten haben. Dann schreibt man zeitkritische Routinen entweder in Maschinensprache oder setzt einen Compiler ein, der ein Basic-Programm in Maschinensprache umwandelt. Letzteres ist natürlich viel einfacher und bequemer, da es lediglich den Umgang mit dem Compiler vorraussetzt. Außerdem benötigt die Compilierung nicht viel Zeit. Das Compilat kann übrigens nicht mehr von Basic aus bearbeitet werden. Wenden wir uns nun der Funktionsweise des Turbo-Basic-Compilers zu.

Wenn Sie ein Basic-Programm eintippen, zum Beispiel die Programmzeile »100 ?3*4«, faßt der Computer diese Zeile zunächst nur als eine Folge von Zeichen im ATASCII-Format auf. Unter dem Begriff ATASCII versteht man die spezielle Atari-Version des ASCII-Zeichensatzes (ASCII ist die Abkürzung für: American Standard Code for Information Interchange). Der Interpreter wandelt den eingegebenen Text in Token um. Gleichzeitig überprüft er noch die Syntax, also ob die Befehlsfolge auch zulässig ist. Diese Aufgabe muß meistens auch ein Compiler übernehmen. Bild 1 zeigt, wie die Programmzeile »100 ?3*4« in Token übersetzt aussieht.

Die Programmzeile wird dann anhand ihrer Nummer einsortiert. Durch diese Umwandlung in Token und gleichzeitiger Syntaxprüfung wird die Aufgabe des Compilers etwas einfacher. Speichert man ein Basic-Programm mit dem SAVE-Befehl, werden auch die Token gespeichert. Mit LIST hingegen liegt es in der Textform vor. Der Turbo-Basic-Compiler kann nur die »SAVE-Ausführung« übersetzen. Ein Transfer der Text-Form wäre prinzipiell auch möglich, allerdings würde der Compiler hierfür mehr Zeit benötigen. Schließlich muß er die Befehlsnamen kennen und die Variablennamen speichern. Dies würde weiterhin die maximale Länge der zu compilierenden Programme vermindern. Oder es müßten Zwischenfiles auf Diskette angelegt werden, was bei den relativ langsamen Atari-Diskettenlaufwerken sehr zeitaufwendig ist.

Keine Variable ohne Nummer

Bevor Sie ein Basic- oder Turbo-Basic XL-Programm compilieren, sollten Sie sicherstellen, daß es fehlerfrei läuft. Zwar ist dann immer noch nicht gewährleistet, daß das Programm anschließend auf Anhieb ordnungsgemäß compiliert und ausgeführt werden kann. Zumindest entfallen aber Syntax-Fehler, die zur Folge hätten, daß Sie Ihr Basic-Programm korrigieren müßten, um es anschließend nochmals zu compilieren. Auch muß die Struktur Ihrer Basic-Programme stimmen. Dazu bietet sich der LIST-Befehl unter Turbo-Basic XL an, der bekanntlich FOR-NEXT-Schleifen und IF-ELSE-ENDIF-Abfragen einrückt.

Atari-Basic und Turbo-Basic XL-SAVE-Files bestehen aus einem Vorspann mit den Variablennamen und einigen anderen Informationen für den Interpreter. Diese werden deshalb vor dem Einlesen entweder überlesen (wie die Variablennamen) oder, in einer angepaßten Form, gespeichert. Dies betrifft in erster Linie die Variablentypen. Der Variablentyp kann aber nicht nur aus der Variablennummer, die im Programm gespeichert ist, definiert werden. Das Dollarzeichen ($) beispielsweise, welches die Stringvariablen kennzeichnet, wird vom Interpreter nicht im eigentlichen Programm gespeichert.

Bei langen Programmen mit vielen Variablen kann man diese Phase der Compilation deutlich erkennen. In der Info-Zeile (der dritten Bildzeile des Turbo-Basic-Compilers) steht dann nur das Wort Zeile ohne Zeilennummer. Sobald die erste Zeile übersetzt wird, erscheint auch die Anzeige der entsprechenden Zeilennummer.

