A BYTE BOOK

Top-Down BASIC for the TRS-80 Color Computer

by Ken Skier

Top-Down BASIC for the TRS-80 Color Computer

By Ken Skier

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland
Bogotá Hamburg Johannesburg London Madrid
Mexico Montreal New Delhi Panama Paris
São Paulo Singapore Sydney Tokyo Toronto

Library of Congress Cataloging in Publication Data

Skier, Ken.

Top-down BASIC for the TRS-80 color computer.

Bibliography: p.

Includes index.

1. TRS-80 color computer – Programming.

2. Basic (Computer program language) I. Title.

QA76.8.T183S55 1982 001.64'24 82-22880

ISBN 0-07-057861-3

1234567890 DOC/DOC 89876543

ISBN 0-07-057861-3

Edited by Bruce Roberts.

Design and Production Supervision by Ellen Klempner.

Production Editing by Tom McMillan and Peggy McCauley.

Typeset by LeWay Composing Service, Inc.

Printed and Bound by R.R. Donnelley and Sons.

INTRODUCTION

You own a Radio Shack TRS-80 Color Computer? Great! Then you're probably already familiar with Color BASIC. If you have a 16K Color Computer, you've had a chance to explore *Extended* Color BASIC, too. These versions of BASIC are very powerful, and the Color Computer manuals that came with your system show you how to write programs in these languages. *Getting Started With Color BASIC* and *Going Ahead With Extended Color BASIC* are well written books. If you work through them, you can congratulate yourself on completing an introductory course in computer programming.

Congratulations! But you don't have your computer science degree, yet.

There's more to computer programming than writing and entering a bunch of statements that do something when you type RUN. That's *part* of programming — an essential part — but an experienced programmer knows that getting a program to RUN is just one step in the process. Other steps are equally important.

To develop as a programmer, you must master these other steps. This book will guide you through the overall *process* of programming by showing you:

- How to *design* a program from the *top down*.
- How to *structure* your programs so that you can *maintain* and *extend* them.
- How to write *readable* programs that are easy to *understand*.
- How to write programs that are *user-friendly:* easy to *use* and hard to *misuse*.
- And — oh, yes — how to write programs that *RUN*.

This book does not replace any books you may already have on the BASIC programming language. Rather, it builds on what you have learned from those books, enabling you to develop programs in a much more organized manner.

I assume you already know many of the common BASIC commands and functions and that you've mastered the syntax of BASIC. Never heard the word "syntax"? Don't worry. If you can write a short BASIC program that runs without error, then you've mastered the syntax of BASIC.

But you've still got a lot to learn about the *process* of programming.

You see, this is not simply a book about BASIC; rather, it's a book about program *design*, intended for the BASIC programmer.

Let me make an analogy. The BASIC books you've already read should have taught you the building blocks of BASIC, just as a class in carpentry can teach you about nails and screws and wood. But carpenters don't design buildings and bridges; architects do. An architect needs to understand the nature of materials, but not as an end in itself: he needs it in order to design elegant structures that work.

When you mature as a programmer, you also become an architect — an architect of computer programs. This book will teach you how to design program structures, even very large ones, that work. And just as an architect defines elegance in terms of *usefulness* and *simplicity*, so shall we define elegance by the usefulness and simplicity of the programs we design.

So, this is not a course in carpentry. It's a course in architecture — the architecture of computer programs.

Welcome to the class!

You may be wondering what I mean by top-down design, structured programming, and the readability of code. I'll introduce you to those concepts in Chapter 1, but only so you'll become familiar with the terms. You won't really understand those concepts until you read through the succeeding chapters, where I'll *show* you what those ideas mean in practice by exemplifying them in substantial programs.

Will those chapters simply present a number of finished programs that embody those principles? Nope. Instead, those chapters will actually *walk you through* the processof programming, showing you how to begin with just an idea of what you want a program to do, and then how to refine that idea, defining the structure of the program, until, finally, you can *code* the program in BASIC. As you walk through this process for a number of different applications, you will come to understand its step-by-step nature and may ultimately make a similar process part of the way that *you* write programs.

