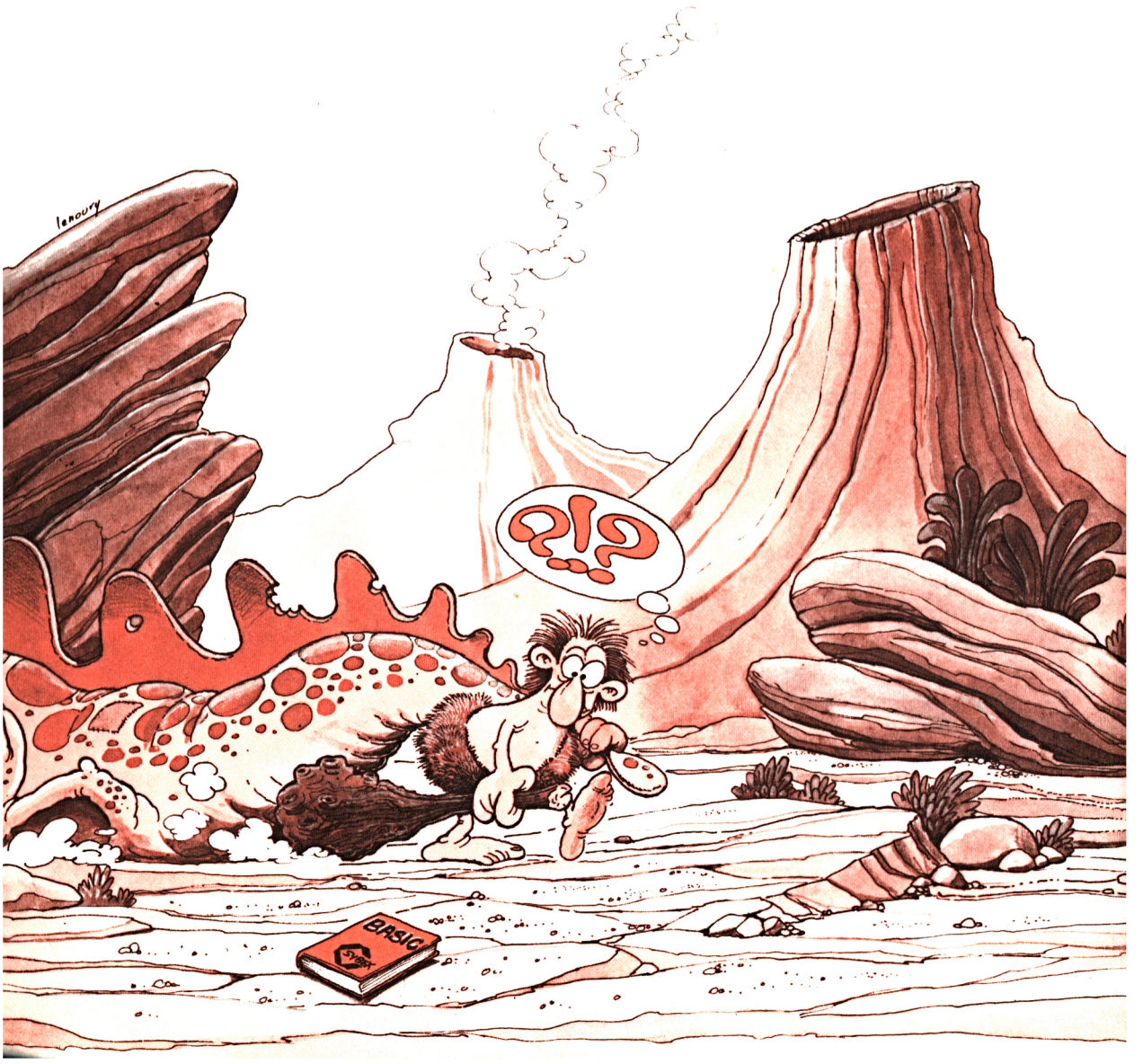
SYBEX

Your First Amstrad Program



Rodnay Zaks

"Yes . . . YOU can!"



Your First Amstrad Program

Your First Amstrad Program

Rodnay Zaks

SYBEX

BERKELEY · PARIS · DÜSSELDORF

The author is indebted to the many persons of the SYBEX editorial and production departments who have watched every detail of the preparation of this book and have contributed to its final appearance. In particular, Salley Oberlin, Richard King, and Eric Novikoff have contributed many valuable improvements to the form and contents of the original manuscript, verifying and challenging assertions, statements, and programs, thus contributing in a major way to the clarity and accuracy of this book.

All efforts have been made to supply complete and accurate information. However, SYBEX assumes no responsibility for its use, nor for any infringements of patents or other rights of third parties which would result.

Titel of the American edition on which this edition is based:
YOUR FIRST COMMODORE 64 PROGRAM
Original Copyright © 1983 by
SYBEX Inc., Berkeley, USA
Adaption: Wolfgang Dederichs, Hattingen
Cover design and cartoons:
Daniel Le Noury / tgr
Book design: Einar Vinje
Layout: tgr – typo-grafik-repro gmbh, remscheid

All English language rights for the adapted version reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN 0-94788-800-4
1st Edition 1985
Printed in Great Britain
Copyright © 1985 by SYBEX-Verlag GmbH, Düsseldorf

1

Speaking BASIC

Introduction 19

Programming 20

The BASIC Interpreter 22

What is BASIC? 24

Which BASIC? 25

Your Computer 26

Of Computers and Syntax 31

2

Communicating with Your Computer

Introduction 33

Using the Keyboard 34

Speaking BASIC 38

A Longer Program 46

Summary 51

Exercises 52

3

Calculating with BASIC

Introduction 55

Printing Numbers 56

Scientific Notation 57

Doing Arithmetic 58

Printing Formats 61

Application Example 63

Summary 64

Exercises 64

4

Memorizing Values and Using Variables

Introduction 67

The INPUT Statement 68

The Two Types of Variables 71

Assigning a Value to a Variable 77

The Variable Counter Technique 82

Summary 84

Exercises 85

5

Writing A Clear Program

Introduction 86

The REM Statement 88

Multiple Statements on a Line 89

Using Spaces 90

Improving the Display 92

Shortcut Input 92

Selecting Variable Names 94

Proper Line Numbering 94

Summary 97

Exercises 97

6 Making Decisions

Introduction 99

The IF Statement 100

An Arithmetic Test 108

The GOTO Statement 112

IF Statement Revisited 114

Counting Ones 115

Arithmetic Test Revisited 117

Validating the Input 117

Mileage Conversion 118

Birthday 119

Summary 120

Exercises 120

7 Automating Repetitions

Introduction 123

The IF/GOTO Technique 124

The FOR . . . NEXT Statement 127

Sum of the First n Integers 129

Tables of Values 130

Lines of Stars 131

Advanced Looping 132

Additional Features 136

Summary 137

Exercises 137

8 Creating a Program

Introduction 139

Algorithm Design 140

Flowcharting 144

Coding 152

Debugging 154

Documentation 156

Summary 158

Exercises 159

9

Case Study: Metric Conversion

Introduction 161

Designing the Algorithm 162

Flowcharting 162

Coding 169

Testing 175

Summary 178

Exercises 179

10

The Next Step

Introduction 181

What Can You Do With BASIC? 182

Improving Your Skills 182

More BASIC 183

Conclusion 187

Appendices

Answers to Selected Exercises 189

AMSTRAD BASIC Keywords 194

BASIC Glossary 195

Index 198

Preface

Hundreds, perhaps thousands, of books have been written about BASIC.

Why another one? Simply because the audience has changed. In the past, using programming languages such as BASIC was the privilege of the few who had access to computers. Programmers were a small elite group. This is no longer the case. Personal computers have made BASIC the most widely-used and most accessible computer language ever, and the majority of computer users today have little or no technical background. They use computers for fun, education, business, or their profession.

This book addresses this new group of users. Not only does it *look* different, but it *is* different. It is intended for the beginner and thus assumes that the reader has no prior technical knowledge. This book is for everyone—whether aged 8 or 88—who wants to learn quickly, how to get started in BASIC with the AMSTRAD computer. The author believes that all new computer users who want to learn how to write their own programs in BASIC are enthusiastic and young, or young-at-heart. They want a simple, straightforward, educational approach to learning BASIC. That is the approach of this book: It is designed to make learning BASIC on the AMSTRAD easy and fun.

Furthermore, this book aims at teaching you the essentials of BASIC in just a few hours. You should be writing your first BASIC program within one hour. And within a few hours, you should know enough to start writing useful and meaningful programs.

Time is passing . . . let's begin.

The author wishes you a pleasant journey along the magical path to knowledge.

Rodnay Zaks
Berkeley, November 1984

How To Read This Book

This is an educational book. You should read every chapter in sequence and understand each one before you continue on to the next. I have provided exercises at the end of each chapter to help you test your new skills. Do as many of them as you can. Answers to selected exercises are provided in Appendix A at the end of this book.

If you have an AMSTRAD computer, try all the programs. To truly learn and remember, you must practice and experiment. This book will bring you the skills and knowledge you need to get started—but remember, nothing can substitute for experience.

The main goal of this book is to get you started programming your AMSTRAD quickly and effectively. To achieve this goal and make your path simple, I had to make choices: This book does not describe every feature or concept of BASIC—just the important ones.

It is my hope that with this book you will understand everything quickly, write your first BASIC programs in no time, and appreciate and enjoy the power of your new programming skills.

What You Will Learn

Chapter One explains the language of computers and introduces you to the heroes of this book: the Computer, the Interpreter, the Program, the Instructions, and other characters of the cast.

Chapter Two shows you how to communicate with your computer, using the resources of the keyboard and the display. You will learn to type your first BASIC programs and execute them.

Chapter Three shows you how to perform calculations with BASIC.

Chapter Four helps you write programs that can be used repeatedly. In addition, you will learn how to use variables correctly and effectively.

Chapter Five shows you how to make your programs clear and readable.

Chapter Six shows you how to make complex decisions based on logic and values.

Chapter Seven explains how to automate repetitive tasks, using program loops.

Chapter Eight shows the correct method for designing a program—from the algorithm to the working, documented program—including designing the flowchart.

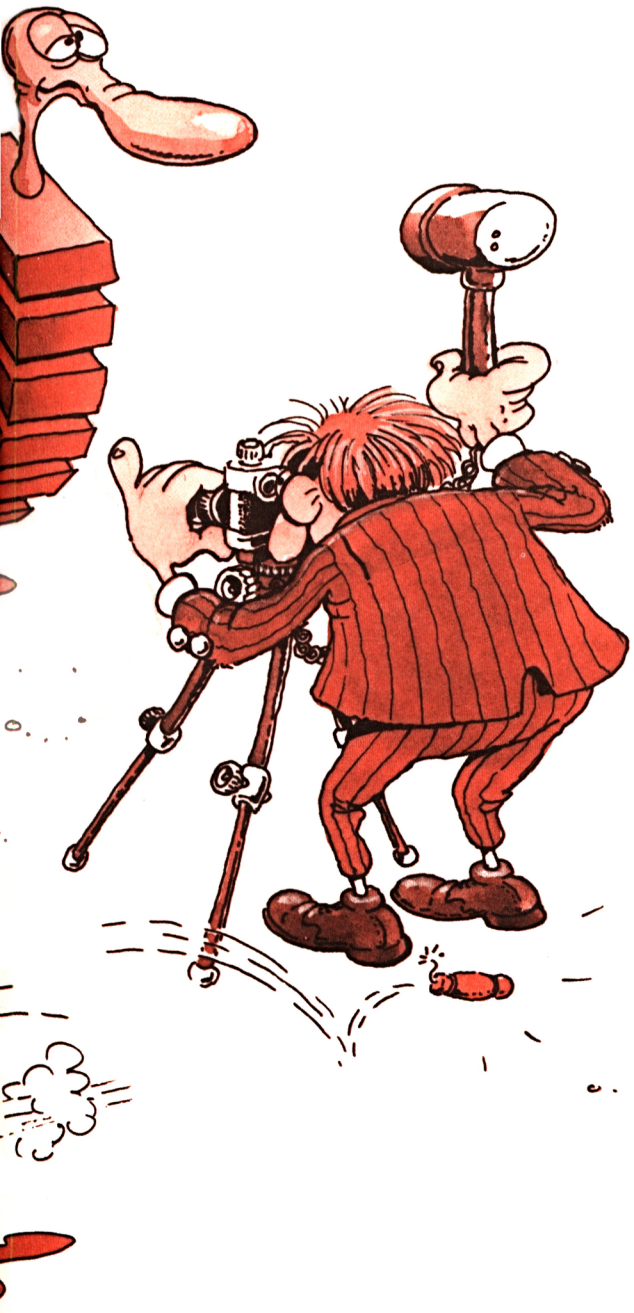
Chapter Nine helps you apply all these concepts to a practical case study.

Chapter Ten helps you examine the next steps to programming expertise.

The Appendices A,B, and C offer answers to selected exercises, a list of keywords, a glossary, and a bibliography that will guide you on your path to advanced knowledge.

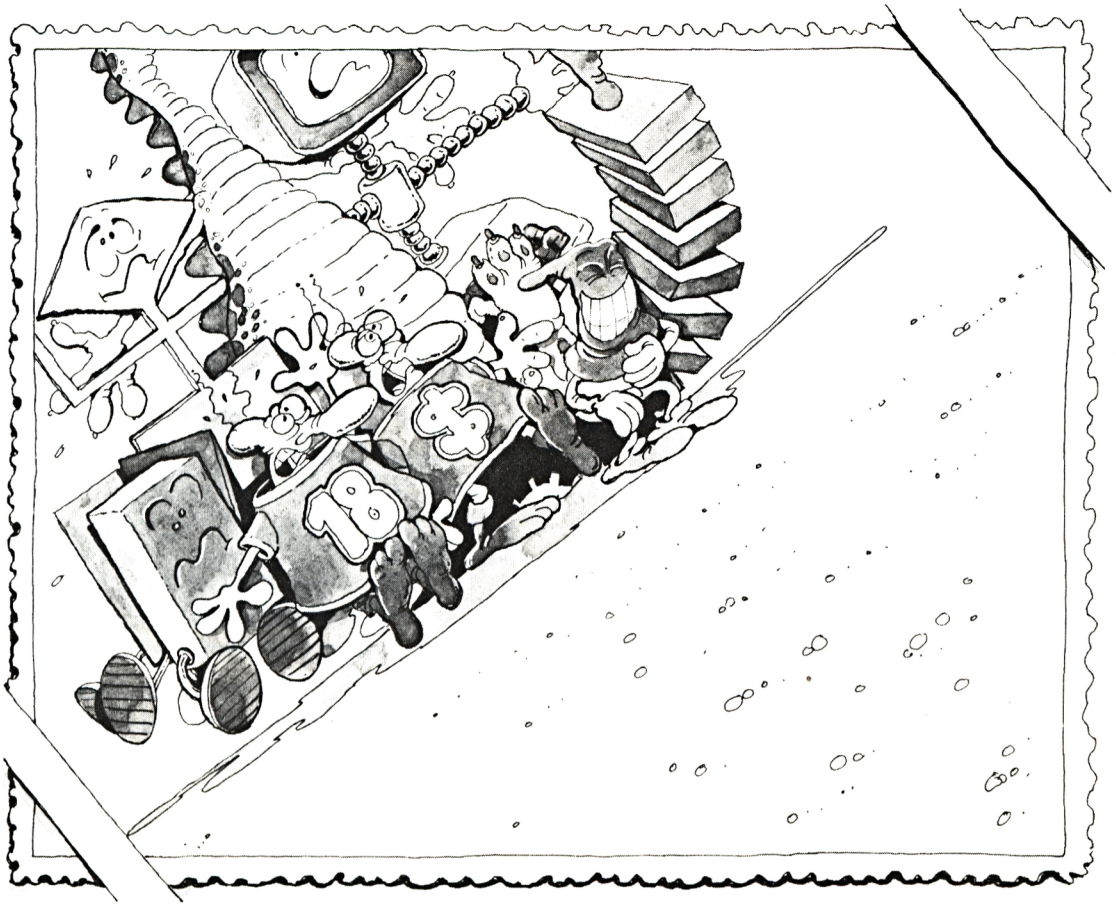
If you are ready, let's open up the family album and I'll introduce you to the heroes of this book . . .


18
$

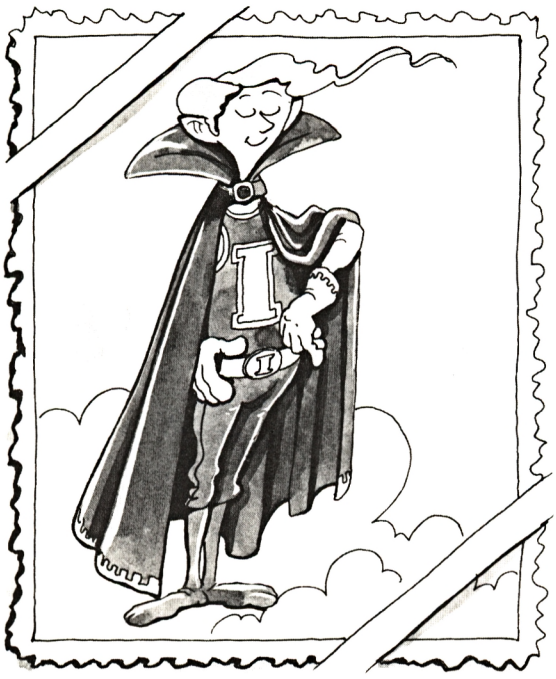


"Meet Our Heroes"

"Featuring (going clockwise): Dino the Programmer, the BASIC Interpreter standing on his friend the Computer, the Program Snake, the mischievous Bug, two Variables, Program Instructions ready to walk to their assigned spot, and the indispensable flowchart on which Dino rests . . . Oops, our bug seems up to something . . ."



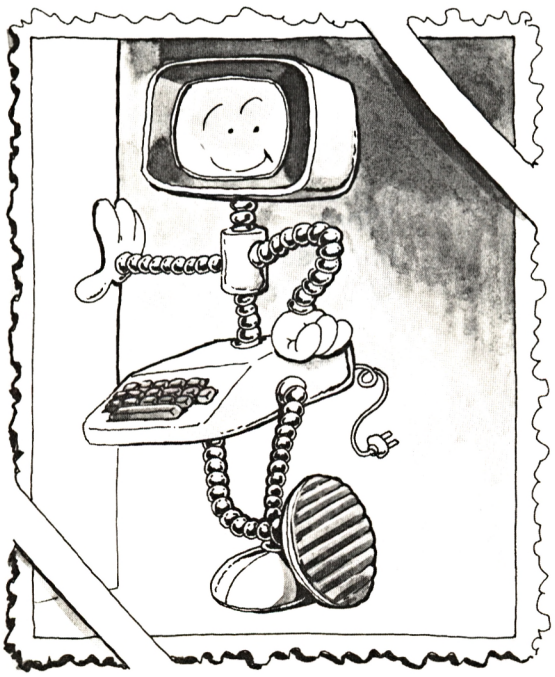
"Sorry . . .
our mischievous
Bug did it again.
So much for
a group shot."



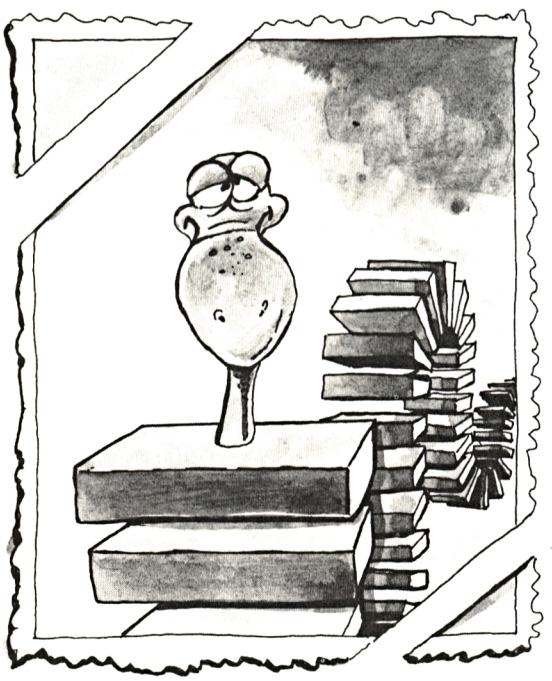
"This is the BASIC Interpreter. When awake, he resides in your Computer's memory. His job is to translate your instructions to the computer. He will help you any way he can."



"Never forget this face. This is a Bug. He will make your life miserable. Do your utmost to keep him away from your programs."



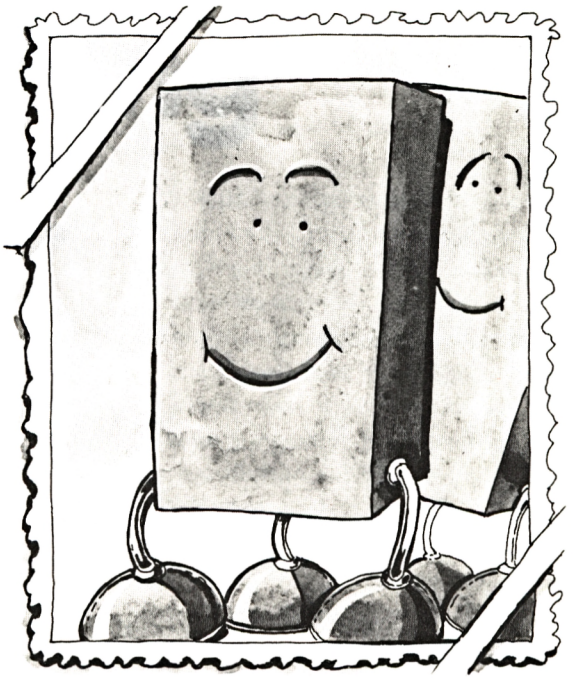
"This is your friend the Computer—at your command."



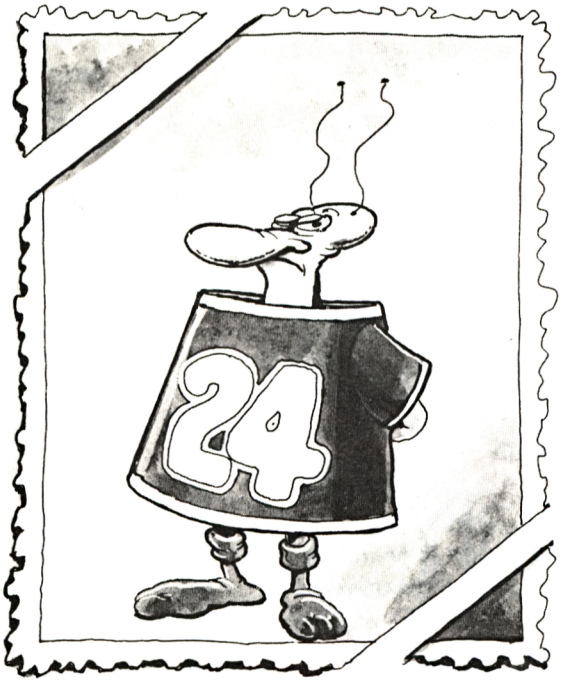
"No, this is no monster—it's the Program Snake. He's made up of instructions. You'll learn to assemble him. He's very tame once you know him. Just keep Bugs away from him."



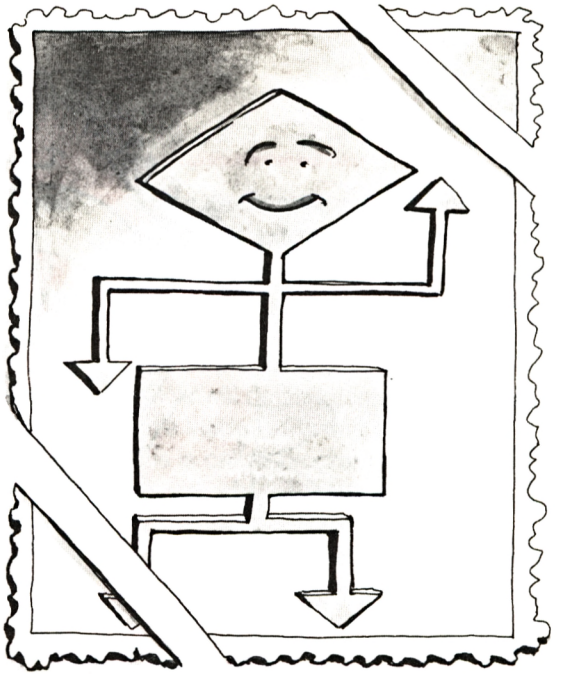
"This is Dino. He's friendly, and although he has no formal education, he will show you how simple it is to write BASIC programs."



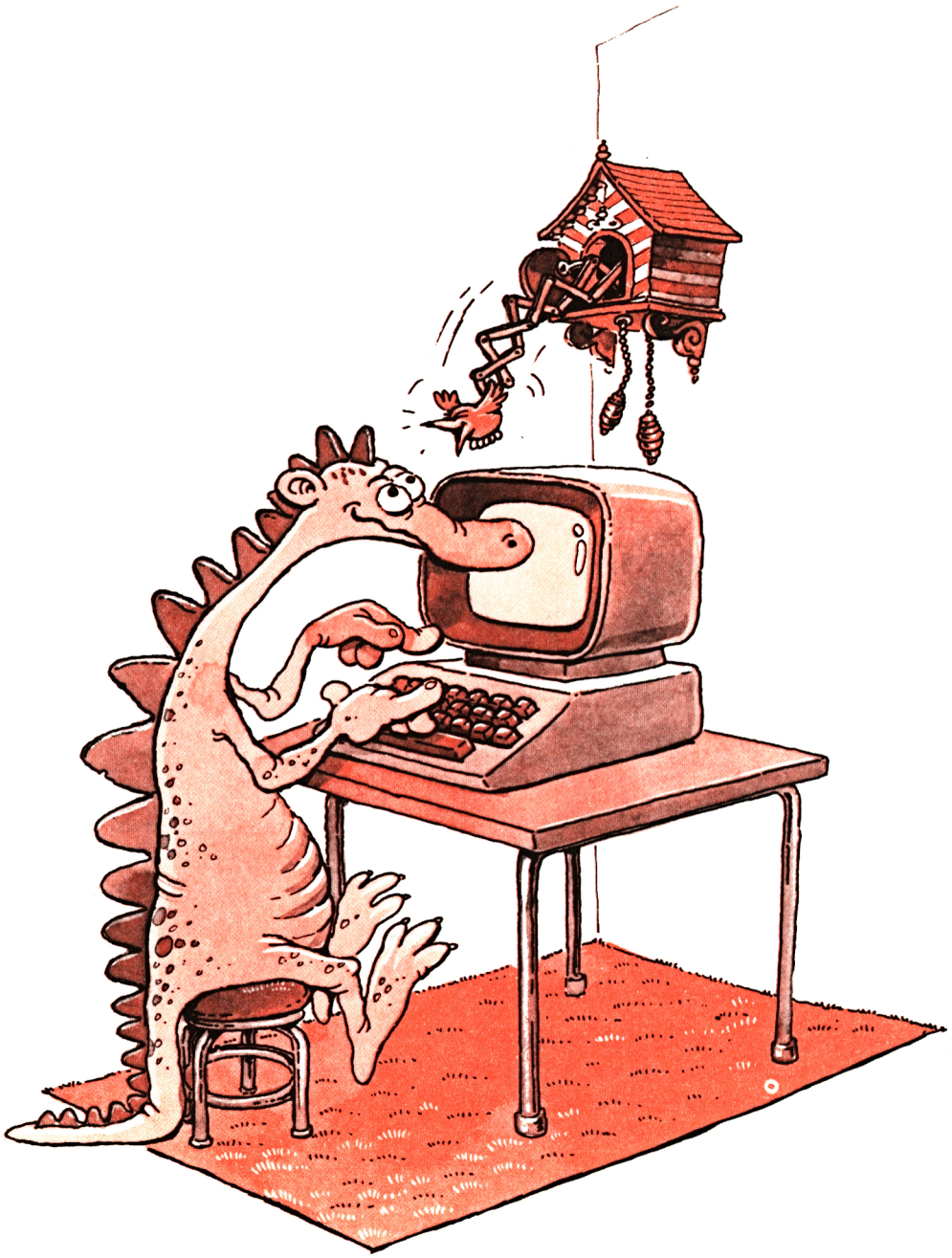
"Here are the BASIC instructions, ready to join the Program Snake."



"This is a numeric variable. His coat is labeled with his name, and he has a value imprinted on him. He's unhappy because he wants to go back to his reserved box in the memory."



"This is your best friend, the Flowchart. He'll help you design programs that work."



Speaking BASIC

1

I claimed in the preface that "You will be writing BASIC programs within one hour."—so, why start with a chapter on concepts and definitions? Aren't we wasting time? On the contrary: Our purpose is to *learn* and *retain* information, and true learning requires depth of understanding. The information presented in this chapter will help you better understand what programming is, how a BASIC program is executed, and the vocabulary of computers.

Before we begin writing our first program, there are a few important definitions and concepts you should learn. Once you understand these terms, I can explain *what happens* and *what to do,* in a simple yet accurate manner, and you should be able to follow along easily. So read this chapter carefully to ensure that you truly understand what you are doing—and so that you won't just be pressing keys.

We will begin by learning how to give instructions to a computer—this is called *programming.* Next, we will explain the need for *programming languages,* such as BASIC. Then we will discuss what a BASIC *interpreter* is, and we will explore the history of BASIC, its *dialects* and its uses. Finally, we will examine the components of a *computer system* and learn some of the technical jargon used to describe these components.

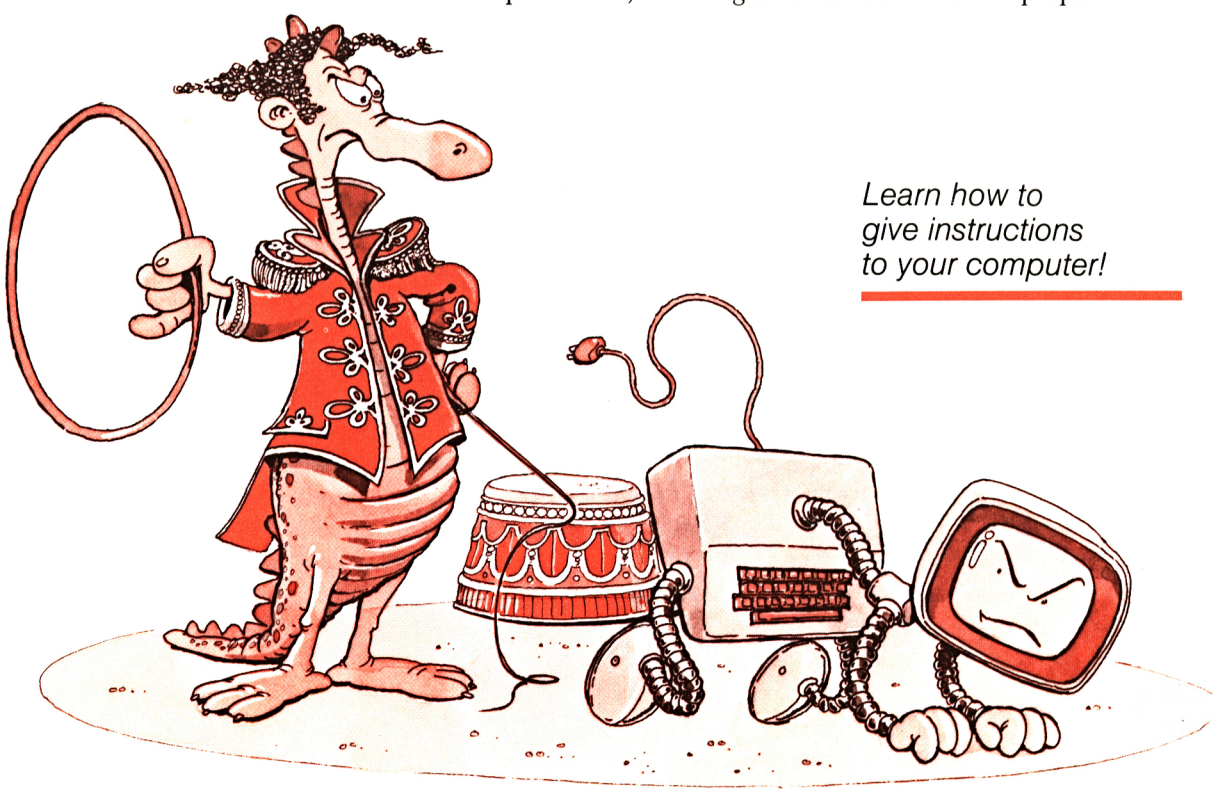
Programming

Your computer is a machine designed to process *information*—both textual and numerical. For example, you can make your computer display words and sentences on a screen—this is known as *text processing* —or you can make it convert a weight expressed in ounces into its value in grams—this is known as *numerical processing.* In order to make your computer perform this processing, it is necessary to issue instructions in a format, or "language" it understands. Each computer can only "understand" (i.e., recognize and execute) a small number of different instructions (say, a few hundred).

Instructions that a computer can understand directly are called *machine language* instructions. These instructions are stored in *binary* format, i.e., in groups of 0's and 1's in the computer's memory. Each 0 or 1 is called a *bit,* and a group of eight bits is called a *byte.*

A sequence of instructions that accomplishes something useful is called a *program.* (A sequence of instructions that accomplishes nothing is an *error.*) Your computer executes a program by executing each instruction in turn. Unfortunately, writing a computer program (a sequence of instructions) in machine language, i.e., in binary form, is a slow and tedious process.

Ideally, we would like to be able to give spoken or written commands in everyday language (say, in English) to the computer and have it execute them. But, this is not possible since a computer cannot understand any of the usual languages—whether spoken or written. The reason for this is quite simple: A computer executes orders strictly and exactly; it is logical and precise; and requires clear, unambiguous instructions in the proper



Learn how to give instructions to your computer!

sequence and form. The problem with a spoken language lies in the language itself—sentences can be ambiguous and often their meanings can depend on context, such as facial expressions or gestures. This type of communication cannot be interpreted by a computer.

Even carefully written English remains insufficiently precise for a computer. For example, you cannot tell a computerized robot to "go to the kitchen and boil an egg," and expect results, unless the robot has been programmed to know its way around your kitchen. A robot has to be trained (or programmed) before it can operate in an environment like ours. And, even if a robot is trained to know its way around *your* kitchen, it may not be successful in a friend's kitchen, because things may be located in different places. Remember, communication with a computer must be clear, precise and unambiguous.

It is for this reason that simplified "languages" were invented to communicate with computers. Recall that the binary language (also known as machine language) is the easiest language for a computer to understand. However, this language is hardly practical for people. Therefore, other languages have been invented to faciliate communications. These languages resemble common English and are called *high-level languages.*

For effective and clear communication with a computer, only a limited number of English words are used, as predefined commands. In addition, sentences or *statements* that specify instructions to a computer must obey strict grammatical rules, called the *syntax* of the language. The combination of a restricted vocabulary and a syntax is called a *programming language.* BASIC is one such language.

In summary, a *programming language* is a collection of rules (the syntax) and words and symbols (the vocabulary) that allow you to issue instructions to a computer in a format that can be understood exactly. A sequence of such instructions is called a *program.*

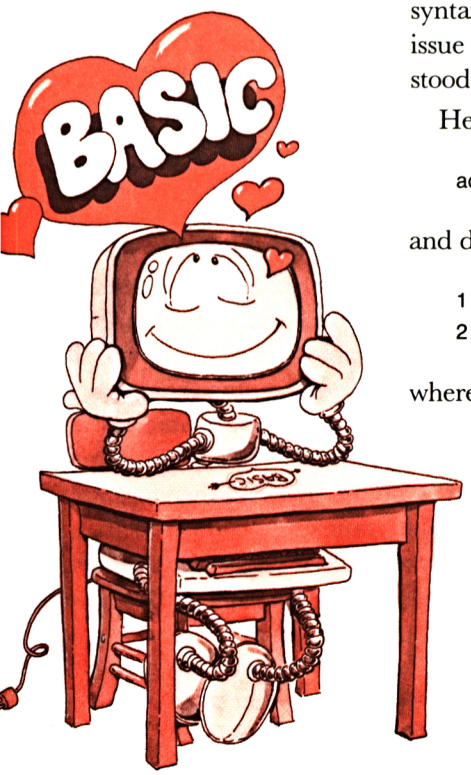
Here is an example. Suppose we want to

```
add 2 + 2
```

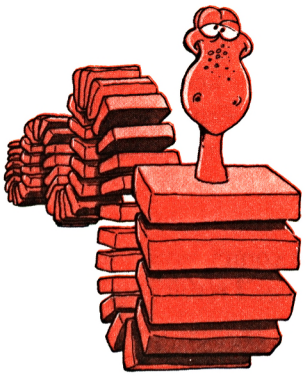
and display the result. Using BASIC, we would write:

```
1 R = 2 + 2
2 PRINT R
```

where R stands for "result."



"I love BASIC.
Please speak BASIC to me."



"Remember me? I'm the program and I'm made up of instructions."

But wait . . . we said earlier that the only language that a computer can understand directly is *machine language;* and we are now issuing instructions to a computer in a language close to English. Isn't there a contradiction?

There is no contradiction. Indeed the bare computer cannot understand BASIC directly, or for that matter, any other *high-level programming language* (a language that uses English-like sentences). Therefore, to be understood by a computer, a program written in a high-level language, such as BASIC, must be *interpreted* by a special program, called quite appropriately an *interpreter.* In other words, you speak BASIC to your computer via an interpreter. Therefore, in order to execute a BASIC program, your computer must have a BASIC interpreter. Let's now go on to learn what an interpreter does.

The BASIC Interpreter

BASIC interpreter reads each BASIC instruction you type at the keyboard, analyzes it, and uses its own special procedures to carry it out. This process is completely invisible to you (i.e., it takes place inside your computer). Once you activate the interpreter program on your computer, for all practical purposes, your computer can speak BASIC. Your computer may be able to speak other programming languages as well (if the appropriate interpreters are provided).

There are several kinds of BASIC interpreters. Here we will explain the type you have on your AMSTRAD; a *resident interpreter.*

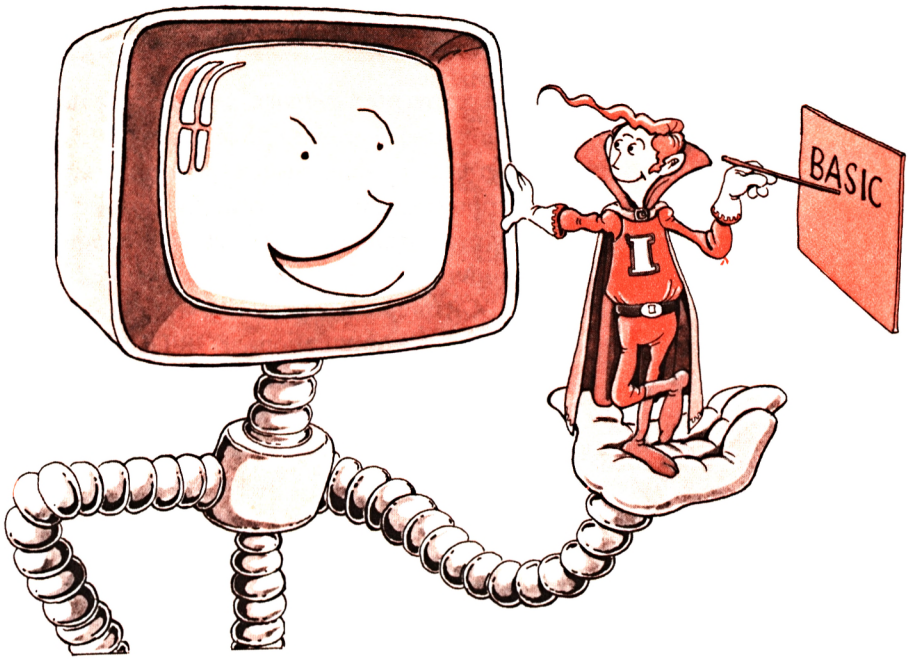
A *resident* BASIC interpreter is provided on most small computers. This interpreter is called resident because it resides permanently in the memory of the computer. On the AMSTRAD it is available immediately, whenever the computer is turned on. Once the BASIC confirmation symbol (or *prompt* (Ready)) appears on your screen, you know that the computer is ready to execute your BASIC instructions for you.



"Remember me? I'm the BASIC interpreter, ready to translate your instructions to the computer. I'm a program, and I'll reside in your computer's memory."

A resident interpreter often has one drawback: it can provide only a limited version of BASIC. Since a resident interpreter is "built-in" to the permanent memory of a computer, it must be kept small in size, since the total memory size of a computer is limited. The memory of a computer must contain the programs, including the BASIC interpreter, and also provide sufficient space for computations, system management, and data to be operated on. These space requirements limit the size, and therefore restrict the complexity, of the resident interpreter. On computers that come with a small amount of memory, the resident interpreter is often a "tiny BASIC" that imposes limits on what you can do with it.

A "mini" or "tiny BASIC" therefore, has fewer features than a "full" or "extended BASIC." A usual limitaion of a "mini-BASIC" is that it operates on integers only; that is, it does not handle fractional numbers. Such a version of BASIC is also called



*"When I'm in the memory,
your computer speaks BASIC."*

"integer BASIC." By contrast, an improved version of BASIC that also handles fractional numbers is called a "floating-point BASIC." This feature is highly desirable if you plan to do fractional computations. By virtue of clever design involving overlapping ROM and RAM spaces, the AMSTRAD has been built to accomodate extended BASIC.

What is BASIC?

High-level languages were invented to make it easier for a user to give instructions to a computer, that is, to program it. Over the years, hundreds of programming languages have been invented.

In the early days computers were used primarily for scientific purposes, and the early programming languages were designed to facilitate numerical computations. Thus, the granddaddy of languages, FORTRAN ("FORmula TRANslator"), was designed purposely to specify numerical computations. FORTRAN, however, suffered from many drawbacks, and many new languages were invented. BASIC was one such language; COBOL, APL and PASCAL were others that became widely used.

The invention of BASIC represented a major breakthrough. BASIC was designed to be simple and easy to learn. In addition, it was *interactive.* Let's see what this means.

BASIC stands for *B*eginners *A*ll-purpose *S*ymbolic *I*nstruction *C*ode. It was invented in 1964 by John Kemeny and Thomas Kurtz at Dartmouth College in New Hampshire, working under a National Science Foundation grant. The goal of the authors' was to design a language that could be used easily by a beginner. They succeeded in reaching this goal. To this day, BASIC is one of the easiest programming languages to learn.

Because BASIC was designed to be interactive—in fact it was the first interactive language—a user could interact with the program at a terminal, rather than submit batches of perforated IBM cards, as with older languages. BASIC originally ran on the GE225 time-sharing system at Dartmouth College. Terminals were available throughout the campus, and many users could access the computer simultaneously.

The success of BASIC was rapid. General Electric (GE) immediately decided to use it commercially. Kemeny and Kurtz published the first book on BASIC in 1967. And, Hewlett Packard (HP) and Digital Equipment Corporation (DEC) decided to make BASIC available on most of their computers.

BASIC offers two major advantages over languages like FORTRAN:

1. **For a user:** BASIC is the easiest language to learn, *especially* for a beginner.
2. **For a manufacturer:** BASIC is the easiest language to provide on a computer. Since the language is simple, the interpreter is also simple and requires only a small amount of memory.

A third factor contributed to the enormous success of BASIC: the advent of low-cost microcomputers. When microcomputers became widely available in the late 1970's, BASIC became the universal programming language on these small computers. Because the BASIC interpreter for a simplified version of BASIC requires only 4K (4,096 bytes) of memory, even the smallest computers could accommodate a resident BASIC. (Remember that a

resident BASIC refers to an interpreter stored in the permanent memory of the computer.) The larger more recent computers have larger memories (64K and more—where 1K refers to 1,024 bytes) and can, thus, provide more powerful versions of BASIC.

Today BASIC is used on almost all computers. Over the years manufacturers have added extensions and "features" to the language, so that today the BASIC language is probably the most nonstandard computer language. No two BASICs are the same. In fact, BASIC has become a family of languages and is no longer a single language. Although many standards have been proposed, none has succeeded, and none is likely to succeed at this late date. Does this mean then that you must relearn BASIC on each computer? Not quite. Once you know the essentials of BASIC that are common to all versions, you can easily learn the enhancements that each version offers. All BASIC's have essentially the same core of instructions. You will learn more about these instructions as you read on in this book.

Which BASIC?

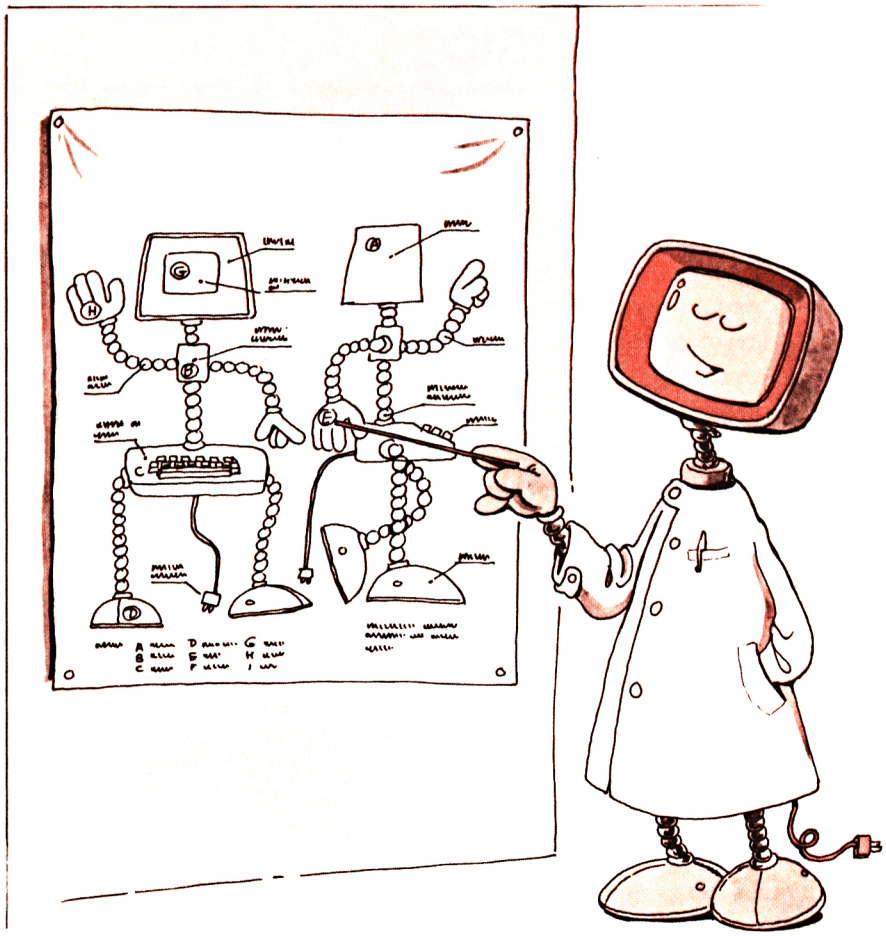
The BASIC interpreter resident in your AMSTRAD has most of the features found in powerful (*full* or *extended*)BASICs, including scientific, or floating-point notation. A full BASIC such as this one offers great convenience and power.