Jetzt beginnt die eigentliche Compilierung. Sie setzt sich aus zwei Durchgängen zusammen. Im ersten Durchgang (Paß 1) wird das Programm Zeile für Zeile und Befehl für Befehl eingelesen und in Maschinencode übersetzt. Im zweiten Durchgang (Paß 2) werden dann noch die offenen Sprungadressen eingesetzt. Dazu werden im Paß 1 GOTO-Sprünge im Code durch einen ungültigen Maschinenbefehl mit angehängter Zeilennummer gekennzeichnet. Im zweiten Durchgang schließlich ersetzen dann 6502-JMP-Befehle (Code $4C) die ungültigen Maschinenbefehle.

Ähnliches gilt für Zahlen und Textkonstanten. Sie werden in einem Block hinter dem eigentlichen Basic-Programm gespeichert. Die endgültigen Adressen stehen dann erst nach der Übersetzung fest. Im Anschluß an den zweiten Compilier-Durchgang folgt dann noch eine Kontrolle, ob offene Schleifen vorliegen. Also ob ein NEXT zu einem FOR fehlt oder ein WEND zu einem WHILE etc. Trifft dies zu, erscheint eine Meldung auf dem Bildschirm. Dann folgt noch die Überprüfung nach fehlenden ENDIFs. Durch diese Aufteilung der Fehlerprüfung (die meisten Fehler werden bereits im ersten Durchgang festgestellt) werden die fehlerhaften Zeilennummern nicht komplett sortiert ausgegeben, sondern gegebenenfalls in vier einzelnen Gruppen.

Nach der Umwandlung in Token sieht die Programmzeile »100 ? 3*4« folgendermaßen aus:

0A 00 15 15 28 0E 40 03 00 00 00 00 24 0E 40 04 00 00 00 00 16

Dabei bedeuten:

0A 00	= Zeilennummer (100)
15	= Zeilenlänge
15	= Abstand vom Zeilenanfang zum Ende der ersten Programmzeile
28	= Token für »?«
0E 40 03 00 00 00 00	= Zahl 3
24	= Token für »*«
0E 40 04 00 00 00 00	= Zahl 4
16	= Token für Zeilenende

Bild 1. Eine Basic-Programmzeile, umgesetzt in Token

Die meiste Arbeit wird im ersten Durchgang geleistet, der Rest dauert nur wenige Sekunden. Das Einlesen des Programms erfolgt Zeile für Zeile. Zuerst zur Zeilennummer: Wenn die Zeilennummer größer als 32767 ist, dann ist Paß 1 beendet (der Interpreter verwendet die Zeilennummer 32768 übrigens als Endekennzeichen). Dann schließt der zweite Durchgang an. Ansonsten wird die Zeilennummer angezeigt und die Zeile Befehl für Befehl eingelesen und übersetzt.

Sollte der Compiler auf einen LIST- oder ENTER-Befehl stoßen, wird einfach eine Fehlermeldung mit Zeilennummer ausgegeben, und der nächste Befehl kommt an die Reihe.

Keinerlei Probleme bereiten REM- oder »--«-Zeilen. Diese Befehle werden einfach überlesen und ignoriert, da sie den Programmablauf nicht beeinflussen.

Befehle ohne Parameter (wie END, POP oder CLR) werden einfach zu einem entsprechenden Unterprogrammaufruf compiliert.

Bei einem DATA-Befehl wird der Text mit Zeilennummer und Länge in einer Tabelle im Anschluß an das Programm eingetragen. Sie nimmt noch folgende Informationen auf:

- eine weitere Tabelle mit konstanten Zahlen
- konstanten Strings
- die Anfangsadressen aller Programmzeilen
- den während der Compilation benötigten Stapelspeicher (der 6502-Stapel ist viel zu klein)

Diese Informationen werden gegebenenfalls auch im Speicher verschoben, wenn das zu compilierende Programm oder eine der Tabellen anwächst. Deshalb nimmt auch die Compilierungsgeschwindigkeit bei langen Programmen gegen Ende ab; vor allem wenn viele DATAs am Anfang des Programms stehen. Diese Verlangsamung betrifft lediglich die eigentliche Compilierung und nicht den Programmablauf.

Alle Befehle, die Parameter erfordern, sind wesentlich schwieriger zu übersetzen. Bild 2 zeigt nur einen Befehl mit einem einzigen Parameter.

Die Umwandlung des Befehls von Schritt 1 nach Schritt 2 hat der Interpreter schon erledigt. Der Compiler liest das Befehlstoken $03 für COLOR und verzweigt in die entsprechende Compilationsroutine. Diese ruft wiederum die Routine für die Übersetzung eines Ausdrucks auf, die eine Integerzahl zurückliefern soll. Dann wird nur noch »STA $C8« angehängt, und damit ist die Compilierung des Befehls schon erledigt. Beim Befehl GRAPHICS würde »JSR @Graphics« angehängt werden. Der Klammeraffe (@) kennzeichnet übrigens eine Routine in der Runtime Bibliothek.