Let's get started.

WARNING

Displaying the same image on your television set for an extended period of time — say, more than 15 minutes — can *burn the phosphor on your picture tube*.

If you burn the phosphor on your TV set, the burned-in image will remain on the tube *forever*.

The author and publisher disclaim any responsibility for such consequences.

To reduce the risk of burning the phosphor on your picture tube, *turn down the brightness* on your TV set. The lower the brightness level, the longer it will take to burn the phosphor.

Be especially wary of any program whose display is essentially unchanging, in whole or in part of the picture tube. Some of the programs in this book have such displays. (The rim of the clock program in Chapter 9, for example, never changes.)

Therefore, never leave a display on the screen unless you're watching it. If you're leaving the room, turn off the TV set or turn it to another

channel, a channel over which your television set may receive a normal broadcast signal. Even better, turn off your Color Computer system altogether.

I've never burned the phosphor on my TV set, and I've run some of these programs for hours at a time, but TV sets and tubes vary, so be forewarned.

COPYRIGHT

CASSETTES AND DISKS

If you would rather not type the programs listed in this book into your Color Computer, you may purchase them on cassette or disk from:

SkiSoft, Inc.
P.O. Box 379
Cambridge, MA 02238-0379

For information on ordering these programs, see Appendix 5.

TRADEMARKS

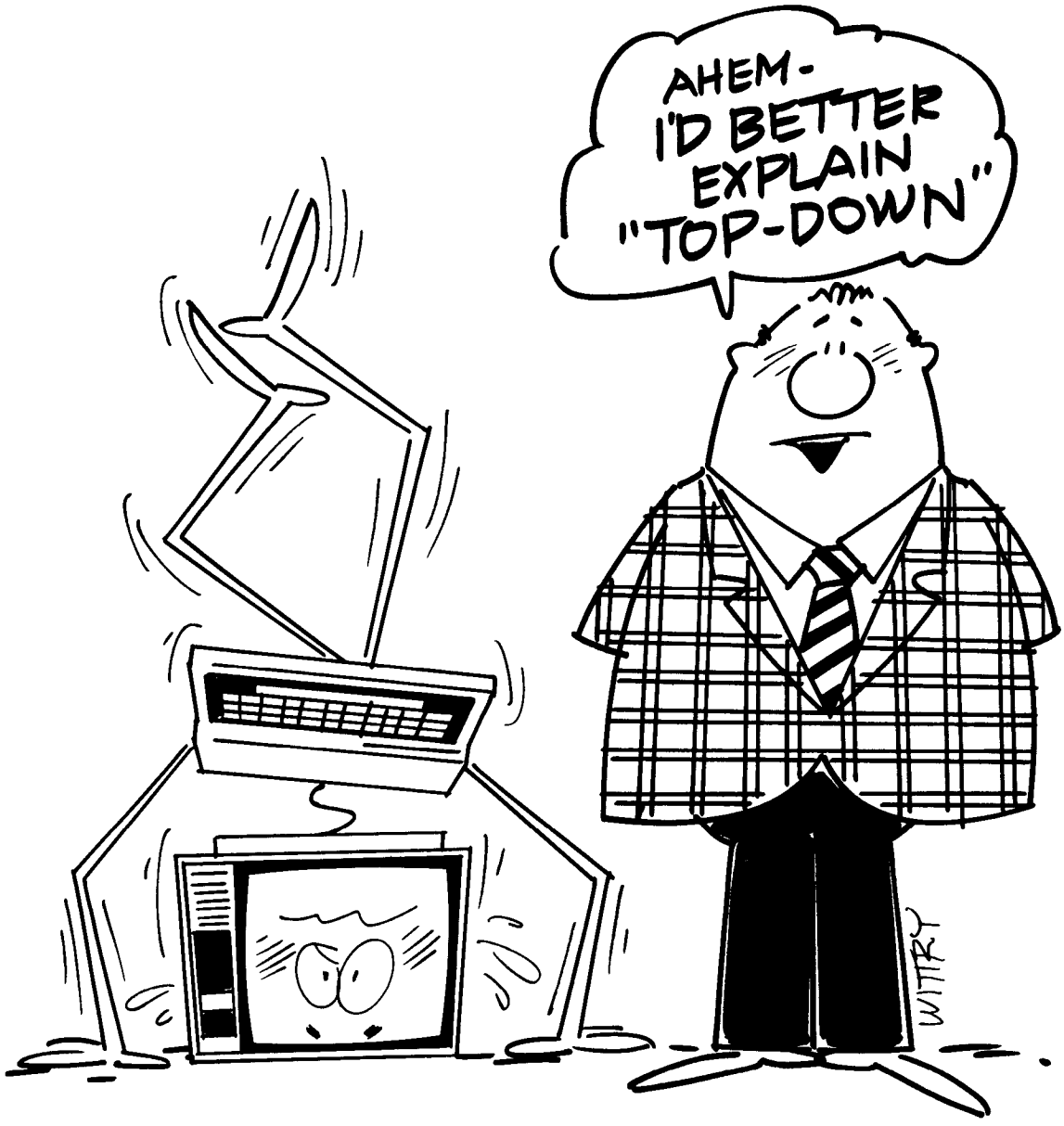
AHEM-
I'D BETTER
EXPLAIN
"TOP-DOWN"
WITTRY