As you type in and run the programs in this book, you will become skilled in the main capabilities of AMSTRAD BASIC. A few of its features, such as colour graphics and sound, will not be covered here. Later on, if you wish to write programs that employ these features, you can study your AMSTRAD User Instructions.

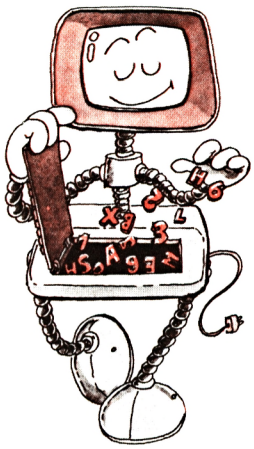
Now that we understand more about programming languages in general and BASIC in particular, let's learn more about your computer and how it processes information.

Your Computer

Your computer processes information and communicates with you via a keyboard and a screen, plus maybe a printer. The *keyboard* is used to send information to the computer. Every time a key is pressed, the electronic code corresponding to the character for that key is sent to the computer, where it is recognized and acted on, or ignored. The keyboard is your *input device;* it provides information to the computer.



"I'm robust, helpful, and friendly once you know me."

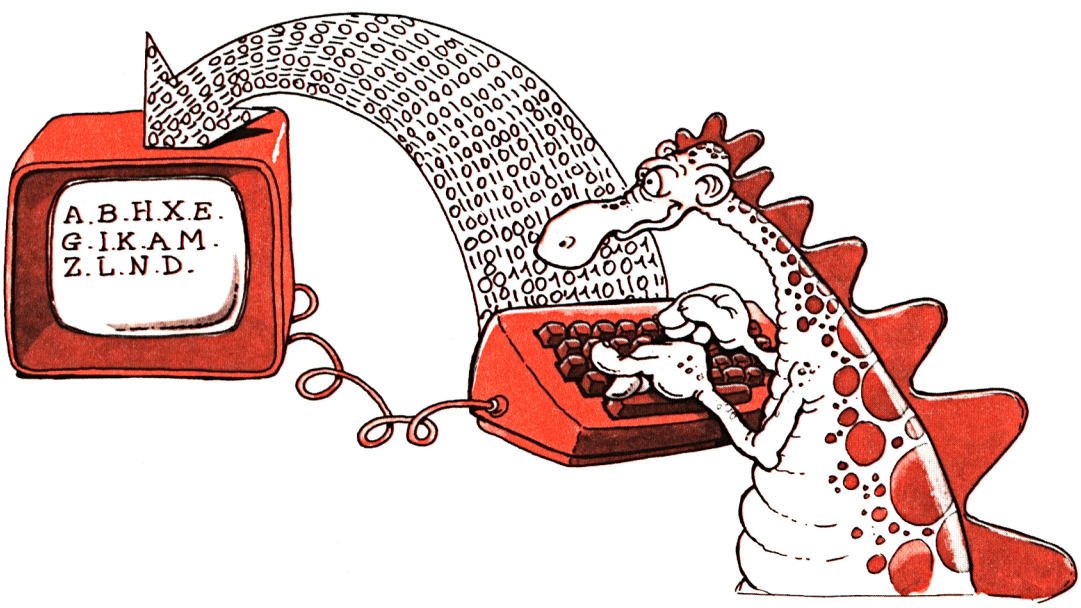


"This is my keyboard."



"I need an input from you to know what to do."

A television *screen,* or *monitor* displays information generated by the program. Normally, each character you press at the keyboard will appear on the screen. It is first sent to the computer, then "echoed" to the screen. Generally, there is no direct connection between the keyboard and the screen. All communications go through the computer. This is illustrated below.



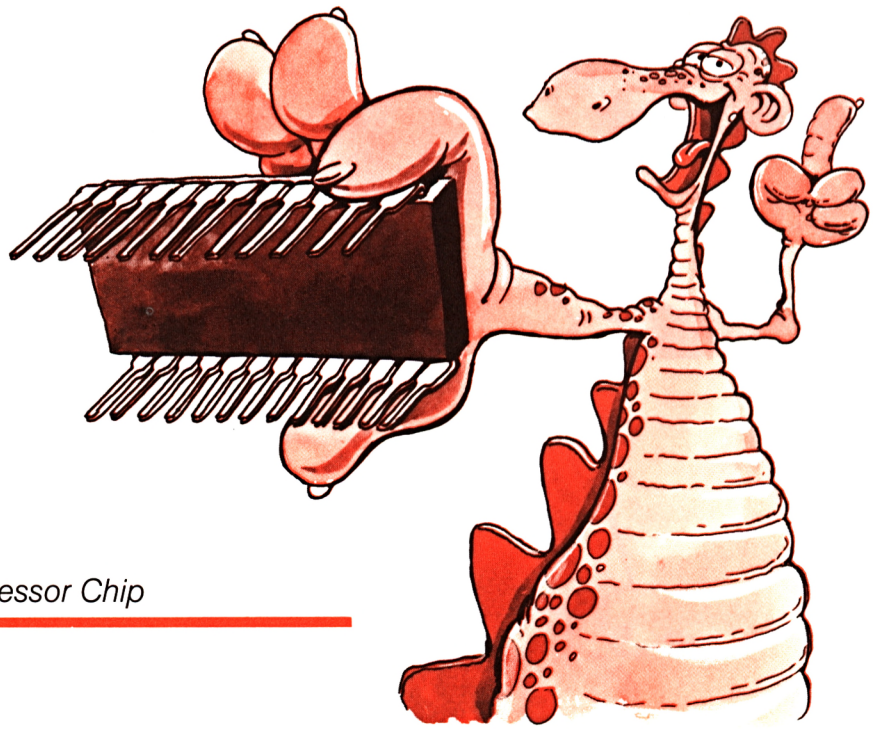
The keyboard sends information to the computer

The computer and the cassette recorder of the AMSTRAD are integrated with the keyboard. The screen and the disk unit are separate. The computer proper includes a processing unit (the central processing unit), a memory, and several interfaces (the electronics for connecting printers and other devices). Let's examine these three elements.

The *central processing unit* (CPU) fetches program instructions, one instruction at a time, from the memory and executes them. The CPU requires few components, called integrated circuits or "chips". All microcomputers use a *microprocessor* chip as the main element of the CPU. A typical chip is shown in the drawing below.

The *memory* stores the programs and all the information that the programs manipulate, read, or generate during their execution. To execute a program, the program must first be placed in the computer's memory. For example, if a program is originally stored on a cassette or a diskette, it must first be transferred into the computer's memory. This is called *loading* the program. There must be enough memory inside the computer to accommodate the largest program size, plus the data the program will manipulate.

Two types of memories are present in your computer: ROM and RAM. The "normal" type of memory that you will use to store your program is the RAM or random access memory. RAM is a read/write memory: Information may be written into RAM and read from it. A RAM looks just like the microprocessor shown below except that there is a different chip inside. Unfortunately, in the present state of the technology, this type of memory is volatile: the contents of RAM disappear once the power is



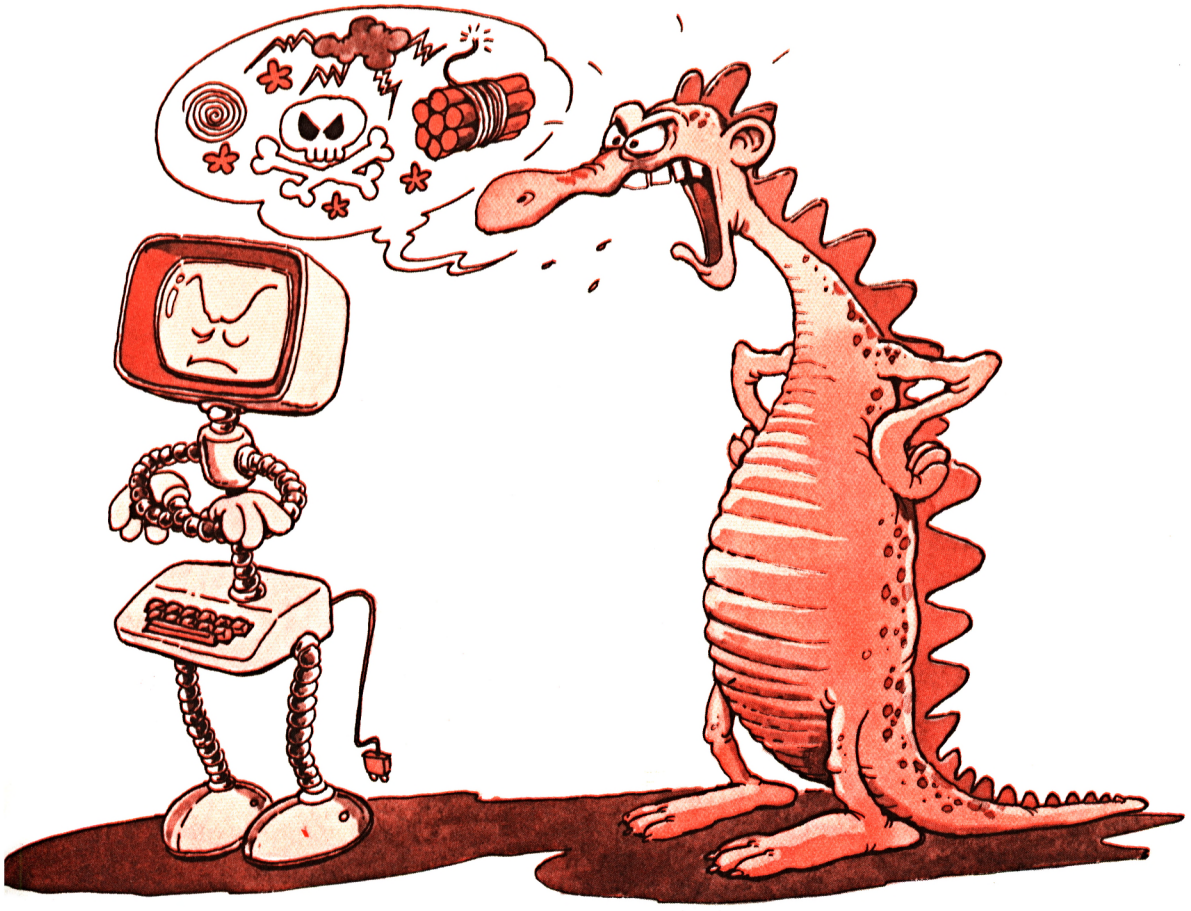
A Microprocessor Chip

turned off. This is why, at the end of a session, you must always store your program on a nonvolatile medium, such as a cassette or a diskette, if you want to keep it.

The AMSTRAD's BASIC interpreter is stored in the computer's ROM. *ROM* stands for read-only memory. This type of memory has been permanently loaded with programs by the manufacturer and cannot be changed. It is nonvolatile and is never erased. In addition to the resident BASIC interpreter, the ROM contains a special program called the *operating system,* or *monitor,* which is needed to communicate with the computer once it is turned on.

If your ROM contained nothing, the computer would not know what to do when you press the keys on your keyboard. At a minimum, your ROM must contain a monitor. This monitor program examines the information sent by the keyboard, and reacts to it by performing housekeeping actions, such as starting the resident BASIC interpreter or loading a program from a cassette.

Your computer will do nothing until you enter a program in its memory.



You cannot use the ROM to store any other programs. All other programs that you enter on your computer are loaded into RAM. Sometimes, you can also purchase program cartridges for your computer. In these cases, the programs are stored in ROM chips inside the cartridge.

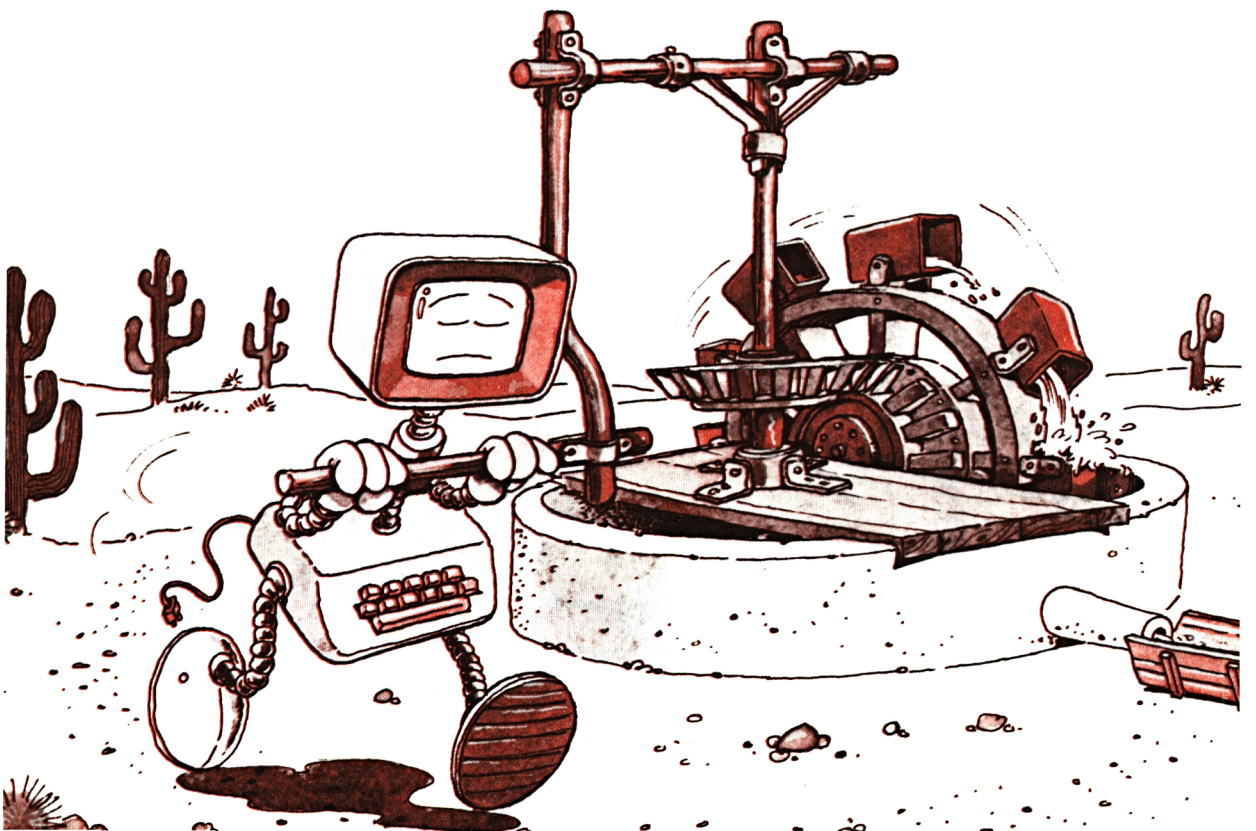
At least two additional devices are commonly connected to a computer: a mass memory and a printer. These devices connect through the *interfaces*—the electronics required to connect special devices. The mass storage device may be a cassette recorder or one or more *disk drives*. (*Note:* Both of these devices use a magnetic medium to record information and can store much more information than the internal electronic memory of the computer.) These special devices require a specific interface in the computer box that allows them to communicate with the computer. AMSTRAD computers have built-in interfaces for a cassette recorder, as well as for a parallel printer, so that the only connection needed is the cable between the computer and the pinter (the cassette s built into the keyboard).

A printer is required to obtain a permanent printout of programs or results. A printer is an *output device,* just like a video display, and specific instructions are provided in BASIC to send information to either the display or the printer.

Finally, a *modem* is another device often used. A modem allows you to communicate with another computer or terminal over ordinary telephone lines. It is useful when using a commercial network or for accessing *data banks* (collections of information).

We have now learned the required vocabulary. Before we go on to Chapter 2 and start using the computer, *one word of caution is in order.*

The monitor program runs all the time, ready to perform all common chores.



Of Computers and Syntax

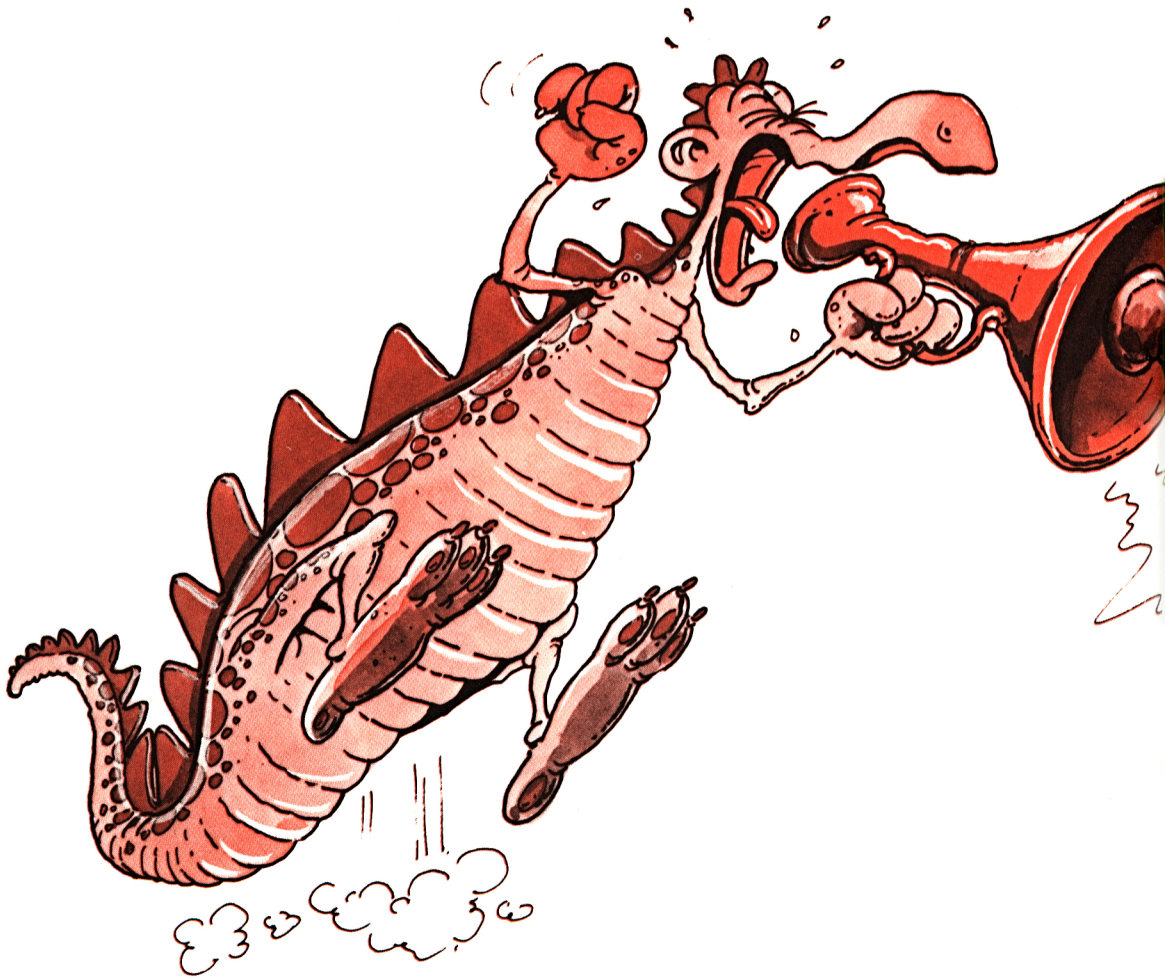
Computers are fast, patient, and accurate. They do only what they are told to do, and they do it exactly. In order to communicate successfully with your computer you will have to be exact. If you make an error or a mistake in writing one of the BASIC instructions, you will not damage anything in the computer, but your program will not execute successfully. It will generally stop and say: "Syntax error."

Recall that the *syntax* is the set of rules that specify the correct way to write a BASIC instruction. Syntax rules are rigid. For example, you may not use an approximate spelling. If the rules specify a full stop, you may not use a comma or a semi-colon instead. Each character is rigidly interpreted by the computer and has a precise meaning. Any deviation will mean failure, or, at the very least, unexpected results.

Remember that if you do not obey the rules exactly, your program will probably not run. Do not attempt to be creative. The rules are simple, straightforward, and easy to follow when writing program instructions. It is best to save your creativity for the overall program design and for planning the tasks you wish to accomplish. In short, it is important that you carefully follow the instructions and recommendations given in this book.



*"Remember . . .
be exact."*



Communicating with Your Computer



2

In this chapter, you will learn how to communicate with your computer. You will learn to issue BASIC instructions and make the computer display words and sentences. The information exchanged between you and the computer will include programs (instructions in BASIC that you send) and data (the numbers and characters that you send or receive).

We will first learn how to use the keyboard of a computer, so that you can start sending instructions. In particular, we will learn how to move the cursor around the screen, and how to correct typing errors. We will then issue your first instructions in BASIC and make the computer display messages on the screen. We will learn the difference between *immediate* and *deferred* execution. Finally, you will learn the steps involved in writing a simple program and executing it.

By the end of this chapter, you should be familiar with the basic skills required to communicate with your computer.

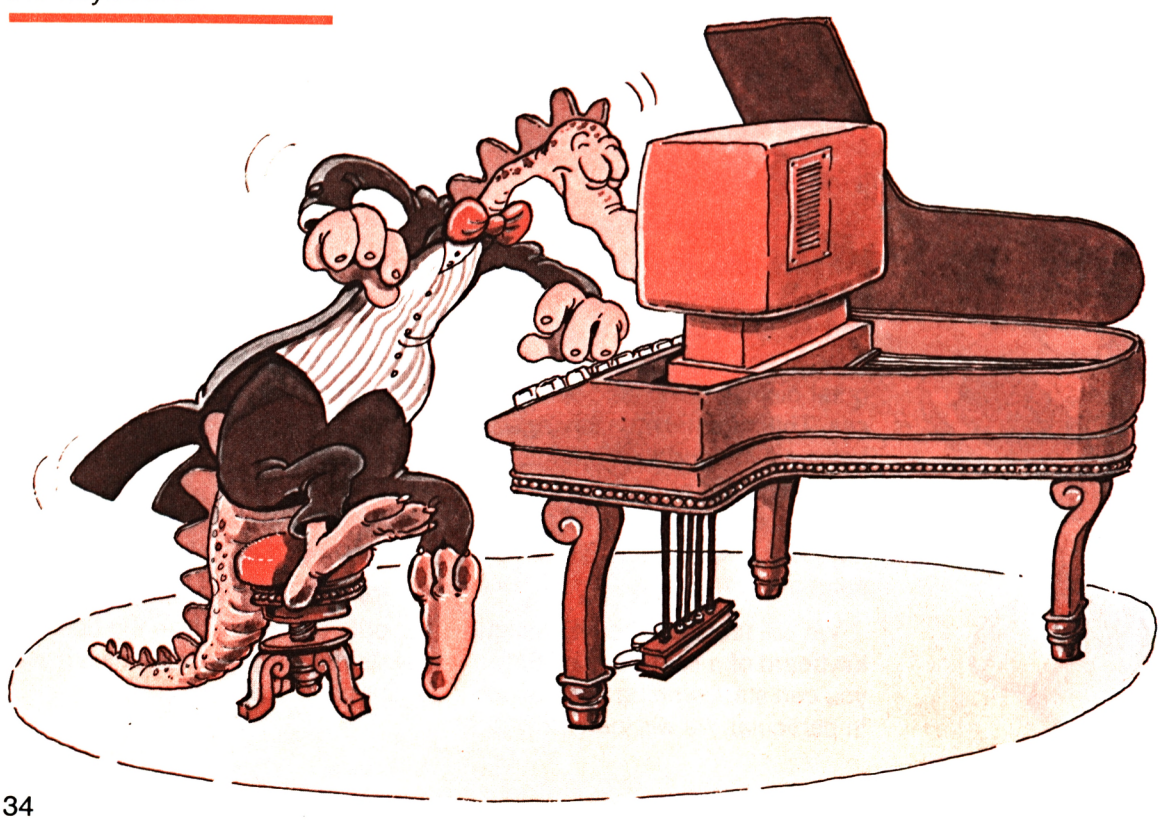
Using the Keyboard

The AMSTRAD's keyboard looks just like a regular typewriter keyboard, except that it has extra keys. Most of its features are similar to those found on other computers, but it does have a few unique ones. The main keys are:

1. the letters of the alphabet (A through Z)
2. the digits 0 through 9
3. symbols, such as =, +, *, ", and $
4. a "carriage return" key marked ENTER,
5. a SHIFT key and a "control" (CTRL) key
6. a CAPS LOCK key
7. an ESCape key
8. a DEL key, a CLR key, and a COPY key

The AMSTRAD also has cursor control keys. We will now examine the role and the use of all these keys.

Learn about the keyboard.



Characters, Numbers and Symbols

Each letter, number, and symbol key on the AMSTRAD keyboard serves the same function as the corresponding keys on a typewriter. However, when you first switch your computer on, pressing a letter key will produce an lowercase letter on the screen. For programming in BASIC it is best to use uppercase (capital) letters. Press the CAPS LOCK key so that, for example, pressing the letter K will produce a K, not a k.

Carriage Return

On a typewriter, the carriage return lever or key performs two actions: it returns the carriage to the beginning of a line, and it advances the paper to the next line. Hence its name. On a computer, however, the carriage return key generally moves the *cursor* to the beginning of the next line on the screen. (The cursor is a white or yellow rectangle that indicates the next position at which a character will be displayed on the screen.) On your AMSTRAD, the carriage return key is appropriately called the ENTER key, since its main function is to ENTER a character or a line of text or data into the computer's memory. Some keyboards label this key RETURN. Others identify it with an arrow (↵).

In any case, each instruction to the computer, including BASIC instructions, must be terminated with a ENTER. The ENTER means, "enter into memory." In fact, anything you type on a line is ignored by the computer until you press ENTER. This way, you can correct any errors you may have made, before you enter the line into the computer's memory.

ENTERs are used only at typing time. Although necessary, an ENTER is not stored as part of a program instruction. In this introductory chapter, to help you learn, we will show all the characters you must type, and display a symbol indicating ENTER at the end of each line. This symbol will appear as ↵.

Remember: you must press the ENTER key to transmit an instruction to the computer or else nothing will happen. Similarly when the computer later asks you for values, you enter each response by pressing ENTER as well.

Shift

The shift keys work just like the one on a typewriter. They allow you to shift between the upper- and lowercase symbols (for example, between 4 and $). Using the SHIFT key doubles the number of symbols or commands that each key can generate.

The SHIFT key when pressed provides uppercase letters whatever the state of the CAPS LOCK key.

Control

The CTRL (control) key is provided to issue frequently-used commands to your computer in shorthand. It does not exist on typewriters. The CTRL key is used like the SHIFT key: by pressing it down, holding it down, and pushing another key on the keyboard at the same time. This is called generating a control character. For example, "CTRL A" is generated by holding down the CTRL key and pressing the A key.

The AMSTRAD also uses the CTRL key to modify the action of the CAPS LOCK and ESC keys. Otherwise, the use of control characters is specific to each of the programs used on your computer, and their effect is explained in the program documentation. We will not use any control codes in this book.

Caps Lock

The AMSTRAD also provide a CAPS LOCK key that allows you to lock the keyboard into the uppercase position. CAPS LOCK on provides you with uppercase letters and lowercase symbols, which is usually the most convenient combination for entry of BASIC programs. When CAPS LOCK is on, pressing the K key provides K, pressing the : key provides the colon, and pressing the 1 key provides 1.

There may be occasions when you want both uppercase symbols and uppercase letters. The AMSTRAD provides an easy way of obtaining this. Press CRTL CAPS LOCK. Then pressing the K key provides K, and pressing the 1 key provides !.

Escape and Break

The ESC (Escape) key is provided as a standard "control code," common to all programs; it can be used in three different ways to "break into" a running program. Each program, whether it be a BASIC interpreter or your own program, must allow a special command, such as "END" or "EXIT," that terminates its execution. However, in case something unexpected happens and you want to stop program execution immediately, you can use the ESC key.

Pressing the ESC key once allows you to stop any program that is running. Pressing any other key will allow the program to continue from where it left off. Pressing the ESC key twice in a row will stop a program that is running, show the line number of the program where the break occurred, and provide the **Ready** prompt.

Pressing the CTRL key, the SHIFT key, and the ESC key all at the same time will reset your AMSTRAD, and remove your program from memory. This is a very effective way of stopping any program you are running! After you have pressed the CTRL-SHIFT-ESC key combination, the AMSTRAD is reset to the almost the same state as when you turned the computer on. Your program will no longer be stored in RAM. Use this only as a last resort.

DEL, CLR, and COPY

The DEL (delete) key is used to erase characters. When pressed, the character to the left of the cursor is erased and the cursor moves one space backwards. Like the other keys, this is a repeating key, so many characters may be deleted by holding it down.

The COPY key is used to copy characters from one place on the screen to another. This is useful for both repeating a BASIC program line and for making changes to an existing line. How to use the COPY key is shown in the **Cursor** section that follows.

The CLR key is also used to erase characters. When pressed, the character within the cursor is removed and all the characters to the right are moved left one position. How to use the CLR key is shown in the **Cursor** section that follows.

The Cursor

Recall that the cursor is a rectangle on the screen that shows your current position. Here is a cursor showing you where you are on the screen after you have typed HELLO:

```
HELLO■
```

By moving the cursor back and forth on the screen, you can modify any text you have typed previously. This is a great convenience for making changes. You will use the cursor extensively to correct typing errors.

If you use the SHIFT key in combination with the cursor control keys (the four arrow keys at the upper right of the keyboard)



Learn to move the cursor on the screen.

you will see a second cursor. The second cursor shows where characters will be copied *from* when you press the COPY key. The characters are copied *to* the location of the original cursor. You can also use the CLR key to delete characters in the line being edited or created.

There is no substitute for experience when learning the functions of the cursor control and line editing keys.

The Numeric Key Pad

The numeric key pad is most useful in business applications when numbers must be typed rapidly. The key pad is used in the fashion of a desktop calculator keyboard. Its keys simply duplicate the functions of the digit keys on the regular keyboard (0 through 9, period, and ENTER).

You may now want to go back to your keyboard and practice some of the things you have just learnt. When you feel reasonably familiar with your keyboard, come back and learn how to issue BASIC instructions to the computer.

Speaking BASIC

To speak BASIC with your computer you need a BASIC interpreter in the memory. Since your computer has a "built-in" (resident) BASIC interpreter, you do not have to do anything to activate the interpreter. The interpreter is activated automatically when you turn on your computer, and a confirmation, or prompt, appears on the screen:

```
Amstrad 64K Microcomputer (v1)

©1984 Amstrad Consumer Electronics plc
        and Locomotive Software Ltd.

BASIC 1.0

Ready
■
```

You may see different version numbers.

"Ready" is the *prompt.* The prompt is a message from the BASIC interpreter; it means, "I'm ready. Go ahead and tell me what to do." As soon as you see this word, you know that the BASIC interpreter is ready and waiting for your instructions.

We will now proceed, assuming that:

1. The AMSTRAD BASIC interpreter in your computer's memory is ready to go to work.
2. The BASIC prompt used by your interpreter, "Ready", appears on your screen. If the prompt does not appear, press ENTER and see what happens. If this does not work, press CTRL-SHIFT-ESC, or turn your computer off, and restart it.

We will now issue our first BASIC instruction to the computer.

Type the following (exactly as it appears here):

```
PRINT "HELLO"
```

The screen should look like this:

```
Amstrad 64K Microcomputer (v1)

©1984 Amstrad Consumer Electronics plc
        and Locomotive Software Ltd.

BASIC 1.0

Ready
PRINT "HELLO"■
```

The word "Ready" is the *prompt* telling you that the BASIC interpreter is waiting for an instruction. The characters in the next line have just been typed by you. Nothing should happen yet. Do you remember why?

This is because you must press the ENTER key to *enter* your instruction. Now press ENTER. Your screen should look like this:

```
Amstrad 64K Microcomputer (v1)

©1984 Amstrad Consumer Electronics plc
        and Locomotive Software Ltd.

BASIC 1.0

Ready
PRINT "HELLO"
HELLO
Ready
■
```

The interpreter has received your instruction and has immediately executed it by displaying HELLO as requested. Next, the interpreter displays a new prompt (Ready), telling you it is ready for a new instruction.

Let's examine our first BASIC instruction more closely:

```
PRINT "HELLO"
```

This instruction has two parts: PRINT and "HELLO". PRINT is a *keyword* that has a specific meaning to the BASIC interpreter. "HELLO" is the message to be PRINTed. It must be enclosed in quotes. You can use any message within the quotes. Let's try another. Type:

```
PRINT "THIS IS ANOTHER TEST"↵
```

The following should appear on your screen:

```
PRINT "THIS IS ANOTHER TEST"
THIS IS ANOTHER TEST
Ready
■
```

Try again. Display your own message, but remember: if you forget one of the quotes or if you misspell PRINT, it will not work, and you will get an error message. Go ahead; try it. You won't damage anything.

You may now be wondering why this instruction is called "PRINT" when in fact it merely displays, and does not print, the information on your screen. Even if you have a printer connected to your computer, nothing will be printed. You have probably guessed the reason. The early terminals were typewriter-like, and

in fact, did print (not display) the information. And, although the technology has changed, the BASIC instruction has not. Another form of the PRINT instruction (PRINT #8...) is used to send information to the printer, rather than to the display.

Before we proceed, make sure that the BASIC prompt (Ready) appears on your screen, i.e., that the BASIC interpreter is ready to accept another command. If the prompt does not appear, you cannot execute a BASIC command. Try pressing ENTER or ESC twice, or if that doesn't work, try restarting your system by pressing CTRL-SHIFT-ESC or using the on/off switch.

We will now write our first BASIC program, rather than just execute a single instruction or "statement." Type the following:

```
10 PRINT "HELLO" ↵
20 PRINT "HOW ARE YOU?" ↵
30 END ↵
```

Your screen should look like this:

```
10 PRINT "HELLO"
20 PRINT "HOW ARE YOU?"
30 END
■
```



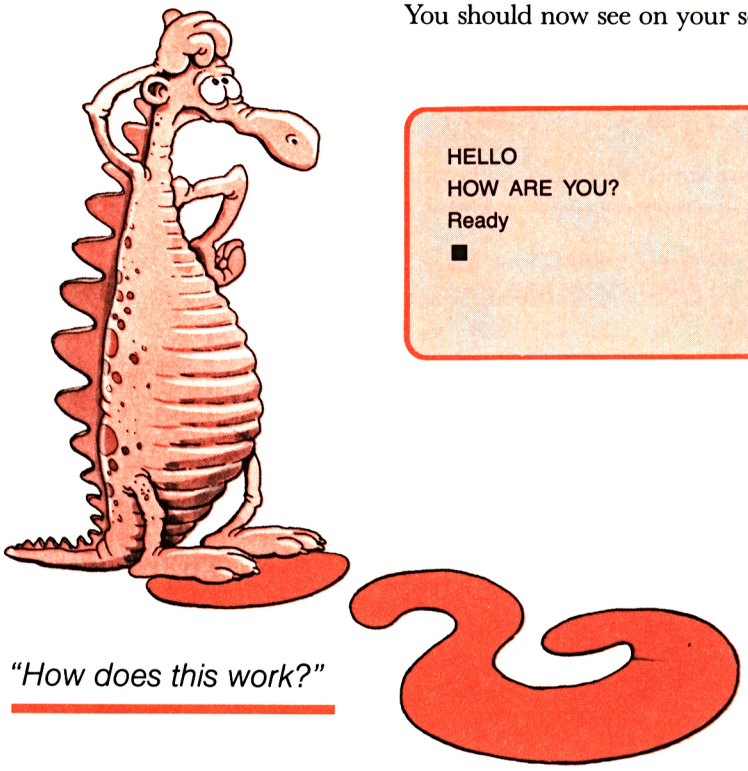
"RUN!"

You may be surprised that nothing has happened. The computer merely responds with "Ready" and a cursor. These three lines are more than just three BASIC statements. They constitute a short BASIC *program.* Note that each line begins with a number; this number is called a line number or *label.* It tells the computer that we want to write a complete program, and not execute each instruction immediately. Now, type:

```
RUN↵
```

You should now see on your screen:

```
HELLO
HOW ARE YOU?
Ready
■
```



"How does this work?"

How does this work? We first created a three-line program and stored it in memory, a line at a time, by pressing ENTER. We then executed the program by typing RUN. This is the normal way of writing and executing a program. We will follow this procedure from now on. All lines will be preceded by a numerical label, and the program will be executed automatically in the order of the labels.

If at any time you type an unlabeled BASIC statement in response to "Ready", the statement will be executed immediately; it will not be memorized. This is called the *immediate mode* or *calculator mode* in BASIC. If at any time you type a labeled BASIC statement in response to "Ready", the statement will be memorized when you press ENTER, but not executed until you type RUN. This is called the *deferred* or normal mode. Let's demonstrate how this works.

Our three-line program has been stored in memory. It can be executed any
number of times, expanded or modified. Now, type:

```
RUN ↵
```

again, and you should see:

```
HELLO
HOW ARE YOU?
Ready
■
```

Let's now examine the computer's memory and display its contents. To do this type:

```
LIST ↵
```

and you should see the following displayed on your screen:

```
LIST
10 PRINT "HELLO"
20 PRINT "HOW ARE YOU?"
30 END
Ready
■
```

Your program is listed on the screen as it is stored in the computer's memory.

To demonstrate the difference between immediate and deferred instructions, type:

```
PRINT "GOOD BYE" ↵
```

You see on the screen:

```
PRINT "GOOD BYE"
GOOD BYE
Ready
■
```

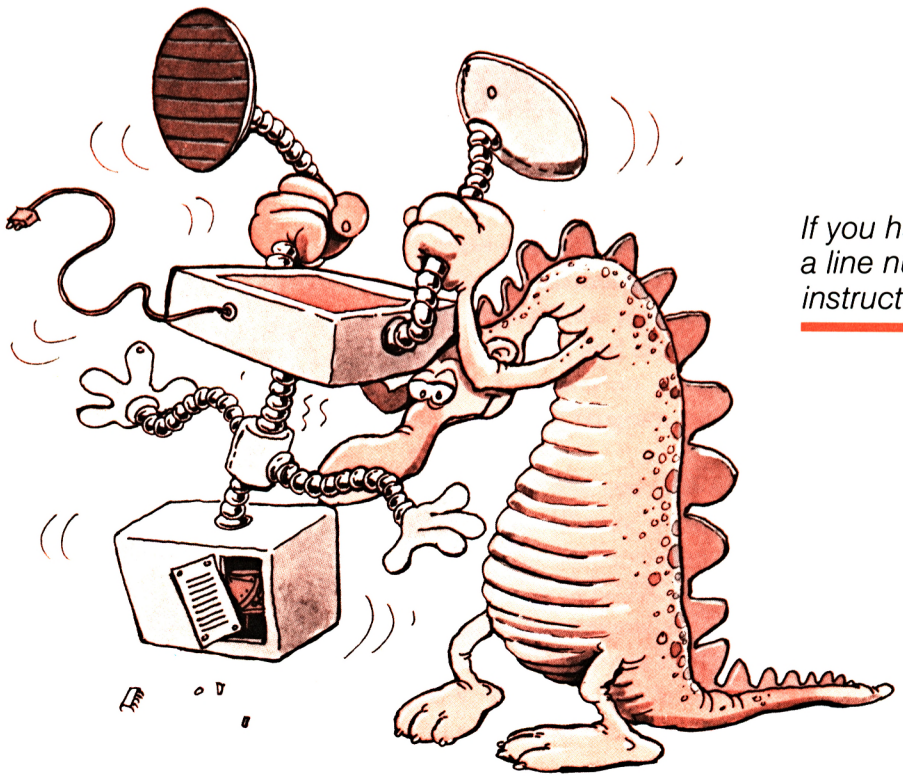
Now type:

LIST ↵

again, and you should see on your screen:

```
10 PRINT "HELLO"
20 PRINT "HOW ARE YOU?"
30 END
Ready
■
```

The new immediate instruction: PRINT "GOOD BYE" is nowhere to be seen. It has not been memorized.



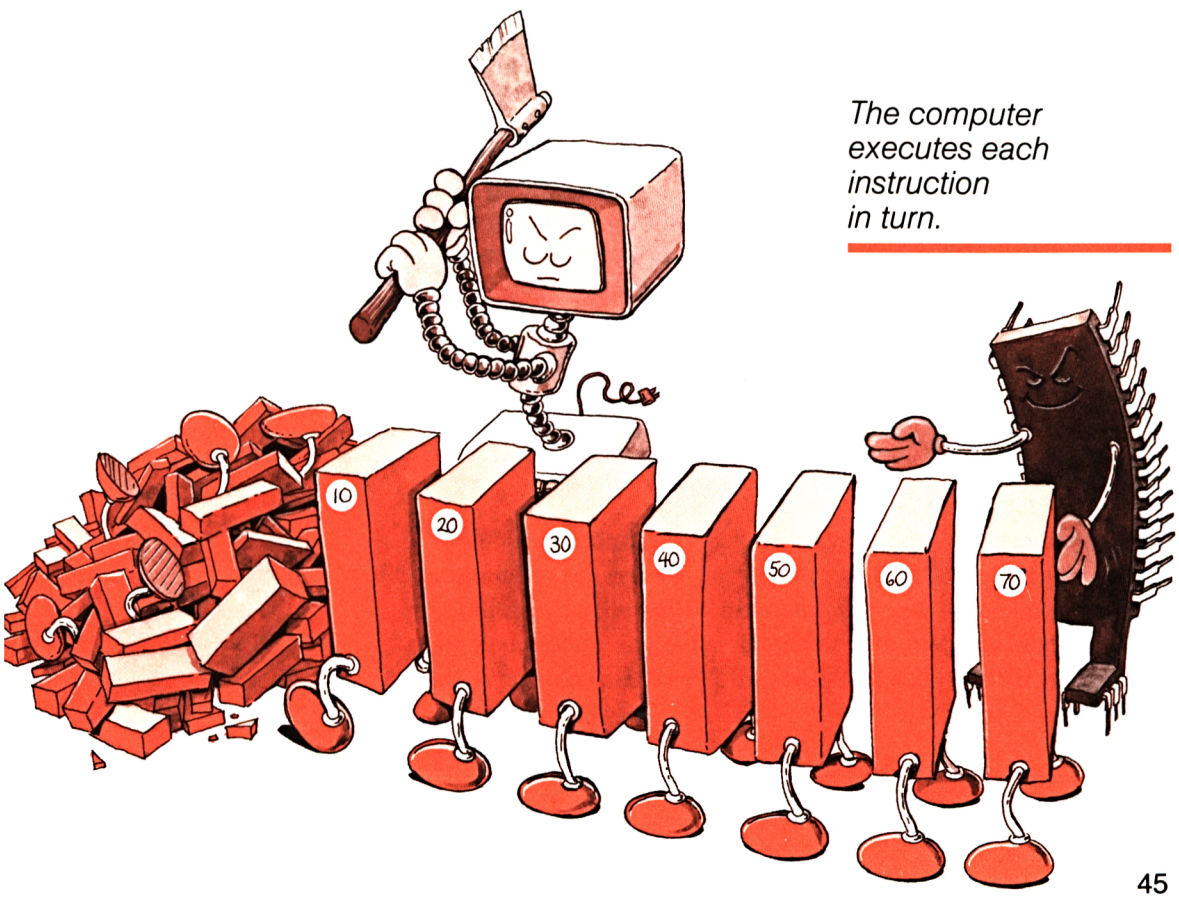
If you haven't used a line number, the instruction will be gone!

Program Summary

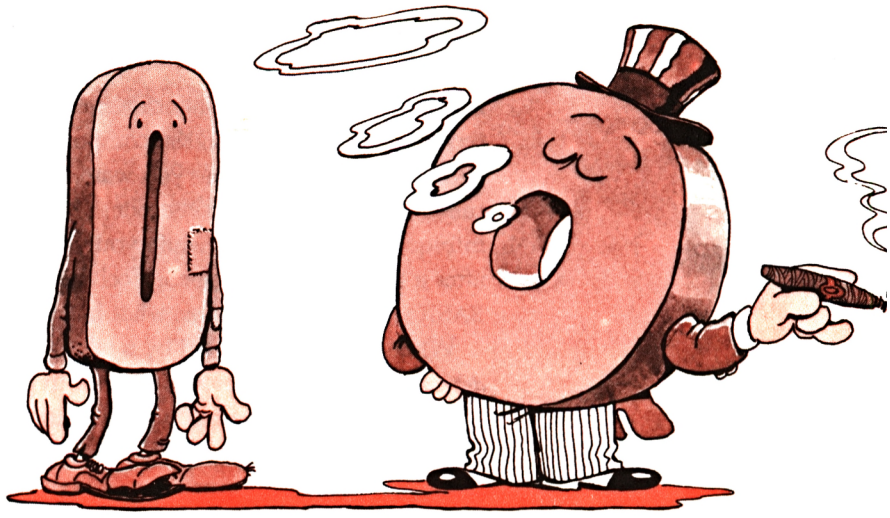
In summary, an immediate statement is executed as soon as you type it. It is not stored in the computer's memory, and you will need to type it every time you want to execute it. This facility is called the *immediate execution* mode. In practice, this mode is used infrequently—generally when you want to verify some values after a program has stopped. Feel free to experiment by typing BASIC instructions and see what happens.

When a line is preceded by a label, the mode is called the *deferred execution* or *programming* mode. In this mode, each line of the program is stored away in the computer's memory. When a complete program has been typed in, you simply issue the RUN command, and the entire program will be executed. Remember that when you turn the computer off, the contents of the RAM memory disappear, including any program you may have typed. If you wish to keep your program for later use, you must SAVE it on a cassette or diskette.

Each line of a BASIC program must start with a label. The label specifies the order in which the line will be executed. Thus, line 10 will be executed before line 20.



The computer executes each instruction in turn.



Don't confuse number 0 and letter O

The END statement, line 30 in our example, is optional with AMSTRAD BASIC, but required with older ones. The END statement tells the interpreter that it has reached a legitimate END, rather than an accidental one, during execution. However, an AMSTRAD BASIC program that runs out of lines to execute will end all by itself.