Ein großer Fortschritt wurde bis jetzt noch nicht verzeichnet. Zwar sind die Befehle jetzt prinzipiell compiliert, das ändert aber leider nichts daran, daß mit dem entsprechenden Parameter, wie in unserem Beispiel die 1, noch nichts passiert ist.

Compilieren Zeile für Zeile

Die dafür benötigte Routine gliedert sich in mehrere Abschnitte:

- P-Code-Erzeugung
- P-Code-Optimierung
- Maschinencode-Erzeugung

P-Code (Pseudo-Code) ähnelt der Maschinensprache eines hypothetischen (gedachten) Prozessors. Manche Compiler erzeugen nur einen P-Code, der dann von einem kleinen Interpreter ausgeführt wird. Die Abarbeitungsgeschwindigkeit eines solchen Codes ist zwar schneller als ein herkömmliches Basic-Programm, allerdings von der Geschwindigkeit einer »richtigen« Maschinensprache noch weit entfernt. Aber platzsparender ist P-Code im Vergleich zu voll compiliertem Basic. Echte Maschinensprache ist jedoch von keinem Compiler zu übertreffen.

Bild 3 zeigt ein komplizierteres Beispiel. »03« entspricht wieder dem Token für COLOR, »46« steht für PEEK, »3A« für die offene Klammer, »2C« für die schließende Klammer, »35« steht für die Addition (+), »16« bedeutet Zeilenende oder auch einen Doppelpunkt, der Basic-Befehle in einer Zeile trennt. Der Wert »0E« kennzeichnet schließlich noch eine Zahl in der internen Darstellung (entspricht sechs Byte).

Das Token »03« ist in unserem Beispiel bereits identifiziert. Es beginnt jetzt der Test, in welcher Reihenfolge die Operationen ablaufen müssen. Der Interpreter hat dabei fast die gleichen Tests durchzuführen, und zwar jedesmal, wenn er diese Zeile erreicht. Der Compiler übersetzt jede Zeile nur einmal bei der Compilierung. Dadurch laufen die Programme schneller ab.

Bei der Übersetzung in P-Code bedient sich der Compiler einer Tabelle, in der die Prioritäten aller Operatoren vermerkt sind. Übrigens handelt es sich hierbei fast um die gleiche Tabelle, die auch der Interpreter benutzt. Daraus ergibt sich dann folgender P-Code. In Bild 4 sind die einzelnen Werte lediglich in Kurzform dargestellt. Der Compiler betrachtet sie natürlich als Zahlen.

Jetzt könnte theoretisch schon die Routine aufgerufen werden, die den Maschinencode erzeugt. Allerdings würden dann im compilierten Programm die gleichen Berechnungen ablaufen, die der Interpreter normalerweise auch durchführt. Schließlich steht jetzt viel Zeit zur Optimierung des P-Codes zur Verfügung, da die Programmzeilen nur ein einziges Mal in Maschinencode umgesetzt werden.

Hier nochmals der P-Code:

```
Zahl 41 02 00 00 00 00
fp->intpeek int->fp push
Zahl 40 01 00 00 00 00
movepull add fp->int
```

Im ersten Schritt faßt der Optimierer die Zahl »41 02 00 00 00 00 fp->int« zusammen, indem er den Wert in die Integer-Darstellung umwandelt. Daraus entsteht:

```
Izahl 200 peek int->fp push
Zahl 40 01 00 00 00 00
movepull add fp->int
```

Anmerkung: Es gibt zwei Fp-Accus, $D4 bis $D9 und $E0 bis $E5. Integerzahlen stehen dabei im Prozessorregister A und Y. Dabei enthält Y das höherwertige und A das niederwertige Byte.