TABLE OF CONTENTS

Chapter 1:

WHAT IS TOP-DOWN, USER-FRIENDLY, READABLE PROGRAMMING?

You may have heard the term "structured programming" or the phrase "program structure." Although we haven't defined them, you may already have a sense of what they mean. Just as a book is more than a bunch of words, and a house is more than a bunch of bricks, so is a program more than just a bunch of statements in some computer language.

Every program has a purpose, and whenever we create something with a purpose, the thing we create has some form, a *structure*, designed to accomplish that purpose. This is true of anything people create, from houses to hand tools, from bicycles to blimps. When the human mind creates something with a purpose, that thing has a suitable structure. Or, as Frank Lloyd Wright's mentor Louis Sullivan said, "Form follows function." He was talking about architecture, but what he said was true of everything the human mind creates, including computer programs.

TOP-DOWN DESIGN

A house may be composed of many small things such as bricks, nails, boards, shingles, and panes of glass. But if you are an architect designing a house, you won't start by thinking about the thousands of shingles and nails that will go into it. Instead of thinking of these many small items, you'll think in terms of the larger, structural units of the house: the foundation, the walls, the roof. Only when you've sketched out a picture of the house showing these large, structural units will you go into detail and design the smaller items from which you will build the foundation, floors, etc. Eventually you'll design each wall, doorway, and window, but that fine, detailed work comes last, after the broad outline is complete.

What I've just described is called *top-down design*, which begins with the broadest possible outline, or sketch, and moves ultimately to the most detailed design, or blueprint. Only the last, most detailed blueprint concerns itself with the smallest things, the discrete units that go into the house: the shingles, nails, bricks, and panes of glass.

THE USER INTERFACE

Similarly, you can design a program from the top down. Begin by thinking about the things that a program should do, and write down those things in the order in which they should occur. Don't go into detail describ-

ing the things your program should do; just jot down brief descriptions or labels.

For example, let's say you want to write a program that calculates the square and cube of a given number. You might write down the following list of things the program will do:

CUBE-SQUARE CALCULATOR

GET A NUMBER FROM THE USER.
CALCULATE ITS SQUARE.
CALCULATE ITS CUBE.
PRINT THE RESULTS.

Note that this program description reads like English, not like BASIC. Of course, this program description could be used by someone to write a BASIC program, but it could equally well be used to write a program in FORTRAN, in machine language, or in any other programming language. The program description is *not* a program; it's a general description of what the program does *from the point of view of the user*. It doesn't matter whether we implement that program description in BASIC or in FORTRAN or in any other computer language; when we RUN the program the person using it will see *exactly the same things happen*.