As a final detail, note that the digit 0 must look different from the letter O. The traditional way to avoid confusion has been to show the digit 0 as Ø, an O with a slash. 0 appears this way on the AMSTRAD keyboard and on the display.

A Longer Program

RUN and LIST are called *commands*. They are keywords used by themselves to specify special actions carried out by the computer system, or more exactly by the interpreter. These commands are also part of BASIC. We will progressively learn more statements and commands as we continue through this book.

Let's now write a more complete program, using the PRINT statement and the NEW command. Type the following:

```
NEW ↵
```

Then:

```
LIST ↵
```

Nothing happens. There is no longer any program in the memory. The NEW command clears the computer's memory. Now type:

```
NEW ↵
10 PRINT "THIS" ↵
20 PRINT "IS" ↵
30 PRINT "ANOTHER" ↵
40 PRINT "EXAMPLE" ↵
50 END ↵
```

Let's find out what this program does. Type:

```
RUN ↵
```

You should now see on your screen:

```
THIS
IS
ANOTHER
EXAMPLE
Ready
■
```

Our program displays the text as expected. Let's verify that the program has been stored correctly in memory by typing:

```
LIST ↵
```

You should now see on your screen:

```
LIST
10 PRINT "THIS"
20 PRINT "IS"
30 PRINT "ANOTHER"
40 PRINT "EXAMPLE"
50 END
Ready
■
```

The NEW command has been used to clear the memory of the computer, i.e. to erase it, to make room for the new program. If you did not use the NEW command, the new statements would erase and replace any older statement with identical labels previously typed into memory. This could lead to errors since old state-

ments might be "leftover" in a new program. If you do not use NEW, then every time you type a statement with the label 10, the new version will automatically erase any older version of that statement. But, *beware,* if you had previously written a statement with the label 15, then typed in the above program without a NEW, the statement "15" would remain in the computer's memory and would automatically be incorporated into your new program in sequence (since the label 15 is not used in the new program). Let us demonstrate this.

Type:

```
15 PRINT "*****" ↵
```

Then:

```
RUN↵
```

Your screen should look like this:

```
THIS
*****
IS
ANOTHER
EXAMPLE
Ready
■
```

As you can see, the new PRINT statement, labeled 15, has been automatically inserted in the older program after statement 10. Let's LIST the program. Type:

```
LIST ↵
```

Your screen should look like this:

```
10 PRINT "THIS"
15 PRINT "*****"
20 PRINT "IS"
30 PRINT "ANOTHER"
40 PRINT "EXAMPLE"
50 END
Ready
■
```

You can verify that statement 15 is now part of your program. Remember that every time you type a program statement with a label, it is automatically inserted in the memory of the computer in the proper sequence.

We will now demonstrate that when you use a label number that *already exists,* your new statement will automatically erase the previous one. Let's now use this feature to erase statement 15.

Type:

```
15 PRINT "....."
```

Then:

```
RUN ↵
```

Your screen should look like this:

```
THIS
.....
IS
ANOTHER
EXAMPLE
Ready
■
```

As you can see, your new PRINT statement with the label 15 has superceded the previous one. Verify it by typing:

```
LIST ↵
```

Your screen should look like this:

```
10 PRINT "THIS"
15 PRINT "....."
20 PRINT "IS"
30 PRINT "ANOTHER"
40 PRINT "EXAMPLE"
50 END
Ready
■
```

To avoid accidents, whenever you write a new program, you should use the NEW statement to clear your computer's memory and avoid interference by "leftover" statements from previous use.

"That's how I erase !"



Let's now erase statement 15. There are many ways to do this. Here we will type:

```
15 ↵
```

This statement merely consists of a label. This is called an *empty statement.* Statement 15 does nothing, except erase any previous version. Now type RUN. You should see the following on your screen:

```
THIS
IS
ANOTHER
EXAMPLE
Ready
■
```

Be careful. This feature can be dangerous. If you type:

```
20 ↵
```

by accident, and hit return, you will erase the previous version of statement 20, and replace it with an "empty" statement that does nothing. To avoid surprises, always verify your program listing prior to execution.



Summary

We have now learned how to write elementary BASIC programs that display information on the screen. We have written a BASIC program using labeled statements. We have discussed why programs should be preceded with the NEW command and terminated with the END statement. We have seen that a program is stored automatically in the computer's memory as it is entered, and that it can be executed by typing the RUN command. We have also seen how a program listing can be displayed with the LIST command.

We have learned that the execution of program statements is in the order of the labels. If you duplicate a label number either intentionally or by mistake, the new statement will automatically erase any previous statement with the same label number. Also, if at any time you add a line with a new label, the interpreter will automatically insert it in its proper sequence within the program.

In this chapter, we have introduced many new concepts. If you truly want to learn how to program, it is essential that you start practicing what you have learned. Several self-test exercises follow. You are strongly encouraged to try them out. Answers to selected exercises are given at the end of this book.

Exercises

2-1: Write a program that prints the following: "HAVE A GOOD DAY."

2-2: Write a program that prints:

```
AAAAA
BBBB
CCC
DD
E
```

2-3: Write a program that prints:

```
*******
* TITLE *
*******
```

2-4: Define the following terms:

a. label

b. deferred execution

c. immediate execution

d. empty statement

e. cursor

f. control key

g. keyword

h. prompt

2-5: Why is "PRINT" used as a command to display on the screen?

2-6: Can you execute a whole program by typing statements one at a time in the immediate mode?

2-7: Why use NEW before typing a new program?

2-8: Can you type labeled program statements out of sequence?

2-9: Give examples of some AMSTRAD BASIC commands.

2-10: Is the following statement a valid way to display the word EXAMPLE?

```
PRINT EXAMPLE
```

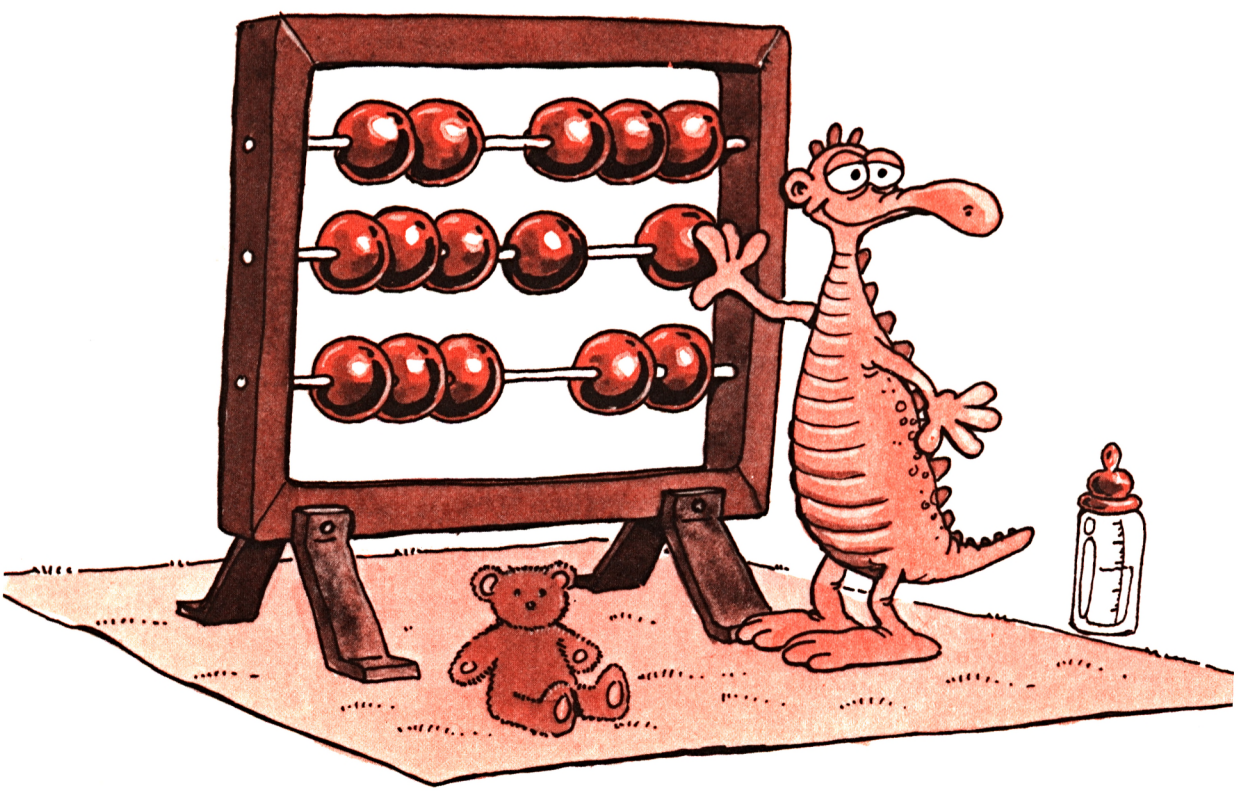
2-11: What is the use of the RETURN key?

2-12: Explain how to erase statement 20 in a program.

2-13: If you have already typed statement 30 and wish to substitute a new statement 30, do you have to erase the old one first?

2-14: Write a program that displays the following:

```
TTTTTT    H   H   EEEEEE
  TT      H   H   EE
  TT      H   H   EE
  TT      HHHH    EEEE
  TT      H   H   EE
  TT      H   H   EE
  TT      H   H   EEEEEE
```



Calculating with BASIC



In this chapter, we will start using numbers. We will display them and add, subtract, multiply, and divide them. We will learn to perform computations using the simple arithmetic operators, and we will describe the other important built-in operators in BASIC.

Printing Numbers

So far we have only printed text. Let's now print a number. Type:

```
PRINT 3 ↵
```

The result should be:

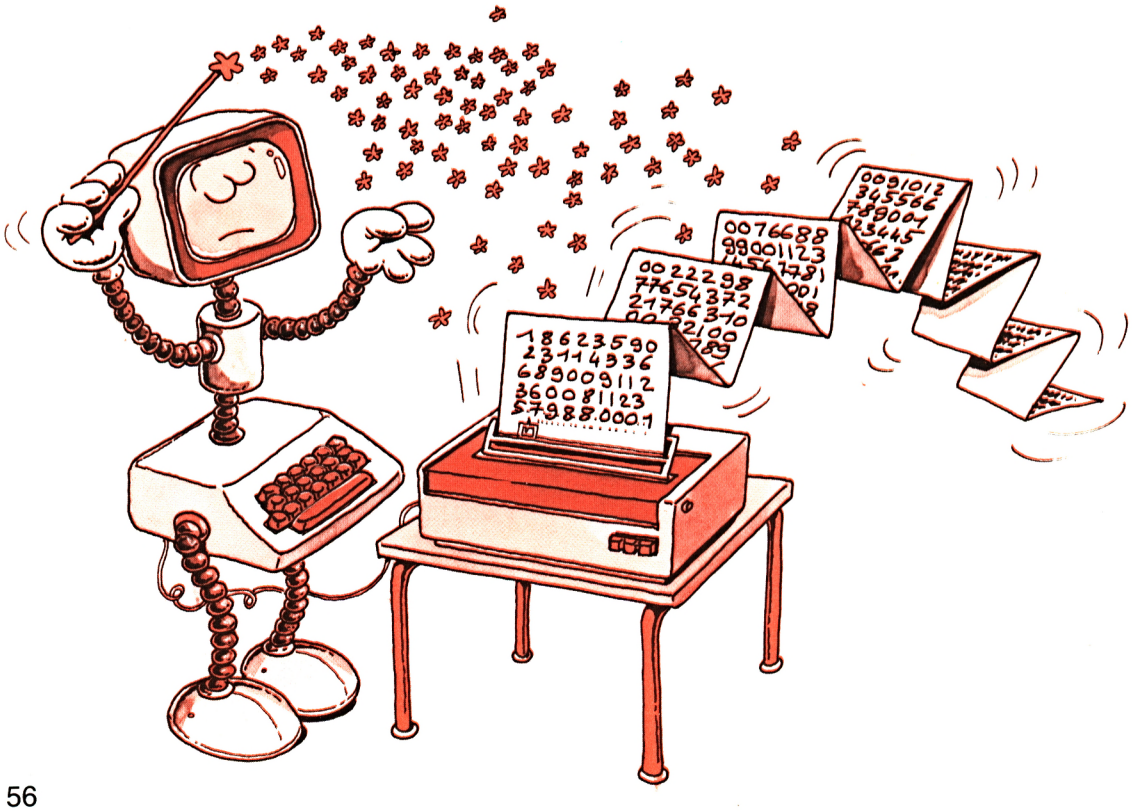
```
3
```

Recall from our discussion in Chapter 2 that this statement is in the immediate mode, where an instruction does not need to be preceded by a label and is executed immediately. In this chapter, we will write all of the examples in the immediate mode, so that you can execute them by pressing only a few keys.

Notice that in BASIC, numbers do not need to be enclosed in quotes, only text. The use of quotes allows the interpreter to easily differentiate between user-provided text and the BASIC reserved words, such as PRINT. The text inside the quotes is called a *string,* and a string may include numbers.

Let's now try to print several large numbers, say 100, 1000, 10000, etc. When you try to print a number with more than nine digits, you will notice that something odd happens—a decimal point appears to the right of the first digit, an "E" appears after the last digit, followed by a plus sign and a number. Your AMSTRAD BASIC interpreter has converted your number into scientific, or exponent, notation. We'll discuss scientific notation shortly.

"I'll show you more ways to print numbers."



Your BASIC interpreter allows you to use decimal numbers. Type:

```
PRINT 1.5 ↵
```

In computer jargon, decimal numbers are called *floating-point numbers* or *real numbers*. A BASIC interpreter that allows the use of floating-point numbers is called a *floating-point BASIC*.

Scientific Notation

Let's discuss decimal numbers further. As with integers, in the case of decimal numbers, the interpreter will only retain a set number of digits. For example, the correct value of one third is:

```
0.33333333333 . . . (etc.)
```

Inside the computer, this value may be stored as:

```
0.333333333 (nine significant digits are retained besides the zero)
```

The correct value is said to have been *truncated* (cut) to nine digits. (*Note:* This is an approximation, but it is generally sufficient.)

Your BASIC interpreter uses a *scientific representation* for these numbers. When a number becomes very large or very small, it will be displayed in scientific or *exponent* notation to save space.

Here is an example:

```
3.2E6
```

means

$$3.2 \times 10^6 = 3200000$$

10^6 means 10 to the power 6, i.e., 10 multiplied 6 times by itself:

$$10 \times 10 \times 10 \times 10 \times 10 \times 10 = 1000000$$

Similarly

```
1.12E-7
```

means

$$1.12 \times 10^{-7} = 0.000000112$$

10^{-7} means 1/10 to the power 7, i.e., 1/10 multiplied 7 times by itself (1/10 is 10^{-1}).

Doing Arithmetic

Let's now perform simple arithmetic calculations. Type:

```
PRINT 2 + 2
```

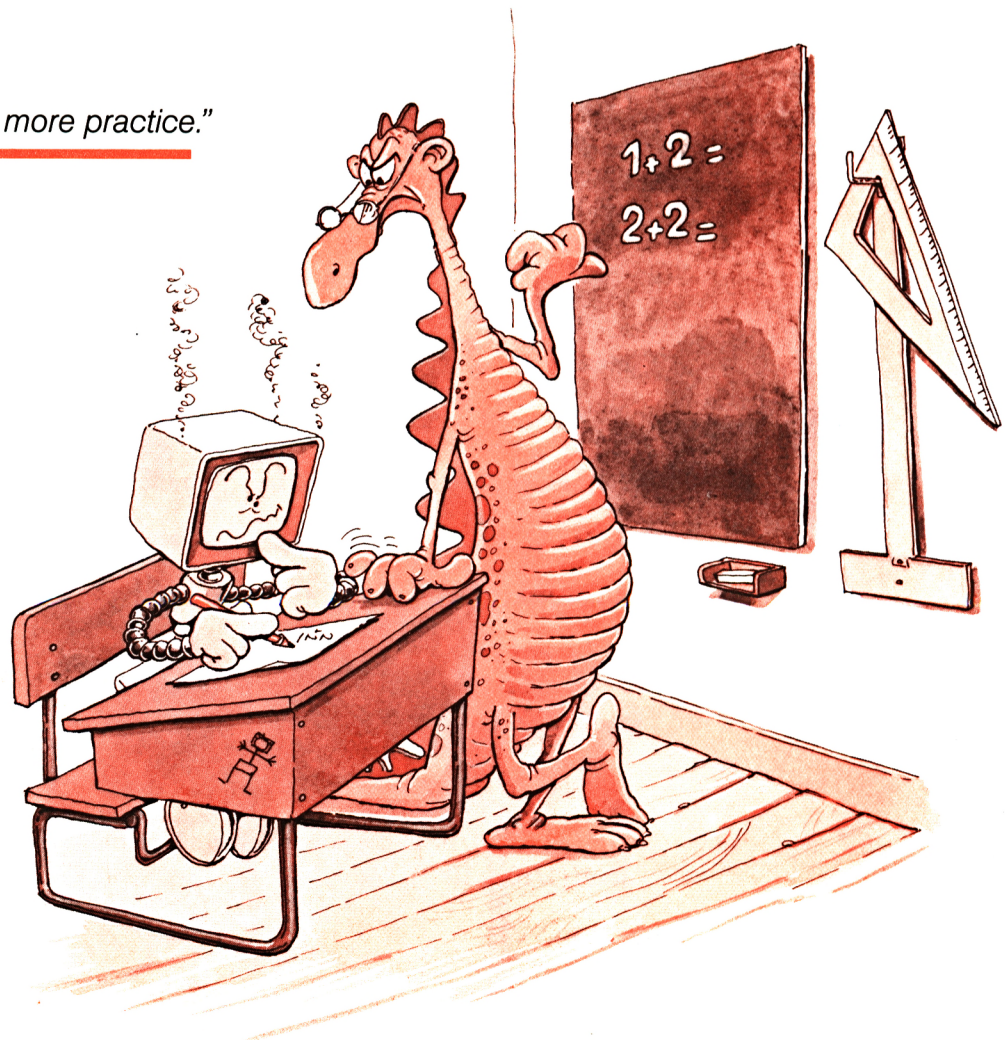
The result appearing on your screen should be:

```
 4
```

We have just performed our first arithmetic computation. The addition symbol, +, is called an *operator.* An operator is a symbol that represents an operation to be performed on one or more operands. BASIC provides at least five built-in arithmetic operators:

- – *(minus)*
- + *(plus)*
- * *(multiplication)*
- / *(division)*
- ↑ *(exponentiation or power)*

"You need more practice."



Now try this example. Type:

```
PRINT 2 * 3
```

The result should be 6. The * symbol is the symbol for multiplication. The usual multiplication symbol, ×, could be confused with the letter X, so programming languages use the * symbol instead.

Here are other examples of valid arithmetic statements:

```
PRINT 1 + 2 * 3
```

The result is

```
7
```

```
PRINT 3 - 2
```

The result is

```
1
```

```
PRINT 8/2
```

The result is

```
4
```

```
PRINT 1 + 2 + 3 + 4
```

The result is

```
10
```

Since AMSTRAD BASIC allows decimal numbers, the following statement is also legal:

```
PRINT (6/3 + 12/4) / 2
```

The result is

```
2.5
```

Note that parentheses have been used in this example to clarify

the grouping of operations. Let's try another example. Type:

```
PRINT 2 + 3 + 4 / 2 ↵
```

The result is:

```
7
```

The division (/) was performed first on the 4. This is due to the fact that in BASIC, if given a choice (i.e., if parenthesis are not used), the division (/) or the multiplication (*) will take place before the addition (+) or the subtraction (−). If you had intended to divide the group 2 + 3 + 4 by 2, then it would have been necessary to type:

```
PRINT (2 + 3 + 4) / 2 ↵
```

The result would then be:

```
4.5
```

The division would then have been performed on the group (2 + 3 + 4). It is good practice to use parentheses freely, to avoid any confusion. For example, the following expression (or group of values and operators)

$$\frac{1 + 2 + 3}{4 + 5} \times 3$$

could be translated into the following BASIC expression:

```
((1 + 2 + 3) / (4 + 5)) * 3
```

or

```
(1 + 2 + 3) / (4 + 5) * 3
```

because execution proceeds from left to right when operators have the same precedence (standing), i.e., the division occurs here before the multiplication.

If you were to write the following in BASIC:

```
(1 + 2 + 3) / ((4 + 5) * 3)
```

it would be equivalent to:

$$\frac{1 + 2 + 3}{(4 + 5) \times 3}$$

Use parentheses to denote groups. Be sure to make sure there is always a matching right parenthesis for each left one.

Let's now use our new computing skill to display useful values.

Printing Formats

If you type:

```
PRINT "THE PRODUCT OF TWO AND THREE IS "; 2 * 3 ↵
```

The result will be:

```
THE PRODUCT OF TWO AND THREE IS 6
```

You will note that after you type space before the last quotation mark in the above PRINT statement, the cursor jumps to the beginning of the next line. This is because some of the lines we ask you to type in this book are longer than 40 characters. Unless your AMSTRAD is operating in mode 2, 40 characters is the maximum that will fit on your screen. This will have no effect on what happens as long as you are sure to press the ENTER key at the end of each line, as shown by the ↵ symbol. This enters the line into the computer's memory.

In the above PRINT statement, we have mixed text and numbers, separated by a semicolon. More precisely, we have used an *expression,* 2 * 3, rather than a number.

Now, type:

```
PRINT "THE PRODUCT OF TWO AND THREE IS, 2 * 3" ↵
```

and you will get:

```
THE PRODUCT OF TWO AND THREE IS, 2 * 3
```

This is a valid BASIC statement, but not the one you had intended. Remember, everything within quotes is displayed literally. The comma or semicolon must be *outside the quotes* to work correctly.

A PRINT statement may be used to print several items on the same line. The items may be separated by a comma, a space, a semicolon or there may be no space between them (which makes them hard to read). A comma between the items results in a large space. Like a tab stop on a typewriter, the comma symbol is used to create *tabs,* i.e. fields on the screen. This technique is convenient for displaying tables.

Items separated by a semicolon in the PRINT statement are printed right next to one another on the line. However, numeric items are usually printed with a space on either side of them.

Let's try this new feature. Type:

```
PRINT 1;2;3 ↵
```

Your display should show:

```
 1  2  3
```

Now type:

```
PRINT 1,2,3 ↵
```

Your display should show:

```
1       2       3
```

To see how text and numeric items are printed, type:

```
PRINT "FRE"; "D IS"; 5; "TODAY"
```

You will see:

```
FRED IS 5 TODAY
```

Let's now compute the value-added tax for a sale of £1234. The tax rate is 6.5%. AMSTRAD BASIC allows decimal numbers. The statement is:

```
PRINT "VALUE ADDED TAX IS"; 1234 * 6.5 / 100 ↵
```

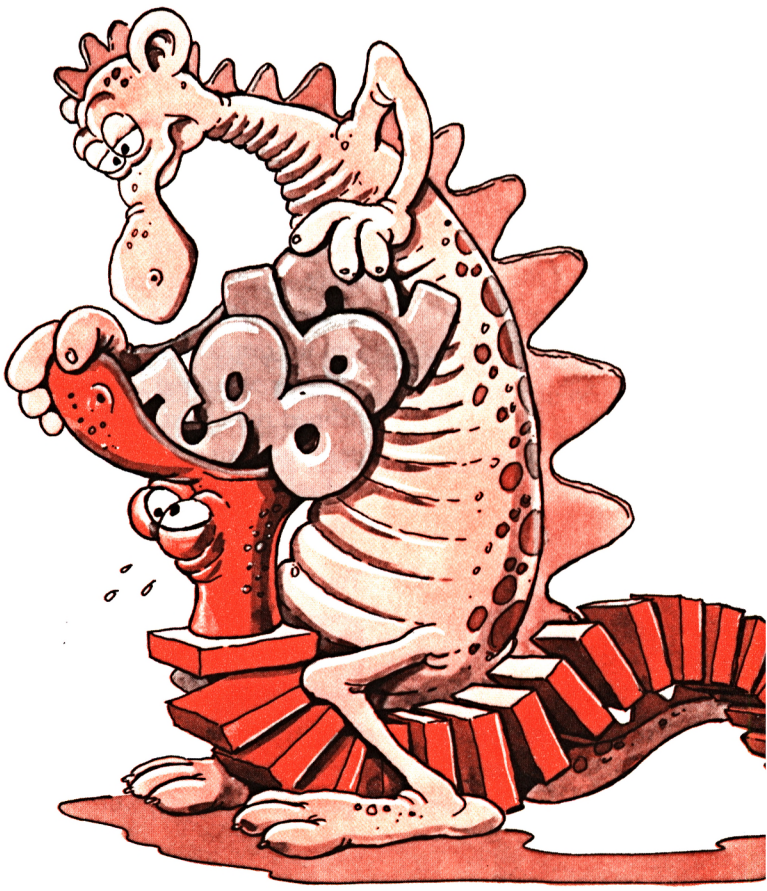
The result is:

```
VALUE ADDED TAX IS 80.21
```

We could also type:

```
PRINT "VALUE ADDED TAX IS"; 1234 * 0.065 ↵
```

and we would obtain the same result. There are many equivalent ways to write a program.



Only feed reasonable numbers to your program.

You can print many items on a line. Look:

```
PRINT 1;2;3;4;5;6;7;8;"MANY ITEMS" ↵
```

The screen will show:

```
1 2 3 4 5 6 7 8 MANY ITEMS
```

We have now learnt how to perform simple arithmetic computations, and how to display the results. Let's use this new skill to solve some simple problems.

Application Examples

Let's compute a car's mileage in miles per gallons. The mathematical formula is:

```
MILEAGE = DISTANCE (in miles) ÷ PETROL (in gallons)
```

Let's assume that the distance was 310 miles and the amount of petrol used was 10.2 gallons. Here is the statement written in BASIC:

```
PRINT "MILEAGE IS"; 310 / 10.2 ; "MPG" ↵
```

For metric readers we will now convert this into liters per kilometers. One gallon is 4.5 liters. One mile is 1.6 kilometer. The consumption in liters per kilometer is:

```
PRINT "CONSUMPTION IS"; (10.2*4.5)/(310*1.6); "L PER KM" ↵
```

Here is another simple problem. Given a temperature in Fahrenheit, the Celsius equivalent is computed by the formula:

$$\text{CELSIUS value} = (\text{FAHRENHEIT value} - 32) \times 5/9$$

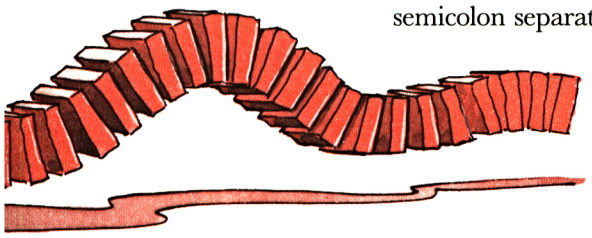
To compute the Celsius equivalent of 79°F, type:

```
PRINT "79 DEGREES F ="; (79 - 32) * 5/9; "DEGREES C"
```

The result is:

```
79 DEGREES F = 26.1111111 DEGREES C
```

To be complete, note that 5/9 has not been enclosed in parentheses, as it does not matter whether * or / is executed first. Also note that the designers of AMSTRAD BASIC have allowed us to write some PRINT statements without using semicolons. Where text and numeric items alternate in the PRINT statement, the semicolon separator may be omitted.





Summary

In this chapter, we have learnt how to perform arithmetic computations and how to display text and results on the same line. We have used this new skill to automate the computation of simple formulas by writing one-line BASIC statements.

So far we have specified all the values within the BASIC statement itself. What we now want to do first to write a program, then supply values repeatedly from the keyboard so different values can be used with the program without having to rewrite it. We will accomplish this with *variables*. This is the topic of the next chapter.

Exercises

3-1: Write a BASIC statement that computes:

$$\frac{5 + 6}{1 + 2 \div 3}$$

3-2: Write a BASIC statement that computes:

$$1 + 1/2 * \frac{1}{1 + 1/2}$$

3-3: Write a BASIC statement that computes the Fahrenheit equivalent of 20°C.

3-4: Given a speed of 100 km per hour, compute the equivalent speed in m.p.h. (1 mile = 1.6 km)

3-5: Compute the number of seconds in a day, a week, a month, and a year.

3-6: Assuming an average speed of 45 m.p.h., compute the time needed to travel 250 miles.

3-7: Assuming 365 days in a year, compute the number of days you have lived so far.

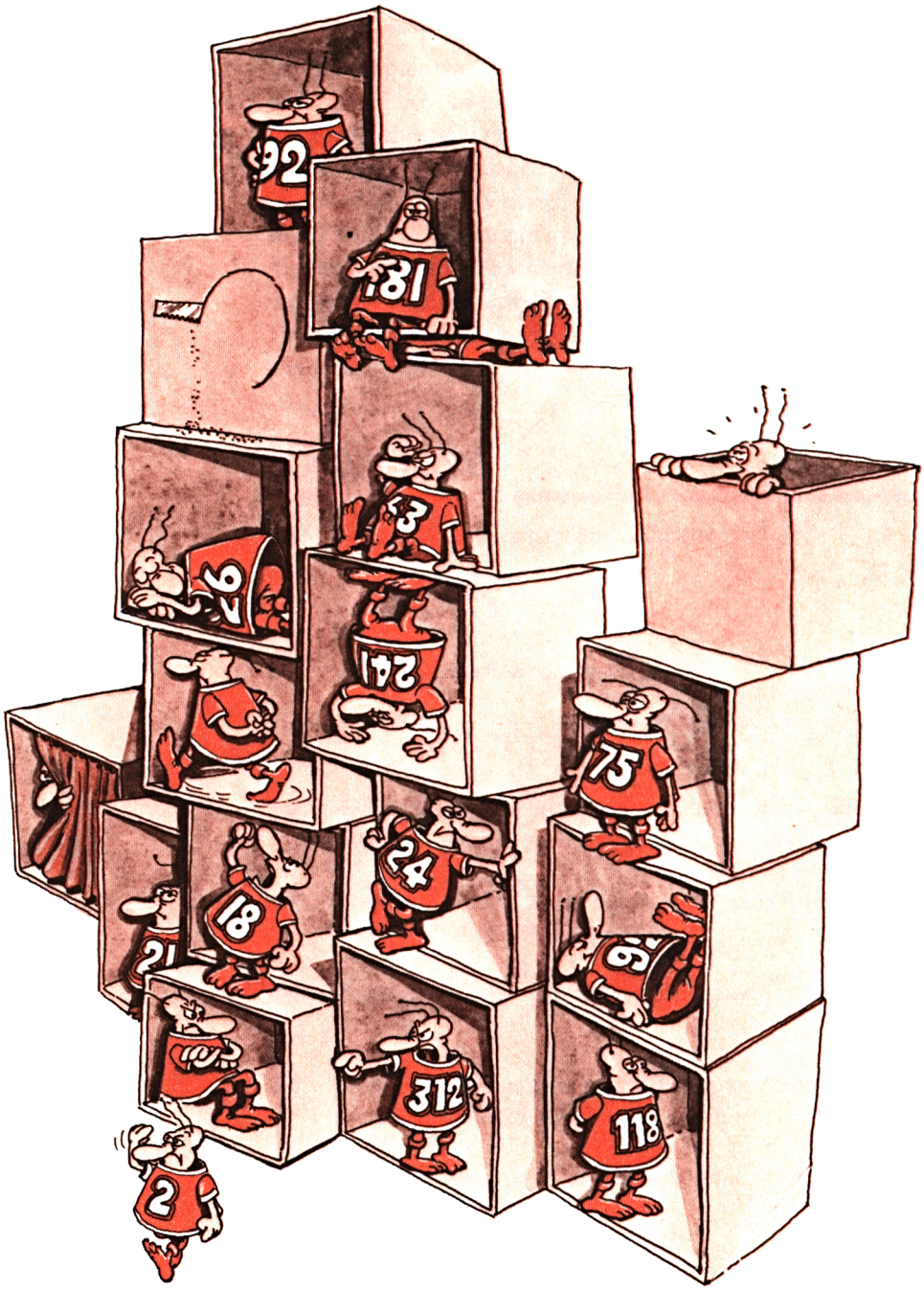
3-8: Compute the equivalent annual salary for a person, given the:

a. weekly pay (there are 52 weeks in a year)

b. fortnightly pay (multiply by 26)

c. monthly pay

d. hourly pay (multiply by 2080)


92
181
3
9
241
75
24
18
21
6
312
118
2

Memorizing Values and Using Variables



In this chapter, we will learn to write programs that can be used repeatedly, without change, and that will display different results, depending on the data supplied at the keyboard. So far, to obtain the result of an arithmetic computation, such as 2 + 3, we have had to include in the program statement the numbers to be operated on. We will now learn to write programs that can be executed with different data each time. The programs will specify the operations to be performed. The data will be supplied at the keyboard by the user at the time the program is executed. This makes the program reusable.

In addition, we will introduce the concept of a variable and learn how to use the INPUT statement, and other ways of giving values to a variable.

Let's now begin by learning how to supply information to a program while it is running.

The INPUT Statement

Type the following program. (**Note:** we will no longer display ↵ to indicate ENTER at the end of each line):

```
10 INPUT A
20 PRINT A, 2 * A, 3 * A
30 END
```

Now execute this program by typing RUN, as usual. You should see on your screen:

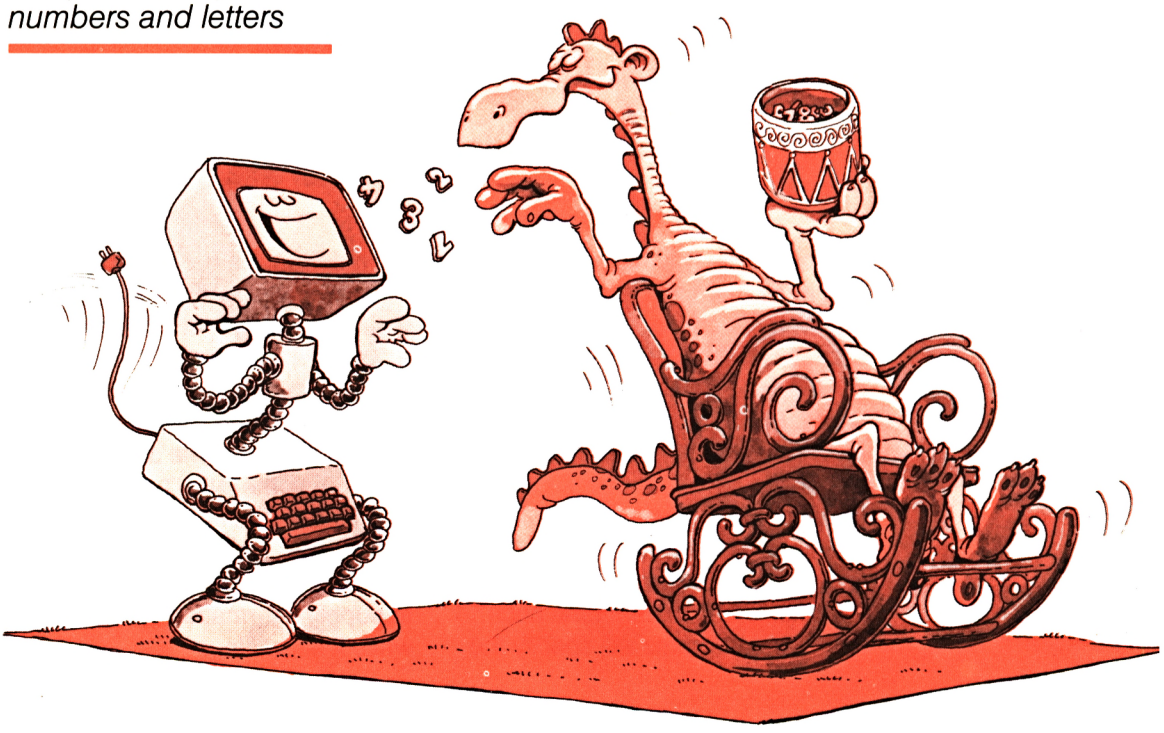
```
?■
```

A "?" appears on your screen with a cursor next to it to remind you to type in the requested data.

Now, type in a number, say 3. Terminate your input by pressing the ENTER key as usual. Your screen should now show:

```
3    6    9
Ready
■
```

On INPUT, your computer accepts numbers and letters



Your program has been executed. Let's see what happened. The first line was:

```
10 INPUT A
```

This statement asked you to provide a number at the keyboard. The program displayed a ?, and stopped—waiting for your input. The value 3 you supplied was then read and stored in A. "A" is called a *variable.* It is a name used to store a value. Formally, a *variable* is a name given to a memory location. Examples of some variable names are: A, B, C, F, Z1, G2. AMSTRAD BASIC allows variable names with more than 2 characters, for example: NUMBER1, SUM, TAX, RESULT.

The second statement was:

```
20 PRINT A, 2 * A, 3 * A
```

This statement resulted in the printing of 3, 2 * 3, 3 * 3, or:

```
3   6   9
```

Using the INPUT statement, it is also possible to enter several values at the same time. Here is an example. Type:

```
10 INPUT A,B
20 PRINT A; A * 2; B; B * 2
30 END
```

Now RUN the program. You should see the usual "?" appearing on your screen. Type two numbers, say 2 and 3, separated by a comma, then press return. You should now see on your screen:

```
2 4 3 6
```

Let us examine what happened. The first line of the program was:

```
10 INPUT A,B
```

This statement resulted in a request for two values from you, which were then stored in the variables A and B. Remember that variables are names of memory locations. The contents of A and B were initially empty, but they became 2 and 3, respectively. The second statement of the program was:

```
20 PRINT A; A * 2; B; B * 2
```

This statement resulted in printing the values 2, 2 * 2, 3, 3 * 2, or:

```
2 4 3 6
```

Let's try running this program again. This time try typing:

```
5,8
```

and the results should be:

```
5 10 8 16
```

We can use this program repeatedly and obtain new results by typing in new values at the keyboard. We have made our program reusable by using variable names (A and B), instead of explicit values.

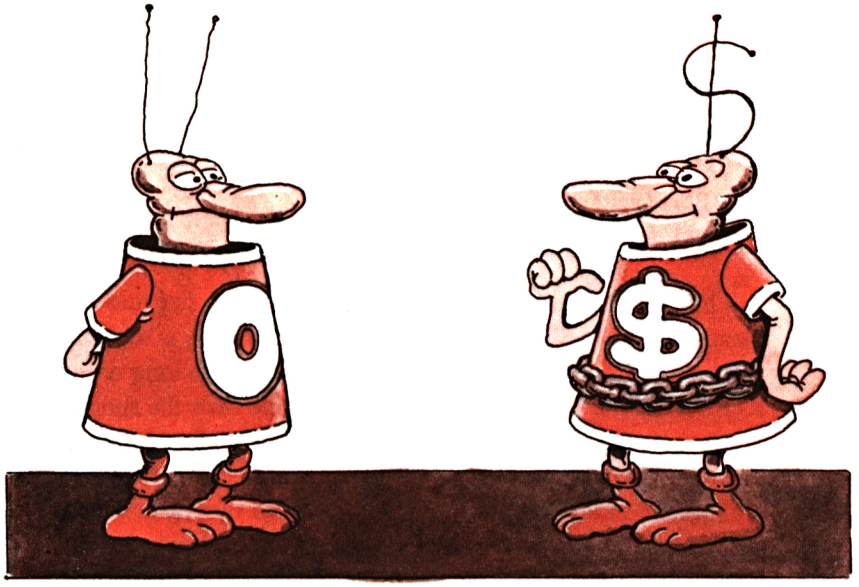
In order to make this program truly reusable, let's now improve its style and readability. Suppose that we want to save the program, and RUN it again in several days. We might forget what it does or we might not remember how many values we must supply. We can rewrite the program to display this information on the screen. Here is an improved version:

```
10 PRINT "*THIS PROGRAM MULTIPLIES*"
15 PRINT "*ANY TWO NUMBERS BY 2*"
20 PRINT "TYPE TWO NUMBERS"
30 INPUT A,B
40 PRINT "FIRST NUMBER:"; A, "DOUBLE:"; 2 * A
50 PRINT "SECOND NUMBER:"; B, "DOUBLE:"; 2 * B
60 END
```

and here is the resulting display: (**Note:** throughout this text, we will display data provided by the user in **boldface** type.)

```
*THIS PROGRAM MULTIPLIES*
*ANY TWO NUMBERS BY 2*
TYPE TWO NUMBERS
? 5,7
FIRST NUMBER: 5        DOUBLE: 10
SECOND NUMBER: 7       DOUBLE: 14
```

We have now learnt how to supply numeric data to the program by using the INPUT statement. We have also introduced the concept of variable. Let's now learn how to use these techniques effectively and how to develop more complex programs.



*"Hello Numeric!
I'm a String.
I store a chain
of characters.
You can recognize
me easily.
Look at
my antennas!"*

The Two Types of Variables

There are two *types* of variables in BASIC: *numeric* and *string*. Numeric variables represent numbers; string variables represent text. These two types of variables look different; a string variable has a "$" at the end of the name. They are also used differently. For example, you can add numbers, but not text. Let's first learn about numeric variables, then about string variables.

Numeric Variables

Let's learn the rules for naming a numeric variable; then we will learn how to use these variables effectively. We have already used two numeric variables, called A and B, at the beginning of this chapter. We gave them a value by INPUTing numbers from the keyboard. Let's now learn how to name a variable.

When naming a variable, all BASICs, including the original Dartmouth BASIC, allow the use of one letter, optionally followed by a single number. Here are examples of valid variable names:

A (*one letter*)
B (*one letter*)
Z (*one letter*)
A1 (*one letter and one digit*)
A2 (*one letter and one digit*)
B2 (*one letter and one digit*)

With this definition, the following names are not legal:

12 (*does not start with a letter*)
A2B (*too many characters*)
BA (*only one letter permitted*)
1B (*must start with a letter*)
AB1 (*too long—also only one letter permitted*)

The advantage of short names is to reduce the size and complexity of the BASIC interpreter. The disadvantage is that short names are difficult to remember. For example, the long name RESULT is more descriptive and memorable than the short name R. To improve readability most newer BASICs allow long names.

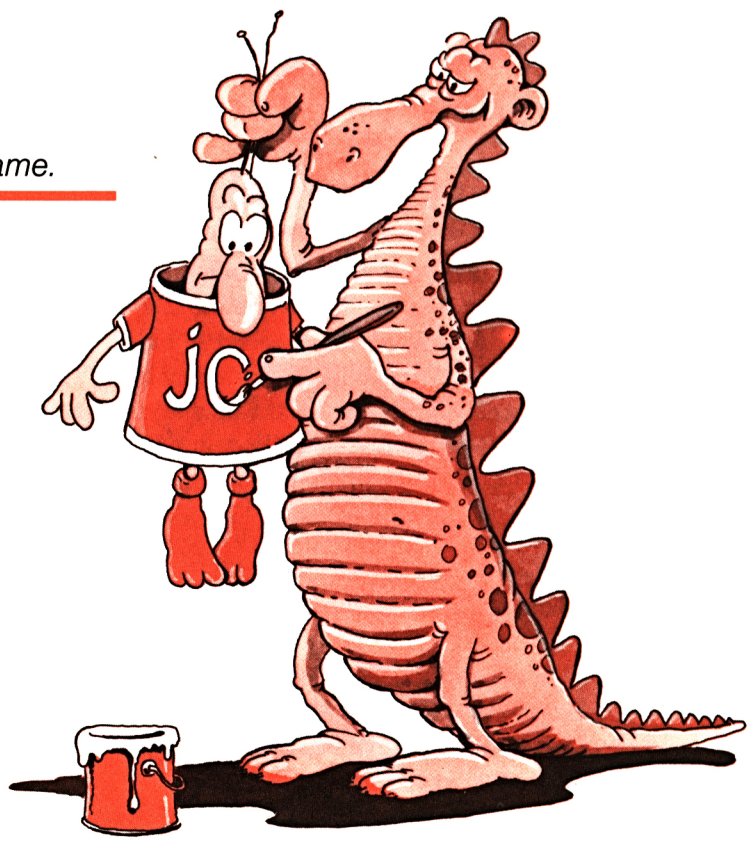
AMSTRAD BASIC allows the use of variable names that begin with a letter, and that are optionally followed by one or more letters or numbers, and do not exceed 40 characters in length. The only punctuation permitted in AMSTRAD BASIC variable names is the full stop. For example, the following are legal longer variable names in AMSTRAD BASIC:

WINNER	STUDENT1
LOSER	STUDENT2
RESULT	STUDENT14
SUM.OF.DIGITS	CASE24T3

The following are not legal:

3TIMES (*starts with a digit*)
A-ONE (*illegal symbol*)

Every variable must have a name.



Clearly a program with long, explicit variable names is more readable. In this chapter, we will use both short and long names, so that you can get used to both conventions. Remember that long names are simply a matter of convenience and will not affect the program in any way.

There is one more restriction on names: you cannot use a name that is a *keyword,* i.e., a name that has a meaning to your BASIC interpreter. For example, you may not use LIST, END or RUN as variable names. (**Note:** a list of keywords appears at the end of this book, as well as at the end of the AMSTRAD User Instructions.)

Now that we know how to create legal names for numeric variables, let's learn how to give a name to a piece of text known as a *string.*



"I'm a string variable.
I contain text.
My name ends with a $."

String Variables

First, let us introduce strings. Here are examples of strings:

"RESULT"

"THIS IS AN EXAMPLE"

"MY NAME IS JOHN"

"25 TIMES 4 = "

Note that a string is normally enclosed in quotes; otherwise, it could be confused with a variable name. A string may contain any sequence of characters, except the quotation symbol. The length of a string is limited to 242 characters.

Let us now introduce string variables. When the value stored in a variable is text (that is a string), rather than a number, the variable is called a *string variable.*

A string variable's name is just like a numeric variable's name, except that it must have a "$" at the end. Here are some valid short string variable names:

A$

R$

A1$

B5$

Since your AMSTRAD BASIC allows long names, here are several more valid names:

NAME$

FIRST$

CITY$

UNIT25$

A name that is a keyword plus a dollar sign, for example AND$, is illegal, although the name ANDY$ *is* legal.