Der Optimierer faßt jetzt »Izahl 200 peek« zu »Ipeek 200« zusammen. Das erste Beispiel würde »LDA #<200:LDY #>200:JSR PEEK« als Code erzeugen und das zweite Beispiel »LDA 200:LDY #0«. Jetzt haben wir also:

```
Ipeek 200 int->fp push
Zahl 40 0100 00 00 00
movepull add fp->int
```

Jetzt wird »push Zahl ... movepull add« zu »add# ...« zusammengefaßt. Daraus ergibt sich:

```
1) COLOR 1                  ;so sieht's aus
2) 03 0E 40 01 00 00 00 00  ;so speichert's der Interpreter
3) A9 01 85 C8              ;das soll daraus werden (LDA #1:STA $C8)
                            ;$C8=200 ist die Speicherzelle, in der sowohl
                            ;Interpreter als auch Compiler die Zeichenfarbe
                            ;für Plot ablegen
```

Bild 2. So wird der Befehl »COLOR« compiliert

```
03 46 3A 0E 41 02 00 00 00 00 2C 35 40 01 00 00 00 00 16
```

Bild 3. So sieht der Basic-Befehl »COLOR PEEK(200)+1« in Token umgesetzt aus

```
Ipeek 200 int->fp add
# 40 01 00 00 00 00 fp->int
```

Als nächstes bemerkt der Optimierer die Folge »int->fp Add# ... fp->int« und versucht diese Befehlsfolge zu vereinfachen. So spart man sich später eine Umwandlung. Da die Zahl kleiner als 256 ist, gelingt dies auch.

```
Ipeek 200 Iadd# 1
```

Jetzt hat der Optimierer seine Arbeit geleistet. Anschließend wird richtiger Maschinencode erzeugt.

Aus »Ipeek 200« entsteht »LDA 200:LDY #0«,
»Iadd# 1« wird zu »LDX #1:JSR IADD« (dieser Unterprogrammaufruf ist nötig, da sonst auch ein Overflow stattfinden könnte; Integerwert größer als 65535!)

Das war auch schon alles. An den Befehl COLOR wird noch »STA $C8« angehängt. Insgesamt ergibt sich also:

```
LDA 200:LDY # 0:LDX # 1:JSR
IADD:STA $C8
```

Ohne Optimierung würde wesentlich mehr Programmcode erzeugt werden, der noch dazu viel langsamer ist:

```
LDA #< Z200:LDY #>Z200:JSR
LDOYA:JSR FPI?:JSR PEEK:JSR
IFP:JSR PUSH
LDA #<Z1:LDY #> Z1:JSR
MOVEPULL:JSR ADD:JSR FPI?:STA $C8
```

Außerdem noch im Konstantenbereich:

```
Z200 .BYTE $41,$02,$00,$00,$00,
$00
Z1   .BYTE $40,$01,$00,$00,$00,
$00
```

Durch Optimierung läßt sich also viel Programmcode sparen. Gegenüber einem echten Maschinencode sieht das Ergebnis allerdings immer noch recht umfangreich aus. In Maschinensprache würde nämlich »INC $C8«, was nur 2 Byte benötigt, das gleiche bewirken. Hexadezimal $C8 entspricht dabei dem Dezimalwert 200. In unserem Beispiel codiert der Compiler den Basicbefehl COLOR 1 jedoch optimal, nämlich »LDA #1:STA $C8«.

Optimal compiliert

Allerdings kann der Compiler nur in den seltensten Fällen optimalen Code erzeugen. Schließlich erübrigt sich nur die Umwandlung von FP nach INT und umgekehrt und dies auch nur bei Verwendung von Konstanten. Bei Variablen ist diese Art der Optimierung nicht zulässig. In den meisten Fällen reicht schon die Einsparung einiger Bytes, weil zum Beispiel das Laden einer Variablen und die Umwandlung in einen statt zwei Integerwerte durch einen einzigen Unterprogrammaufruf erledigt wird.

Auch wird nicht alles, was möglich wäre, optimiert. Zum Beispiel berechnet sich »A=10/3« nicht im voraus, obwohl es durchaus sinnvoll wäre. Aber solche Optimierungen kann der Programmierer gegebenenfalls selbst vorwegnehmen. Allerdings sind solche Fälle viel zu selten, als daß sich dies hier lohnen würde. Je länger nämlich der Compiler, desto kürzer die entsprechenden Basic-Programme. Es gibt jedoch einige Berechnungen, deren Optimierung sich besonders lohnt: »^2« erhält eine eigene, schnelle Routine. Im Compiler benötigt »A=B^2« die gleiche Zeit wie »A=B*B«. Es existiert ebenfalls eine schnelle Exponential-Routine, die bereits im Mathematik-Teil der Runtime-Bibliothek vorhanden ist. So oder ähnlich werden alle Befehle mit Parametern behandelt. Sind mehrere Parameter vorhanden, werden diese natürlich zwischengespeichert.