In the computer industry, we have a name for what you see and do when you're running a program. We call it the *user interface*. When a professional programmer sits down to design a program, it is likely that he or she will begin by defining the user interface. After all, you can't write a program to do something unless you know exactly what the program is supposed to do.

Many beginning programmers overlook this step and launch right into writing lines of BASIC or some other program language. That would be as misguided as an architect starting to design a house by drawing a picture of two boards and shingle. You can't even begin to think about boards and shingles until you know whether you're designing a ranch or a colonial or an apartment house. You must know how the occupants intend to use the house before you can design its fundamental structure. And you must understand its fundamental structure before you can figure out where the boards and shingles go.

The same thing is true of programming. Form follows function.

FLOWCHARTS

So a programmer writes down the user interface (or it is given to the programmer as a *specification* to be fulfilled) and the programmer looks at it for a while:

CUBE-SQUARE CALCULATOR

GET A NUMBER FROM THE USER.
CALCULATE ITS SQUARE.
CALCULATE ITS CUBE.
PRINT THE RESULTS.

. . . and then the programmer takes a pencil and starts to draw lines about each of the things the program is supposed to do. See Figure 1.1.

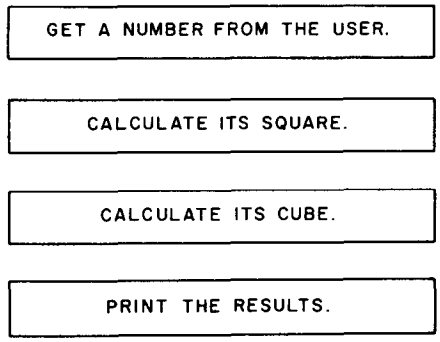

Figure 1.1: *Cube-Square Calculator.*

Having drawn boxes about each of the things the program is supposed to do, the programmer joins the boxes with vertical lines, indicating the start and end of the program with labeled circles. (See Figure 1.2.)

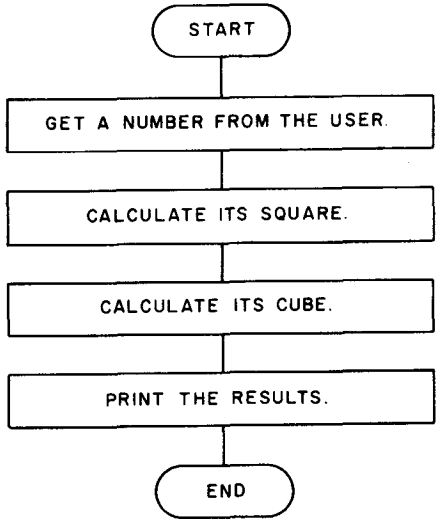

Figure 1.2: *Cube-Square Calculator, step two.*

If you're the programmer who's drawn these boxes and circles and connecting lines, you won't call it a program description or a user interface

anymore; you'll call it a *flowchart*. Each of the boxes represents a thing that the program should do: an *operation*. The flowchart shows the sequence of operations, or the entire *process*, using connecting lines to guide your eye from box to box, thus revealing the *order* of operations.

Like a program description, a flowchart is not a program itself and may be turned into a program written in *any* computer language. BASIC programmers draw flowcharts, as do machine-language programmers, FORTRAN programmers, and many other programmers who want to see graphically the things a program will do and the order in which it will do them.

In a way, flowcharts are similar to board games in which you move a piece, or token, from square to square. In a flowchart, the computer executing the program performs the operation described in one square and then moves on to the next.

How do we know which is the *next* square? We always assume that the flow of control is *down* (when a vertical line connects two boxes) or to the *right* (when a horizontal line connects two boxes). If the flow is ever *up* or to the *left*, that unconventional direction must be indicated explicitly with one or more arrows (as you'll see when we begin developing more advanced program structures involving loops and decision nodes). No arrows are needed when the flow from one box to the next is down or to the right.