Let's now illustrate the use of string variables, with a program that greets the user by name.

```
10 PRINT "I AM HAL, THE COMPUTER."
20 PRINT "WHAT IS YOUR FIRST NAME";
30 INPUT FIRST$
40 PRINT "AND WHAT IS YOUR LAST NAME";
50 INPUT LAST$
60 PRINT "HELLO, "; FIRST$; " "; LAST$; "!"
70 PRINT "NOW I KNOW YOUR NAME!"
80 PRINT "I LIKE "; FIRST$; " AS A FIRST NAME."
90 END
```

Here is a sample run. Note that the characters you type appear in boldface type.

```
I AM HAL, THE COMPUTER.
WHAT IS YOUR FIRST NAME? JOHN
AND WHAT IS YOUR LAST NAME? SULLIVAN
HELLO, JOHN SULLIVAN!
NOW I KNOW YOUR NAME!
I LIKE JOHN AS A FIRST NAME.
Ready
■
```

You can now communicate with your computer! Let's point out some of the basic features of this program. Let's start with line 20:

```
20 PRINT "WHAT IS YOUR FIRST NAME";
```

Note that this line is terminated with a semicolon. The semicolon means "display the next character immediately after this text." The result of statement 20 and your response to statement 30 is:

```
WHAT IS YOUR FIRST NAME? JOHN
```

If you had written:

```
20 PRINT "WHAT IS YOUR FIRST NAME"
```

with no semicolon at the end, the result would have been:

```
WHAT IS YOUR FIRST NAME
? JOHN
```

Naturally, you may choose either way. The format is a matter of preference. But remember, if you want the characters you type to appear on the same line as the preceding message, use a semicolon at the end of the PRINT statement. Otherwise, the next position on the screen will be the beginning of the following line.

Now that we know more about numeric and string variables, let's use them as we continue our dialogue with Hal, the computer. Let's add the following to our program:

```
90 PRINT "WHAT IS THE CURRENT YEAR (2 DIGITS)";
100 INPUT CURRENTYEAR
110 PRINT "WHICH YEAR WERE YOU BORN (2 DIGITS)";
120 INPUT YEARBORN
130 PRINT "DEAR "; FIRST$;
140 PRINT ",THIS YEAR YOU ARE OR WILL BE";
    CURRENTYEAR - YEARBORN
150 END
```

Here is a sample dialogue. Again, the characters you type are shown in boldface type.

```
WHAT IS THE CURRENT YEAR (2 DIGITS)? 85
WHAT YEAR WERE YOU BORN (2 DIGITS)? 50
DEAR JOHN, THIS YEAR YOU ARE OR WILL BE 35
```

Let us examine the program in detail. The statement

```
90 PRINT "WHAT IS THE CURRENT YEAR (2 DIGITS)";
```

prints the obvious message. Again a semicolon is used so that the two digits to be input will be displayed on the same line. In the next statement,

```
100 INPUT CURRENTYEAR
```

CURRENTYEAR is a numeric variable. The value 85 that you typed will be read into it. From now on, whenever the name CURRENTYEAR is used, the value 85 will be substituted automatically by the BASIC interpreter. This will happen in statement 140.

The statement

```
110 PRINT "WHICH YEAR WERE YOU BORN (2 DIGITS)";
```

is just like statement 90. Note the semicolon at the end. The statement

```
120 INPUT YEARBORN
```

is just like statement 100. YEARBORN is a new numeric variable. It will soon contain the value 50. In statement 140 we use this variable name. Note that the value 50 is substituted automatically by the interpreter. Let's examine this:

```
130 PRINT "DEAR "; FIRST$
140 PRINT ",THIS YEAR YOU ARE OR WILL BE";
    CURRENTYEAR - YEARBORN
```

When this statement is executed, the following is displayed:

```
DEAR JOHN, THIS YEAR YOU ARE OR WILL BE 35
```

Let's break this display down and examine it.

DEAR

(*This is called a literal string. A literal string is a string that was part of a program statement.*)

JOHN

(*This is the string value that is read at the keyboard and stored in the string variable named FIRST$. It remains there as long as you don't use the NEW command or enter a new value into FIRST$.*)

THIS YEAR YOU ARE OR WILL BE

(*This is another literal string.*)

35

(This is the result of: CURRENTYEAR − YEARBORN

i.e. 85 − 50 = 35.)

In typing and examining this program, you may already have developed some frustrations. For example, you may have wanted to enter actual dates, including the day and month, so that you can compute your age, as of today. However, this requires comparing today's month and day with that of the birthdate's. To do this, we must learn about a new BASIC statement, the

```
IF (condition) THEN (do)
```

statement. We'll discuss this statement in detail in Chapter 7. Another thing that you might want to do is to design the program so that it executes repeatedly (so that you don't need to type RUN everytime). We'll learn how to do this in Chapter 6, when we discuss the GOTO statement.

Now that we are familiar with both numeric and string variables, let's learn how we can use them in a longer program—first, by assigning a value to a variable, then by using the counter technique.

Assigning a Value to a Variable

So far, the only way we have given a value to a variable has been with the INPUT statement. For example, when the following statement is executed:

```
20 INPUT A
```

you type in a value, such a 5.2 (followed by a ↵) and the value of A is then 5.2.

There is, however, another way to assign a value to A. This is called the *assignment* statement. Here is an example:

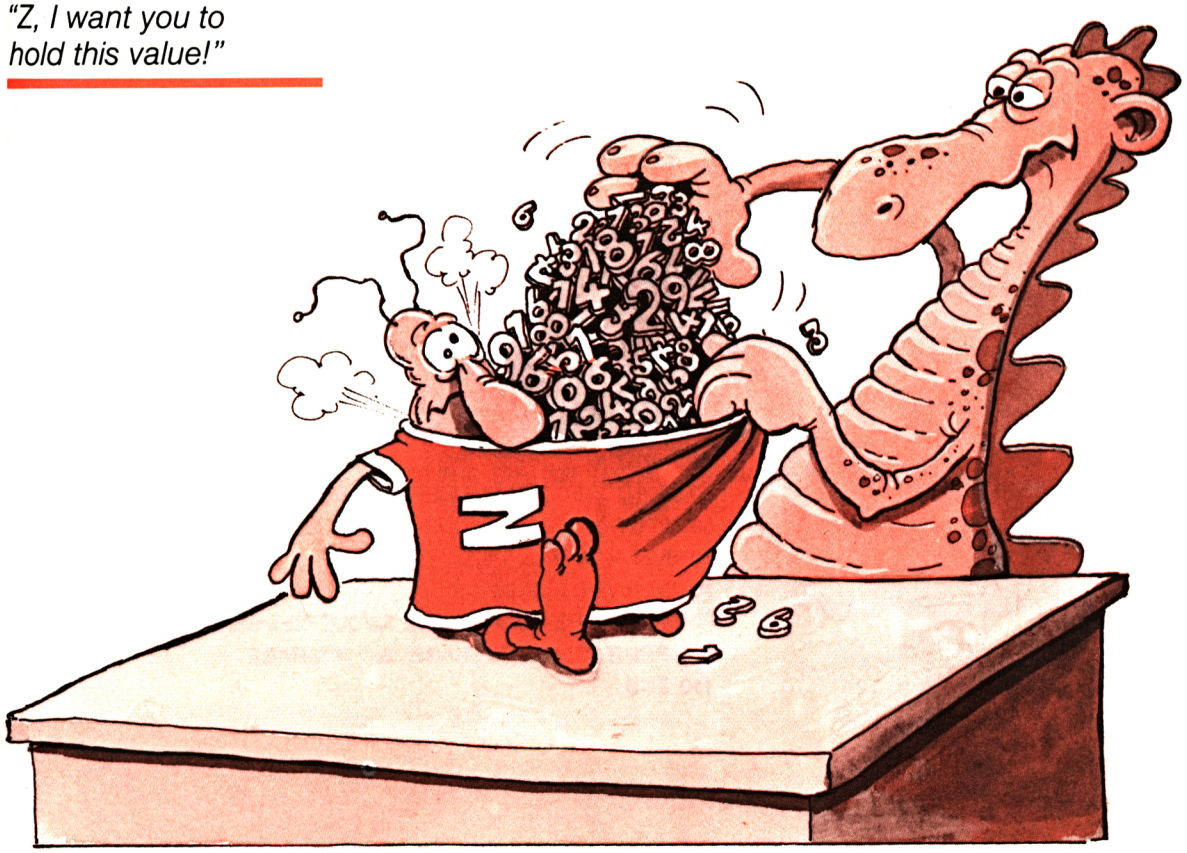
```
10 A = 5.2
```

This statement assigns the value 5.2 to A as part of the program, so that you do not have to supply it from the keyboard. You might also write:

```
10 B = 1
20 C = 2
30 A = B + C
```

As you can see, the value of A will be set at 2 + 1 = 3 when statement 30 is executed.

"Z, I want you to hold this value!"



Let's now show the value of the assignment statement. We will examine two examples. In both examples, we will compute the sum and the average of two numbers. Here is our first program, without the assignment statement:

```
10 PRINT "GIVE ME TWO NUMBERS"
20 PRINT "I WILL COMPUTE THEIR SUM AND AVERAGE"
30 PRINT "FIRST NUMBER PLEASE:";
40 INPUT A
50 PRINT "SECOND NUMBER PLEASE:";
60 INPUT B
70 PRINT "THE SUM OF"; A; "AND"; B; "IS:"; A + B
80 PRINT "THEIR AVERAGE IS:"; (A + B) / 2
90 END
```

And here is a typical RUN:

```
GIVE ME TWO NUMBERS
I WILL COMPUTE THEIR SUM AND AVERAGE
FIRST NUMBER PLEASE:? 24
SECOND NUMBER PLEASE:? 41
THE SUM OF 24 AND 41 IS: 65
THEIR AVERAGE IS: 32.5
Ready
■
```

Notice that the expression A + B is repeated twice in the PRINT statements. This is just a minor inconvenience. However, in a longer program, this would increase the probability of typing errors. In addition, if we were to change the program to use a different formula, much rewriting would be needed.

Here is an equivalent program that uses an *intermediate variable,* called SUM, to store the result. It is easier to read, and less error prone.

```
10 PRINT "GIVE ME NUMBERS"
20 PRINT "I WILL COMPUTE THEIR SUM AND AVERAGE"
30 PRINT "FIRST NUMBER PLEASE:";
40 INPUT A
50 PRINT "SECOND NUMBER PLEASE:";
60 INPUT B
70 SUM = A + B
80 AVERAGE = SUM / 2
90 PRINT "THE SUM OF THE ABOVE NUMBERS IS:"; SUM
100 PRINT "THEIR AVERAGE IS:"; AVERAGE
110 END
```

Two new variables are used:

```
70 SUM = A + B
80 AVERAGE = SUM / 2
```

Using additional variable names has two advantages: the program is clearer, and it is easier to modify. For example, let's assume that we now want to modify this program to obtain the average of three numbers. To do this, we would type:

```
62 PRINT "THIRD NUMBER PLEASE:";
64 INPUT C
70 SUM = A + B + C
80 AVERAGE = SUM / 3
```

We would leave the rest of the program unchanged. We have simply read in a third number, and modified the formulas at a single location. Here is the complete program:

```
10 PRINT "GIVE ME NUMBERS"
20 PRINT "I WILL COMPUTE THEIR SUM AND AVERAGE"
30 PRINT "FIRST NUMBER PLEASE:";
40 INPUT A
50 PRINT "SECOND NUMBER PLEASE:";
60 INPUT B
62 PRINT "THIRD NUMBER PLEASE:";
64 INPUT C
70 SUM = A + B + C
80 AVERAGE = SUM / 3
90 PRINT "THE SUM OF THE ABOVE NUMBERS IS:"; SUM
100 PRINT "THEIR AVERAGE IS:"; AVERAGE
110 END
```

And a typical run:

```
GIVE ME NUMBERS
I WILL COMPUTE THEIR SUM AND AVERAGE
FIRST NUMBER PLEASE:? 5
SECOND NUMBER PLEASE:? 3
THIRD NUMBER PLEASE:? 10
THE SUM OF THE ABOVE NUMBERS IS: 18
THEIR AVERAGE IS: 6
```

We have now learnt two methods for associating a value with a variable:

- We can use the INPUT statement—where a value is supplied at RUN time, i.e., when the program is executed.
- We can use the assignment statement—where a value or a method to compute the value (a formula) is stored within the program itself.

The first method (using the INPUT statement) should be used when you expect that the value explicitly supplied to the variable (that is, not a *computed* value) will be different every time the program is executed.

The second method (using the assignment statements) should be used whenever you are using a formula to compute the value of the variable, or whenever you expect the explicit value (i.e., not a *computed* value) to remain the same every time the program is executed.

Let's now learn the complete rules for writing an assignment statement.

The Syntax of an Assignment

The rules (or syntax) for writing an assignment statement are simple. The general form of an assignment statement is:

```
<variable> = <expression>
```

There must always be a variable on the left and an expression on the right. More precisely, an expression is:

- a number or a variable, or
- a number or a variable followed by an operator (such as +, −, *, /), and another expression.

Here are a few sample expressions:

3	(a number)
A	(a variable)
2 + 2	(number, operator, number)
A + 2	(variable, operator, number)
A + B * 3	(variable, operator, expression)

Expressions may be enclosed in parentheses, for example:

```
3 + (A + 2) / 2
B + ((C * 2) + (D / 2)) / 4
```

You may want to think of an expression as a value, or as something that will be computed and will result in a value, (in other words, as a formula for computing a value).

The equal (=) sign used in an assignment statement is not interpreted the same as in standard mathematics. In an expression it means "receives the value of." For example, you may write:

```
10 A = 1
20 A = A + 1
```

(The expression A = A + 1 would be meaningless or absurd in ordinary mathematics.) In BASIC, after statement 20 is executed, the value of A will be 1 (the previous value of A) + 1 = 2. Remember that in an assignment statement in AMSTRAD BASIC, the = sign means that the variable on the left receives the value of the expression on the right.

Here are examples of valid assignment statements:

A = −3 + 2 (−3 is a negative integer)

B = A + 1

C = (2 * 3) + (A / B)

AVERAGE = SUM / NUMBER

SQUARE = A↑2

X = B↑2 − (4 * A * C)

Let's examine this last assignment and verify that it satisfies our definition:

```
B ↑ 2
```

is

<variable><operator><number>

followed by

```
- (4 * A * C)
```

i.e.,

<operator><parenthesized expression counting as a value>

Inside the parentheses:

```
4 * A * C
```

is

<number><operator><variable><operator><variable>

Yes, it is a valid expression.

The following assignments, however, are not valid:

B + C = SUM	*(only one variable (not an expression) on the left of the = sign)*
2 = A	*(there must be a variable, not a value on the left)*
SUM = B + C (D / 3)	*(missing operator after C)*
(AVERAGE) = (B + C) / 2	*(no parentheses allowed on the left)*
A =	*(missing value on the right)*

Finally, note that at the time an assignment is executed, all variables on the right of the = sign must have received a value. If you write

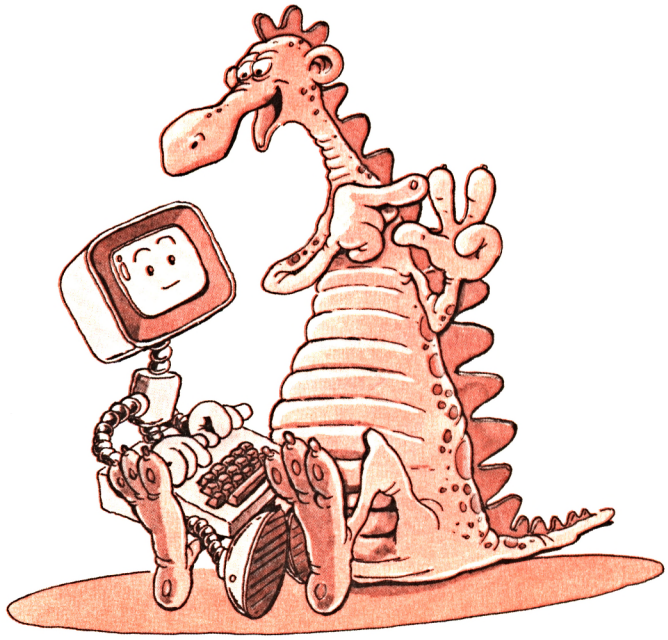
```
10 B = 2
20 SUM = B + C
30 INPUT C
```

the program will fail at line 20, since C does not have a value. You probably meant to write:

```
10 B = 2
20 INPUT C
30 SUM = B + C
```

We've now learnt the syntax of assignments. Let's use this new skill and introduce an important technique that uses assignments: it is called the *counting technique*. We will use it in many of our programs.

"Let's play counter





"I'm a counter variable."

The Variable Counter Technique

Remember that a variable is simply a name given to a memory location. A value can be stored in a memory location by using an INPUT statement or an assignment statement (=). In this example, we want to change the value of a variable repeatedly, in order to count events. The technique we will use to do this is called the *variable counter* technique.

Let's now demonstrate how successive assignments can change the value of the variable N. Type the following in the immediate mode (**Note:** this mode is also called the *calculator* mode):

```
N = 1
```

The value is automatically stored in N. Let's verify that it is. Type:

```
PRINT N
```

The value 1 appears. Now type:

```
N = 2
```

N now contains 2. Type:

```
PRINT N
```

The response is:

```
2
```

The value 2 has been substituted for the value 1 in N. Type:

```
N = 3
```

Now type:

```
PRINT N
```

and verify that the value 3 has been stored in N. The mechanism works. We will soon use this technique to count events. In other words, a variable can be used as an event counter. Here is a preview of an advanced program that counts how many times you supply a number at the keyboard. It stops when you press zero.

```
10 SUM = 0
20 SUM = SUM + 1
30 PRINT "ENTER ANY NUMBER. TYPE 0 TO STOP";
40 INPUT NUMBER
50 PRINT "YOU HAVE ENTERED"; SUM; "NUMBERS"
60 IF NUMBER <> 0 THEN 20
70 END
```

Later on in Chapter 6, we will examine statements like Statement 60 in greater detail. The statements means: IF NUMBER is not

equal to (<>) zero, THEN execute statement 20. Here is a typical run:

```
ENTER ANY NUMBER. TYPE 0 TO STOP? 5
YOU HAVE ENTERED 1 NUMBERS
ENTER ANY NUMBER. TYPE 0 TO STOP? 1
YOU HAVE ENTERED 2 NUMBERS
ENTER ANY NUMBER. TYPE 0 TO STOP? 2
YOU HAVE ENTERED 3 NUMBERS
ENTER ANY NUMBER. TYPE 0 TO STOP? 3
YOU HAVE ENTERED 4 NUMBERS
ENTER ANY NUMBER. TYPE 0 TO STOP? 4
YOU HAVE ENTERED 5 NUMBERS
ENTER ANY NUMBER. TYPE 0 TO STOP? 0
YOU HAVE ENTERED 6 NUMBERS
Ready
■
```

In this program, the value of SUM is *initialized* to 0 in the first statement. It increases by one every time a new number is entered. This is a counter variable. We will see many examples of this technique as we write more programs. Later, we will learn how to tune up such a program, if we don't want to have 0 count as a number when we stop the program.



Summary

In this chapter we have learnt to write programs that can be used repeatedly. These programs will supply new results depending on values provided at the keyboard. We have achieved this by using variables and assigning values to them in various ways.

A variable should be thought of as a name given to a memory location in which a value or text may be stored or accumulated.

We have learnt to change the contents of a variable by using an INPUT or assignment statement. We can now write a simple program that automates a dialogue or a simple computation.

However, our programs now consistently exceed ten lines. They have become long. Let's keep them clear. In the next chapter, before proceeding any further and learning new tools and techniques, we will learn how to write a clear program.

Exercises

4-1: Read four numbers at the keyboard and display the total, the average, and the product of the four numbers.

4-2: AMSTRAD BASIC allows "long variable names." Are the following variable names valid?

a. 24B	e. ALPHA2D	i. PI
b. B24	f. EXAMPLE	j. 3$
c. A+B	g. INPUT	k. THREE
d. APLUSB	h. INPUT1	l. NAME$

4-3: Write a program that asks the name of the user and says: "I THINK I KNOW A (name here)!."

4-4: Write a program that requests:

— the name of an object

— the name of a piece of furniture

— the name of a friend

then says: "DOES YOUR FRIEND (name) HAVE A (object) ON A (piece of furniture)?

4-5: Write a program that requests the colour of your eyes, then says: "I LIKE (colour) EYES."

4-6: Are the following assignments valid:

a. A + 1 = A	d. B + C = A
b. A = A + A + A	e. 3 = 2 + 1
c. A = B + C	f. NUMBER = FIRST + LAST * 2

5

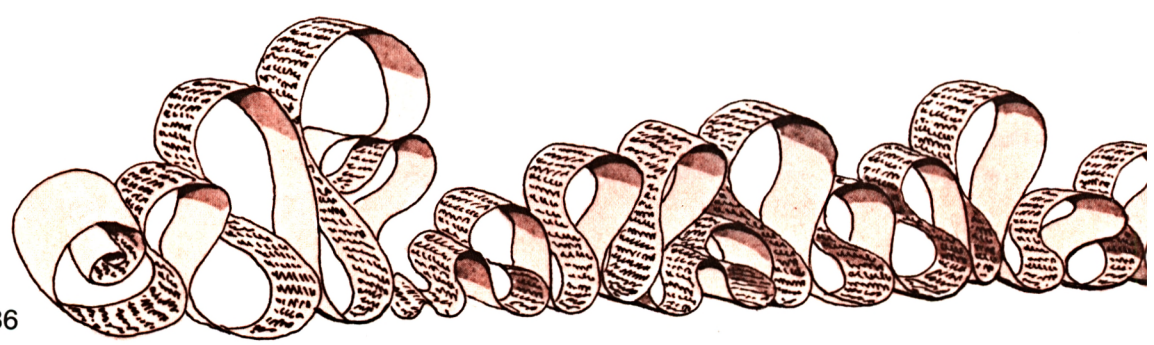
So far, we have written short programs using three different types of statements: PRINT, INPUT, and assignment (=). We have also learnt how to write simple arithmetic expressions. In the chapters that follow, we will learn new techniques and statement types, and we will write longer programs. Before we proceed, however, let's learn to make our programs clear and readable.

Making a program readable is important. If you write a program today and wish to use it or change it in a few days, it will probably take you some time to remember and understand what the program does, and how. As a rule, plan in advance and make each program as readable as possible as soon as you write it. The purpose of this chapter is to help you improve your program readability.

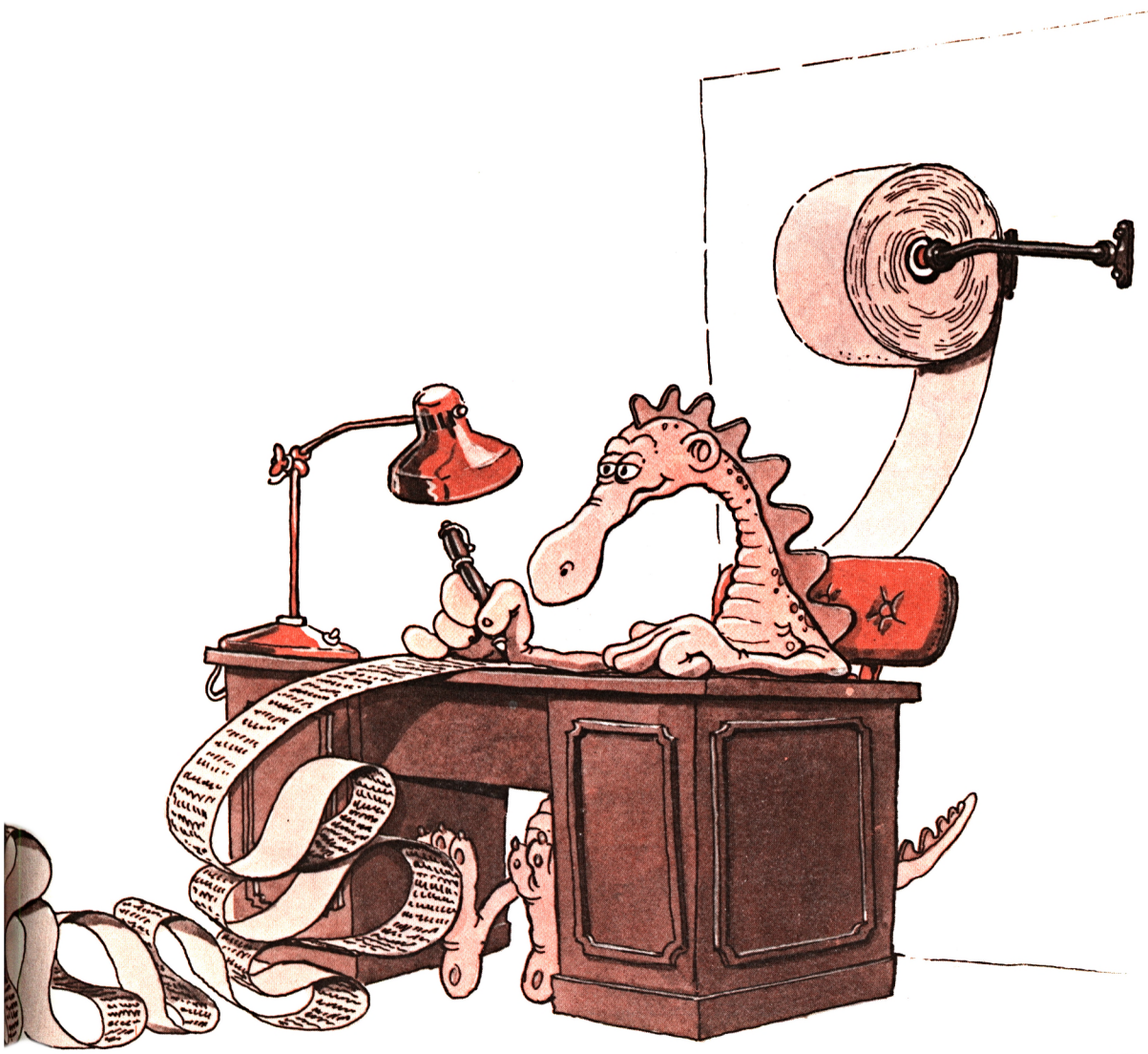
We will examine seven techniques:

1. The use of the REM statement (to introduce *remarks* in a program)
2. The use of multiple statements on one line
3. The use of spaces within a statement
4. The use of the "empty PRINT" and the CLS statements (to improve the display on the screen)
5. The use of the "shortcut INPUT" statement (to reduce the number of lines in a program)
6. The selection of meaningful names for variables
7. The proper line numbering.

We will now describe each technique in turn.



WRITING A CLEAR PROGRAM



The REM Statement

Here is an example using the REM statement:

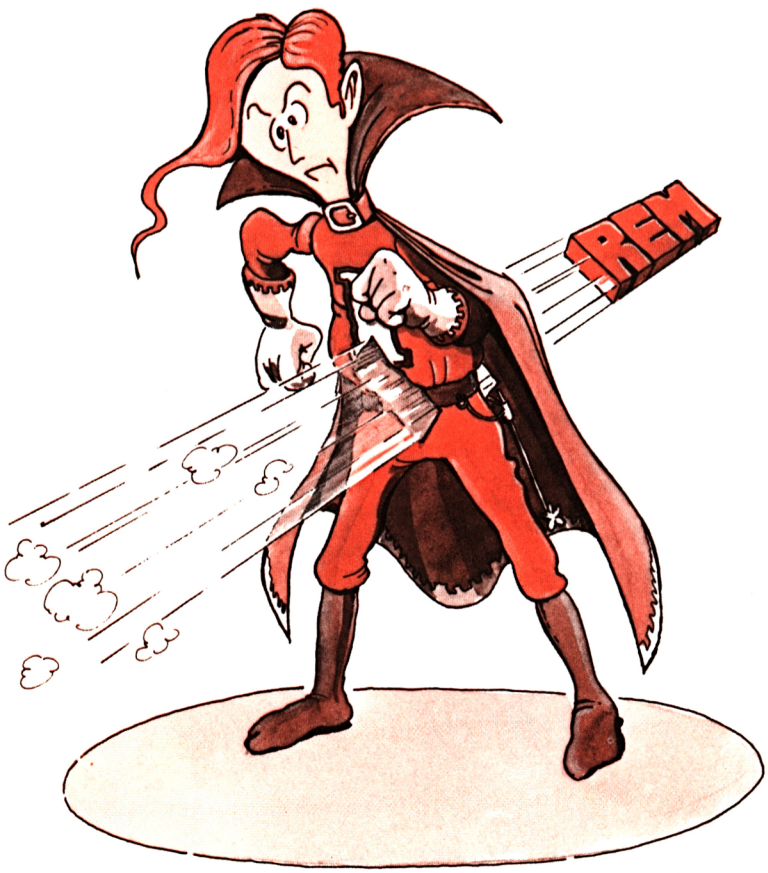
```
10 REM *** ADDITION PROGRAM ***
20 PRINT "GIVE ME TWO NUMBERS";
30 INPUT FIRST,LAST
40 PRINT "THEIR SUM IS:"; FIRST + LAST
50 END
```

The REM statement is used to make the program more readable by introducing comments in the text. This statement is ignored by the interpreter when the program is executed and has no effect on the program. Here are additional examples of remarks you could use:

```
25 REM NOW READ THE TWO NUMBERS
45 REM WE COULD ALSO MULTIPLY THEM AS PART OF THE
     ABOVE STATEMENT
```

and here is the resulting program

```
10 REM *** ADDITION PROGRAM ***
20 PRINT "GIVE ME TWO NUMBERS";
25 REM NOW READ THE TWO NUMBERS
30 INPUT FIRST,LAST
40 PRINT "THEIR SUM IS:"; FIRST + LAST
45 REM WE COULD ALSO MULTIPLY THEM AS PART OF THE
     ABOVE STATEMENT
50 END
```



REMarks are invisible to the interpreter.

"See how a few stars make my program more readable."



For improved readability, feel free to use stars, dashes or other symbols:

```
10 REM *** ADDITION PROGRAM ***
60 REM ---SECOND PART---
100 REM =====GRAND FINALE=====
200 REM $$$$$CHANGE THIS SECTION LATER$$$$$
```

Multiple Statements on a Line

AMSTRAD BASIC allows you to write two or more statements on the same line, separated by a colon. Here is an example:

```
50 PRINT "TYPE NUMBER"; : INPUT NUMBER
```

This is equivalent to:

```
50 PRINT "TYPE NUMBER";
60 INPUT NUMBER
```

Note that there can be only one line number per line. Therefore, when two statements are written on the same line, there is only one line number on the left. Here is another example:

```
100 REM *** COMPUTE EVERYTHING ON ONE LINE ***
110 SUM = A + B : PRODUCT = A * B : AVERAGE = SUM / 2
```

There are three statements on line 110.

Writing multiple statements on the same line is beneficial in at least two cases: to clarify an INPUT and to introduce a REMark to the right of the statement. Here is an example showing how the INPUT is clarified:

```
70 PRINT "ENTER 2 NUMBERS"; : INPUT N1, N2
```

Line 70 is written to correspond to what happens on the screen. This makes the program easier to read.

Here is an example showing how a REMark is introduced on the same line as the statement it refers to:

```
60 RESULT = A + 2 * B - 5 : REM RESULT MUST BE POSITIVE
```

Here is another one:

```
50 TEMPERATURE = (FAHR - 32) * 5 / 9 : REM WE CONVERT
    TO CELSIUS
```

Using Spaces

With the exception of names, strings, input data, or the required spaces after keywords, spaces are ignored by AMSTRAD BASIC. For example, you may write:

```
20PRINT4+2*3
```

However, a statement like this is hard to read. To facilitate reading back your program, use spaces liberally. Use spaces:

► *before each keyword, such as PRINT and INPUT.*

Here are examples:

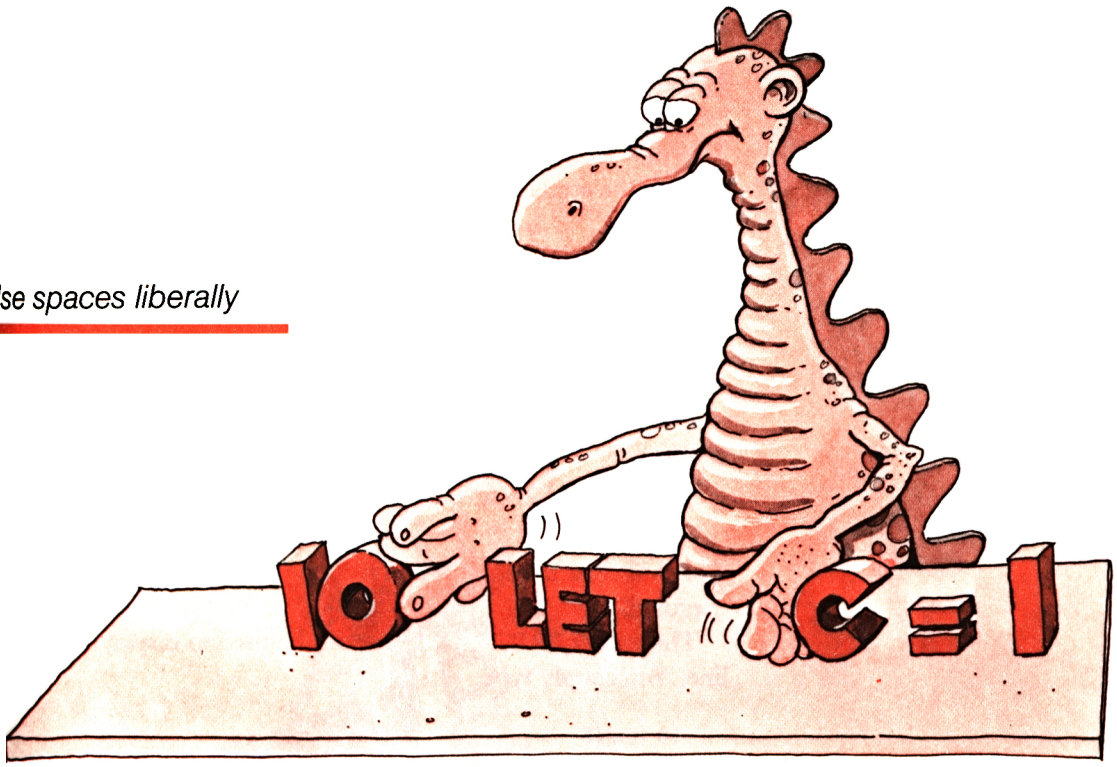
```
50 PRINT 4
60 INPUT NUMBER
```

► *before and after each operator.*

Here are examples:

```
30 PRINT 4 + 2 * 3
40 RESULT = A1 / ((B - C) * D)
```

Use spaces liberally



Finally, here is an example of using spaces within a REM statement to even further facilitate reading:

```
1 REM THIS PROGRAM MAINTAINS MY INVENTORY
2 REM COPYRIGHT MYSELF 1982
3 REM THESE ARE THE VARIABLES:
4 REM C    IS THE COLOUR ( 1 TO 10)
5 REM U    IS THE NUMBER OF UNITS (UP TO 1000)
6 REM S    IS THE SIZE (1 TO 50)
7 REM CST  IS THE UNIT COST
8 REM R    IS THE RETAIL PRICE
9 REM Q    IS THE REORDER QUANTITIES
```

Note how the spaces are used to enhance readability. You may not, however, add spaces in the middle of a keyword, a variable name or during a response to INPUT, unless the spaces are part of a string.

Improving the Display

Two new techniques can be used to improve the way your results look on the screen: the CLS statement and the empty PRINT statement.

The CLS statement clears the screen at the time it is used, Thus, you may want to start each new program with:

```
10 CLS
```

Here is a trick you may want to use. At the end of your program, you can write:

```
110 CLS : LIST
```

This will clear the screen, and list the entire program—after it is executed.

The empty PRINT statement can be used to display a blank line. You simply type:

```
50 PRINT
```

and a blank line will be displayed. If you want to skip three lines on the screen, type:

```
50 PRINT : PRINT : PRINT
```

Or, equivalently:

```
50 PRINT
60 PRINT
70 PRINT
```

Shortcut INPUT

Everytime you use an INPUT statement, you should prompt the user of the program by explaining what to enter. Here is an example:

```
50 PRINT "TYPE TWO INTEGERS";
60 INPUT AGE1, AGE2
```



Keep it simple!

The sequence PRINT-INPUT is used so often that most modern BASICs, including AMSTRAD, allow a shortcut INPUT statement, and you may write:

```
50 INPUT "TYPE TWO INTEGERS"; AGE1, AGE
```

which is equivalent to the two statements above. This shortcut technique reduces the typing required and shows clearly what will be displayed on the screen.

The INPUT statement displays a question mark to indicate when the computer is ready for you to type something. The shortcut INPUT statement provides an opportunity to use punctuation other than the question mark. For example, the statement:

```
70 INPUT "TYPE TWO NUMBERS: ", A1, A2
```

will show on the screen:

```
TYPE TWO NUMBERS: 8,27
```

when answered with the numbers 8 and 27.

In this example, INPUT did not supply its usual question mark. When there is a comma between the closing quotation marks of an INPUT string and the name of the variable, INPUT will not supply its usual "? " prompt (note the space after the question mark). It is up to you to provide the punctuation you prefer.

Selecting Variable Names

You should always select variable names so that you can easily remember what the name represents. Otherwise, you may find it difficult to write a long program, and you may make many accidental errors. Also, if your variable names are not clear, you may find that at a later date you are unable to figure out what your program does.

AMSTRAD BASIC allows multiple letter names, so use them. Here is an example, showing variable names that are easy to remember:

```
10 REM HERE ARE THE VARIABLES
20 REM    RESULT    FOR THE RESULT
30 REM    FIRST     FOR THE FIRST NUMBER
40 REM    LAST      FOR THE LAST NUMBER
```

Three restrictions apply:

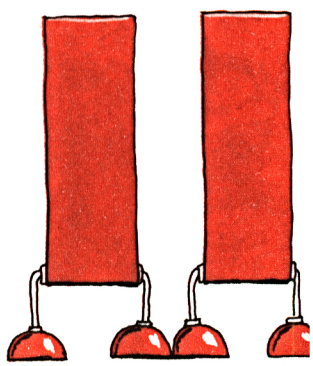
- All names must begin with a letter
- The only non-alphanumeric character that may be included in a name is the full stop
- You may not use a keyword, such as PRINT, REM or INPUT, as a variable name.

It is a good idea to identify all your variable names at the beginning of each program, as well as any formulas or equations you will be using.

Proper Line Numbering

So far in this chapter we have numbered lines by multiples of 10:

```
10 (statement)
20 (statement)
30 (statement)
```



You may, however, use any sequence you want, as long as you use positive integers, and you do not go over 65,535 limit for line numbers. For example, you may write:

```
1 (statement)
2 (statement)
3 (statement)
```

or:

```
100 (statement)
200 (statement)
300 (statement)
```

We have taken the precaution to leave a regular gap between consecutive statement numbers so that we can easily add corrections or improvements later on. For example, here is version 1 of a program:

```
10 REM *** MULTIPLICATION PROGRAM ***
20 INPUT "GIVE ME 2 NUMBERS: ", N1, N2
30 PRINT "THE PRODUCT IS:"; N1 * N2
40 END
```

We now want to clarify the program and improve the display. To do this, we will type the following additional statements:

```
5 CLS
15 PRINT "THIS IS AN AUTOMATIC MULTIPLICATION PROGRAM"
16 PRINT : PRINT
35 PRINT : PRINT
```



Number your lines properly.

The new statements will be inserted automatically. Let's now LIST the result:

```
5 CLS
10 REM *** MULTIPLICATION PROGRAM ***
15 PRINT "THIS IS AN AUTOMATIC MULTIPLICATION PROGRAM"
16 PRINT : PRINT : PRINT
20 INPUT "GIVE ME 2 NUMBERS: ", N1, N2
30 PRINT "THE PRODUCT IS:"; N1 * N2
35 PRINT : PRINT
40 END
```

Here is a sample run of our improved program:

```
THIS IS AN AUTOMATIC MULTIPLICATION PROGRAM

GIVE ME 2 NUMBERS: 12,15
THE PRODUCT IS: 180

READY
■
```

If you plan to make many corrections, or add many statements, you may want to leave large gaps in your line numbering, and use for instance:

```
10 (statement)
50 (statement)
60 (statement)
100 (statement)
```

AMSTRAD BASIC has a special command called RENUM that will automatically renumber all the lines in your program with an increment of 10 (you can specify other values for the increment). This is very useful in a long program where you may run out of sequential numbers when inserting a "patch," or correction.



Summary

Writing programs that work involves one key ingredient: discipline. It is important to be as orderly and organized as possible when writing programs. Shortcuts increase the probability of mistakes. In particular, take the time to clarify your programs and your displays. In this chapter we have described and stressed the techniques necessary to write clear programs.

As you write programs in BASIC, you should make every effort to follow the suggestions offered in this chapter. It is essential that you acquire good programming habits, or else your programs may be unreadable, or may not even work, as you begin writing more and more complex programs.

Exercises

5-1: Describe techniques that improve the readability of the display.

5-2: Are the following legal?

a. A = A + 1

b. A = A + 1

c. PRINT ALPHA+2

d. SUM=2 + (3 +(4/5))/2

e. IN PUT NUMBER

f. SUM= 2 2 + 3 3

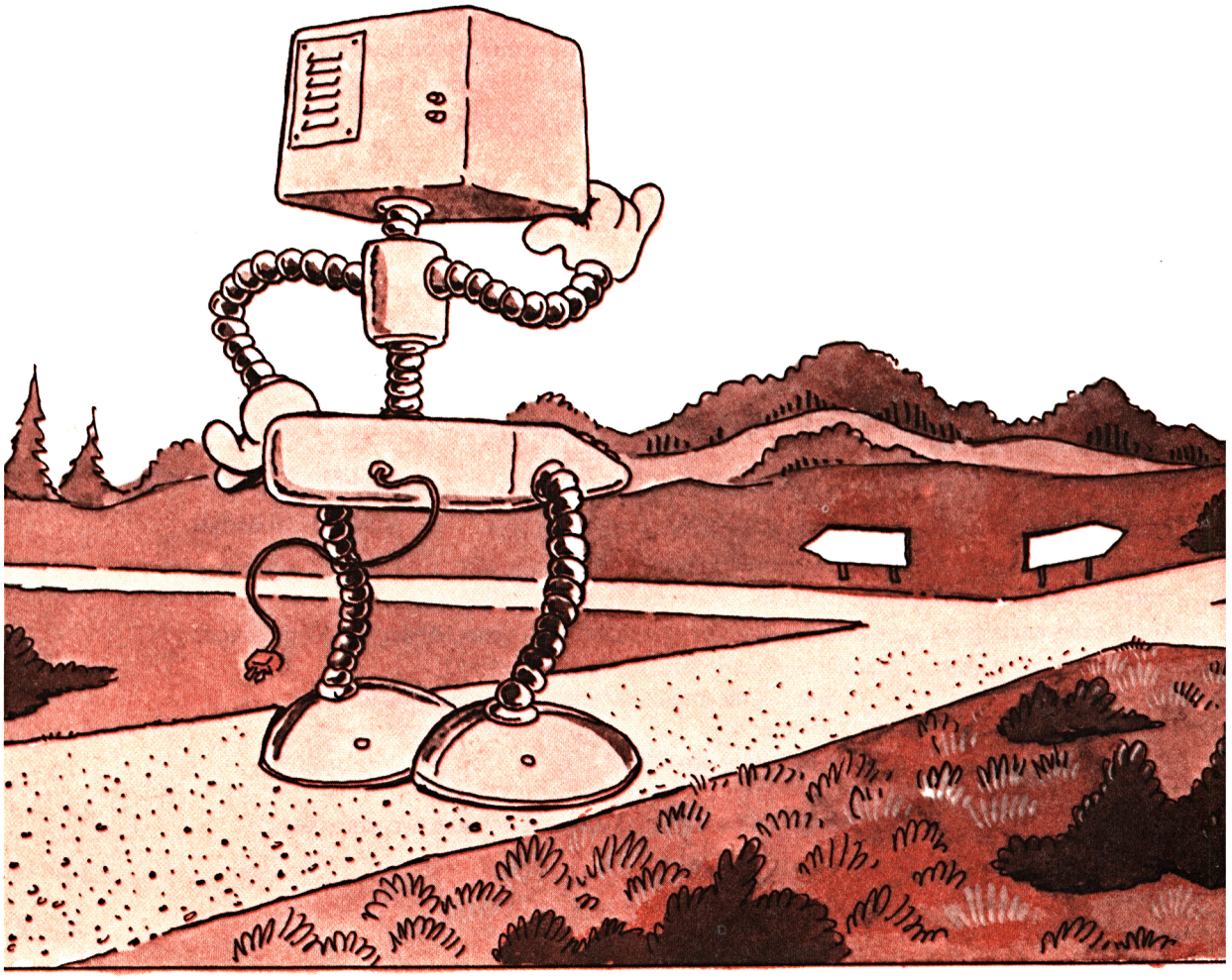
5-3: Explain why most INPUTs should have a prior message.

5-4: Write three examples of shortcut INPUTs.

5-5: Why use REMs?

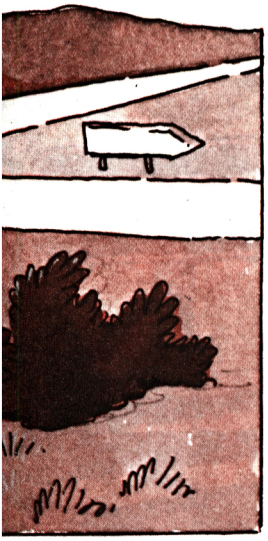
5-6: What is the value of A after these two statements:

```
30 A = 3
40 REM A = 4
```



MAKING DECISIONS

6



So far, we have learnt how to communicate with the computer and how to perform simple arithmetic. However, our programs have been somewhat dull, and we could easily perform the same tasks by hand. This is because we have used only the elementary resources of the computer. We have not taken advantage of the more advanced resources. For example, computers are particularly good at performing two tasks: making complex decisions (based on logic and values) and executing repetitive tasks many times in a following chapter. In particular, we will learn how to make decisions. Our programs will become "intelligent," as they decide what to do.

In BASIC, program decisions are made by testing a value, using the IF statement. If the test succeeds, one part of the program is executed. If it fails, another part is executed. We will also learn to use the GOTO statement to force the program to execute a group of instructions out of numerical sequence.