Ein weiterer lohnenswerter Befehl ist

```
Zahl 41 02 00 00 00 00      ;die 200
fp->int peek int->fp        ;der PEEK-Befehl benötigt einen Integer als Adresse,
                            ;deshalb muß die 200 zuerst umgewandelt werden.
                            ;Dann folgt der eigentliche PEEK-Befehl und dann
                            ;die Umwandlung des Ergebnisses in eine
                            ;Floating-Point-Zahl (fp). Diese
                            ;Umwandlungen muß der Interpreter auch immer
                            ;durchführen.
push                        ;retten des Ergebnisses auf den Rechenstapel
Zahl 40 01 00 00 00 00      ;Zahl 1
Movepull                    ;Schiebt die Zahl 1 aus dem FP-Accu 1 in den
                            ;FP-Accu 2 und holt das Ergebnis des PEEKs
                            ;in den FP-Accu 1
Add                         ;die Addition
fp->int                     ;dies wird von der Compilationsroutine für Color
                            ;angehängt, es wird ja ein Integer benötigt
```

Bild 4. Der P-Code, der sich aus »COLOR PEEK(200)+1« ergibt

der POKE-Befehl. Beispielsweise wird »POKE 16,64« durch »LDA #64:STA 16« übersetzt.

Beim PRINT-Befehl sind noch die Kommata zu beachten. Auch INPUT, READ, GET und LOCATE beinhalten noch einige Probleme. Eine ausführliche Erläuterung würde den Rahmen dieses Artikels jedoch sprengen.

Gesondert zu behandeln sind alle Befehle, die den linearen Programmablauf beeinflussen. Der GOTO-Befehl läßt sich vergleichsweise einfach übersetzen. Zuerst wird die Routine aufgerufen, die einen Integer-Ausdruck übersetzt und optimiert (wie oben); allerdings generiert sie ihn noch nicht in den endgültigen Maschinencode. Zunächst wird nur der P-Code erzeugt. Dann wird anhand des P-Codes überprüft, ob eine Zahl folgt. Wenn ja, wird ein JMP-Befehl compiliert. Zunächst nur »07 Zeile«, woraus sich nach Paß 2 »4C Adresse« ergibt. Wenn nein, wird die Routine für die Erzeugung von Maschinencode aufgerufen und »JSR GOTOX« angehängt. Bei einem berechneten GOSUB wird beispielsweise entweder »JSR GOSUBX« oder vor dem »JMP-($07)«-Befehl der GOTO-Anweisung ein Unterprogrammaufruf compiliert. Dieser enthält dann, auf dem Basic-Stack, die um drei Werte erhöhte Rücksprungadresse des Befehls. Es wird aber nicht einfach ein JSR compiliert, da sonst die Anzahl der Unterprogrammebenen durch den zu kleinen 6502-Stack begrenzt wäre. Außerdem müssen die Parameter für die FOR-NEXT-Schleifen ebenfalls auf diesem Stapel abgelegt werden. Sie belegen jeweils 16 Byte, so daß sehr schnell ein Overflow des 6502-Stack drohen würde. Dies könnte katastrophale Folgen haben und die Kompatibilität zum Interpreter-Basic stark einschränken. Zumindest ist die untere Hälfte der Seite 1 frei, und es gibt Programme, die mehr als 10 geschachtelte Schleifen verwenden. Der Zeitverlust durch diesen simulierten Stack hält sich jedoch in Grenzen.

Einen ähnlichen Stack verwendet der Compiler auch zur Übersetzung von DO-LOOP, REPEAT-UNTIL, WHILE-WEND und FOR-NEXT-Schleifen sowie des EXIT-Befehls.

Bei DO wird die aktuelle Adresse auf diesem Stapel abgelegt. Sie erhält außerdem noch die DO-Befehlskennung. Bei Übersetzung des zugehörigen LOOP wird die Kennung daraufhin überprüft, ob ein DO fehlt und dann ein JMP-Befehl auf die zuvor festgestellte Adresse compiliert. Sollte zwischen DO und LOOP ein EXIT gestanden haben, so hat dieser Befehl seine eigene Kennung auf dem Stack hinterlassen und einen JMP-Befehl mit »Dummy«-Adresse erzeugt. LOOP ersetzt diese Adresse durch die auf den eigenen JMP-Befehl folgende.