STRAIGHT-LINE CODE

Figure 1.2 shows what we call *straight-line* code, because the connecting lines all go in a straight line, with one action following another from start to end. Straight-line code is the simplest of all program structures.

Given that you've drawn a flowchart such as Figure 1.2, how would you turn it into a program? First you must decide what program language you will use. I'll assume that you wish to write it in Microsoft Color BASIC, for the TRS-80 Color Computer. Now look at each box in the flowchart, and write one or more lines of BASIC that will accomplish the action stated in the box.

Listing 1.1 shows one implementation of Figure 1.2 as a BASIC program.

```
100 INPUT X
110 Y=X*X
120 Z=X*X*X
130 PRINT Y
140 PRINT Z
150 END
```

Listing 1.1: *Cube-Square Calculator.*

That's a terrible program!

About all you can say for it is that it works, if you know exactly what to expect when you use it. But that's a poor review for any program. What

makes this piece of code so terrible is that it's hard to use and hard to understand. Or, to use the terminology of the computer industry, it's not *user-friendly* and it's not *readable*. We'll discuss the issue of readability shortly, but now let me show you why the Cube-Square Calculator in Listing 1.1 is not very user-friendly.

USER-FRIENDLINESS

A friendly program, like a friendly person, introduces himself and lets you know what he can do for you. What does the program in Listing 1.1 tell you? Nothing.

Run the Cube-Square Calculator in Listing 1.1 and a question mark appears on the screen. That's it. No title, no introduction, no prompt. Just a question mark. If you're at the keyboard, you must *know* that a question mark means the computer is waiting for you to INPUT something. But even if you know what a question mark means, it doesn't help you much. What does the program want you to INPUT? You can't tell. Unless you know in advance what the program wants, you must guess. It displays no prompts, no hints.

But let's say you know that the program wants you to INPUT a number (as opposed to a string). You type in, say, 5 and press (ENTER).

In an instant, the Color Computer prints two numbers directly underneath yours. The screen now looks like Figure 1.3.

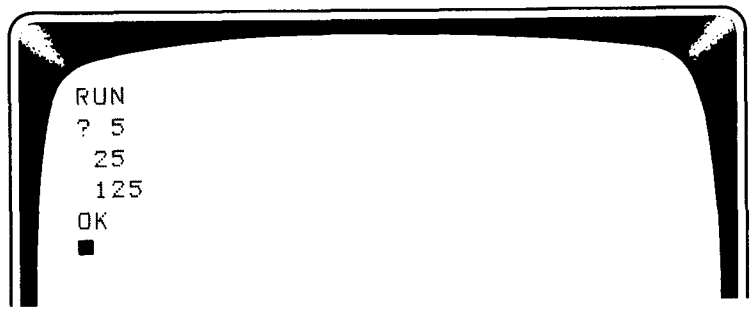

Figure 1.3

What do these numbers represent? You don't know. The program doesn't say. That's not user-friendly; it's downright inscrutable.

Let's make our Cube-Square Calculator friendlier to the user. See Listing 1.2.

```
100 CLS
110 PRINT "I'LL CALCULATE THE SQUARE"
120 PRINT "AND CUBE OF A NUMBER."
130 PRINT
140 PRINT "TYPE IN A NUMBER AND PRESS ENTER"
150 INPUT X
160 Y=X*X
170 Z=X*X*X
180 PRINT "THE SQUARE OF "; X; " IS "; Y
190 PRINT "THE CUBE OF "; X; " IS "; Z
200 END
```

Listing 1.2: *Cube-Square Calculator with title and prompt.*

This version of the Cube-Square Calculator is much friendlier to the user. Let me demonstrate. When you type RUN and press (ENTER), it clears the screen and introduces itself, telling you what it can do for you:

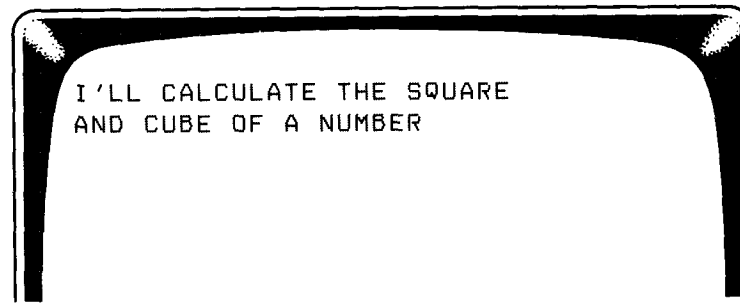

Figure 1.4

Then it goes on to tell you what *you* should do:

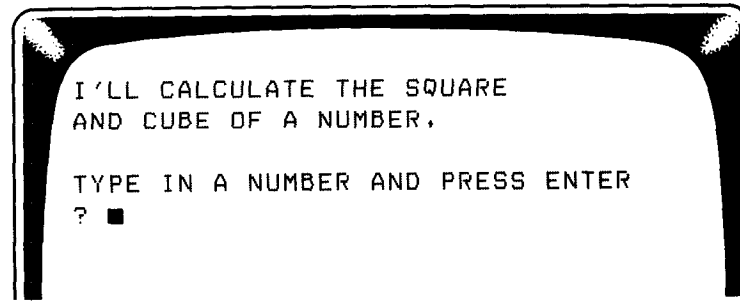

Figure 1.5

You follow its instructions by typing in, for example, 5 and pressing (ENTER).

The program then performs its calculations and prints its results. It doesn't just print a mysterious pair of numbers, it tells you what they mean.

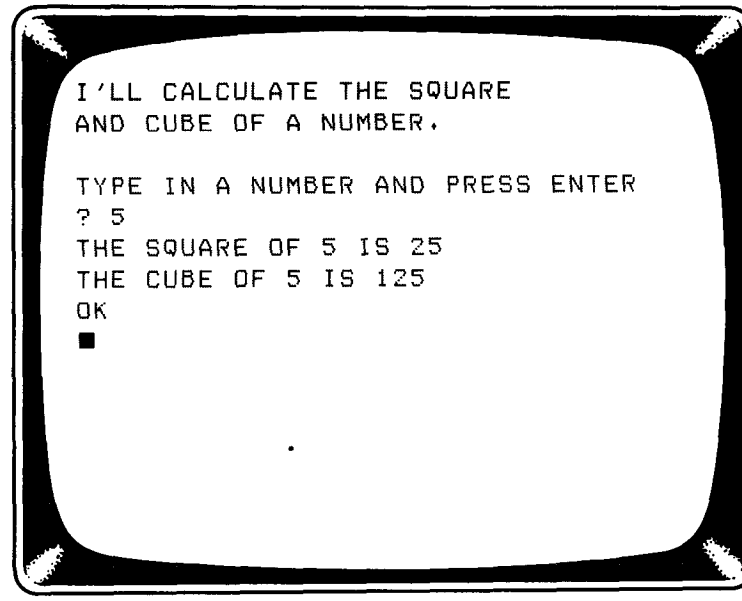

Figure 1.6

THE SQUARE OF 5 IS 25 . . . THE CUBE OF 5 IS 125 — that's clear! Polite. *User-friendly*. Even if you have no idea what this program will do, once you type RUN and press (ENTER), you'll have no trouble using it. For example, the prompt TYPE IN A NUMBER AND PRESS ENTER is designed so that someone who has *never* used a computer can enter the number properly. If the prompt said only INPUT THE NUMBER or ENTER THE NUMBER, a novice might type the number but never get around to pressing (ENTER). The prompt in Listing 1.2, however, is so precise that it virtually holds your hand, telling you what keys to press.

Listing 1.1 is a program that only its author could use, and then only for a brief period before forgetting what it does, what input it expects, and what its output means. In contrast, Listing 1.2 presents a program that even someone who's never seen it before can use. That's the best test of user-friendliness: a user-friendly program can be used without difficulty by a novice.