The IF Statement

The IF statement is written:

```
IF (condition) THEN (statement, i.e., do something)
```

Here is an example:

```
IF I = 1 THEN PRINT "ONE"
```

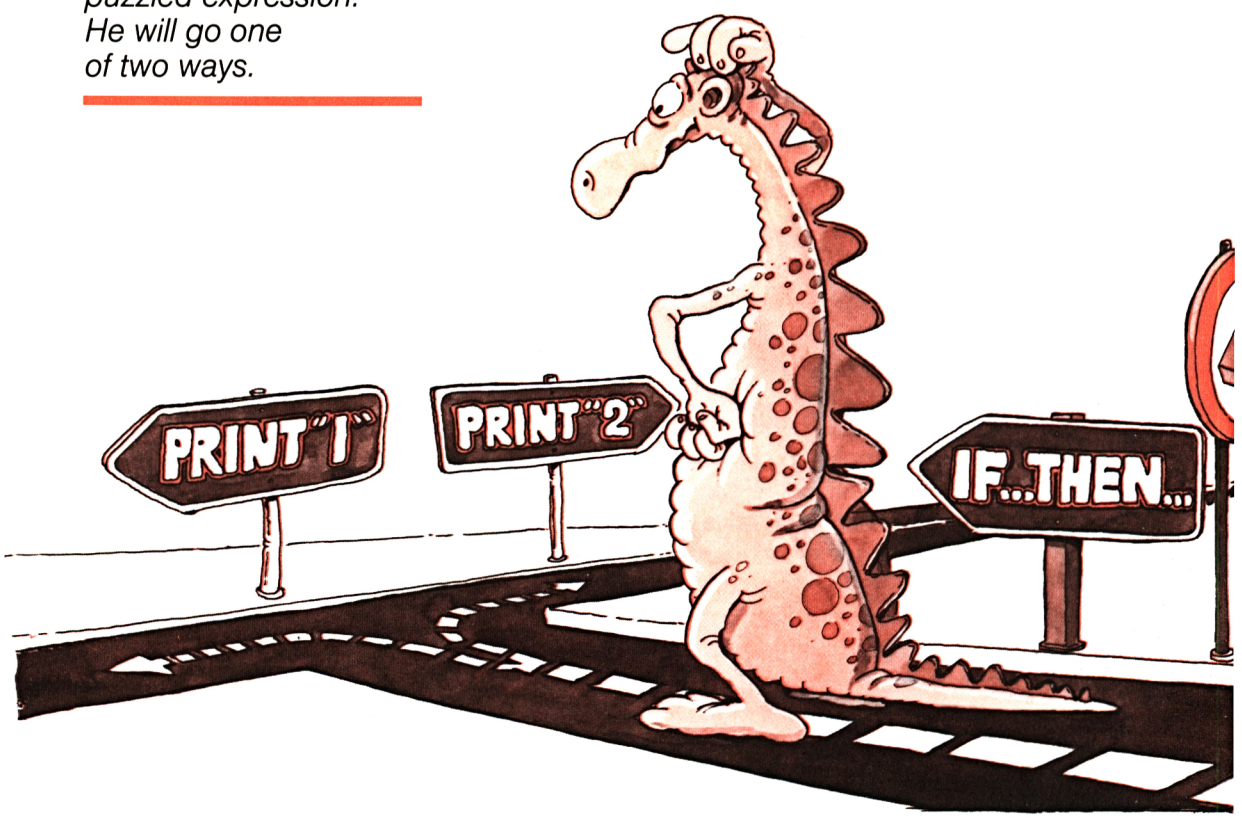
The effect of this statement should be clear: IF the value of the variable I is equal to 1 at the time this statement is executed, THEN the word ONE is printed. If I is not equal to 1, nothing happens and the next statement in the program is executed.

I = 1 is called a logical *expression.* The expression I = 1 is *true* when I is equal to 1; otherwise, it is *false.*

The IF . . . THEN statement allows you to test the value of an expression and execute one statement or another—that is, to make a decision—depending on the results of the test. Here is another example:

```
10 INPUT I
20 IF I = 1 THEN PRINT "ONE"
30 END
```

Look at Dino's puzzled expression. He will go one of two ways.



Now RUN this program. Type "1" at the keyboard. Your screen should look like this:

```
RUN
? 1
ONE
Ready
■
```

RUN this program again. Type "2" at the keyboard. Your screen should look like this:

```
RUN
? 2
Ready
■
```

This time no message was printed in response to the 2.

Let's now teach our program to recognize the numbers 1 through 4:

```
10 REM THIS PROGRAM RECOGNIZES THE NUMBERS 1 TO 4
20 INPUT "TYPE AN INTEGER: ", NUMBER
30 IF NUMBER = 1 THEN PRINT "ONE"
40 IF NUMBER = 2 THEN PRINT "TWO"
50 IF NUMBER = 3 THEN PRINT "THREE"
60 IF NUMBER = 4 THEN PRINT "FOUR"
70 END
```

Let's run the program. Here are two typical runs as they appear on the screen (with emphasis added).

```
RUN
TYPE AN INTEGER: 3
THREE
Ready
RUN
TYPE AN INTEGER: 5
Ready
■
```

This is good, but not yet perfect. Ideally, when we type 5, we would like the program to respond with something like:

```
I DON'T KNOW THIS NUMBER.
```

or else request a new integer.

There is a special feature of the IF statement that can make this happen. For example, you may write:

```
70 IF NUMBER = 5 THEN 20
```

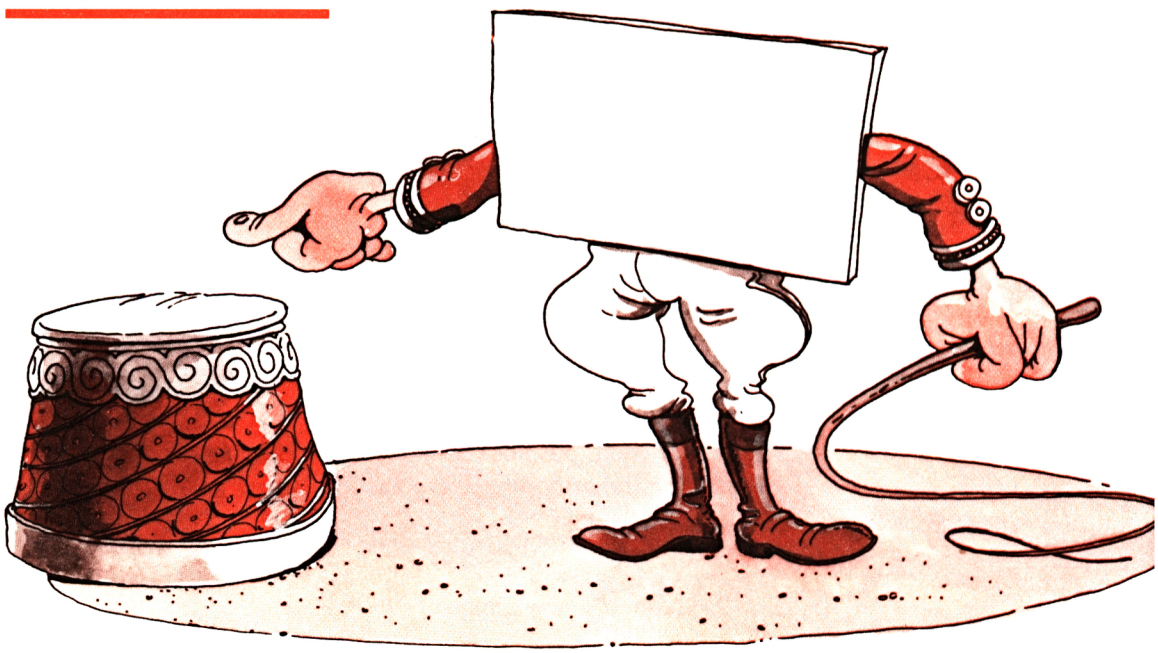
where 20 is the number of the line to be executed if the test is successful. This is a new form of the IF statement. This statement means that if NUMBER is equal to five, then execute line number 20 next. We can now jump out of sequence! Here is an example:

```
10 INPUT I
20 IF I = 1 THEN 50
30 PRINT "YOU DID NOT TYPE A 1"
40 END
50 PRINT "YOU TYPED A 1"
60 END
```

RUN this program and type "1" at the keyboard. Your screen should look like this:

```
RUN
? 1
YOU TYPED A 1
Ready
■
```

We can make the program jump out of sequence.



RUN the program again, and type a "2" at the keyboard. Your screen should look like this:

```
RUN
? 2
YOU DID NOT TYPE A 1
Ready
■
```

Our program has become "intelligent", i.e., it gives an appropriate message whether or not the input is 1. You may be wondering if we could have achieved the same result using the original form of the IF statement. Let's try it:

```
10 INPUT I
20 IF I = 1 THEN PRINT "THIS IS A ONE"
30 PRINT "THIS IS NOT A ONE"
40 END
```

Now, RUN this program and type a 1 at the keyboard. The display is the following:



I'm the bug.
I got you.

```
RUN
? 1
THIS IS A ONE
THIS IS NOT A ONE
Ready
■
```

It does not work. Regardless of the success or failure of the IF, the next statement following it in the program (statement 30 here) is executed.

In this example, we first get the correct message as the IF is executed:

```
THIS IS A ONE
```

Then the second message is printed anyway:

```
THIS IS NOT A ONE
```

The new form of the IF statement:

```
IF I = 1 THEN 50
```

eliminates this problem. We will use this statement frequently in our programs.



A successful IF forces the Interpreter to activate an instruction.

Let's now take a closer look at the IF statement so that we can fully utilize it. The general form of the IF . . . THEN statement is the following:

IF (logical expression) THEN (executable statement or line number)

Let's now examine logical expressions and executable statements in turn.

Logical Expressions

In our example, I = 1 is a *logical expression,* i.e., it can be either *true* or *false.* True and false are called *logical values.* Here are some examples of logical expressions:

I = 1	(*I equals 1*)
I > 4	(*I is greater than 4*)
NUMBER < 100	(*NUMBER is less than 100*)
YEAR <> 5	(*YEAR is not equal to 5*)
AGE < 13	(*AGE is less than 13*)

A logical expression combines values or variables with logical operators. For completeness, the operators you can use with logical expressions are:

=	equal	
<>	not equal	(*in mathematics, this is written as ≠ or #*)
<	less than	
>	greater than	
<=	less or equal	(*in mathematics, this is written as ≤*)
>=	greater or equal	(*in mathematics, this is written as ≥*)

Here are some even more complex logical expressions:

(NUMBER + 2) > 4

(AGE − 5) >= 10

((2 * I − 5) / 2) < 10

2 > I

You may also write:

4 > 2 (*this is always true*)
4 = 2 (*this is always false*)

The following are *not* valid logical expressions:

2 < I < 10	(*only one relational operator may be used*)
(2 AGE − 2) < 5	(*invalid expression—missing an *. This should read: (2 * AGE − 2) < 5*)



You may even combine logical expressions using the *logical operators* AND, OR, and NOT. For example, you may write:

```
IF (AGE > 12) AND (AGE < 20) THEN PRINT "THIS IS A
   TEENAGER"
```

This statement will print "THIS IS A TEENAGER" whenever AGE is greater than 12 and less than 20. The AND is satisfied, i.e., true, when *both* of its clauses are true. Note, that you may *not* write:

```
IF AGE (> 12 AND < 20) THEN . . .
```

as each pair of parenthesis must enclose a valid expression.

Here is another example using two logical expressions:

```
20 INPUT ANSWER$
30 IF (ANSWER$ = "YES") OR (ANSWER$ = "NO") THEN 60
40 PRINT "INVALID ANSWER"
   . . .
60 PRINT "VALID ANSWER—LET US PROCEED"
```

This program segment collects an answer in the variable ANSWER$. (Recall that the $ at the end of the name is used to denote a string variable, i.e., a collection of characters.) YES or NO are the only valid answers; this program checks for the validity of what you have typed.

If you type YES, then (ANSWER$ = "YES") is true, and the IF succeeds: statement 60 is executed next, and the program prints:

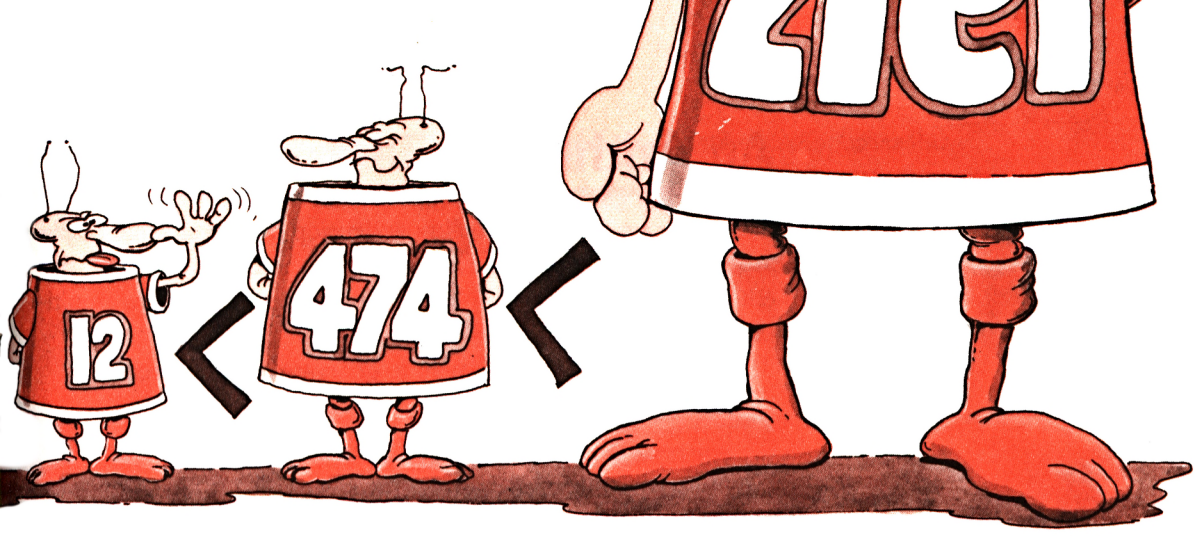
```
VALID ANSWER—LET US PROCEED
```

Similarly, if you type NO, the same occurs.

If you type anything else, you get a diagnostic:

```
INVALID ANSWER
```

An OR is satisfied, i.e. *true,* whenever at least one of its two clauses is true. The OR fails when both of its clauses are false. For example, if you type "YA", then (ANSWER$ = "YES") fails,



Relational operators can be used on variables

and the second clause of the OR is tried: (ANSWER$ = "NO"). It fails, and the program displays the message:

```
INVALID ANSWER
```

Finally, NOT can be used to negate a condition. Here is an example of a complex IF statement:

```
IF ((AVERAGE < 67) AND (LASTEXAM < 60) AND NOT (ORAL >
80)) THEN PRINT "FAILED"
```

In this statement, we check for three conditions at once. We will not discuss such complex statements any further here, but you may want to experiment with them on your own.

Executable Statements

Let us recall the definition of the IF statement:

```
IF (logical expression) THEN (executable statement or line number)
```

Now that we are familiar with logical expressions, let's examine the right part of this definition:

```
THEN (executable statement or line number)
```

An executable statement is simply any statement that is executed. It may be an assignment, an INPUT or a PRINT statement, or even another IF statement. Now that we understand the theory of the IF statement, let's put it into practice.

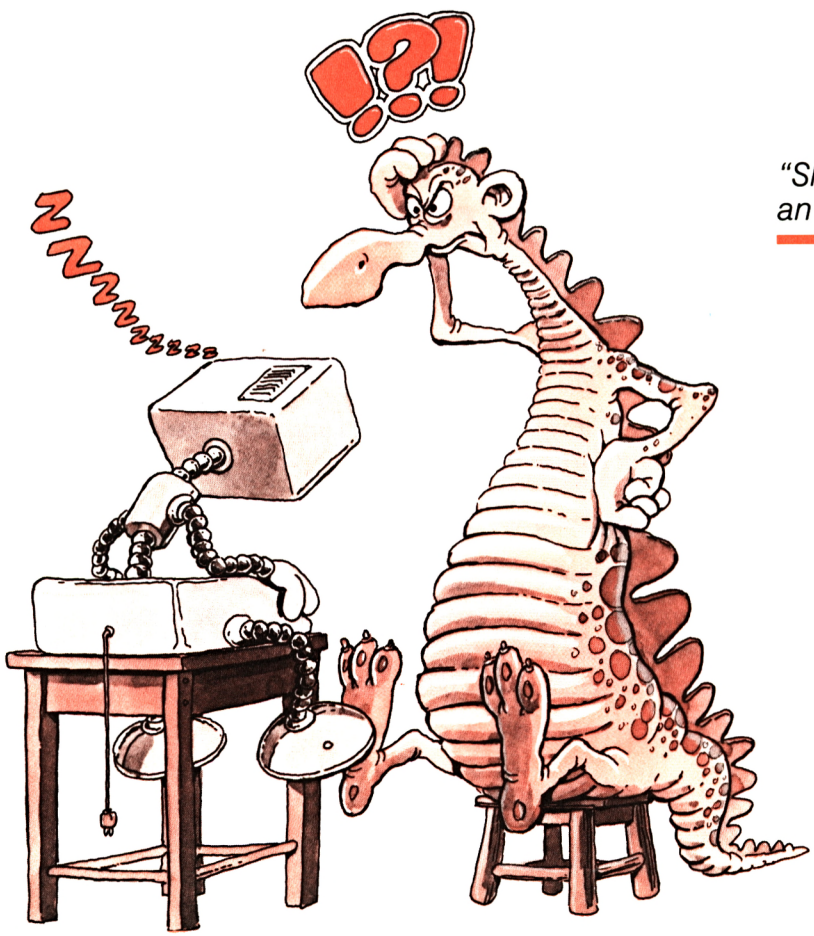
An Arithmetic Test

Using our new skills, we will now develop a program that displays a "menu" on the screen. Depending on the user's selection, this educational program will perform additions, subtractions, multiplications or divisions.

Here is the dialogue we plan to generate on the screen:

```
WELCOME TO COMPUTER TEACHER
I WILL CHECK YOUR ARITHMETIC SKILLS
WHAT DO YOU WANT TO PRACTICE?
  - ADDITION (TYPE 1)
  - SUBTRACTION (TYPE 2)
  - MULTIPLICATION (TYPE 3)
  - DIVISION (TYPE 4)
WHAT IS YOUR CHOICE? 3
```

```
ALL RIGHT. LET'S MULTIPLY
HOW MUCH IS 2 TIMES 3? 6
THAT'S RIGHT. CONGRATULATIONS
```



"Should I present an arithmetic menu?"

Now here is the program that accomplishes this:

```
10 REM *** THIS PROGRAM TESTS YOU IN ARITHMETIC ***
20 PRINT "WELCOME TO COMPUTER TEACHER"
30 PRINT "I WILL CHECK YOUR ARITHMETIC SKILLS"
40 PRINT "WHAT DO YOU WANT TO PRACTICE?"
50 PRINT "   - ADDITION (TYPE 1)"
60 PRINT "   - SUBTRACTION (TYPE 2)"
70 PRINT "   - MULTIPLICATION (TYPE 3)"
80 PRINT "   - DIVISION (TYPE 4)"
90 INPUT "WHAT IS YOUR CHOICE"; CHOICE
100 IF (CHOICE = 1) THEN 200
110 IF (CHOICE = 2) THEN 300
120 IF (CHOICE = 3) THEN 400
130 IF (CHOICE = 4) THEN 500
140 PRINT "INCORRECT CHOICE."
145 PRINT "YOU MUST SELECT A NUMBER BETWEEN 1 AND 4"
150 PRINT "GOOD BYE" : END
190 REM --- ADDITION ---
200 PRINT "ALL RIGHT. LET'S ADD"
210 INPUT "HOW MUCH IS 4 + 7"; NUMBER
220 IF (NUMBER < > 11) THEN 600
230 PRINT "THAT'S RIGHT. CONGRATULATIONS" : END
290 REM --- SUBTRACTION ---
300 PRINT "ALL RIGHT. LET'S SUBTRACT"
310 INPUT "HOW MUCH IS 9 - 5"; NUMBER
320 IF (NUMBER < > 4) THEN 600
330 PRINT "THAT'S RIGHT. CONGRATULATIONS" : END
390 REM --- MULTIPLICATION ---
400 PRINT "ALL RIGHT. LET'S MULTIPLY"
410 INPUT "HOW MUCH IS 2 TIMES 3"; NUMBER
420 IF (NUMBER < > 6) THEN 600
430 PRINT "THAT'S RIGHT. CONGRATULATIONS" : END
490 REM --- DIVISION ---
500 PRINT "ALL RIGHT. LET'S DIVIDE"
510 INPUT "HOW MUCH IS 9 DIVIDED BY 3"; NUMBER
520 IF (NUMBER < > 3) THEN 600
530 PRINT "THAT'S RIGHT. CONGRATULATIONS" : END
590 REM --- FAILURE EXIT ---
600 PRINT "WRONG. SORRY AND GOOD BYE." : END
```

This program looks imposing in size, but it is actually quite simple. Let's examine it.

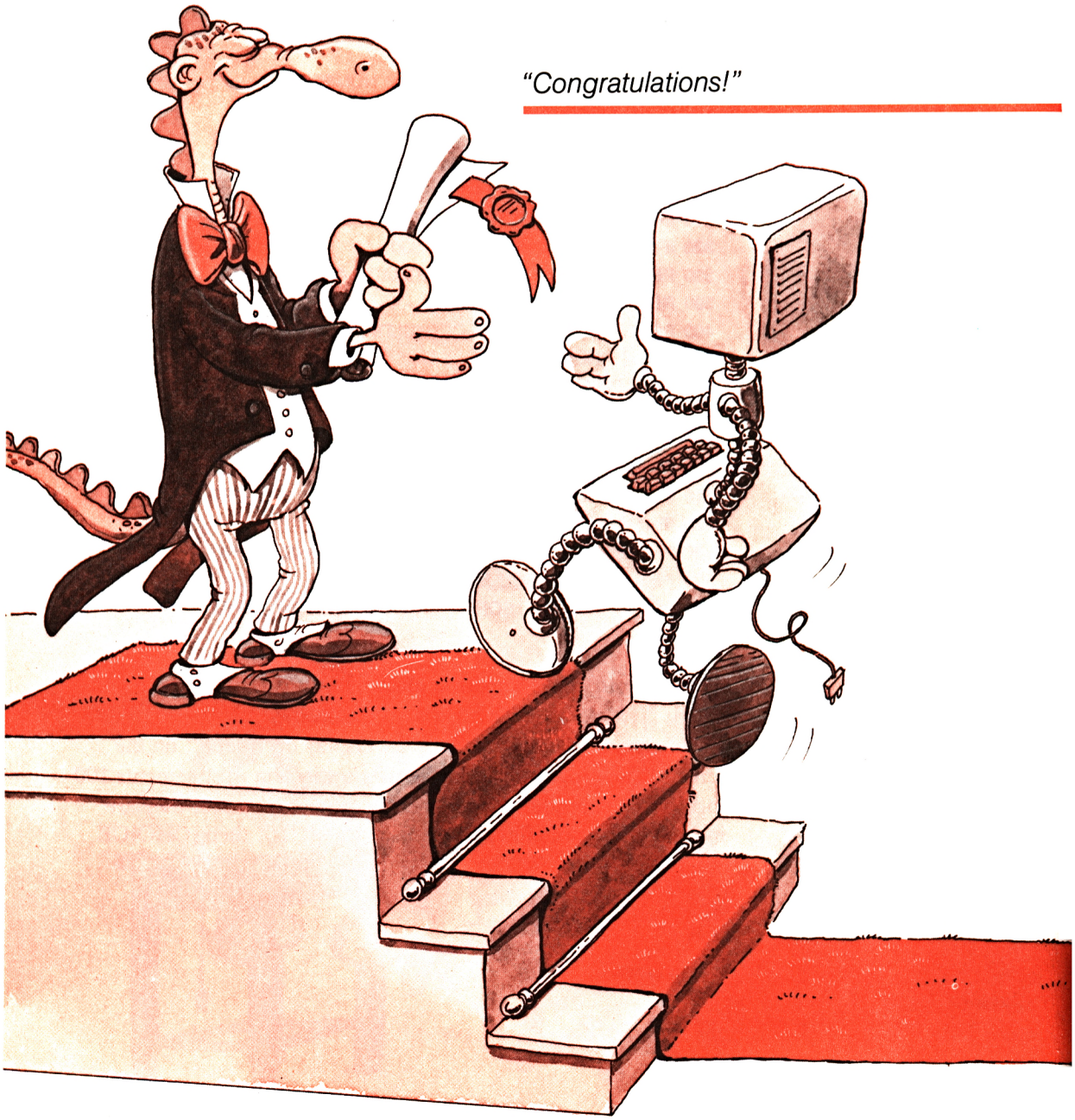
Statements 20 to 90 produce the display or "menu" on the screen. The program checks the user's selection in statements 100 to 130 (the parentheses after each IF are not required; they are included for readability). If the user typed "1", then (CHOICE

= 1) is true, and statement 200 is executed next. If the user typed something other than 1, 2, 3, or 4, then statement 140 is executed and the program says:

```
INCORRECT CHOICE.
YOU MUST SELECT A NUMBER BETWEEN 1 AND 4
GOODBYE
Ready
■
```

and stops (this is the : END statement on line 150).

In our example, we type 3. Statement 100 fails, so statement 110 is executed next. Statement 110 fails, so statement 120 is



executed next. Statement 120 succeeds, since (CHOICE = 3) is true, and statement 400 is executed next. Here is the corresponding program segment:

```
400 PRINT "ALL RIGHT. LET'S MULTIPLY"
410 INPUT "HOW MUCH IS 2 TIMES 3"; NUMBER
420 IF (NUMBER < > 6) THEN 600
430 PRINT "THAT'S RIGHT. CONGRATULATIONS" : END
```

In our example, we type 6 in response to statement 410. When statement 420 is executed, (NUMBER < > 6) is false since NUMBER = 6. (Recall that < > means not equal to.) Therefore, the next statement to be executed is statement 430, and the program responds with:

```
THAT'S RIGHT. CONGRATULATIONS
Ready
■
```

and stops since line 430 contained *two* statements. The second statement is:

```
: END
```

Looking at this program, you may quickly spot a new frustration: If you type a number other than 1, 2, 3, or 4 after the menu is shown, or if you give an incorrect arithmetic answer, the program will stop abruptly. Ideally, we would like the program to continue. For example, it would be good if, after the program tells the user that a number other than 1 to 4 is not valid, it would then ask for a new choice. We would like to be able to go back to the beginning of the program and restart it, or more generally, to be able to go to *any* part of the program next. This is possible with the GOTO statement. Let us examine this statement.

The GOTO Statement

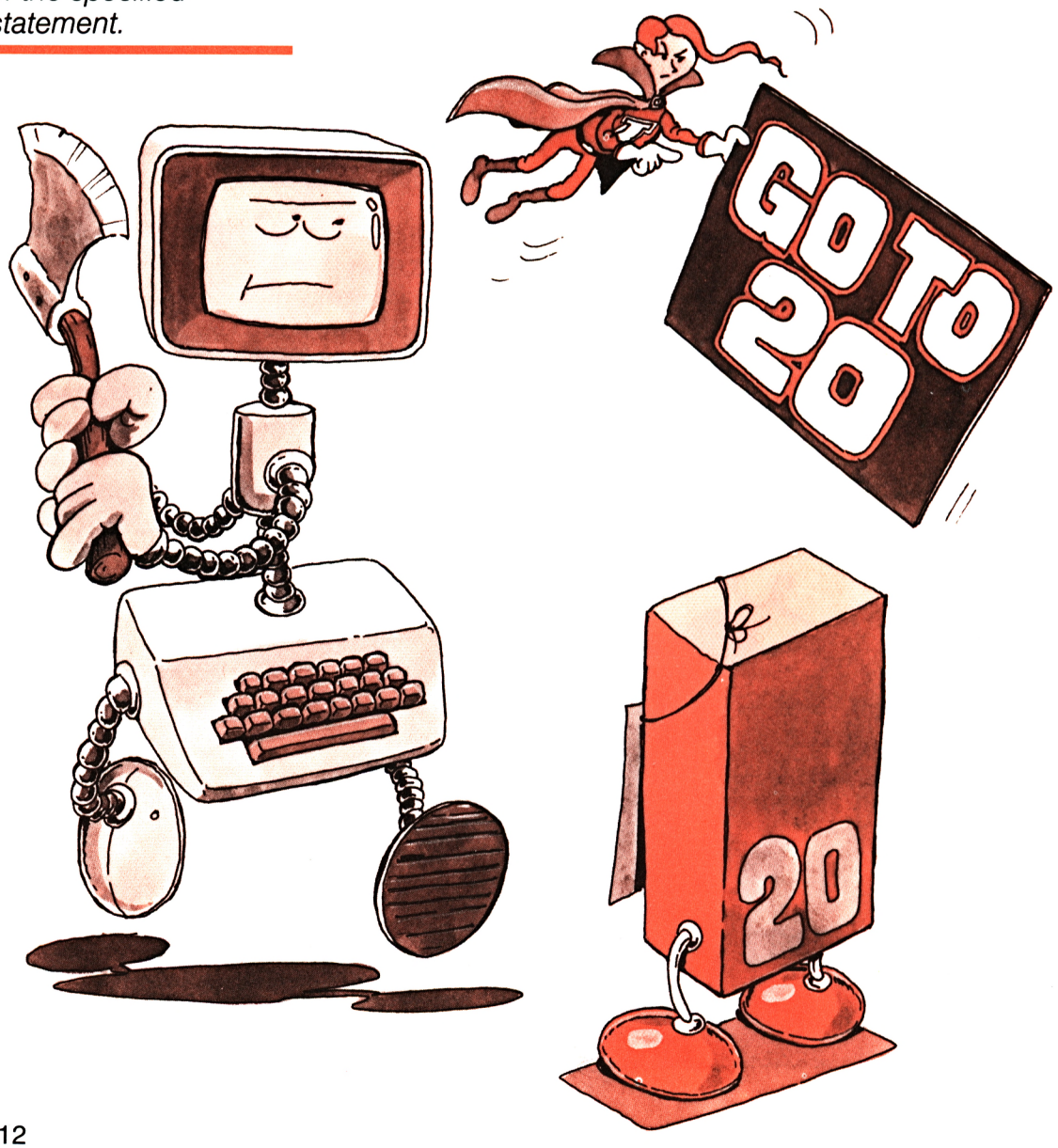
The GOTO statement is written as:

```
GOTO (line number)
```

It forces execution of the specified statement. Here is an example:

```
10 PRINT "THIS PROGRAM RECOGNIZES 1'S."
15 PRINT " TYPE 0 TO STOP."
20 INPUT "TYPE A NUMBER: ", NUMBER
30 IF NUMBER = 1 THEN PRINT "ONE"
40 IF NUMBER = 0 THEN 60
50 GOTO 20
60 END
```

GOTO forces execution of the specified statement.



Here is a sample run:

```
RUN
THIS PROGRAM RECOGNIZES 1'S.
TYPE 0 TO STOP.
TYPE A NUMBER: 1
ONE
TYPE A NUMBER: 5
TYPE A NUMBER: 25
TYPE A NUMBER: 1
ONE
TYPE A NUMBER: 0
Ready
■
```

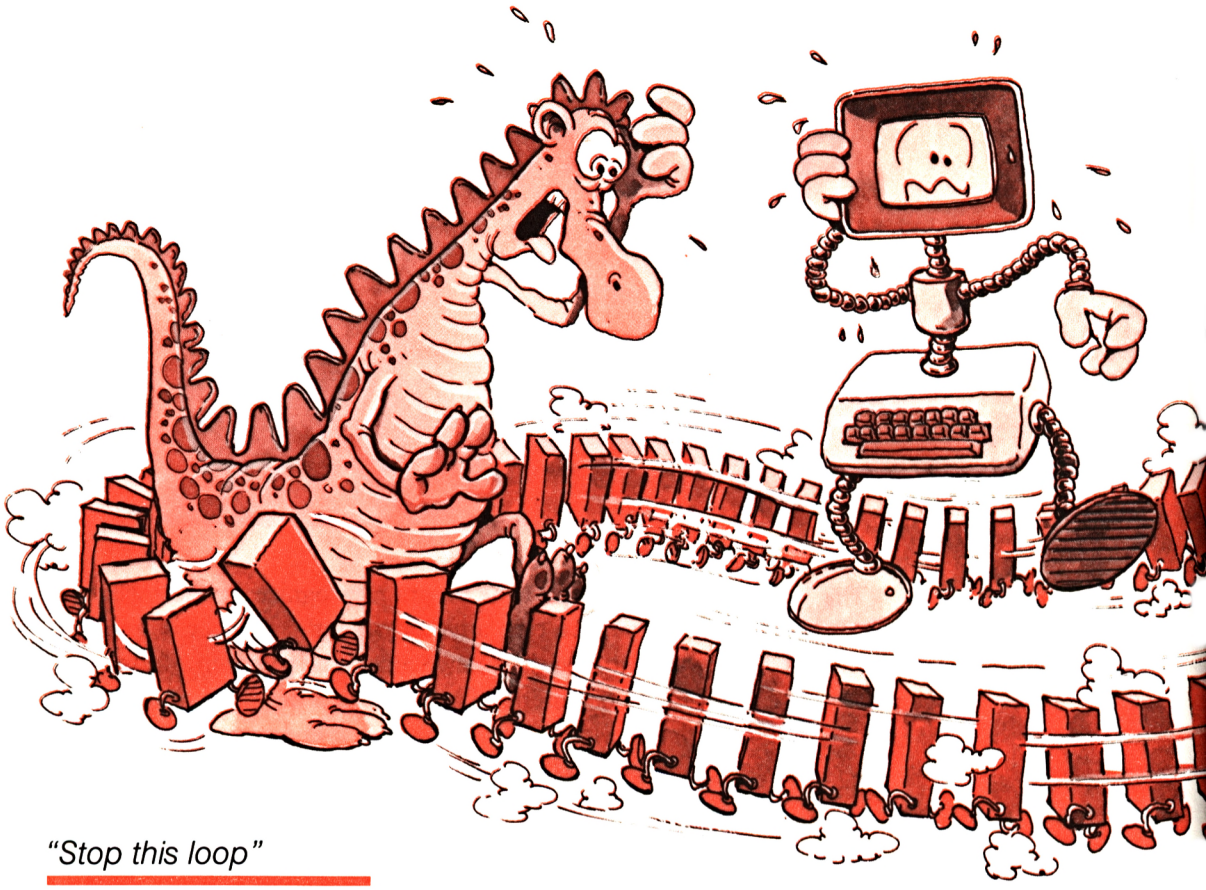
Every time you type 1 the program recognizes it and spells out ONE. Every time you type anything else, the number is ignored and the program requests a new value. The program continually goes back to the beginning. This is called a *loop*. The program is said to loop upon itself. If you type a 0, it is detected by statement 40, and the program jumps to statement 60 and ends.

Let's now remove statement 40. The program looks like this:

```
10 PRINT "THIS PROGRAM RECOGNIZES 1'S."
15 PRINT "TYPE 0 TO STOP."
20 INPUT "TYPE A NUMBER: ", NUMBER
30 IF NUMBER = 1 THEN PRINT "ONE"
50 GOTO 20
60 END
```

Here is a sample run:

```
THIS PROGRAM RECOGNIZES 1'S.
TYPE 0 TO STOP.
TYPE A NUMBER: 2
TYPE A NUMBER: 1
ONE
TYPE A NUMBER: 5
TYPE A NUMBER: 0
TYPE A NUMBER:
```



"Stop this loop"

Like the apprentice sorcerer, we have created a terrible problem: this program will never stop! This is a common programming error called an *endless loop*. The program may continue executing forever. Don't worry. This will not damage anything. To stop it, you must press the ESC key. If worst comes to worst, and you can't remember what to do, turn your computer off and then turn it back on again. But remember: if you turn the computer off, you will lose everything you have typed in so far and have not previously saved on cassette or diskette.

On the AMSTRAD pressing the CTRL-SHIFT-ESC combination is the next best thing to turning the computer off and back on again. Reinitializing the computer like this terminates the endless loop, but you will also lose the program and almost everything else. We will strive to avoid this unpleasant situation by providing a normal (programmed) exit for each program from now on.

Having introduced the GOTO statement, let's now go back to our definition of the IF statement, and simplify it.

IF Statement Revisited

Recall that one of the forms of the IF statement is

IF (logical expression) THEN (line number)

The other is:

IF (logical expression) THEN (executable statement)

Here is an example of the first form:

```
IF NUMBER = 0 THEN 60
```

This is equivalent to:

```
IF NUMBER = 0 THEN GOTO 60
```



GOTO 60 is an executable statement, and you will recognize that the form

```
THEN 60
```

is simply a convenient shorthand statement for

```
THEN GOTO 60
```

So, in reality, the general form of the IF statement is, in fact, simpler than our previous definition:

IF (logical expression) THEN (executable statement)

Here is one more simplification: the use of THEN before a GOTO is usually optional. With AMSTRAD BASIC, the following forms are all equivalent:

```
IF NUMBER = 0 THEN 100
IF NUMBER = 0 THEN GOTO 100
IF NUMBER = 0 GOTO 100
```

We will now demonstrate the use of IFs and GOTOs on program examples.

Counting Ones

In Chapter 5, we introduced the counter technique. Let's now use it to count the number of 1's typed in the last program of the previous section. Here is the improved program:

```
1 REM ONE'S COUNTER
10 PRINT "I COUNT HOW MANY SINGLE 1'S YOU TYPE."
20 PRINT "TYPE 0 TO STOP."
30 SUM = 0
40 INPUT "TYPE A NUMBER: ", NUMBER
50 IF NUMBER = 0 THEN 100
60 IF NUMBER <> 1 THEN GOTO 40
70 SUM = SUM + 1
80 PRINT "ONE. TOTAL SO FAR:"; SUM
90 GOTO 40
100 END
```

Here is a sample run:

```
I COUNT HOW MANY SINGLE 1'S YOU TYPE.
TYPE 0 TO STOP.
TYPE A NUMBER: 10
TYPE A NUMBER: 1
ONE. TOTAL SO FAR: 1
TYPE A NUMBER: 9
TYPE A NUMBER: 5
TYPE A NUMBER: 1
ONE. TOTAL SO FAR: 2
TYPE A NUMBER: 2
TYPE A NUMBER: 1
ONE. TOTAL SO FAR: 3
TYPE A NUMBER: 410
TYPE A NUMBER:
```

Let us examine the program. Statements 10 and 20 display messages:

```
10 PRINT "I COUNT HOW MANY SINGLE 1'S YOU TYPE."
20 PRINT "TYPE 0 TO STOP."
```

Statement 30 initializes the counter variable SUM to zero:

```
30 SUM = 0
```

Then the number is collected from the keyboard:

```
40 INPUT "TYPE A NUMBER: ", NUMBER
```

If the number is 0, we are done:

```
50 IF NUMBER = 0 THEN 100
```

where 100 is the END statement. Let's assume that the number was 10, and see what happens:

```
60 IF NUMBER < > 1 THEN GOTO 40
```

If the number is not 1, then we jump back to 40 and request a new number. If the number is 1, we proceed:

```
70 SUM = SUM + 1
```

The counter variable SUM is incremented by one. Recall the meaning of an assignment statement. You can read line 70 as:

SUM receives the new value of (old value of SUM) + 1

At this point, SUM receives the value 0 + 1 = 1. The next statement is:

```
80 PRINT "ONE. TOTAL SO FAR: "; SUM
```

Then the program loops back to line 40, requesting a new number:

```
90 GOTO 40
```

Arithmetic Test Revisited

Recall that we developed an arithmetic test program at the beginning of this chapter. We regretted the fact that it was too simple and could not recycle. However, we can now make it recycle.

Since the program is rather large, let's look at the relevant segment only. First, here is the section that asks the user to select a number between 1 and 4:

```
90 INPUT "WHAT IS YOUR CHOICE"; CHOICE
100 IF (CHOICE = 1) THEN 200
110 IF (CHOICE = 2) THEN 300
120 IF (CHOICE = 3) THEN 400
130 IF (CHOICE = 4) THEN 500
140 PRINT "INCORRECT CHOICE."
145 PRINT " YOU MUST SELECT A NUMBER BETWEEN 1 AND 4"
150 PRINT "GOOD BYE" : END
```

And here is our improvement:

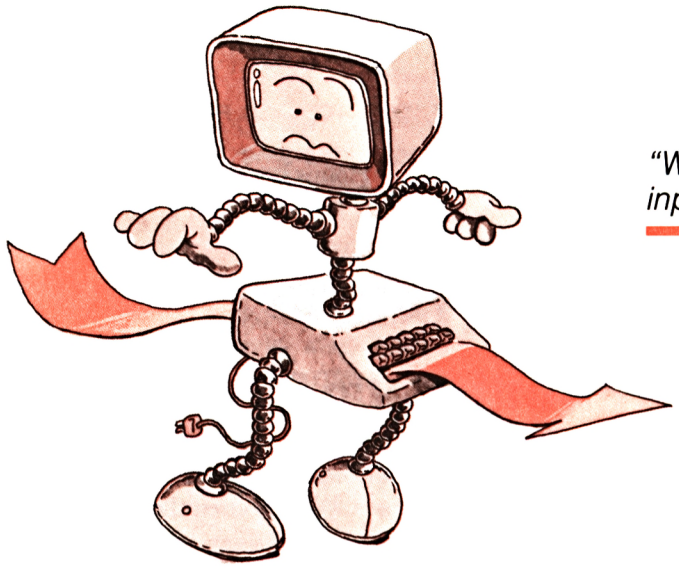
```
150 GOTO 90
```

That is all. Check it.

Now, we would also like the program to present more than one arithmetic question. Say, we want it to ask ten different questions. We could do this by adding GOTO's and a counter.

Validating the Input

The examples we have just examined demonstrate an important rule when designing a program: Whenever you request data at the keyboard, do not assume that it will always be supplied correctly. A user might hit the wrong key, either deliberately or accidentally. To avoid strange or erroneous program behavior, always *validate the input.* If the information typed at the keyboard is not



"Watch my input please."

valid, generate a polite message and request the input again. We will perform input validation in most of our examples.

Let's now develop two complete programs that make decisions.

Mileage Conversion

In Chapter 3, we learned how to perform a simple conversion of miles into kilometers. Here is a way to automate it:

```
10 REM *** MILEAGE CONVERSION ***
20 REM
30 PRINT "I CONVERT MILES INTO KILOMETERS"
40 PRINT "TYPE 0 TO STOP"
50 INPUT "HOW MANY MILES"; MILES
60 IF MILES = 0 GOTO 100
70 KM = MILES * 1.6
80 PRINT MILES; "MILES ="; KM; "KILOMETERS"
90 GOTO 50
100 END
```

Here is a sample run:

```
I CONVERT MILES INTO KILOMETERS
TYPE 0 TO STOP
HOW MANY MILES? 7
7 MILES = 11.2 KILOMETERS
HOW MANY MILES? 10
10 MILES = 16 KILOMETERS
HOW MANY MILES? 0
Ready
■
```

Birthday

Here is one more example. Let's now improve our earlier program that computed a person's age. You will supply today's date and the day, month, and year you were born, the program will tell you your exact age. Here is the program:

```
10 REM *** AGE COMPUTATION ***
20 INPUT "WHAT IS YOUR FIRST NAME"; FIRST$
30 PRINT "HELLO "; FIRST$; " I WILL COMPUTE YOUR AGE"
40 PRINT "WHAT IS TODAY'S DATE? (DD/MM/YY)"
50 INPUT "THE DAY FIRST: ", DD
60 IF DD < 1 OR DD > 31 THEN 50
70 INPUT "THE MONTH (1 TO 12): ", MM
80 IF MM < 1 OR MM > 12 THEN 70
90 INPUT "THE YEAR (2 DIGITS): ", YY
100 IF YY < 0 OR YY > 99 THEN 90
110                                                    REM
120 PRINT "NOW GIVE ME YOUR BIRTHDATE"
130 INPUT "DAY: ", DBORN
140 IF DBORN < 1 OR DBORN > 31 THEN 130
150 INPUT "MONTH (1 TO 12): ", MBORN
160 IF MBORN < 1 OR MBORN > 12 THEN 150
170 INPUT "YEAR (2 DIGITS): ", YBORN
180 IF YBORN < 0 OR YBORN > 99 THEN 170
190                                                    REM
200 REM --- AGE COMPUTATION ---
210 IF MBORN < MM THEN 270
220 IF MBORN > MM THEN 320
230 REM --- BIRTHDAY THIS MONTH ---
240 IF DBORN < DD THEN 270
250 IF DBORN > DD THEN 320
260 PRINT "TODAY IS YOUR BIRTHDAY. CONGRATULATIONS"
270 AGE = YY - YBORN
280 PRINT "YOU ARE"; AGE; "YEARS OLD"
290 END
300                                                    REM
310 REM BIRTHDAY NOT YET REACHED THIS YEAR
320 AGE = YY - YBORN - 1
330 GOTO 280
340 END
```

Despite its length, this program is quite simple. Note how we validate each input. However, to keep the program short our validation is rough. We do not verify that each number is an integer. Nor do we verify the number of days in each month. This is left as an exercise for the thorough (and patient) reader.

Here is a way you could check that M is equal to an integer between 1 and 12:

```
IF NOT (MM = 1 OR MM = 2 OR . . . OR MM = 12) THEN 70
```



Summary

Using the IF and the GOTO statements, we have learnt how to write programs to perform tests on values and to make decisions. We have also learnt how to perform program loops so that a portion of a program can be repeated indefinitely. In addition, we have learnt how to systematically check and validate inputs typed at the keyboard. We have now learnt all the basic skills required to write common programs, and we have examined several meaningful examples as well. We will now make our programs more convenient to write.

Because of the frequency and importance of loops and automation in programs, AMSTRAD BASIC offers additional facilities in the form of additional statements. We will discuss these facilities in the next chapter.

Exercises

6-1: What is the use of the IF statement?

6-2: What is the effect of the following:

```
10 INPUT ANSWER$
20 IF ANSWER$ = "YES" THEN PRINT "THANK YOU"
30 IF ANSWER$ = "NO" THEN PRINT "TOO BAD"
40 PRINT "YES OR NO" : GOTO 10
```

If the result of the above is illogical, suggest a better program.

6-3: Are the following logical expressions valid when used within an IF statement?

a. A = 4

b. B = 2 OR C = 3

c. A > 5

d. 5 > A

e. 1 > 2

f. SUM > NUMBER

g. LETTER$ = "A"

6-4: Is the following valid?

```
IF A = 5 THEN IF B = 2 THEN 18
```

6-5: What is a program loop?



AUTOMATING REPETITIONS



Using the IF and the GOTO statements, we can execute a program segment repeatedly. The corresponding program segment is called a loop; and most programs use loops. In this chapter we will learn improved techniques for creating loops. We will also develop sophisticated programs that automate tasks.

We will begin this chapter with a review of the IF/GOTO technique of generating a loop. We will then introduce a new statement, the FOR . . . NEXT statement, designed to facilitate the creation of loops. We will use this important statement extensively in our programs.