Geschwindigkeit ist Trumpf

Bei REPEAT ist der Vorgang ähnlich wie bei DO. Es unterscheidet sich lediglich die Kennung. Wenn UNTIL auf eine REPEAT-Anweisung folgt, wird zuerst der folgende Ausdruck übersetzt. Erst dann wird ein bedingter Sprung angehängt. Da der 6502 nur relativ kurze, bedingte Sprünge kennt, wird mit »BNE *+5:JMP ad« gearbeitet. Natürlich muß auch UNTIL eventuelle EXIT-Befehle berücksichtigen.

WHILE-WEND entspricht dem eben Besprochenen, nur steht hier die Bedingung am Schleifenanfang.

Bei FOR-NEXT-Schleifen ist noch zu beachten, daß zusätzlich der Basic-Stack benutzt wird. Deshalb wird automatisch ein POP-Befehl in den EXIT-Befehl eingeschlossen.

Damit ist die Wirkungsweise des Compilers erklärt. Vielleicht ist Ihnen aber noch nicht ganz klar, wie der Interpreter und der Compiler die Reihenfolge der Berechnungen feststellen. Der Interpreter führt die Berechnungen aus, der Compiler erzeugt einen P-Code. Trotzdem funktionieren beide nach dem gleichen Prinzip. Zur Zwischenspeicherung der Operatoren dient jeweils ein Stack (Stapel). Dazu ein einfaches Beispiel:

```
8*Sin(A)+16
```

Der Compiler liest die Ziffer 8. Als Zahl wird sie sofort compiliert. (Der Interpreter würde sie auf dem Zahlenstapel ablegen.) Dann wird das Token für Multiplikation (*) mit der Priorität 5 gelesen. Da der Stapel für Operatoren noch leer ist (Priorität 0), wird das Token dort abgelegt. Als nächstes folgt SIN mit der höchstmöglichen Priorität (9). Weil diese größer ist als die auf dem Stapel (5), wird auch SIN auf den Stapel gelegt. Dann folgt die geöffnete Klammer. Als Spezialfunktion ruft die Klammer dieselbe Routine noch einmal, allerdings rekursiv auf. Es wird also der Wert in Klammern entsprechend übersetzt oder berechnet. Dann kommt die Variable A an die Reihe. Variablen werden, genauso wie Zahlen, sofort compiliert. Dann schließt als Endekennzeichen die Klammer ab. Weiter geht es mit der ursprünglichen Übersetzung. Es folgt nun das Token für Addition (+), dessen Priorität (4) kleiner als die auf dem Stapel ist. Deshalb wird jetzt der SIN-Befehl compiliert. Anschließend wird noch die vorliegende Priorität mit der des »*«-Tokens (5) verglichen. Da sie wieder kleiner ist, wird jetzt die Multiplikation compiliert.

Der leere Stack hat die Priorität 0, also wird das Token für Addition auf dem Stack abgelegt. Dann folgt die Zahl 16 und schließlich das Endekennzeichen mit der Priorität 0. Der Stapel ist anschließend leer, die Addition wird compiliert, und der Vorgang ist beendet. Der entstandene P-Code lautet »Zahl 8 Var A Sin Mult Zahl 16 Add«.

So einfach wie hier beschrieben, ist das Compilieren von Basic-Programmen jedoch nicht. Es sollte an dieser Stelle vielmehr gezeigt werden, wie der Turbo-Compiler prinzipiell funktioniert. Sollten alle Vorgänge beim Compilieren beschrieben werden, würde dies ein Buch füllen. Noch ein Tip: Das Standard-Atari-Basic ist wegen eines Betriebssystem-Fehlers nicht in der Lage, »A=NOT NOT A« zu berechnen. Dies hängt mit der Stapelverarbeitung zusammen. Probieren Sie diese Befehlsfolge doch einfach einmal auf Ihrem Atari 800XL aus. Aufgrund der Syntaxkontrolle ist der Befehl nicht zulässig. Die alten Atari-Computer reagierten auf »A=NOT NOT A« mit einem Systemabsturz. Geben Sie die Programmzeile doch einfach einmal unter Turbo-Basic-XL ein. Sie werden sehen, daß auch solche komplexen, logischen Verknüpfungen ordnungsgemäß funktionieren. (Frank Ostrowski/wb)