To use the best programs — the most user-friendly — *you need NO documentation and NO training*.

Now let's get to the issue of readability.

READABILITY

No doubt you've heard the expression, "coding a program" as in "I'll be down to dinner in a minute! I'm coding a program." But coding a program is not like coding a CIA communiqué for a secret agent in the field. A secret agent's code is meant to be incomprehensible, so it will reveal its meaning only when deciphered by the intended party. But coding a program is something quite different. Coding a program means putting it into a form that the computer will understand. That's necessary, but it doesn't mean you have to make it hard for *people* to understand, too.

When you write a program you are writing it not just for a machine but for *human* readers as well — most importantly, for yourself. After all, if you can't understand a program you've written, who else can? And if you can't understand it, how can you know what lines to change if it turns out the program has a bug, or if — as often happens — you wish to modify it at some later date to make it serve different functions?

With smaller programs, readability might not matter, but with larger programs it becomes essential. After a program grows to a certain size — say, twenty lines — you will probably never make it work if you can't read it easily. So readability is not just an admirable but nonessential goal like good penmanship: it's a top priority if you are to write programs that *work*.

What do I mean by readability?

A program is readable if you can LIST it and read through the listing without wondering, "What is the purpose of this program? What is its structure? What does this variable represent? Why is this line doing this? Where is this program going?" These reactions are all variations of the question, "What's going on here?" If you can read through a program without finding yourself asking that question, then you know you're looking at a readable program. The more often you ask that question, the less readable the program.

You've been reading English for a long time now, and you know that you can understand a well-written paragraph by reading it once; you don't have to reread it. A good writer works long and hard at his or her craft, so that people won't have to reread. What is true of English is equally true of BASIC. You should be able to LIST a program, read it once, and understand it. You may reread certain lines in order to gain a greater appreciation of what they do, but you won't have to read the whole program twice before you begin to comprehend it.

What makes a program readable? There are many factors. Nor is there *one proper way* to write a readable program, any more than there is one proper way to write a readable paragraph. There is such a thing as an author's *style* in a program listing as well as in an essay or a work of fiction. There are different programming styles, each of which may result in a readable listing. Here are some factors that I consider important.

- Remarks
- Variable names
- Line numbering
- Multiple statements per line
- Indentation
- Use of Subroutines

Let's take a look at each of these factors.

Remarks

Imagine that you've never seen the Cube-Square Calculator before, and take another look at Listing 1.2. Can you tell what purpose the program serves? That should be obvious, because the first few lines of the program tell you:

```
100 CLS
110 PRINT "I'LL CALCULATE THE SQUARE"
120 PRINT "AND CUBE OF A NUMBER."
```

We might call the two PRINT statements in lines 110 and 120 a form of *internal documentation*, documentation that exists as *part of the program*. Internal documentation is one way to make a program readable. But you can't always document a program internally with PRINT statements. What should you do then? Use remarks. Remarks can make a program understandable to the programmer, who LISTs the program. They will never be seen by the user, who RUNs the program.

What happens when the BASIC interpreter encounters a remark? Nothing! It ignores the remark and anything that follows the remark on that line. From the computer's point of view, there are two kinds of remarks: REM and apostrophe ('). Listing 1.3 demonstrates how the Color Computer ignores remarks that consist of REMs.

```
100 REM  THIS IS IGNORED.
110 PRINT "HI THERE."
120 REM PRINT "HOW ARE YOU?"
130 END
```

Listing 1.3: *Demonstration of "REM" remarks.*

When you run this program, the computer prints:

```
HI THERE.
```

But it doesn't print

```
HOW ARE YOU?
```

Instead, after printing `HI THERE` it prints:

```
OK
```

— to indicate that the Color Computer is once again in the immediate mode. The Color Computer did nothing at all with lines 100 and 120, because each of those lines begins with a REM.