The IF/GOTO Technique

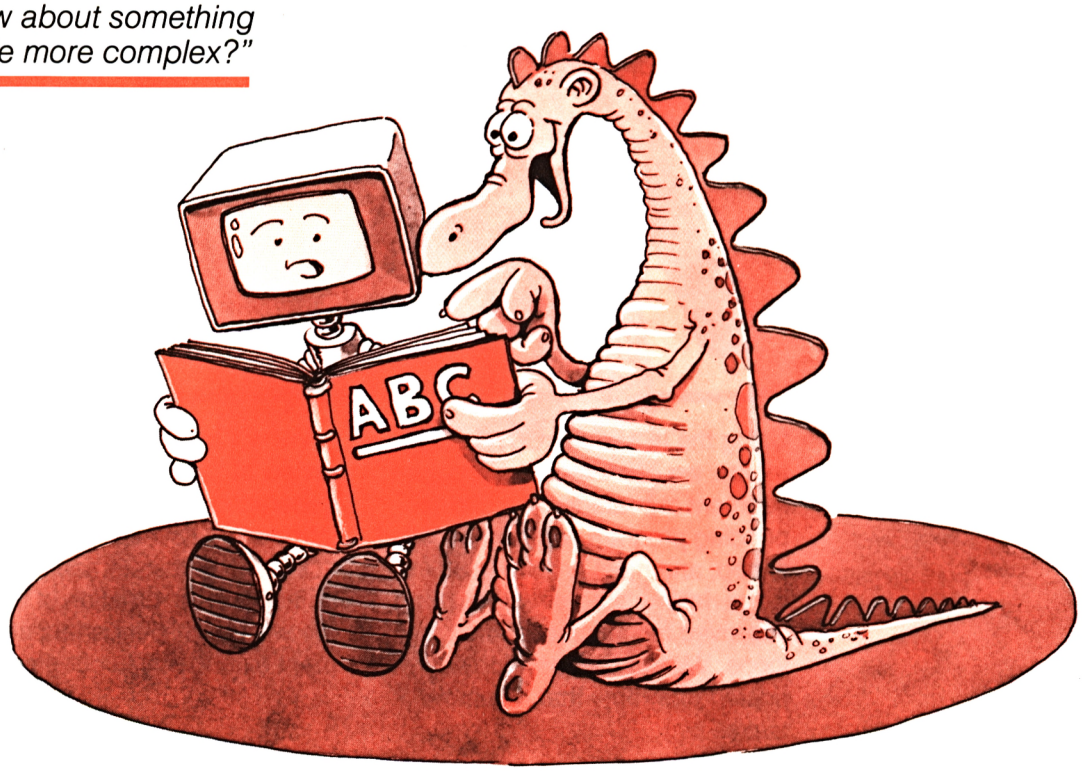
We will begin by examining a program that automates a loop, using the IF/GOTO technique. As we examine this program we will point out certain features common to all loops. For example, we will examine the use of a counter variable, incrementation, initialization, and test before exit. Here is the program. It computes the sum of the first ten integers.

```
1 REM *** SUM OF FIRST 10 INTEGERS ***
10 SUM = 0
20 I = 1
30 SUM = SUM + I
40 I = I + 1
50 IF I = 11 THEN 70
60 GOTO 30
70 PRINT "THE SUM OF THE FIRST 10 INTEGERS IS:"; SUM
80 END
```

Two variables are used in this program: SUM and I. The variable SUM accumulates the sum of the first 10 integers as we keep adding them—it is the equivalent of a subtotal on a calculator. I is the integer that is being added to SUM.

Recall that a variable must have a value the first time it is used. Thus, before we use SUM and I in a formula, we set their values to an *initial* value (0 and 1, respectively). This is accomplished by statements 10 and 20. These statements are called *initialization* statements. If we do not initialize a variable, AMSTRAD

"How about something a little more complex?"



BASIC will set its value to 0 before using it. Readers of your program may not be familiar with this feature of AMSTRAD BASIC, so we include statement 10 for clarity.

The next statement is:

```
30 SUM = SUM + I
```

This statement adds the value of the current integer I to the current SUM. When this statement is executed for the first time, the value of SUM is 0 and the value of I is 1. As a result this statement assigns the value 0 + 1 = 1 to SUM. Following execution of that statement, SUM contains the value 1.

The next statement is:



```
40 I = I + 1
```

The current value of I is 1. The result of this statement is to give I the new value, 2. This is the counter technique: I is incremented by one in order to generate the next integer. At the same time, the value of I indicates how many integers have been added so far. In other words, I is used as the current integer and as a counter.

So, all we have left to do is to go back to statement 30 and keep adding integers:

```
50 GOTO 30
```

"Remember me?"

Wrong. This program will (in theory) never stop (actually it will once the value of SUM becomes larger than the maximum allowed by your interpreter). This is not what we want. We want the program to stop after executing the loop ten times. We must introduce a *test* instruction. Here it is:



```
50 IF I = 11 THEN 70
```

Once I reaches the value 11, statement 70 is executed and the program stops. This is called *exiting* from the loop. Let's now verify that the value 11 (rather than 10) is indeed correct in statement 50. If we write:

```
50 IF I = 10 THEN 70
```

It does not work. Once I reaches the value 10, SUM contains the sum of 1 through 9 only. The loop should be executed one more time.

Remember that each loop contains a counter. You should always carefully check the value of the counter that causes exit from the loop. In our example, as long as I is not equal to 11, the loop will be reentered:

"Got you again!"

```
60 GOTO 30
```

Once I reaches the value 11, the first ten numbers will have been added. This is because in our program the addition (SUM = SUM + I) takes place before the incrementation (I = I + 1).

The final two statements in the program are the *exit* from the loop:

```
70 PRINT "THE SUM OF THE FIRST TEN INTEGERS IS:"; SUM
80 END
```

Here is a sample run of this program:

```
THE SUM OF THE FIRST TEN INTEGERS IS: 55
```

The illustration in Figure 7.1 shows the flow of control in the program. The numbers in parentheses are the statement num-

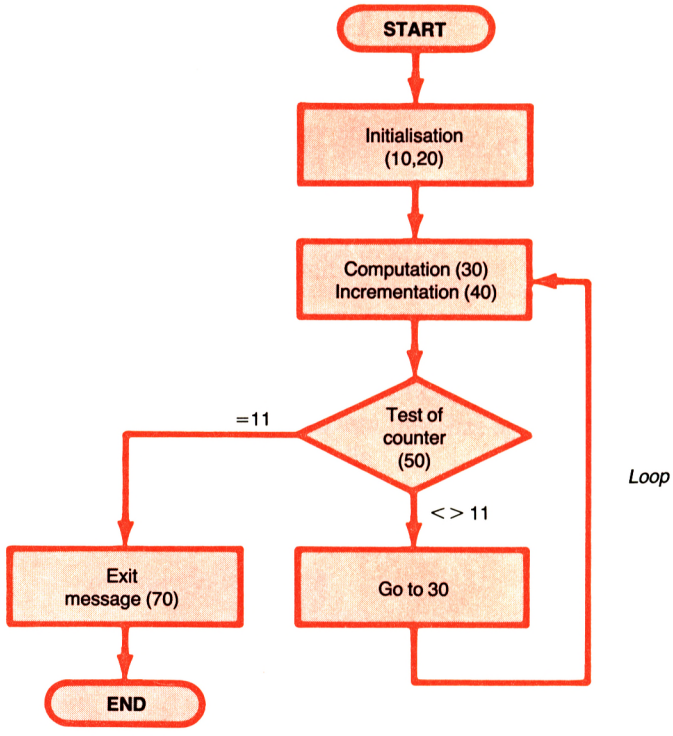


Figure 7.1: Flow of Control in the Integer SUM Program

bers. This diagram is called a *flowchart.* We will discuss the subject of flowcharts in detail in Chapter 8. For now, simply note the general organization of the program: initialization, computation plus incrementation, test, and exit. All program segments with loops perform these functions.

Variations

Let's now play with our integer addition program and sharpen our programming skills. This will demonstrate how many alterna-

tives can be used to write a program. For example, on line 50 we could have written:

```
50 IF I > 10 THEN 70
```

and the result would be the same (when I reaches the value 11, it is greater than 10). Also, we could have written:

```
40 IF I = 10 THEN 70
50 I = I + 1
```

instead of:

```
40 I = I + 1
50 IF I = 11 THEN 70
```

With this change, I is tested first, then incremented. Note that this time I is tested for the value 10 (instead of 11).

We could also have written:

```
50 IF I < 11 THEN 30
60 REM
```

You can verify that these versions are indeed correct. All these variations are valid and equivalent. Even such a short program as the SUM program can be written in many equivalent ways. There is no unique way to write a program. Just like in a spoken language, you can express the same concept in many different ways.

This short program has illustrated the use of a loop and the use of a counter variable. We have also examined the typical phases involved in such a program: initialization, computation, incrementation, test, and exit. In view of the frequent use of loops in programs, a special statement has been provided to facilitate their use in BASIC. This is the FOR . . . NEXT statement.

The FOR . . . NEXT Statement

The FOR . . . NEXT statements automate much of the programming required for a loop. We will explain their use and operation by using actual examples.

Here is one way we can rewrite our addition program using these new statements:

```
1 REM *** INTEGER ADDITION - VERSION 2 ***
10 SUM = 0
20 FOR I = 1 TO 10
30 SUM = SUM + I
40 NEXT I
50 PRINT "THE SUM OF THE FIRST 10 INTEGERS IS:"; SUM
60 END
```

Note that this program has two less instructions than the first one. It is shorter and more readable. Let's examine it in detail.

The first executable statement initializes SUM to zero:

```
10 SUM = 0
```

The next statement is the FOR instruction:

```
20 FOR I = 1 TO 10
```

This instruction has several roles:

- It marks the beginning of the automatic loop. (This is where the loop starts.)
- It specifies that I (the counter variable) starts with the initial value 1 when this statement is first executed. This eliminates the need for an initialization statement for I.
- I is incrementated by 1 (up to a maximum value of 10) every time the statement is reactivated by a matching NEXT statement. An automatic test is performed, and when I exceeds the value 10, the loop is no longer executed, and the statement following the NEXT is executed instead. (This is the loop exit).

The body of the loop simply contains the accumulation of the sum:

```
30 SUM = SUM + I
```

The NEXT statement:

```
40 NEXT I
```

marks the end of the loop and causes reactivation of the FOR. This replaces two statements in the previous version:

```
40 I = I + 1
60 GOTO 30
```

Every time NEXT I is executed, the program jumps to the beginning of the loop , i.e. the FOR statement. When FOR is activated:

- I is incremented by 1
- the new value of I is automatically compared to 10.

As long as I does not exceed 10, execution proceeds. The looping stops when I equals 10, and NEXT is reached. At this point, exit occurs, and statement 50 (following the NEXT) is executed. This sequence is illustrated in Figure 7.2 (a flowchart).

Figure 7.2 shows that the FOR statement automates three tasks:

- initialization of the counter variable (I is set to 1 initially)
- incrementation of the counter variable (I is incremented by 1 each time)
- test of the counter variable against a maximum value (I is compared to 10).

The NEXT statement simply marks the end of the loop and

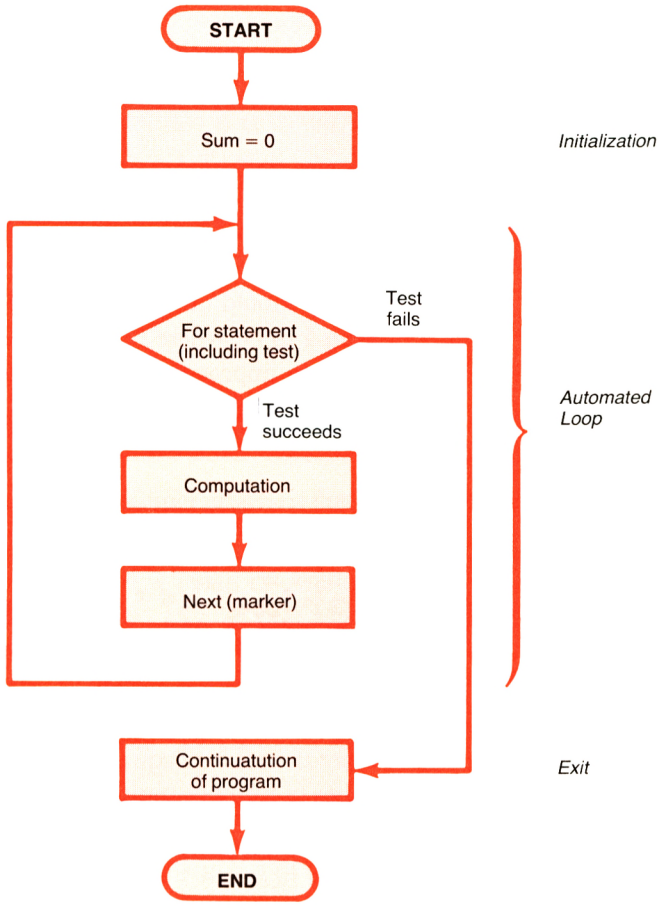


Figure 7.2: Automatic looping with FOR . . . NEXT

causes a "GOTO the FOR" statement. After using the FOR statement a few times, you will appreciate how it simplifies loop design and clarifies the program.

FOR . . . NEXT is a convenience statement. You do not have to use it, but you will probably find it very valuable. Generally, the clearer the program, the lesser the risk of errors. We will now give practical examples to illustrate the use of FOR . . . NEXT statement and the use of automated loops.

Sum of the First n Integers

We will compute the sum of the first N integers. This time the user specifies the value of N at the keyboard. Here is the program:

```
10 REM * SUM OF FIRST N INTEGERS *
20 SUM = 0
30 INPUT "I WILL ADD THE FIRST N INTEGERS. TYPE N: ", N
40 FOR I = 1 TO N
50 SUM = SUM + I
60 NEXT I
70 PRINT "THE SUM OF THE FIRST"; N; "INTEGERS IS"; SUM
80 END
```

Here is a sample RUN:

```
I WILL ADD THE FIRST N INTEGERS. TYPE N: 5
THE SUM OF THE FIRST 5 INTEGERS IS 15
```

You should understand the program readily. This time, we loop from 1 to N, where N is supplied at the keyboard (statement 30). You can improve this program by validating the input: N should be an integer greater than 1.

Tables of Values

Using the powerful FOR . . . NEXT statement, we will show how easy it is to automate computations and print tables of values. Here is a table of squares (a number multiplied by itself) and cubes (a number multiplied by itself and multiplied by itself again):

```
10 REM TABLE OF SQUARES AND CUBES
20 REM FOR THE FIRST 10 INTEGERS
30 FOR I = 1 TO 10
40 PRINT I, I ↑ 2, I ↑ 3
50 NEXT I
60 END
```

Here is the result:

```
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

Let take a closer look at statement 40:

```
40 PRINT I, I ↑ 2, I ↑ 3
```

I ↑ 2 means I to the power 2, i.e., I * I. For example, if I = 2, then I ↑ 2 = 2 × 2 = 4.

Similarly, I ↑ 3 means I to the power 3, i.e., I * I * I. If I = 4, then I ↑ 3 = 4 × 4 × 4 = 64.

Notice that we use a comma this time in the PRINT statement so that the results are neatly aligned as they are listed. The comma forces an automatic spacing (or *tabulation*) on the line. The spacing between columns on your AMSTRAD BASIC is 13 characters.

As an exercise you could rewrite this program to display the sum of the squares and cubes of the first N integers, where N is read at the keyboard. We learnt how to do this in the previous section.

Lines of Stars

Here is a simple program that prints N lines of stars, where N is a number you specify at the keyboard:

```
10 REM * LINES OF STARS *
20 PRINT "I DISPLAY LINES OF STARS"
30 INPUT "TELL ME HOW MANY LINES: ", N
40 REM N IS THE NUMBER OF LINES TO BE DISPLAYED
50 FOR I = 1 TO N
60 PRINT "**************************"
70 NEXT I
80 END
```

And here is a sample run of this program:

```
I DISPLAY LINES OF STARS
TELL ME HOW MANY LINES: 6
**************************
**************************
**************************
**************************
**************************
**************************
```

Again, every time an input is supplied by the user, it is a good idea to validate it in order to avoid strange program behavior. We expect that the person using this program will supply a positive number. Let's assume you want no more than 20 lines of stars. You would tell the user in an appropriate PRINT statement, and use a validation statement like:

```
IF (N < 1) OR (N > 20) GOTO 20
```

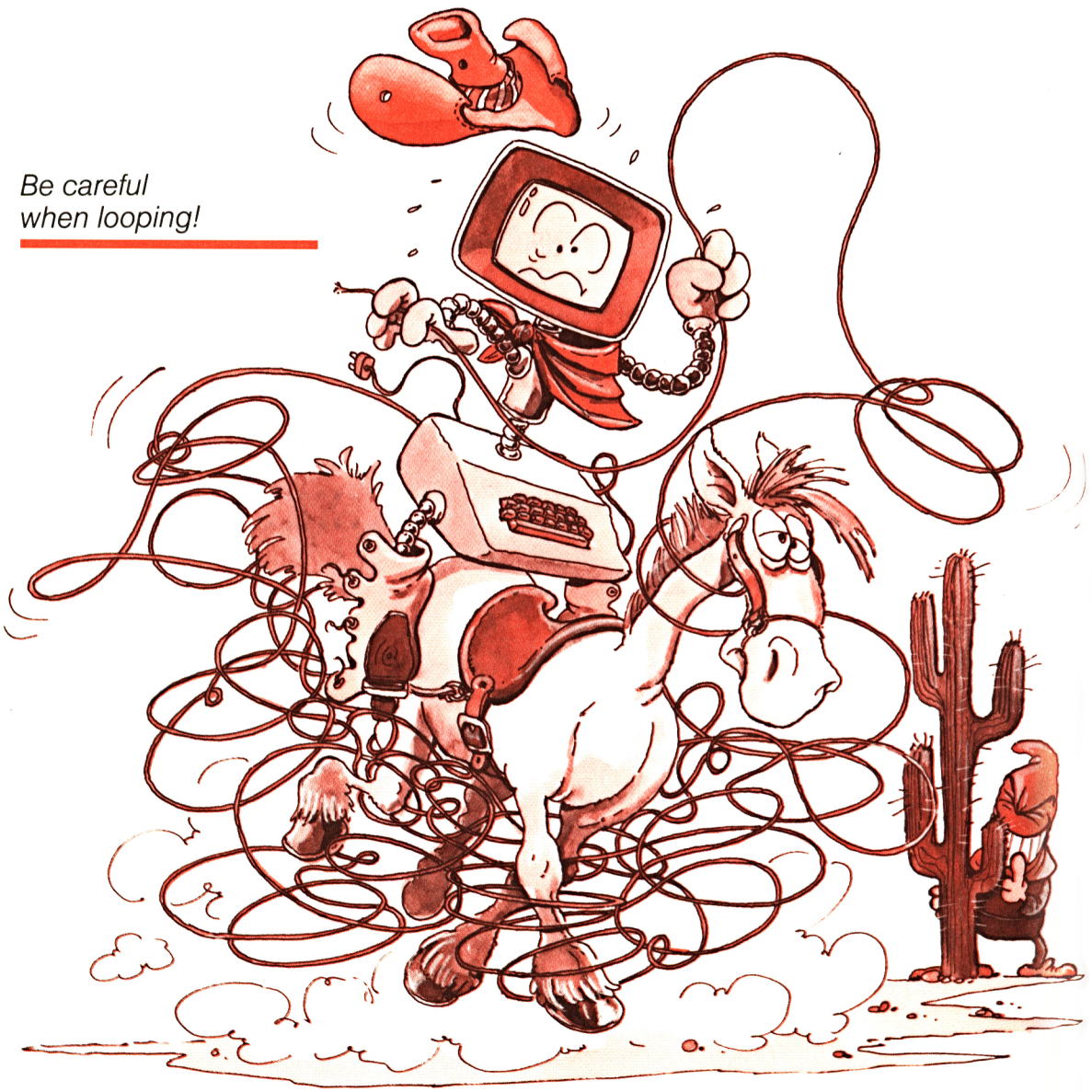
Advanced Looping

The FOR . . . NEXT statement offers two advanced facilities that we have not yet described:

- you can increment the counter by any integer or decimal value, such as 2, 3, 6.4, or even −1 (rather than just incrementing by 1). This is called the *variable step* feature.
- you can create a loop within another loop. Such loops are called *nested loops*.

Let us examine these two techniques.

Be careful when looping!



Variable Step

Here is an example of variable step:

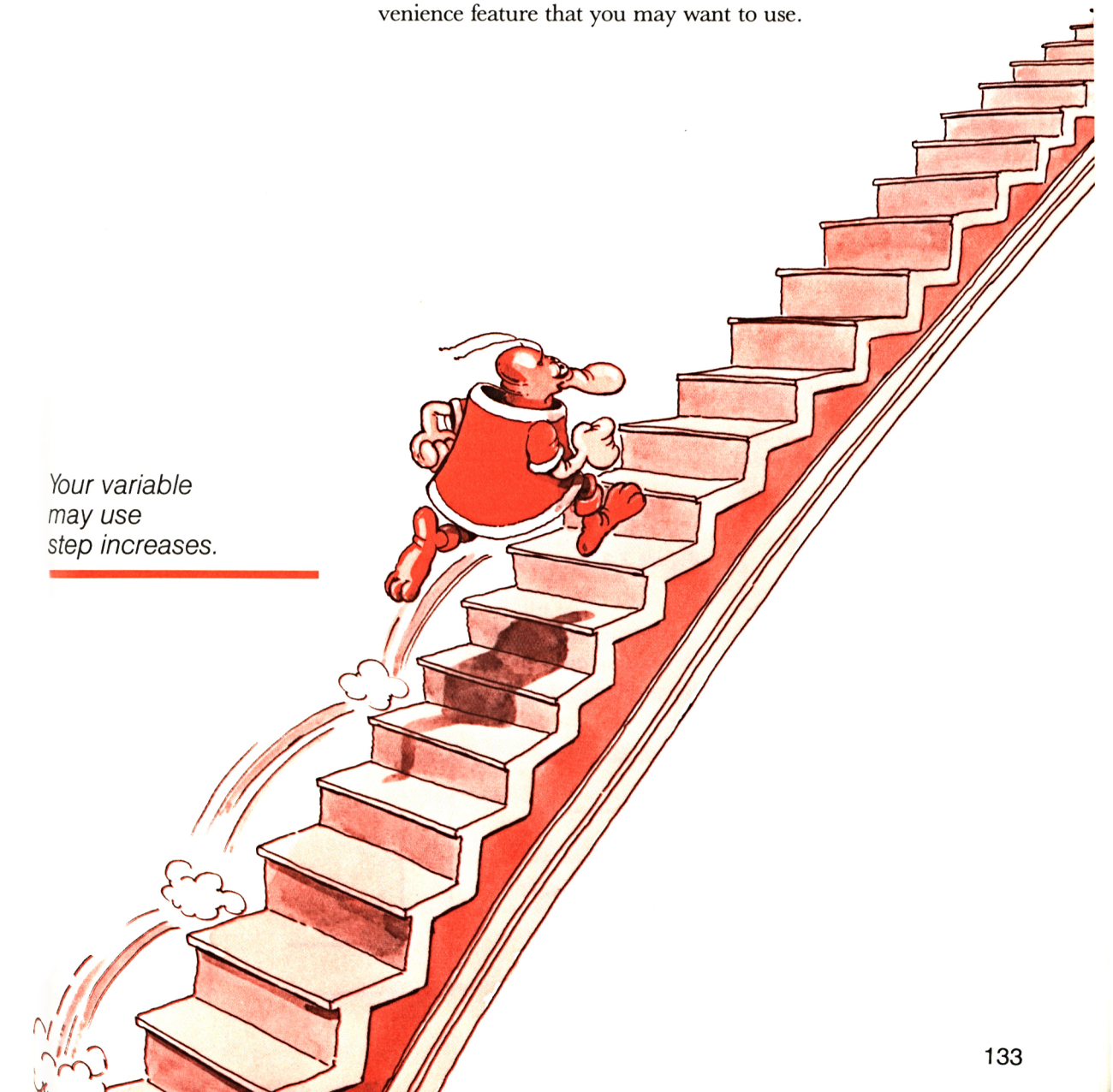
```
FOR I = 1 TO 5 STEP 2
```

Every time the loop is re-entered I will be incremented by 2. (*Note:* we have added boldface here for emphasis.) You could even write:

```
FOR I = 10 TO -5 STEP -1
```

using a *negative step* increment. Because the upper limit for the counter variable (− 5 here) is less than the starting value (10), it is a "negative step" for the interpreter. The value of I will be decreased by 1 each time. The first value of I will be 10. The next will be 9; the next 8, etc. The last one will be − 5. In other words, I will take the following values in turn: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, − 1, − 2, − 3, − 4, − 5. Negative step is another convenience feature that you may want to use.

Your variable may use step increases.



Nested Loops

The nested loop technique is an important and powerful facility used to automate complex processing. A nested loop is created whenever you use a FOR . . . NEXT group of statements within a loop; that is, whenever you use another FOR . . . NEXT group of statements.

In general, you may use as many statements as you wish between the FOR and the NEXT statements. Specifically, you may even include another loop within these statements. When this takes place it is called a nested loop. This concept is illustrated in Figur 7.3.

When using nested loops, notice how the program becomes more difficult to read. To remedy this, you are encouraged to use *indentation.* Indentation is another program clarification technique. Figure 7.4 shows an indented version of the program of Figure 7.3.

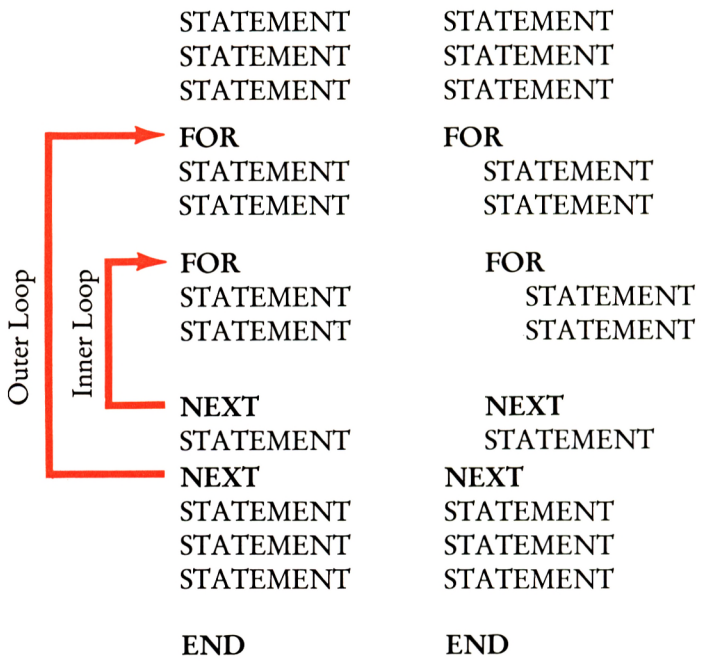
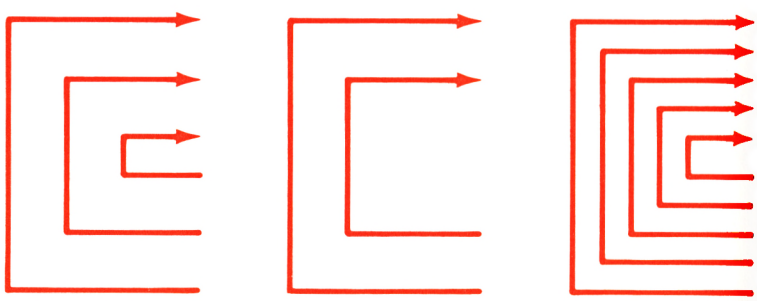


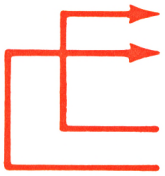
Figure 7.3: A Nested Loop

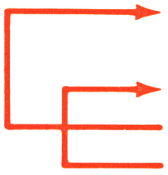
Figure 7.4: Indented Program

You can nest loops to any level, up to a maximum number depending on your interpreter or the amount of memory you have. However, you may not overlap loops. The following loops are legal:

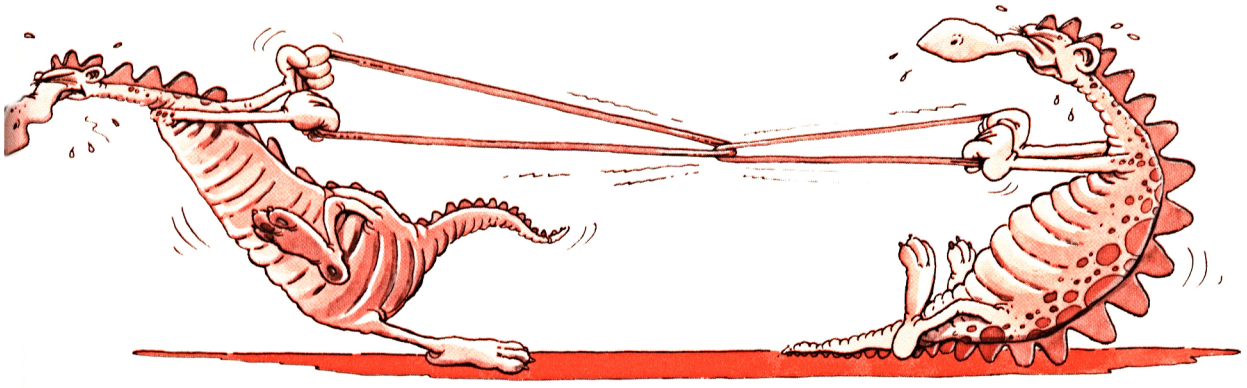


The following are not legal:

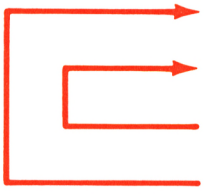




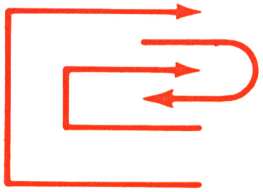
Don't overlap loops!



In addition, you may not jump (i.e, specify a GOTO) from a point *inside* the outer loop to a point inside the inner one:

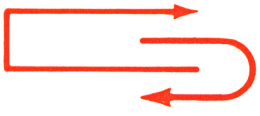


Correct Nesting

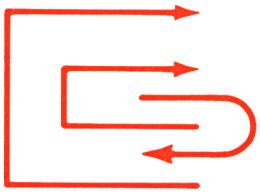


Illegal jumping
(with IF or GOTO)

However, you may jump out of the inner loop:



Correct Jumping



Correct Jumping

Here is an example of program which includes a nested loop and the use of indentation to improve readability. This program displays accelerated time in minutes and hours:

```
10 REM *** SIMULATED CLOCK ***
20 FOR HOUR = 0 TO 23
30     FOR MINUTE = 0 TO 59
40         PRINT "THE TIME IS"; HOUR; "HOURS AND"; MINUTE;
"MINUTES"
50     NEXT MINUTE
60 NEXT HOUR
70 PRINT "END OF THE DAY"
80 END
```

Here is a portion of the run:

```
THE TIME IS 0 HOURS AND 0 MINUTES
THE TIME IS 0 HOURS AND 1 MINUTES
THE TIME IS 0 HOURS AND 2 MINUTES
THE TIME IS 0 HOURS AND 3 MINUTES
THE TIME IS 0 HOURS AND 4 MINUTES
                    .
                    .
                    .
THE TIME IS 0 HOURS AND 59 MINUTES
THE TIME IS 1 HOURS AND 0 MINUTES
```

Additional Features

As a final note, decimal values are allowed in the FOR statement. For example, the following is valid:

```
FOR MEASURE = 0.1 TO 13.5 STEP 0.2
```

Expressions are not allowed in the FOR statement. For example, the following is *not* valid:

```
FOR VALUE = N TO (N * 2) STEP 1
```



Summary

Loops are used extensively to automate the repetition of a program segment. The FOR . . . NEXT statement is used to automate loops in BASIC. In most cases, the FOR . . . NEXT statement can replace several other BASIC statements. In this chapter, we have examined typical uses of the FOR . . . NEXT statement, including nested loops, and we have developed several advanced programs. Now that you have learned all the basic programming techniques, you are almost ready to start writing your own programs. In the next chapter we will explain how you can get started.

Exercises

7-1: Display the first 15 integers on a line (4 statements).

7-2: Write a program that reads the time in hours and minutes at the keyboard and displays it as follows:

Input:	3 (*hours*),	31 (*minutes*)
Display:	H H H	(*3 letters*)
	M M M	(*3 letters*)
	M	(*1 letter*)

7-3: What is the counter variable in a loop?

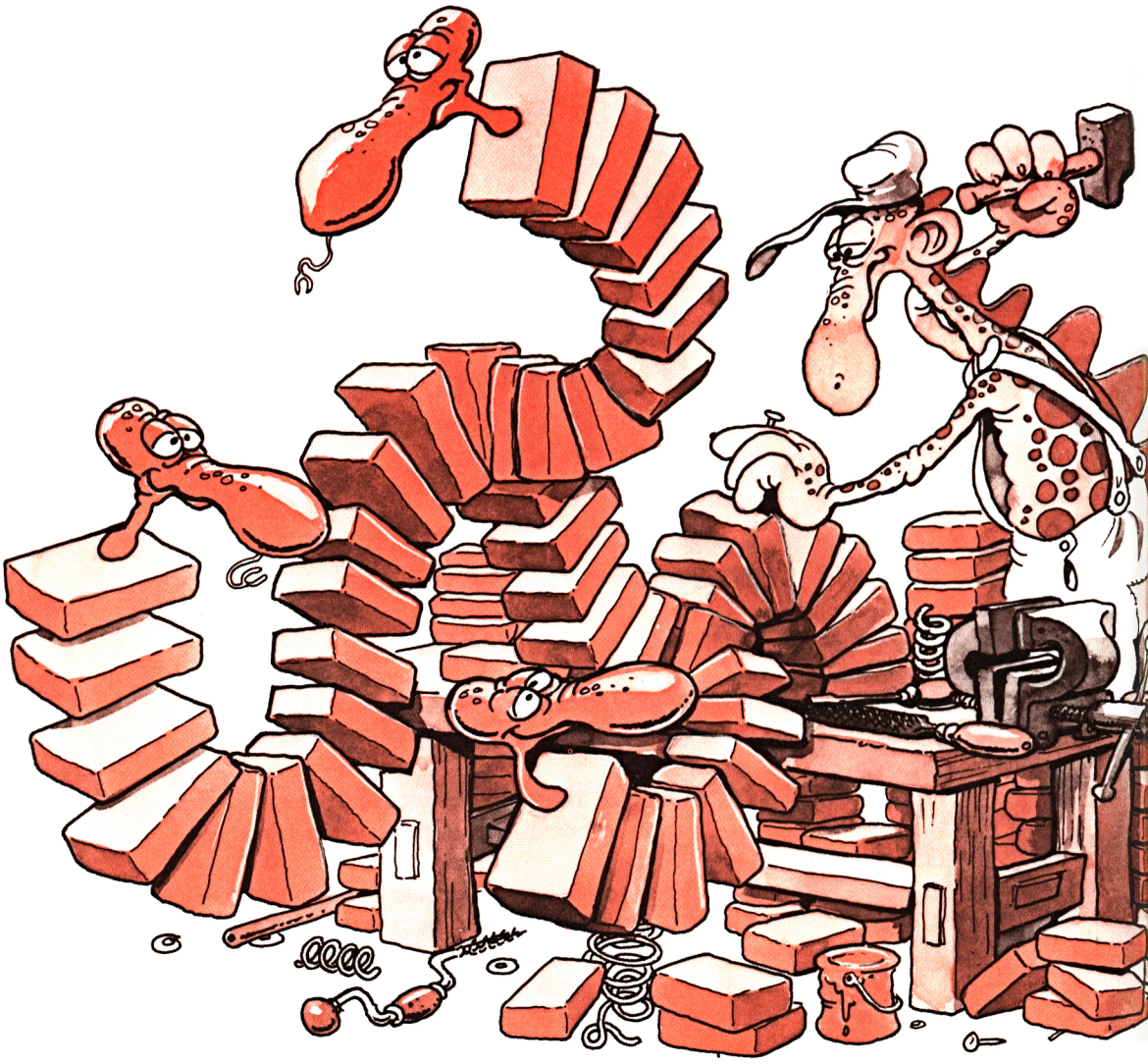
7-4: Can you jump into the middle of a loop?

7-5: Display a table that converts ounces to grams (1 ounce = 28 grams).

7-6: Compute the sum of the first N odd integers—where N is supplied at the keyboard—and display it for each integer.

7-7: Read scores for five students that each took four tests, graded 0 to 10. Display the grades, then the total and average for each.

7-8: Display a value added tax table for prices of £1 to £100 in £1 increments. Supply the tax rate at the keyboard.



CREATING A PROGRAM

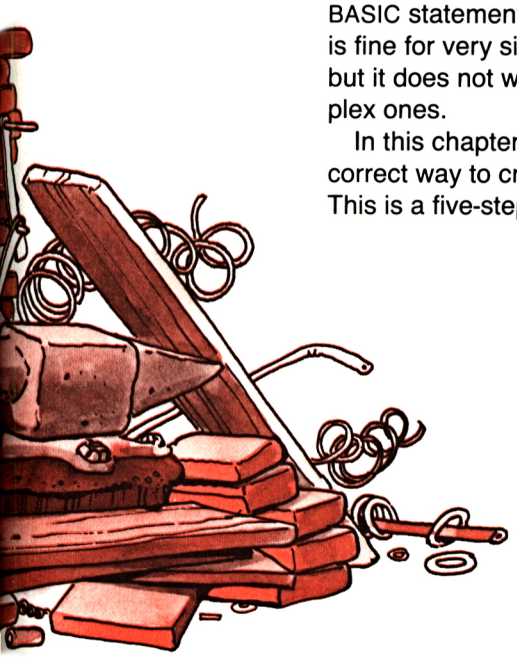
8

Programming involves designing a program that automates a task. So far, we have written several short programs. We have done so without any intermediate steps—directly writing a sequence of BASIC statements. This technique is fine for very simple programs, but it does not work well for complex ones.

In this chapter, we will learn the correct way to create a program. This is a five-step process:

1. Specify the sequence of steps involved in solving the problem. This is called designing the *algorithm.*
2. Draw a diagram showing the sequence of events and the logical steps. This is called drawing the *flowchart.*
3. Write the program in BASIC. This is called *coding.*
4. Verify and test the program. This is called *debugging.*
5. Clarify and document the program. This is called *documenting.*

So far, we have only learned and practiced steps 2 and 5. But this sequence will work only for short programs. Before we proceed to long programs, let's study the complete sequence involved in program development.



Algorithm Design

We want to design a program that will solve a given problem or automate a task. So far, we have designed programs to solve simple problems. The sequence of steps required to solve each problem was generally obvious so that there was almost no design phase. In the general case, however, given a problem we must first design a solution. In order to write programs, this solution must be specified as a sequence of steps. This sequence of steps is called an *algorithm.* Formally, an algorithm is defined as the step-by-step specification of the solution to a problem. Technically, an algorithm must also terminate—it should not go on indefinitely. An algorithm that does not stop is called an *error!*

Here is a simple problem: let's convert a weight measured in ounces into its equivalent in grams. Recall that one ounce is equivalent to 28.35 grams. The solution is obvious: we multiply the weight in ounces by 28.35. This is a simple one-step algorithm.

Let's now examine a slightly more complex problem: let's read a number at the keyboard and test that it is within a certain range. We will accept the number as being valid if it falls in the range between 0 and 100. The sequence of steps involved in solving this problem is the following:

1. Read the number
2. Check to see if it is greater than zero. If so, proceed; if not, reject the number.
3. Check to see if it is less than 100. If yes, accept the number; if not, reject it.

This is a 3-step algorithm.

In practice, most problems are more complicated, and their solutions require longer and more complex algorithms. Here are several everyday examples of algorithms. You can find many more in your cookbook, or your car or appliance manuals.

Let's first examine an algorithm for boiling a three-minute egg. Here are the steps:

1. Take a saucepan
2. Fill it with water
3. Turn on the cooker
4. Place the saucepan of water on the cooker
5. Bring the water to a boil
6. Place an egg in the boiling water
7. Start the three-minute timer
8. When the timer rings, remove the egg
9. Turn off the cooker



"I'll demonstrate the three-minute egg alorithm."

This algorithm looks straightforward, but if it were to be executed as a program by a computerized robot, it would have to be much more precise. For example, we would need to specify exactly which saucepan to use, as well as the precise amount of water to put in the saucepan.

Many algorithms presented in everyday books assume that the user has a specific cultural or technical background, and they are therefore generally incomplete. In other words, they assume that a user can fill in the blanks. Unfortunately, this is precisely why so many manuals are so difficult to comprehend!

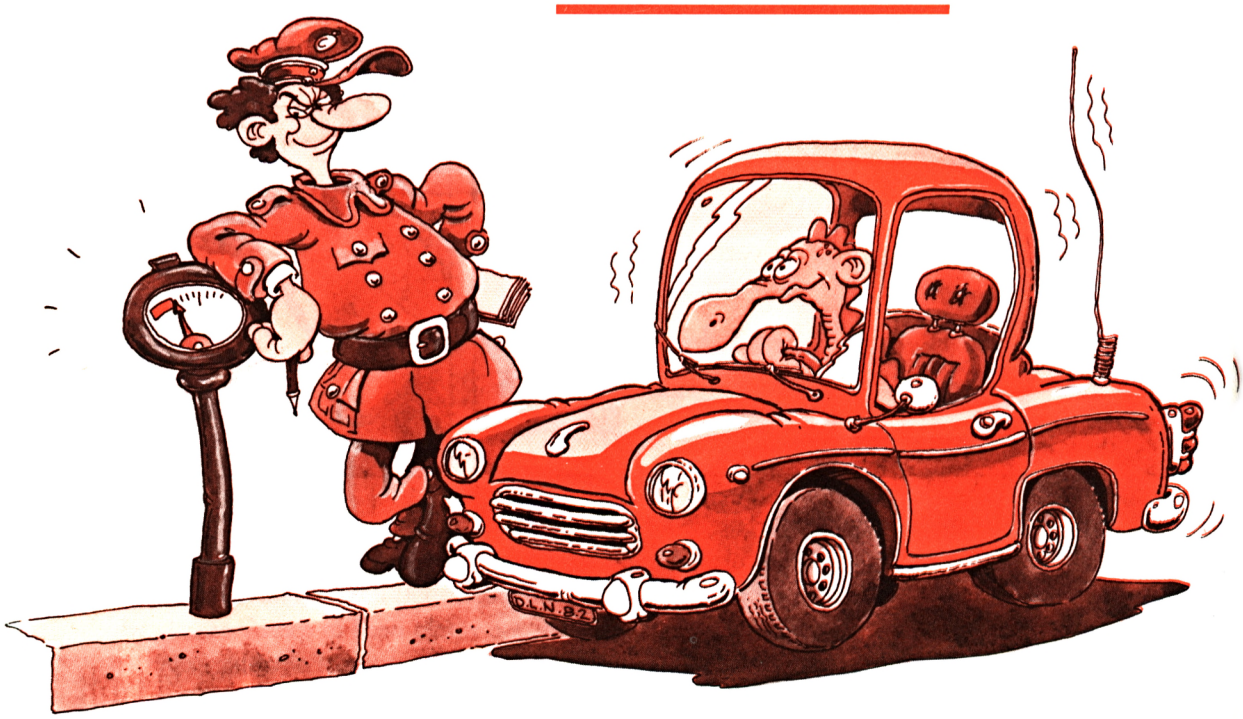
We will not make the same mistake. Our algorithms will be completely specified in order to become useable programs.

Here is one last example: an algorithm for starting a car. If we assume that the car works perfectly, then the algorithm is quite simple:

1. Insert the key in the ignition.
2. Turn the key all the way to the right.
3. Release pressure on the key while applying gentle pressure on the accelerator.

However, we know that is possible that a car may not start. This is because there are other factors involved, such as temperature, or the mechanical condition of the engine. Preparing a complete algorithm for starting a car under all conditions would require several pages—if we take into account everything that could possibly go wrong.

Starting a car is an interesting algorithm.



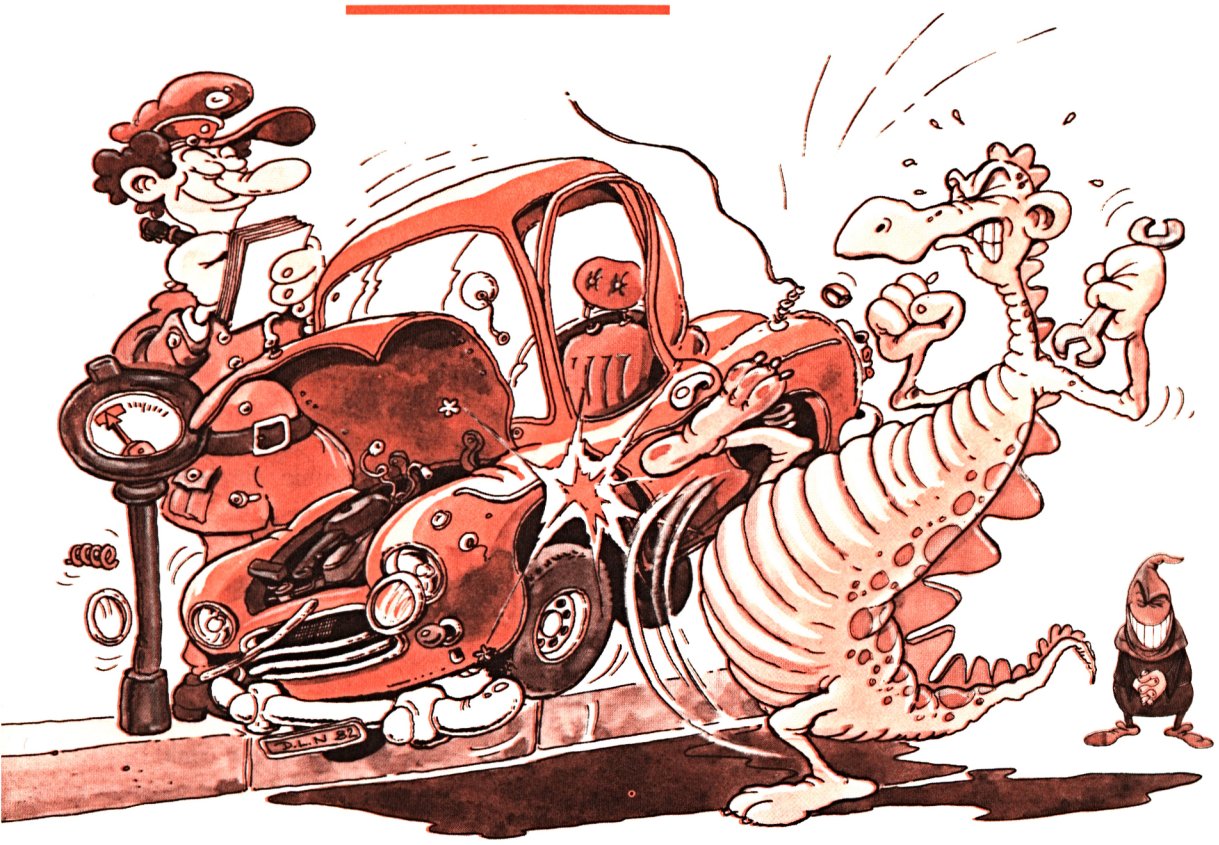
In everyday life we can often simplify the steps of an algorithm. But, in a computer program we may not. An algorithm has to be correct and complete.

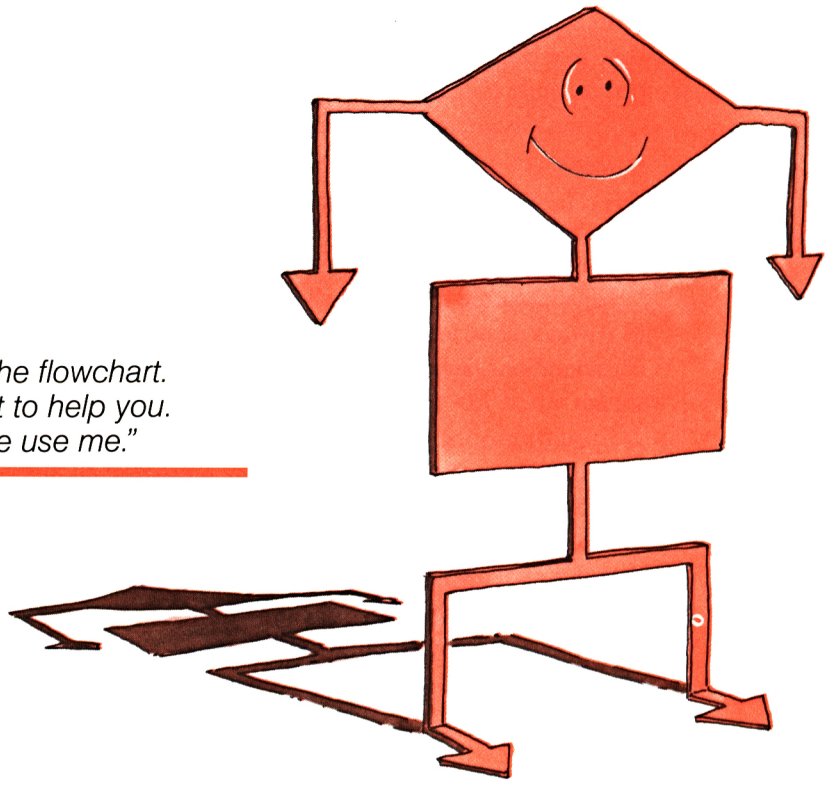
When designing an algorithm for a computer solution, you must be thorough and anticipate every reasonable case that may arise, or your program may eventually fail. Successful programming requires a special attitude: you must continuously challenge everything you do, always assuming that it could be wrong or incomplete.

Never assume an input will be reasonable. Check it and verify it. Always allow for the possibility of errors. We will illustrate these considerations later in this chapter when we examine an actual case study.

In summary, in order to automate the solution to a problem by writing a computer program, start by preparing an algorithm. The final algorithm should be perfect—although, in the beginning it rarely is. In fact, you will probably design an approximate solution (i.e., a rough algorithm) in the beginning and then continue to refine the algorithm as you go along, until it reaches the state you consider to be perfection. Always make sure your algorithm is complete before you start actually writing instructions

Remember, the algorithm is supposed to work!





*"I'm the flowchart.
I want to help you.
Please use me."*

Flowcharting

So we have designed an algorithm. The thought that now comes to mind is: let's translate it promptly into a BASIC program and run it! Wrong. There is one more step that may save you hours of programming time: it is called flowcharting. If you skip this step, you will probably not write a working program, and you will probably waste much time later trying to correct the program—with no guarantee of success. By contrast, once you have a good flowchart, writing the program is one easy step away.

A flowchart is simply a diagram showing the sequence of events. Figure 8.1 shows a flowchart of the steps involved in boiling a 3-minute egg.

As you can see, this flowchart is a graphic representation of the algorithm we have already presented. In this case, each box in the flowchart represents one step of the algorithm. The purpose of a flowchart is to show the sequence of the steps over time. Later, once you become familiar with flowcharting, you may even skip the algorithm design step and begin directly with flowcharting, since the flowchart is just a representation of the algorithm.

In the case of an algorithm as simple as the one for preparing a 3-minute egg, the flowchart is not very useful and could be dispensed with. The true value of a flowchart becomes apparent when

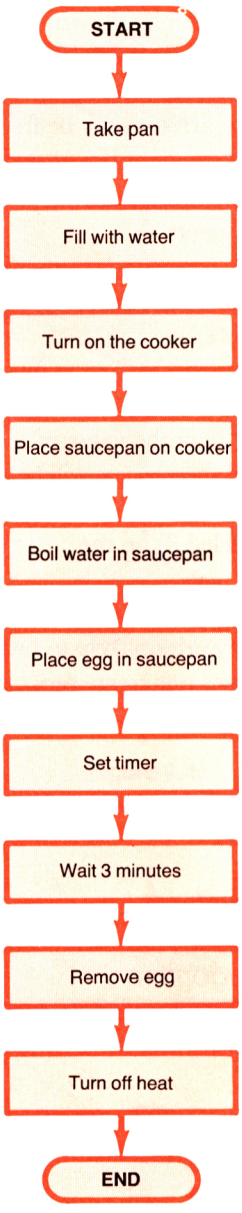


Figure 8.1: Boiling a Three-Minute Egg Flowchart

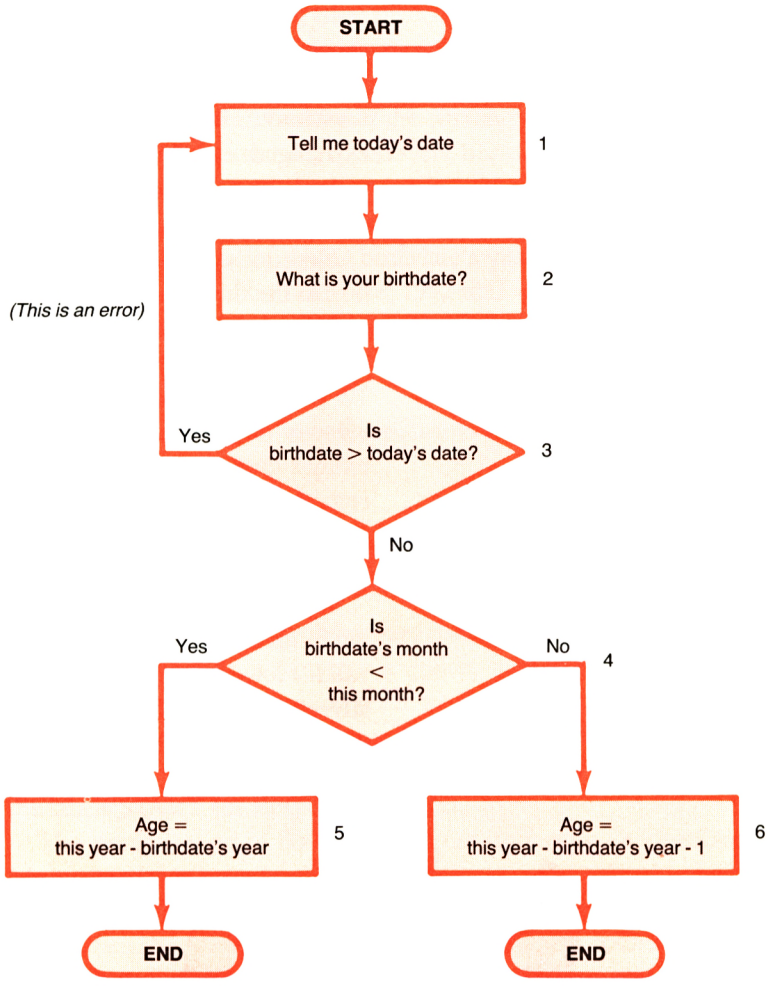


Figure 8.2: Age Computation Flowchart

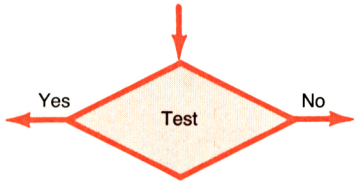


Figure 8.3: Diamond Symbol

you begin designing more complex algorithms that involve numerous choices and decisions.

Let's now design a new program that asks for your birthdate and the current date, and then computes your age. The algorithm is obvious. The flowchart is shown in Figure 8.2. The diamond-shaped boxes in the flowchart indicate a test, i.e., a choice in the sequence. At this point, to facilitate converting the flowchart into a program, make sure that each choice has only *two* results: "yes" or "no." Label the arrows accordingly. Now examine Figure 8.2 and verify that each arrow coming out of the diamond-shaped boxes is labeled either "yes" or "no," depending on the result of the test (see Figure 8.3).

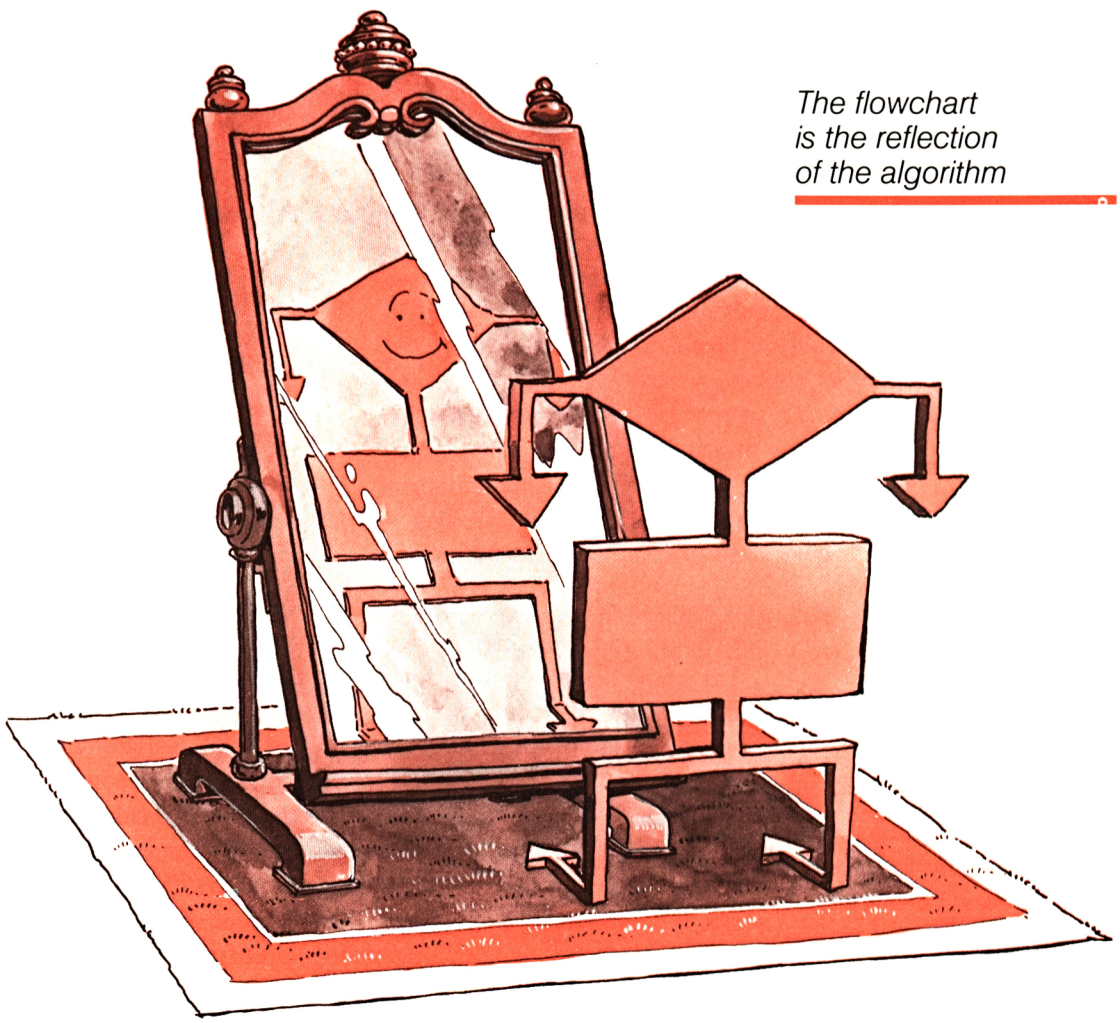
In general, there are three ways the arrows can be drawn, as shown in Figure 8.4. You may select any way you prefer. The purpose of your selection should simply be to facilitate reading the flowchart. The position of the yes and the no arrows may be freely exchanged; in other words, "yes" can be on the right if you prefer.

Let's now go back to Figure 8.2 and the age computation flowchart and examine the algorithm it represents.

First, today's date is requested. This step corresponds to the first box in the flowchart (labeled 1). Second your birth date is requested. This is the box labeled as 2 on the flowchart.

Next, we must verify that the birthdate supplied is reasonable. We must check that the birthdate is earlier than today's date. If the value of the date of your birthdate is greater than the value of today's date, an error will be recognized and the process will be restarted. If not, the birthdate will be presumed valid. This step corresponds to the diamond shaped box (labeled 3) on the flowchart.

To be even more refined, we could also reject birth dates resulting in an age of 150 years or more, since they are unlikely to be valid. However, accepting such birthdates does not create serious adverse affects, and it may not be worth the trouble to check for them.



The flowchart is the reflection of the algorithm

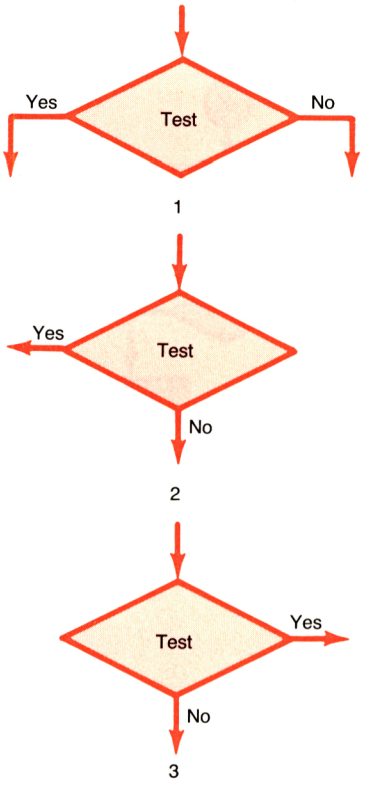


Figure 8.4:
The Three Ways of Drawing Arrows

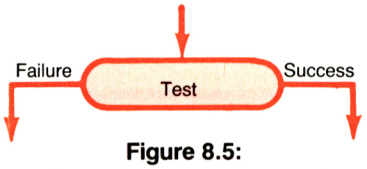


Figure 8.5:
Other Shape for Decision Box

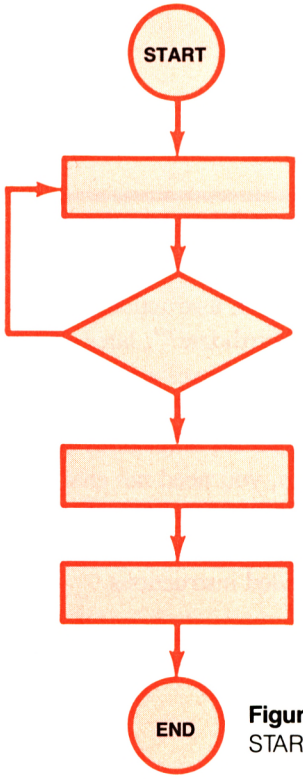


Figure 8.6:
START and END Symbols

In box 4 on the flowchart, we determine whether the month of your birthday is less than today's month. If so, your birthday has passed for this year, and your age may be computed (in box 5 of the flowchart) as the difference of the current year and the year of your birthday. For example, if you were born in Feburary 1946 and we are in March 1984, your age is 1984 – 1946 = 38.

Otherwise (this is box 6 on the flowchart), your age is computed as the current year minus your birthdate's year, minus 1. For example, if you were born in June 1942 and we are in March 1984, your age is 1984 – 1942 – 1 = 41. To keep this example simple, we will not check for the day of the month. We'll add this improvement later.

The steps of the algorithm should now be clear. Let's examine the flowchart symbols.

The Symbols in the Flowchart

In a flowchart, rectangular boxes are used for computations and direct actions that do not involve a choice, such as input or display. Diamond-shaped boxes are used for tests or choices. They must always have at least two arrows coming out of them. Finally, each algorithm must have a beginning and an end. This is denoted by the labels START (or BEGIN) and END.

The symbols used in flowcharts are not uniformly standardized. Many standards have been proposed but none has gained universal acceptance. The rectangular box is always used. The diamond-shaped box, however, may be replaced by one with rounded edges, as shown in Figure 8.5. In addition, the START and END symbols can be placed in a small circle, as shown in Figure 8.6. Finally, some special symbols may be used to indicate the use of specific peripherals. For example, a PRINT operation may appear in one of two ways, as shown in Figure 8.7.

In practice you need not worry about the symbols. A flowchart is simply a way of conveniently visualizing an algorithm (especially when it contains many choices). You could even use different symbols if you wished. However, it is better to use common ones so that your programs may be more easily shared with others, and so that you can read and follow other flowcharts more easily.

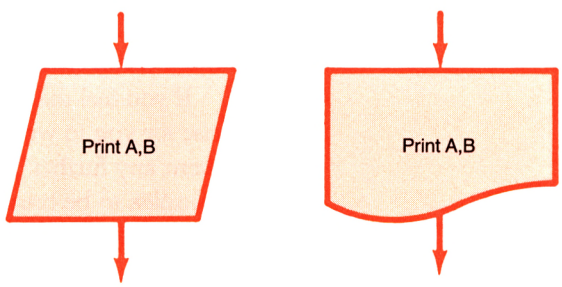


Figure 8.7: Symbols for PRINT

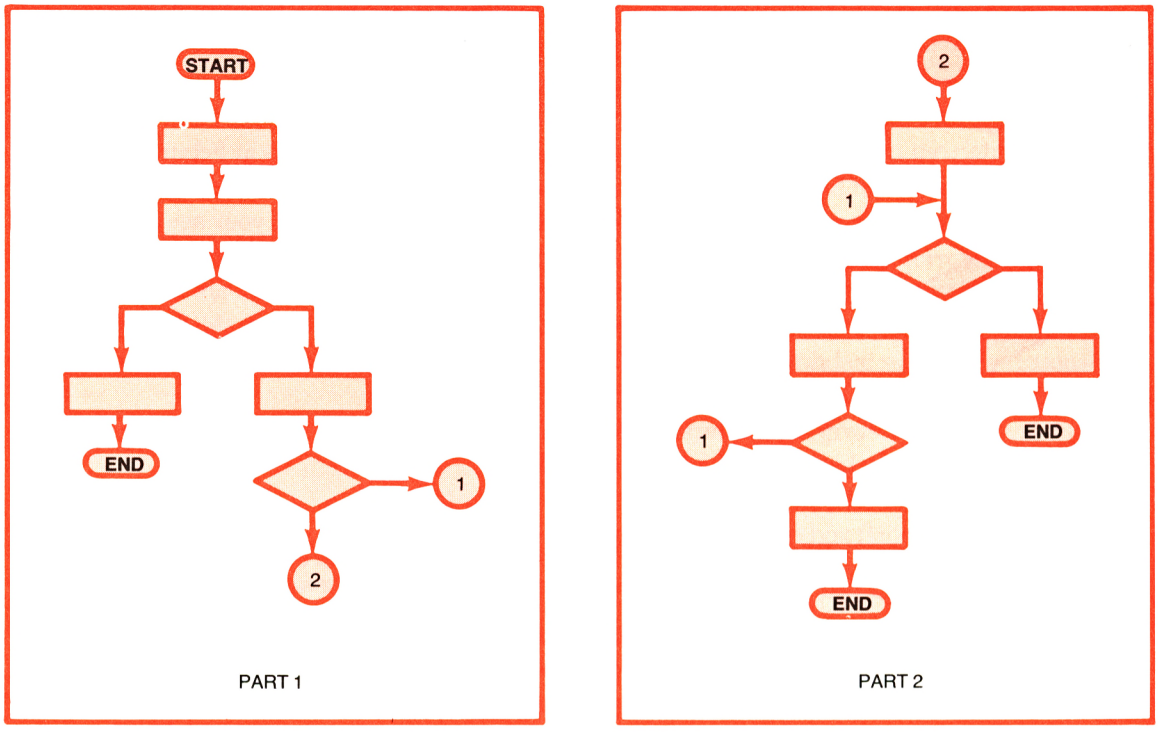


Figure 8.8: Cutting a Flowchart

Cutting the Flowchart

Here is one more useful convention. If a flowchart spreads out over several pages, you can cut it into pieces. Label each cut arrow with a number or name, and make sure you have matching entry points to the flowcharts on the other pages. Figure 8.8 shows an example of using numbers to connect arrows.

Refining the Flowchart

The instructions placed in the boxes of the flowchart may be written as you please. They are not BASIC statements. When writing your flowchart for the first time, you may write instructions that are somewhat vague such as "tell me your birthdate?"; later on, you may refine the statements contained in the boxes and design a more detailed chart that will be easier to translate into a program.

If you feel that the instructions in the boxes are sufficiently precise for you to write an equivalent program, you need not change them any further. If you feel, however, that they are too vague or complex to be translated directly into a program, then you should replace them with a sequence of more-detailed instructions.

As an example, you may recall that the flowchart in Figure 8.2 checks for the month of your birthdate but not the day, when determining whether your birthday has already occurred this month. Let's refine it so that it checks for the month and the day. The corresponding segment of the initial (rough) flowchart appears in Figure 8.9. The new or refined flowchart is shown in Figure 8.10.

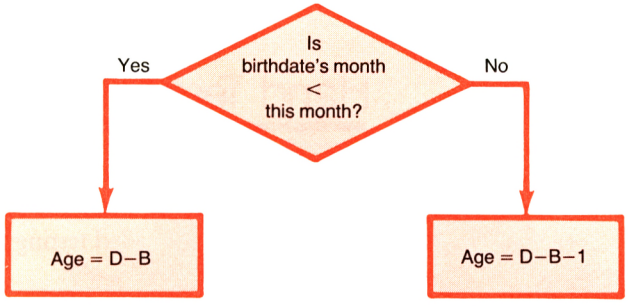


Figure 8.9: Rough Flowchart

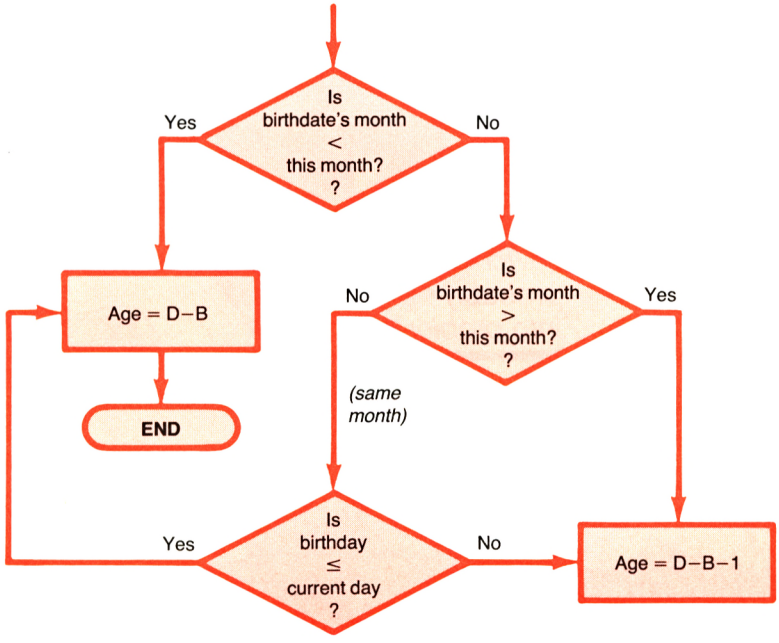


Figure 8.10: Refined Flowchart

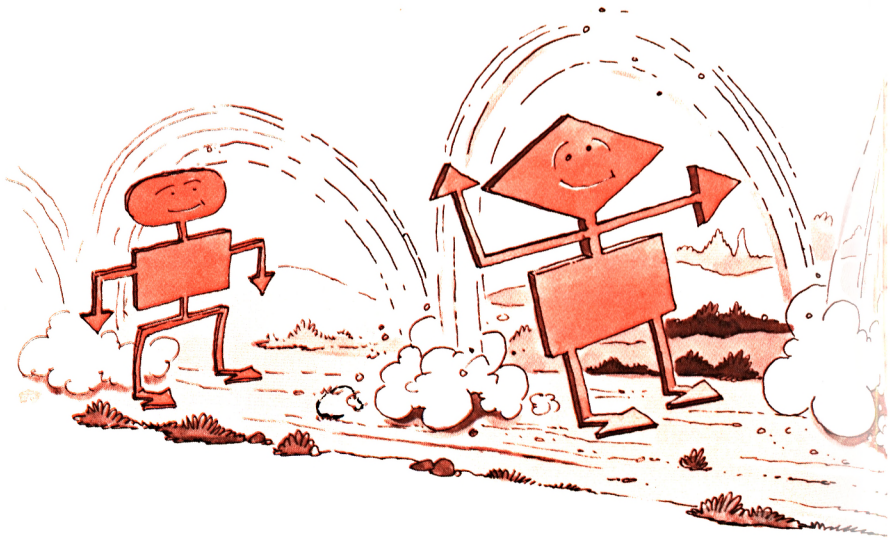
In practice most people do not write out the algorithm; they proceed directly to the flowcharting phase. This is fine. However, sometimes experienced (and not-so-experienced) programmers skip the flowcharting phase as well, and start by writing the *program* on paper. This is highly dangerous and error prone. I strongly recommend that you always draw a flowchart before writing any program. Later on as you become an expert programmer, you may be able to dispense with the detailed flowchart, and draw only a rough one.

Hand Testing

Once you have written a flowchart, test it by trying out actual examples. Make sure that the result is correct or at least appears to be correct. This is called testing *by hand,* as opposed to using the computer.

For example, refer to the age computation flowchart in Figure 8.2 and supply today's date and your own birthdate, as indicated. Does it give you the correct age? If so, things look good. If not, there is an error. Now, try it again with different values, using birthdates that fall before and after today's date. Does it still work? If so, the algo-

IF you skip flowcharting, you'll probably run into problems



rithm is probably correct. If not, there is an error. Hand testing is a quick way to insure that there is no obvious mistake.

Having written a flowchart that seems to work, we have now accomplished all of the preliminary steps required before writing an actual program. Let us now write the program.



Coding

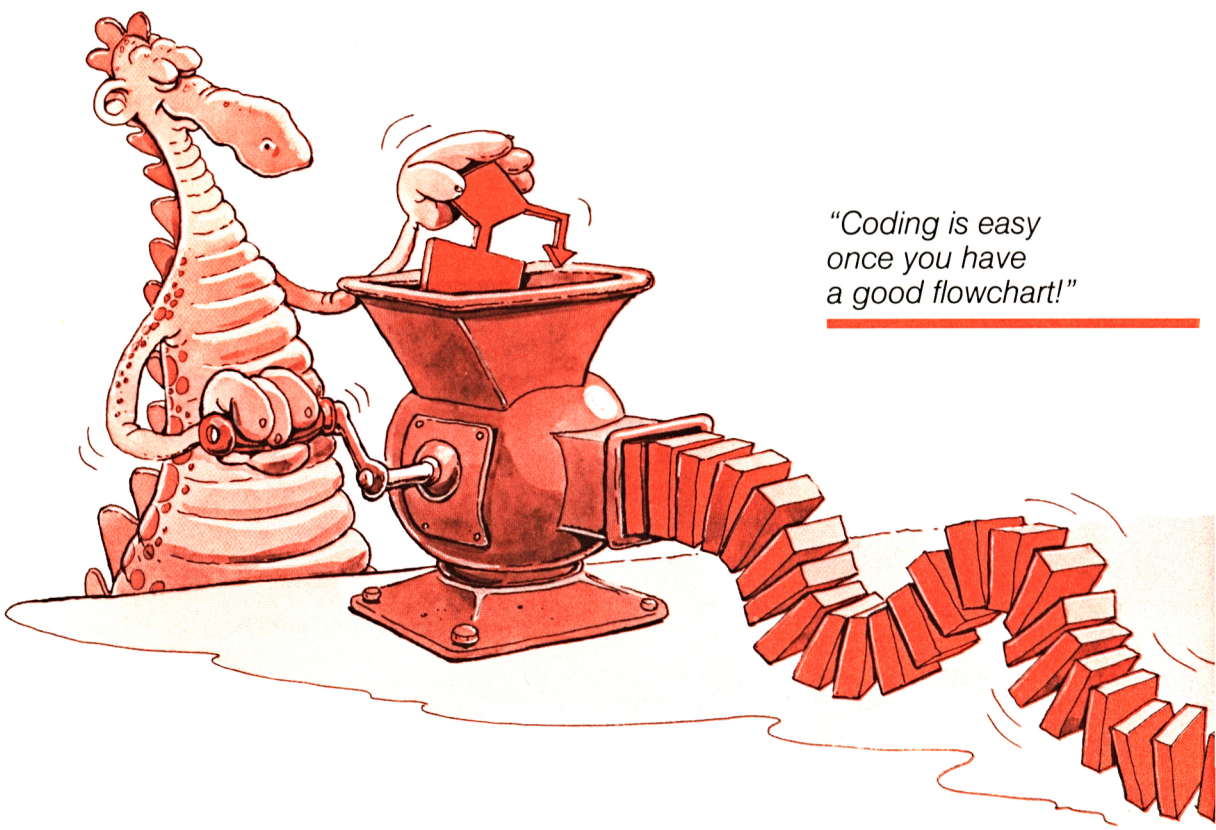
Writing program instructions is called *coding*. *Programming* normally refers to the entire sequence required to create a program: designing the algorithm, flowcharting, coding, debugging, and testing. Coding involves translating the contents of the flowchart boxes into program instructions expressed in a programming language—in this case, expressed in BASIC.

This is what we have learnt so far: we have learnt how to translate statements, formulas, tests, and conditions into BASIC instructions.

The key to easy coding is to write a detailed enough flowchart so that you can easily code each box of the flowchart into a few simple BASIC instructions. At the beginning, each box in your flowchart will be translated into just a few BASIC instructions, say one or two; i.e., there is a straightforward equivalence between boxes and instructions. Later as your programming skills and experience grow, you will be able to write "looser" flowcharts where each box is coded into many BASIC instructions.

Despite appearances, the coding phase is often the one that requires the least time in the program development sequence. Testing the program usually requires much longer than the actual coding. This is why it is so important to write a good flowchart—so that you can minimize errors and testing time.

When coding a program, remember to make your program accurate, clear, and readable, so that it will work quickly and can be tested or modified easily.



"Coding is easy once you have a good flowchart!"

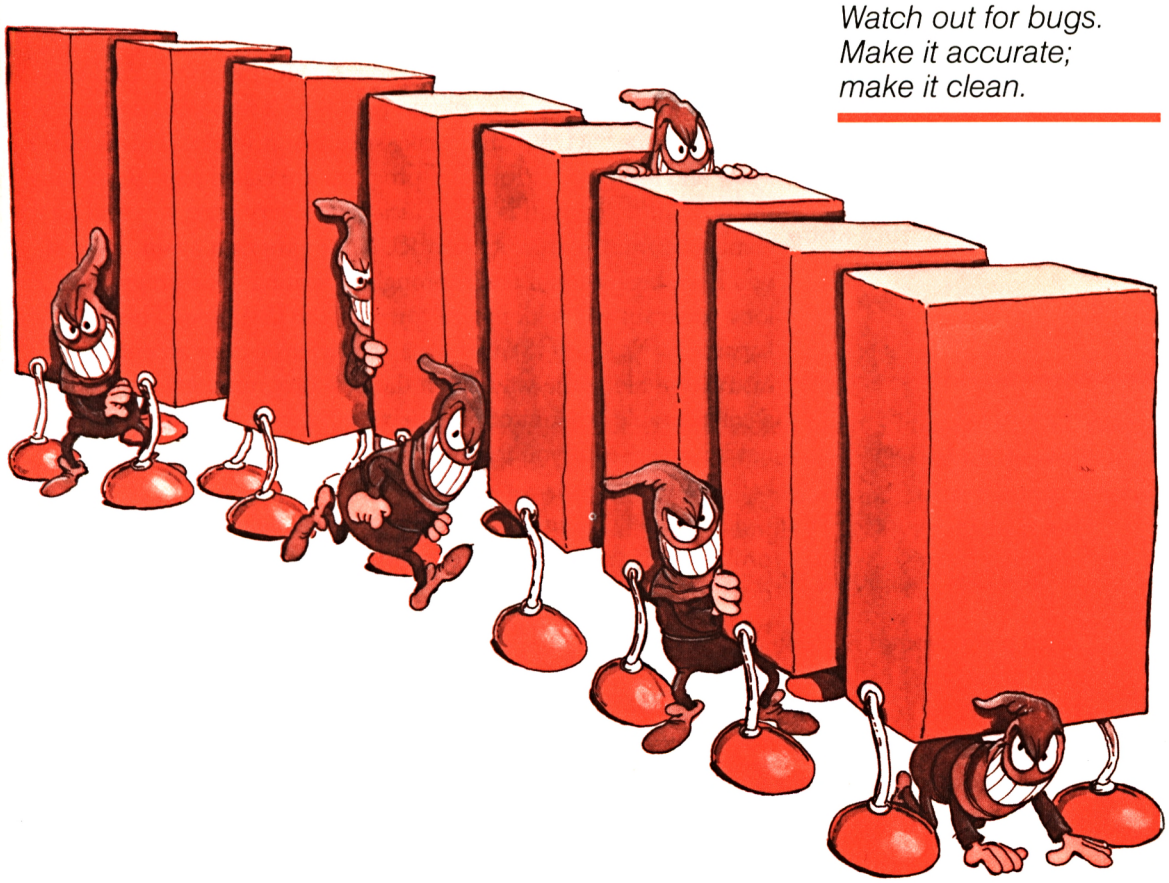
Make it Accurate

Write your program instructions with utmost care since any error in the placement of a punctuation sign or symbol will probably make the program fail.

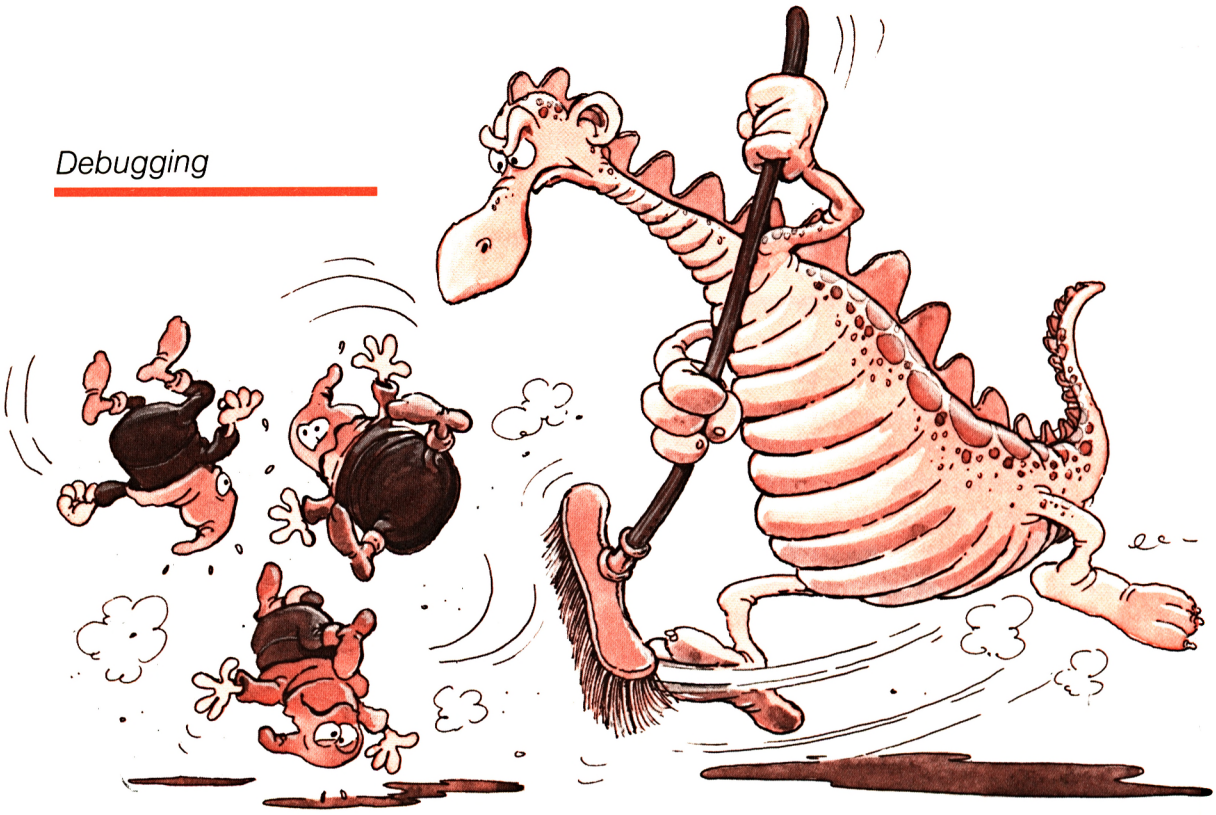
Make it Clear

Use variable names that are easy to remember. Leave intervals or even gaps in your sequence of instruction numbers in case you might have to insert other instructions in between. Use remarks (the REM statement) liberally throughout the program to clarify what it does.

So now you have designed an algorithm, drawn a flowchart and written the corresponding program. And you seriously expect it to work. Don't. Sorry. In the majority of cases, *programs do not work the first time.* It usually takes several attempts or lots of experience before a program of any length will run correctly. This is the topic of the next phase.



Watch out for bugs.
Make it accurate;
make it clean.



Debugging

You have coded your flowchart into a BASIC program. Your program is still on paper. You should now type it into your computer's memory, type "RUN," and make sure that it works. This is called *testing* and *debugging*. Errors in a program are called *bugs*. Removing the bugs in a program is called *debugging*. Every time you find an error, you must correct it, then RUN the program again. Even if you have been careful and thorough in writing your program, a long program will seldom run correctly the first time. This is because it is so easy to misplace a character or even an entire instruction along the way. Even the best programmers must spend a considerable amount of time debugging their programs. So do not be surprised if you must correct your program several times before it finally runs correctly.

Fortunately, your BASIC interpreter will help you diagnose some problems. If your program contains a type of error that can be detected by the interpreter, program execution will stop after you type RUN, and the interpreter will give you a diagnostic, such as "Syntax error in line 84." The interpreter will mostly help you detect *syntax errors* (the use of illegal symbols or operations). Unfortunately, it will not help you detect the most dangerous errors, the *logical* or *design errors*. That is your responsibility. This is why you should invest the time to carefully design your flowcharts. Also, this is why every time a number is supplied to a program or generated by it, you should validate that number by checking its range. In



The interpreter helps diagnose syntax errors

the event that your program contains a logical flaw, this technique will help you isolate the program section that contains the error.

Usually, in the case of a simple program, a few typographical errors will be detected by the BASIC interpreter. You will then correct them and your program will work. You should still make sure that it works correctly by testing it for various cases or values; your program might contain logical flaws. In most cases, however, you will determine that your program works correctly.

Practical Hints

Here is a practical hint: you should insert additional PRINT statements throughout your program to verify key values throughout. This will help you detect strange values and isolate the instructions that caused them. Here is a sample PRINT you might insert:

```
1235 PRINT "TEST FOR AVERAGE VALUE IS "; AVERAGE
```

Then, once the program works, you can remove these extra PRINT statements. This technique is called *tracing* a variable.

Another form of program tracing is offered by AMSTRAD BASIC. The TRON and TROFF commands allow you to turn on and off a trace of the line by line execution of a program. After you have typed:

```
TRON
```

as each statement in the program is executed its line number is printed. This shows exactly what a program is doing.

As another practical hint, any time that your program stops, either on its own or because you or the interpreter stopped it, you can use the *immediate execution mode* to verify the value of various variables in your program. For example, you might type:

```
PRINT AVERAGE
```

then:

```
PRINT SUM
```

in order to check the current values of these two variables.

The key to successful debugging is experience and a lot of prevention. Next time, spend more time on the program design and the flowchart, and you will spend less time debugging.

So your program now works correctly. You think it is correct; and you do not want to touch it any more. However, an error might be found later, or you might use the program again or share it with someone else. In order to make a program reusable, one more step is required: you should document the program while you remember what it does.

Documentation

You have just finished designing and coding a program. You are thoroughly familiar with its operation and what each instruction does. You will be surprised to discover how quickly you will forget what the program does and how difficult it may be to read or understand your own program later on. If you intend to reuse your program or to correct errors found at a future date, it is vital that you clarify the program promptly and completely. This means clarifying the program itself, as well as documenting everything that might require an explanation, on paper or as REMarks within the program.

Here is a summary of the techniques we have described for clarifying the program:

• **Use a clear layout**: Separate sections with blank lines or empty statements. Use alignments or indentations for clarity. You may want to use line numbers that all have the same length. This way all program statements will be aligned vertically. In addition, every time you use a FOR . . . NEXT statement, it is a good idea to indent the block of instructions that are enclosed between the FOR and the NEXT.

Also, use spaces liberally to clarify complex instructions, in particular those containing mathematical expressions. Use parentheses to clarify the results of a computation.

• **Explain what you do:** Use REM statements to explain formulas, tests, names, or conventions. It is also a good idea to provide a short written explanation for any methods or techniques you are using which are not obvious or are not described by REM statements within the program itself.

• **Clean up flowcharts:** Produce one clean flowchart or set of flowcharts that correspond exactly to your program. Often, during the debugging process, last minute changes are made directly to the program. Make sure they are reflected on the original flowchart or else you may find it very difficult to change or correct your program later on.

• **Renumber your lines:** Often, in the process of correcting the program, i.e., in the debugging phase, you will need to insert additional statements. Once you think your program is correct, it is a good idea to renumber the lines so that all line numbers are spaced regularly. This will facilitate later changes to the program.

Document your program.





Summary

In this chapter, we have described the five steps to a finished program: designing, flowcharting, coding, debugging, and documenting. Let us review them.

Each program requires designing an algorithm. The algorithm must be designed at least mentally if not on paper. The algorithm may be sketched as a series of formulas or notes describing the essential steps.

The next step is the flowchart that describes the complete sequence of events. Consider it as a mandatory step for any program that involves more than a few lines. Remember that, the more carefully you design your flowchart, the better the chance that your program will be correct.

The next step is coding. The flowchart is translated into BASIC instructions. Practice will speed up this phase considerably. In fact, the coding phase quickly becomes the shortest phase.

Then comes testing and debugging. This phase is always required and is often the longest phase. Each program must be carefully checked out.

Finally, the quality of the documentation will facilitate or impede the later use of, and changes to, the program.

Exercises

8-1: Describe the five phases of program development.

8-2: What is the difference between coding and programming?

8-3: What is the purpose of debugging?

8-4: How do you trace a variable?

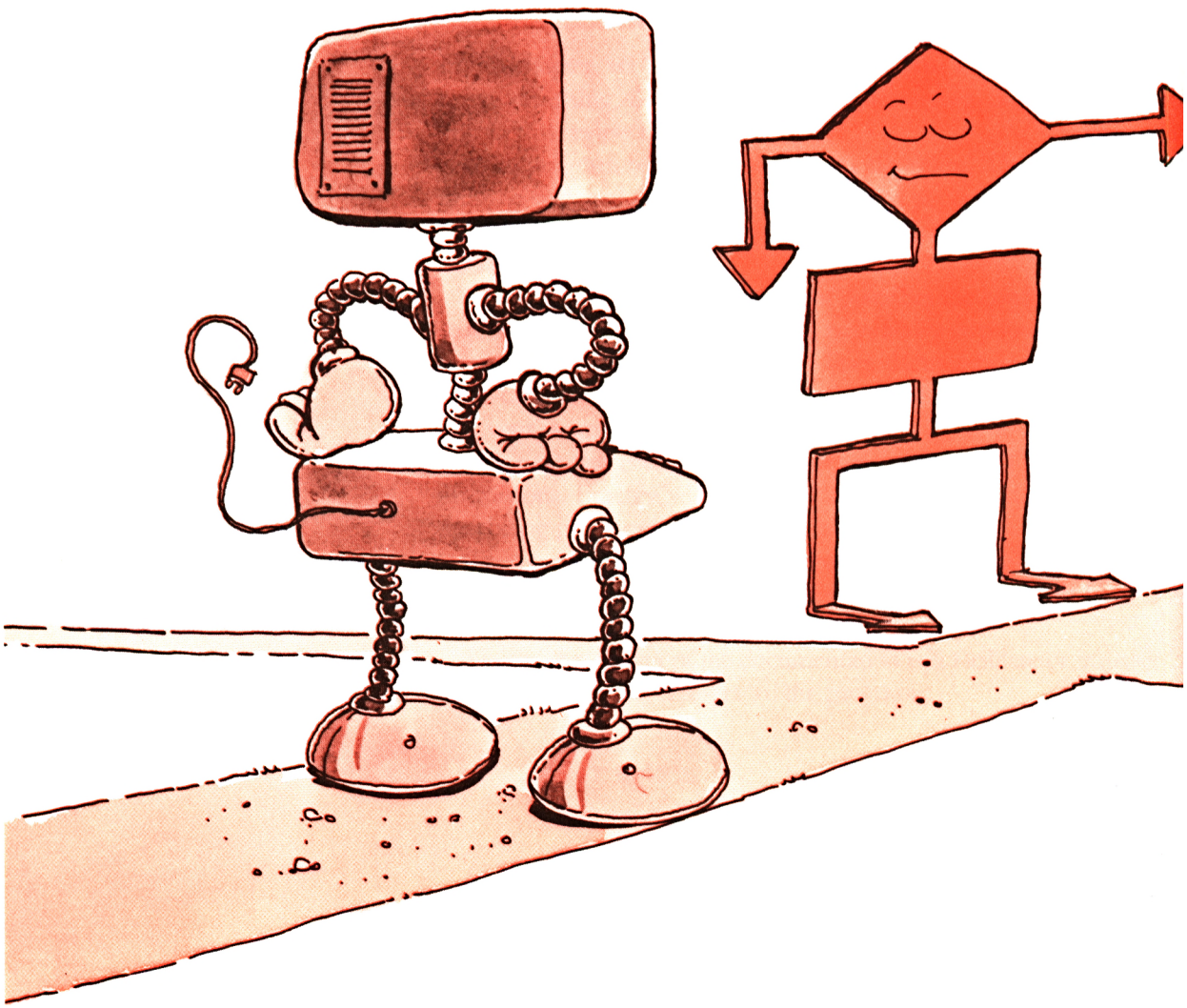
8-5: Why renumber a program after making many changes?

8-6: What is a flowchart?

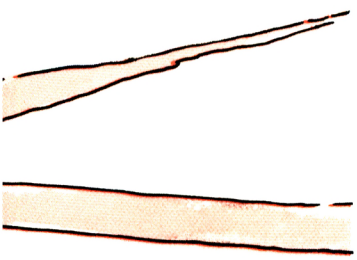
8-7: Write a flowchart to start your car or operate an appliance.

8-8: What are the advantages of a clearly written program?

8-9: Describe the techniques that can be used to clarify a program.



CASE STUDY: METRIC CONVERSION



9

We will now develop a complete program and describe each step in turn. Here is the problem to be solved:

We need to write a program that will automatically convert a weight expressed in ounces into its equivalent value in grams. The program should either convert a number typed at the keyboard or print a weight conversion table between two specified values.

Designing the Algorithm

The rough sequence of steps that we will follow to solve this problem is quite straightforward: We will ask the user what he or she wants (a single conversion or a table of values) and then perform the action requested. This is our rough algorithm. Let us refine it.

One ounce is worth 28.35 grams. The conversion from ounces to grams is therefore accomplished by the following formula:

$$W_{grams} = W_{ounces} \times 28.35$$

or, in short:

$$W_g = W_{oz} \times 28.35$$

Here is the basic algorithm:

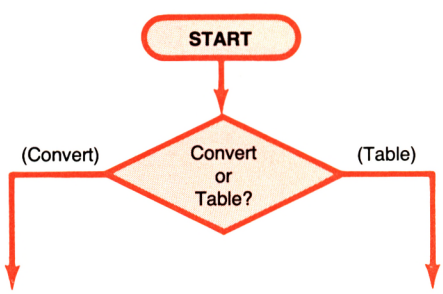
- Specify either single conversion or table
- If conversion, request weight in ounces
- Convert to grams (using the formula above) and display result
- END
- If table, request maximum weight in ounces
- Convert to grams and display results up to the limit
- END

In practice, there is no need to write out the algorithm, as long as you write the flowchart.

Flowcharting

In preparing the flowchart, we must first know whether the user wants the program to perform a single conversion or display a table of values. Here is the corresponding flowchart element:

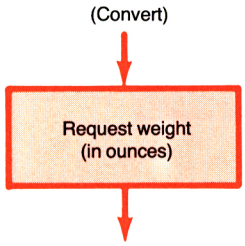




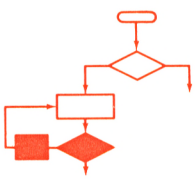
This is a decision box, with two possible outcomes (i.e., branches): CONVERT and TABLE.

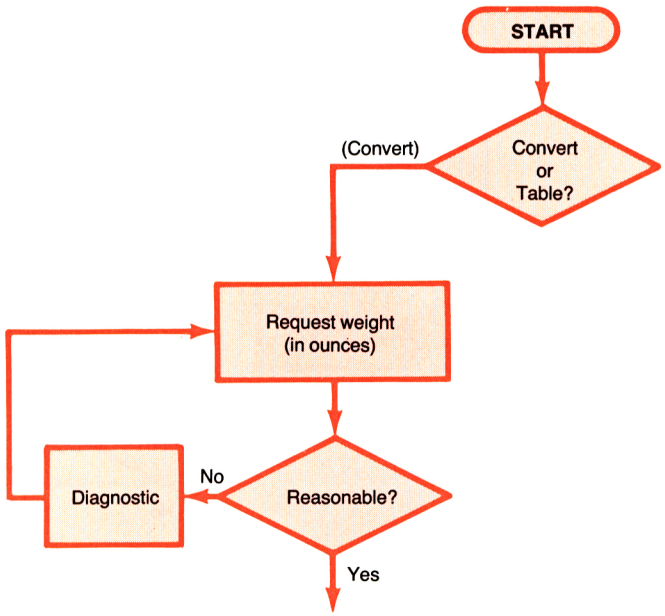
Let's first examine CONVERT: The user wants to convert a single weight from ounces to grams. We must request the value of the weight. Here is the corresponding flowchart entry:





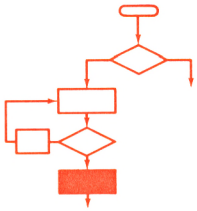
We could now convert this value into grams. However, let's refine the flowchart immediately. As a precaution, we will validate the value supplied by the user, by checking whether it is reasonable or not. Here is the way our flowchart looks with this validation added:

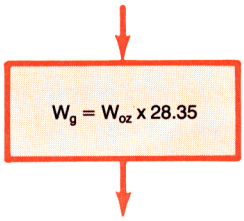




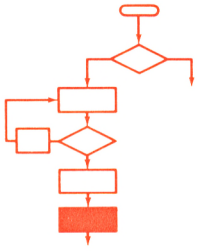
Note that we have added a box to check whether the input is reasonable. If the input is reasonable, we proceed. If not, a diagnostic, such as "UNREASONABLE INPUT, TRY AGAIN" is issued, and the program requests another value for the weight.

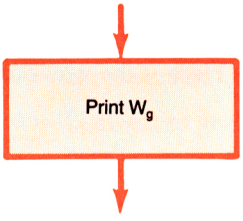
We now have a validated value for the weight. Let's convert it into grams. This is accomplished by the following:



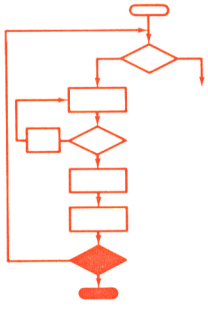


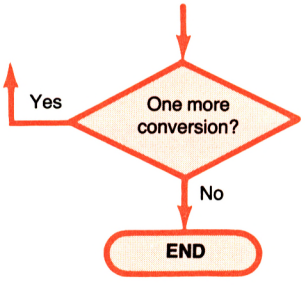
We can now display the results. This is accomplished by the following box:



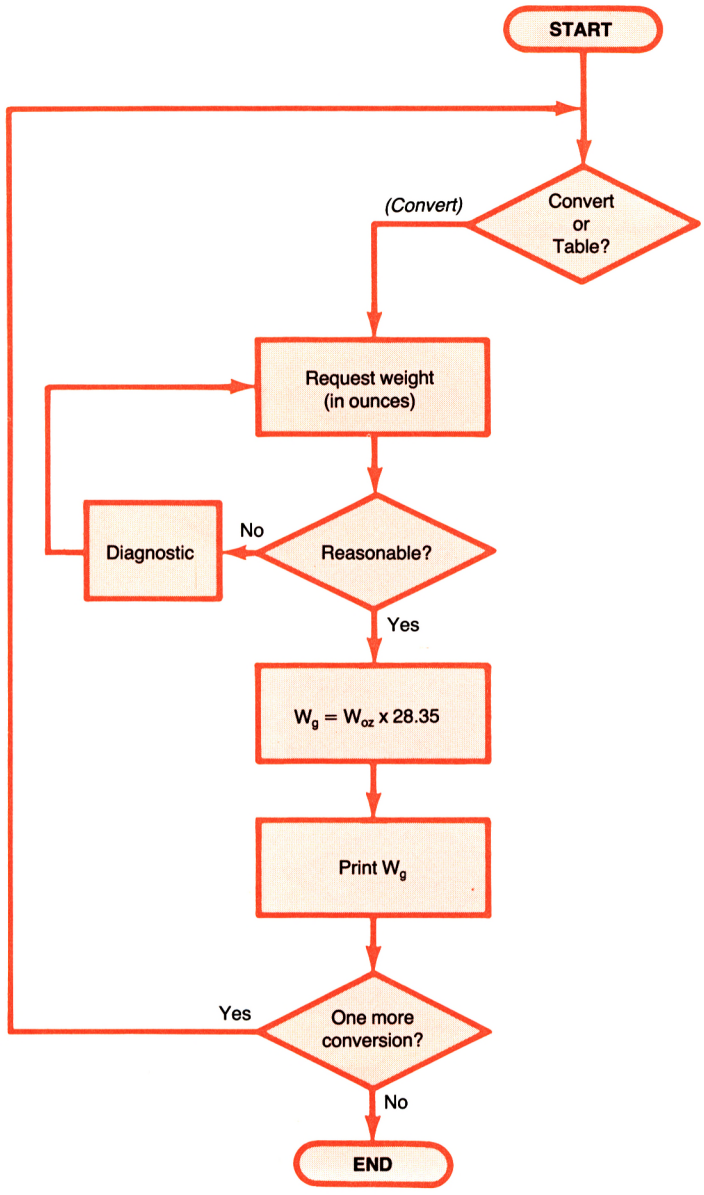


The single conversion is now done. We could end this part of the flowchart here. However, let's add a convenience feature and ask the user whether he or she wants to do another conversion. This is accomplished by the following box:



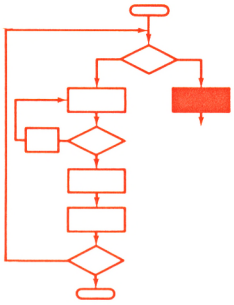


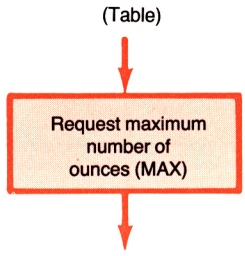
The START symbol on the left arrow indicates that this arrow will be connected back to the beginning of the flowchart. At this point, our flowchart looks like this:



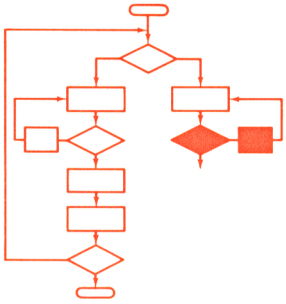
Let us now go back to our first decision box and examine what happens when the user wants to display a table of values. This is the TABLE option at the top of the flowchart.

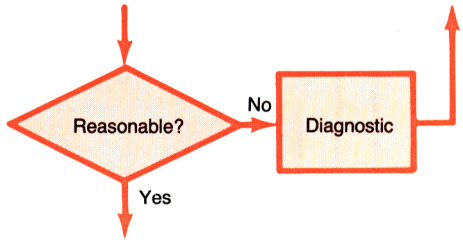
We must know the maximum value to be converted. This is accomplished by the following box:





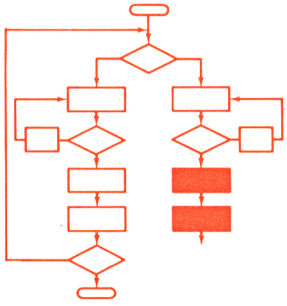
Again, for safety, we will check whether this number is reasonable:

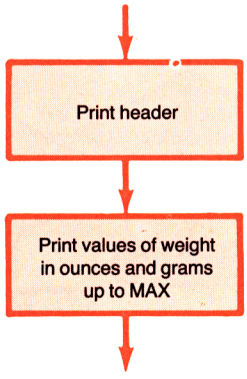




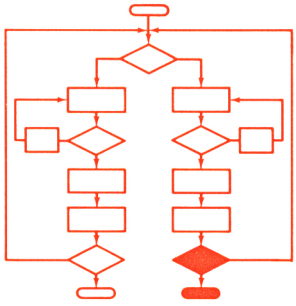
As before, the program will not proceed until a reasonable value of MAX is supplied by the user.

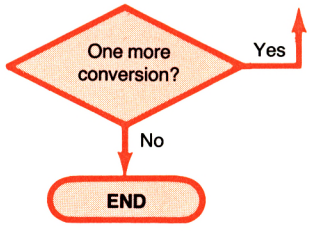
Once a reasonable value is supplied, we can proceed and display a table that converts ounces to grams, up to the desired limit:





Finally, let's add the same convenience feature that we used in the single conversion case and ask the user whether he or she would like to perform one more conversion:





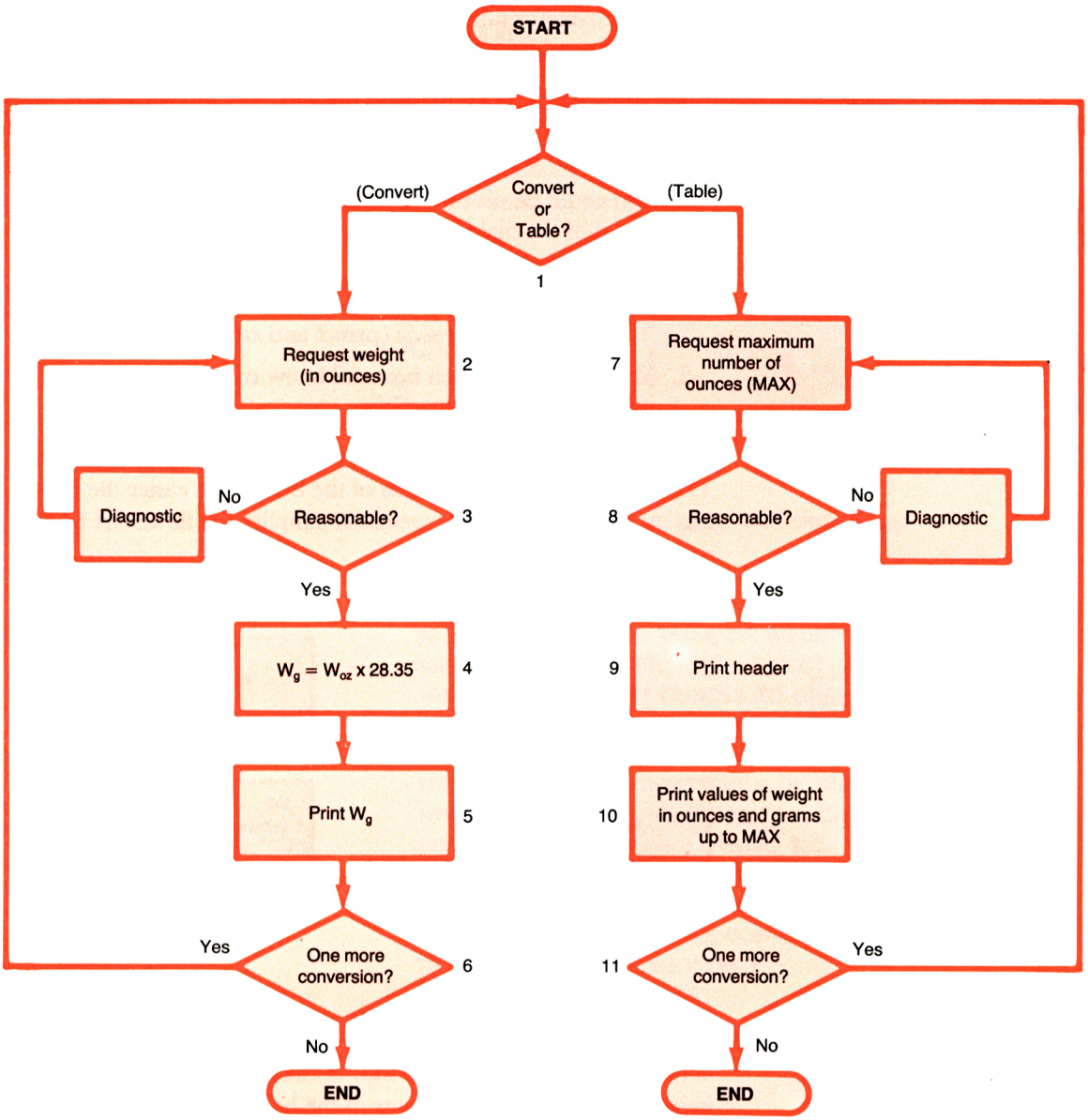


Figure 9.1: Weight Conversion Flowchart

Figure 9.1 shows the complete flowchart. This flowchart is typical. The contents of the boxes are somewhat "loosely" written. Some boxes in the flowchart will be coded into just one or two BASIC instructions, while others will require many more. However, it is the sequence that counts. The fact that some boxes may be more detailed than others does not matter. Remember that the flowchart sequence must be exact, but the details may be written in any manner that is convenient for you. There is no need to spend a lot of time optimizing the contents of the boxes, as long as you feel you can code from it in a straightforward manner.

The flowchart must simply be clear and easy to read. A clear, well-organized flowchart improves your chances of writing a correct program.

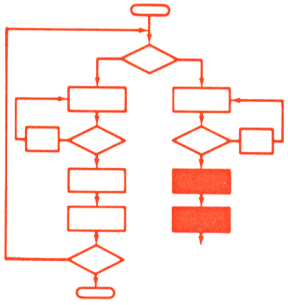
For completeness, here are some refinements or alternatives you might consider:

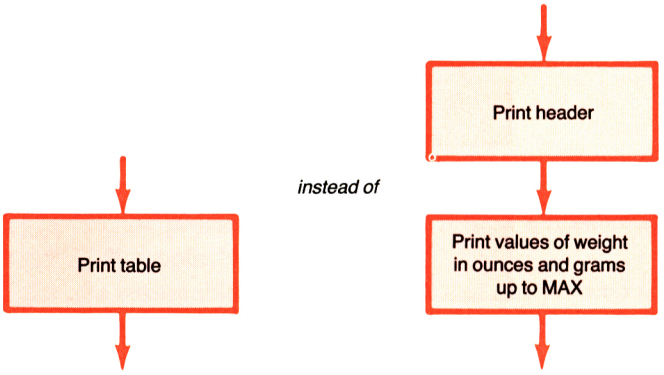
- You could explain in detail the way you will check the weight for "reasonableness." (Here we will simply check that WOZ is a positive number.)
- You could be more explicit about the diagnostic and about much of the dialogue.

In general, the advice is: *keep it simple.* Do just enough so that:

1. The sequence of steps is correct and complete.
2. You understand each box and know roughly how to convert it into program instructions.

The simpler the contents of the boxes, the clearer the flowchart. The more detailed the contents of the boxes, the easier the coding. Applying this advice, we could simplify the flowchart by writing:





Both options are correct. Use the one you feel most comfortable with. Here, I decided to hint at the programming steps involved and make the contents of the box more explicit.

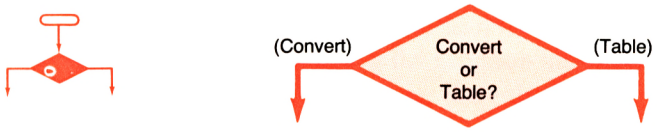
You can always rewrite the contents of a box, or for that matter any part of your flowchart, on a separate piece of paper to facilitate the coding, or to try out an alternative.

Now that we have a flowchart, let's try it out by hand with actual numbers, to make sure it works. This hand-checking process is obvious, so let's proceed.

Coding

We will now write a program that corresponds to our flowchart. Let's now code each box of the flowchart into the corresponding BASIC instructions.

Here is box 1 of the flowchart:

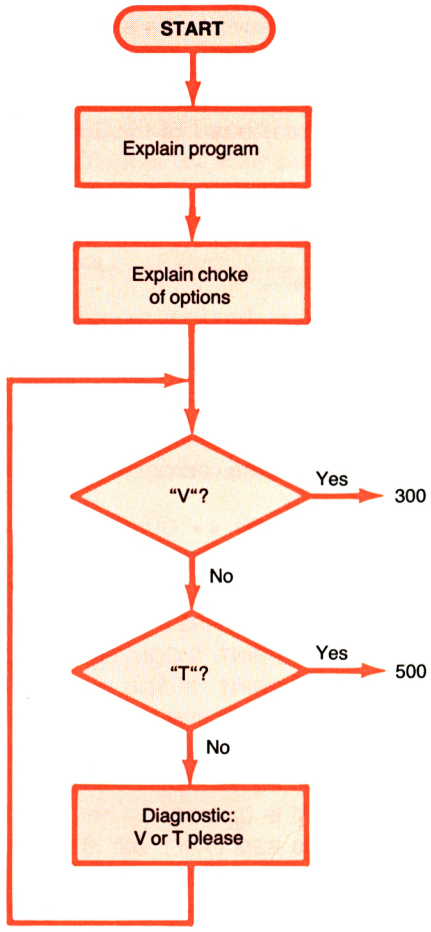


Here are the corresponding program instructions:

```
100 REM ** OUNCES TO GRAMS CONVERSION **
110 REM THIS PROGRAM CONVERTS DIRECTLY
120 REM OR PRINTS A TABLE OF VALUES
130 REM FIRST, SPECIFY MODE: DIRECT OR TABLE
140 PRINT "I CONVERT OUNCES TO GRAMS"
150 PRINT "IF YOU WANT TO CONVERT ";
160 PRINT "VALUES DIRECTLY, TYPE V"
170 PRINT "IF YOU WANT A TABLE OF VALUES, TYPE T"
180 INPUT "YOUR CHOICE (V OR T): ", CHOICE$
190 IF CHOICE$ = "V" THEN GOTO 300
200 REM LABEL 300 IS VALUE CONVERSION
210 IF CHOICE$ = "T" THEN GOTO 500
220 REM LABEL 500 IS TABLE CONVERSION
230 REM INPUT MUST BE V OR T
240 PRINT "V OR T PLEASE"
250 GOTO 180
```

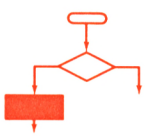
As you become more experienced, you may be able to go directly from box 1 of the flowchart to the corresponding program instructions. However, at the beginning, you will need to write down the detailed version of the flowchart first.

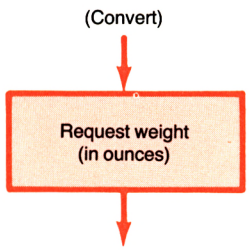
Here is the detailed version of box 1:



Looking at this detailed version, note how closely the flowchart corresponds with the BASIC instructions. In addition, note that we have introduced a *validity test* for CHOICE$. We aren't going to simply assume that the user will cooperate and type "V" or "T". Check it. **Remember**: every time you request an input, you should validate it.

The rest is easier. Let's convert box 2:





The corresponding instructions are:

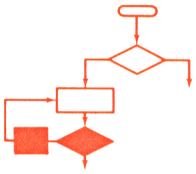
```
300 REM *** VALUE CONVERSION ***
310 PRINT "TYPE THE WEIGHT IN OUNCES. . .";
320 INPUT WOZ
```

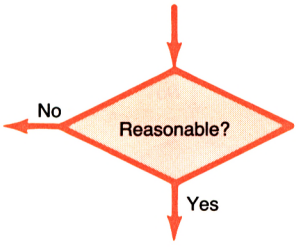
As an aside, we could also write:

```
310 INPUT "TYPE THE WEIGHT IN OUNCES. . .";  WOZ
```

Here, I decided to separate the PRINT and INPUT instructions; thus making it easier to show how the program lines correspond with the flowchart. You may, however, use either form.

Let's proceed. Here is box 3:





Here is its program equivalent:

```
330 IF WOZ < 0 THEN GOTO 310
340 REM WEIGHT MUST BE POSITIVE
350 REM ERROR MESSAGE COULD BE PUT HERE
```

The equivalent of box 4 is:

```
360 WG = WOZ * 28.35
```

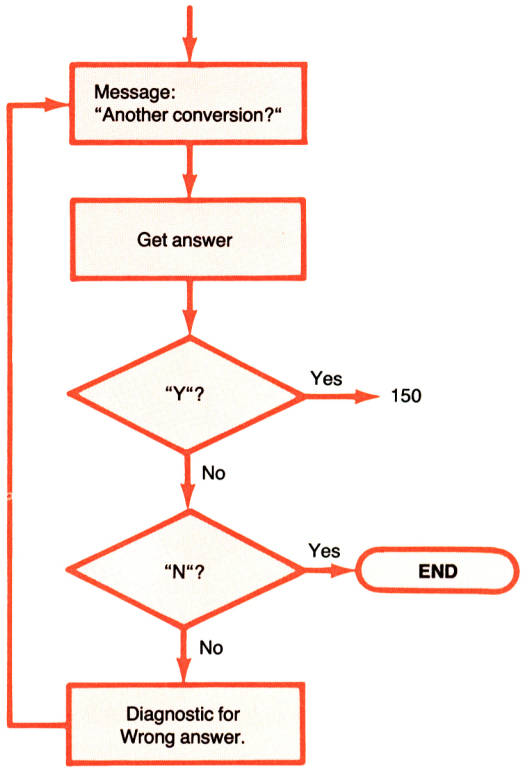
And for box 5:

```
370 PRINT WOZ; "OUNCES ARE EQUAL TO"; WG; "GRAMS"
```

And for box 6:

```
410 PRINT "ANOTHER CONVERSION? Y FOR YES, N FOR NO"
420 INPUT AGAIN$
430 IF AGAIN$ = "Y" THEN GOTO 150
440 IF AGAIN$ = "N" THEN END
450 REM INPUT WAS NOT Y OR N. TELL USER
460 PRINT "Y OR N PLEASE"
470 GOTO 390
```

If the above is not clear to you, here is the equivalent detailed flowchart:



Let's now look at the right part of the flowchart in Figure 9.1. Here is the equivalent of box 7:

```
520 PRINT "I WILL DISPLAY A CONVERSION"
530 PRINT "TABLE OF OUNCES TO GRAMS"
540 INPUT "MAXIMUM NUMBER OF OUNCES: ", MAXIMUM
```

And for box 8:

```
560 REM CHECK MAX FOR 1 OR MORE
570 IF MAXIMUM < 1 THEN GOTO 540
```

For clarity, let's skip a line on the display, before we print the table:

```
580 PRINT
```

Remember: that's an empty PRINT. It displays a blank line. Now here is the equivalent of box 9:

```
590 PRINT "OUNCES", "GRAMS"
```

Note that we use a comma rather than a semicolon for a pleasant spacing of the columns. We also use leading spaces to align the column titles.

And here is box 10:

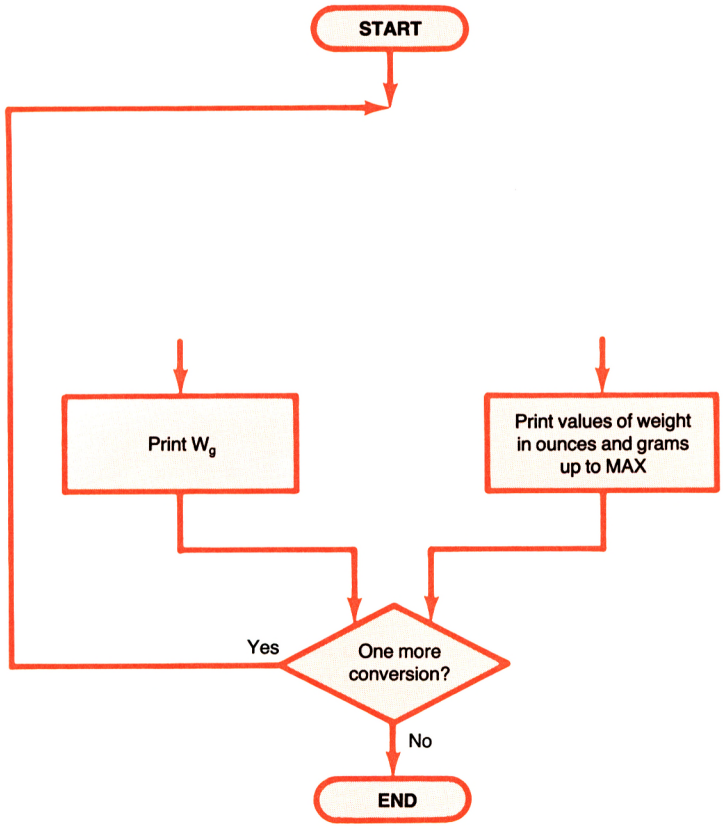
```
600 I = 1
610 PRINT I, I * 28.35
620 I = I + 1
630 IF I <= MAX GOTO 610
```

Finally, here is box 11:

```
650 GOTO 390
```

Box 11 is the same as box 6, so we can simplify our program by jumping (GOTO) to the instructions for box 6.

Here is the corresponding (corrected) flowchart:



The complete program is shown in Figure 9.2.

```
100 REM ** OUNCES TO GRAMS CONVERSION **
110 REM THIS PROGRAM CONVERTS DIRECTY
120 REM OR PRINTS A TABLE OF VALUES
130 REM FIRST, SPECIFY MODE: DIRECT OR TABLE
140 PRINT "I CONVERT OUNCES TO GRAMS"
150 PRINT "IF YOU WANT TO CONVERT ";
160 PRINT "VALUES DIRECTLY, TYPE V"
170 PRINT "IF YOU WANT A TABLE OF VALUES, TYPE T"
180 INPUT "YOUR CHOICE (V OR T): ", CHOICE$
190 IF CHOICE$ = "V" THEN GOTO 300
200 REM LABEL 300 IS VALUE CONVERSION
210 IF CHOICE$ = "T" THEN GOTO 500
220 REM LABEL 500 IS TABLE CONVERSION
230 REM INPUT MUST BE V OR T
240 PRINT "V OR T PLEASE"
250 GOTO 180
290 REM
300 REM *** VALUE CONVERSION ***
310 PRINT "TYPE THE WEIGHT IN OUNCES. . .";
320 INPUT WOZ
330 IF WOZ < 0 THEN GOTO 310
340 REM WEIGHT MUST BE POSITIVE
350 REM ERROR MESSAGE COULD BE PUT HERE
360 WG = WOZ * 28.35
370 PRINT WOZ; "OUNCES ARE EQUAL TO"; WG; "GRAMS"
380 REM
390 REM *** EXIT MODULE ***
400 PRINT
410 PRINT "ANOTHER CONVERSION? Y FOR YES, N FOR NO"
420 INPUT AGAIN$
430 IF AGAIN$ = "Y" THEN GOTO 150
440 IF AGAIN$ = "N" THEN END
450 REM INPUT WAS NOT Y OR N. TELL USER
460 PRINT "Y OR N PLEASE"
470 GOTO 390
490 REM
500 REM *** TABLE CONVERSION ***
510 REM REQUEST UPPER LIMIT
520 PRINT "I WILL DISPLAY A CONVERSION TABLE"
530 PRINT "OF OUNCES TO GRAMS"
540 INPUT "MAXIMUM NUMBER OF OUNCES: ", MAXIMUM
560 REM CHECK MAX FOR 1 OR MORE
570 IF MAXIMUM < 1 THEN GOTO 540
580 PRINT
590 PRINT "OUNCES", "GRAMS"
600 I = 1
610 PRINT I, I * 28.35
620 I = I + 1
630 IF I <= MAX GOTO 610
640 REM I IS NOW > MAX. THIS IS END OF TABLE
650 GOTO 390
660 END
```

Figure 9.2: Weight Conversion Program

Some improvements have been added. For example, REMs have been added for clarity (see instructions 290, 300, 380, 390, 490, 500).

Other improvements could be made. For example, the instructions 600 to 630 could be replaced with a FOR . . . NEXT statement. This might improve readability, but it is hardly worth the trouble.

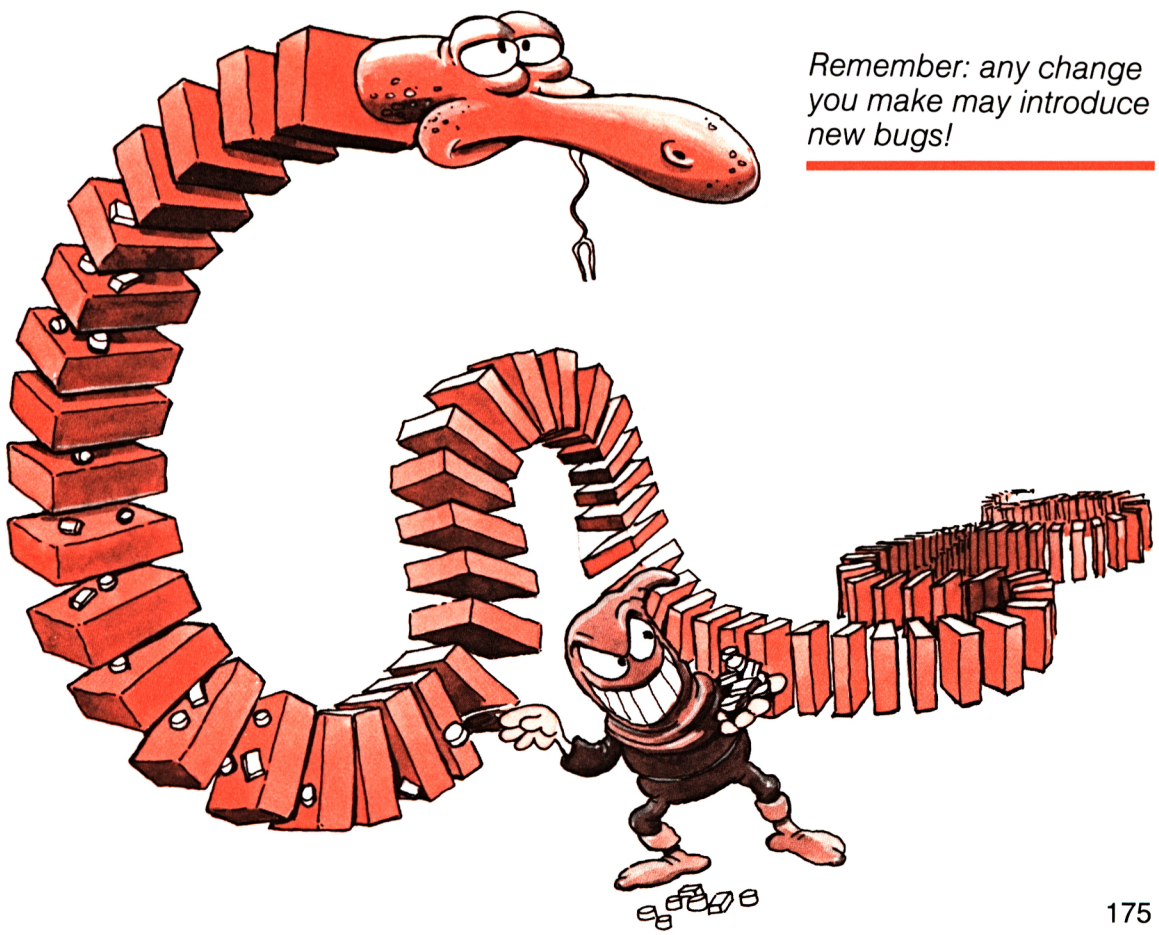
Remember any change you make to a working program may introduce new bugs. Change your program only if there is a clear benefit.

We now have a complete program. Let's try it out.

Testing

Let's RUN the program. Here is a sample run:

```
RUN
I CONVERT OUNCES TO GRAMS
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): V
TYPE THE WEIGHT IN OUNCES. . .? 3
3 OUNCES ARE EQUAL TO 85.05 GRAMS
```



Remember: any change you make may introduce new bugs!

Here is another one:

```
ANOTHER CONVERSION? Y FOR YES, N FOR NO
? Y
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): T
I WILL DISPLAY A CONVERSION TABLE
OF OUNCES TO GRAMS
MAXIMUM NUMBER OF OUNCES: 4

OUNCES      GRAMS
1           28.35
2           56.7
3           85.05
4           113.4
```

Our program seems to work for a single conversion as well as a table.

Let's try to fool it:

```
ANOTHER CONVERSION? Y FOR YES, N FOR NO
? Y
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): D
V OR T PLEASE
YOUR CHOICE (V OR T): V
TYPE THE WEIGHT IN OUNCES. . .? 7
7 OUNCES ARE EQUAL TO 198.45 GRAMS
```

Let's try the repeat option:

```
ANOTHER CONVERSION? Y FOR YES, N FOR NO
? Y
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): V
TYPE THE WEIGHT IN OUNCES. . .? 85
85 OUNCES ARE EQUAL TO 2409.75 GRAMS

ANOTHER CONVERSION? Y FOR YES, N FOR NO
? Y
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): V
TYPE THE WEIGHT IN OUNCES. . .? 2.5
2.5 OUNCES ARE EQUAL TO 70.875 GRAMS

ANOTHER CONVERSION? Y FOR YES, N FOR NO
? Y
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): V
TYPE THE WEIGHT IN OUNCES. . .? 3.1
3.1 OUNCES ARE EQUAL TO 87.885 GRAMS
```

Let's try and fool it again:

```
ANOTHER CONVERSION? Y FOR YES, N FOR NO
? Y
IF YOU WANT TO CONVERT VALUES DIRECTLY, TYPE V
IF YOU WANT A TABLE OF VALUES, TYPE T
YOUR CHOICE (V OR T): V
TYPE THE WEIGHT IN OUNCES. . .? -5
TYPE THE WEIGHT IN OUNCES. . .? □
```

Well it seems to work. But you should really try it several more times before you are completely satisfied with it.

In this case, we have been very careful—and very lucky. Our program worked right the first time.



Summary

In this chapter we have illustrated the complete sequence involved in writing a program that solves a given problem. You should now close this book, write your own flowchart, and convert it into a working program.

The key to successful programming is practice.

Exercises

9-1: Add to this program the option of converting grams to ounces.

9-2: Expand the program to include distance conversion.

1 meter = 39.37079 inches.

1 km = 0.62138 mile.

1 inch = 25.3995 mm

1 foot = 30.479 cm

1 yard = 0.91438 m

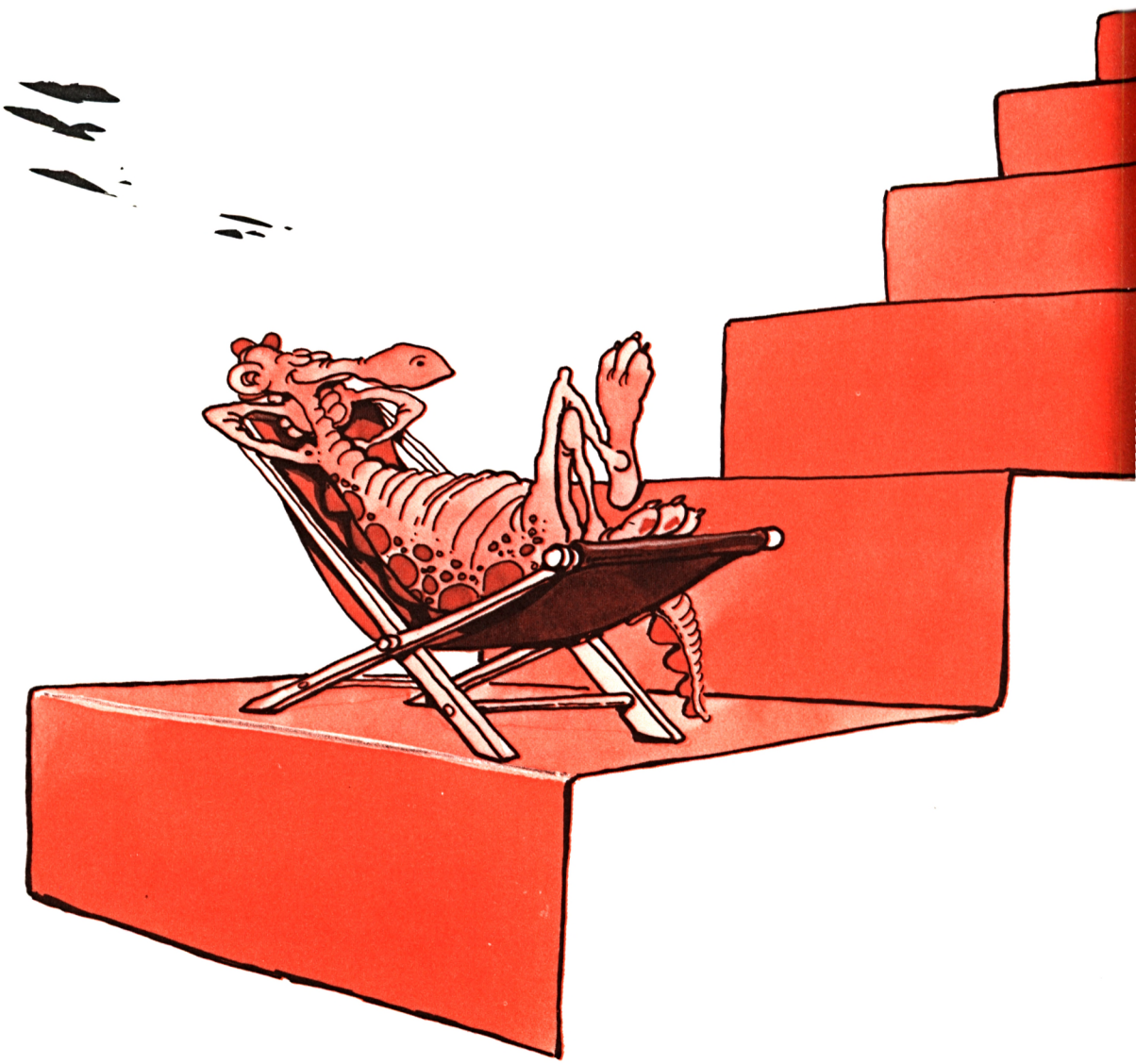
1 mile = 1609.3149 m

9-3: Expand the program to include temperature conversion.

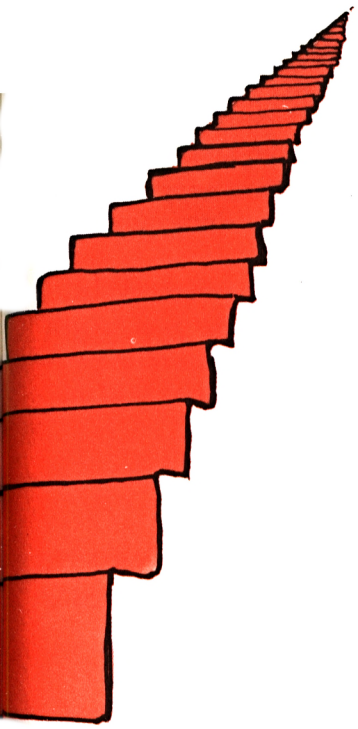
$C = (F - 32) \times 5/9$

$F = (9/5) \times C - 32$

9-4: Suggest additional ways to improve or clarify our final program.



The Next Step



10

You have now learnt how to write your own BASIC programs. In this chapter, we will examine the next step you can take towards improving your programming skills. We will review what you can do with BASIC, then describe the additional skills and techniques that can help you more easily write complex programs.

What You Can Do With BASIC

You can write a BASIC program to automate most tasks, unless they require extremely precise mathematical computations, complex decision-making, or a very fast response (such as real-time process control). You will find that BASIC is well-suited to simple business applications, such as data processing, mailing lists, and common financial computations. With extensions to the language, AMSTRAD BASIC also lends itself well to graphics and games.

Other typical application areas for BASIC include computer-assisted education, personal and business record-keeping, mathematical and technical computations, and many more. Applications are generally limited mostly by one's programming skills.

With the knowledge you have acquired so far, you should be able to write a wide variety of BASIC programs. However, as you progress, you will want to improve your skills and use more powerful and convenient programming tools. That is the topic of the next sections.

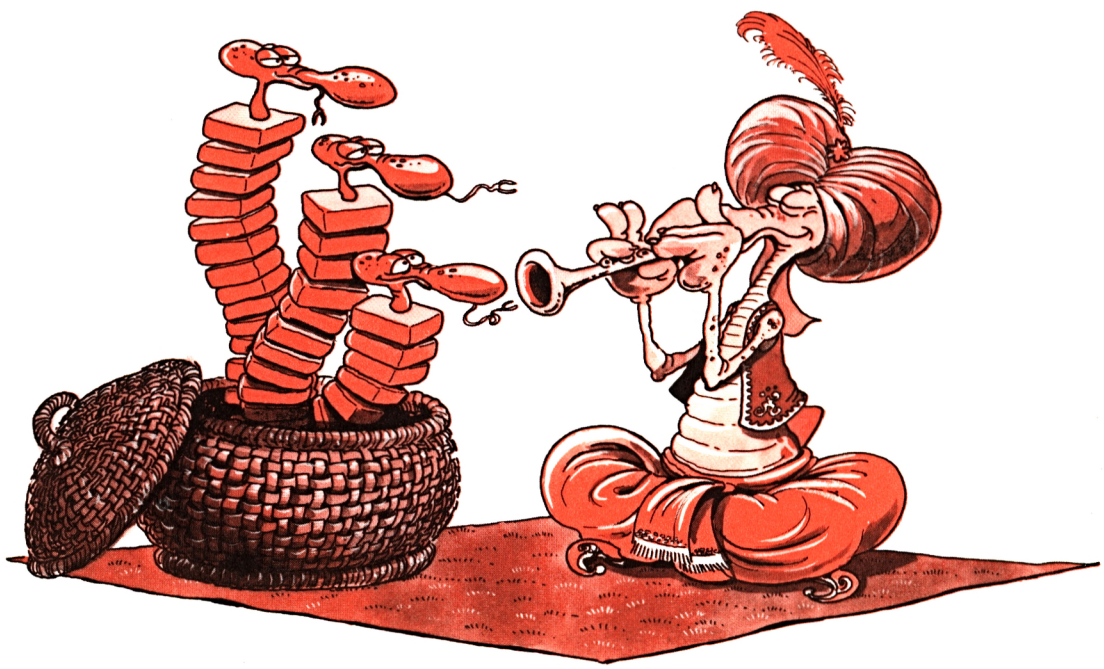
Improving Your Skills

You can take three essential steps towards increasing your skills in BASIC:

1. gain more practice
2. obtain a better knowledge of the complete resources of your AMSTRAD BASIC
3. learn additional programming techniques.

Let us review each step in turn.

Improve your skills





More Practice

The key to programming effectiveness is practice: write as many programs as possible, and get them to work. Develop good programming habits by following all the recommendations presented in this book. If your programs work consistently after just a few tries, you may be well on your way to becoming a disciplined and effective programmer. If they do not, watch your habits, or perhaps read parts of this book again, and then practice some more. Remember: there is no way you can become a good programmer unless you write many programs. This book will get you started, but it can never be a substitute for actual experience.

Practice more

Specific BASIC Features

Your AMSTRAD BASIC interpreter provides specific capabilities, including statements, commands, shorthand facilities, extensions to "standard BASIC" (such as graphics and sound), and an operating environment (including cassette storage commands, file facilities, an editor program, and more). You can enhance your programming ability by learning these additional features and facilities. In the next section we will describe the additional statements found in AMSTRAD BASIC. Facilities such as graphics, sound, and files are specific to your computer and your interpreter. You will gain by learning about them.

Additional Techniques

Once you develop longer programs (say, longer than one page), you will want to learn the usual techniques for solving common problems, such as ordering items, sorting them, formatting data, filing, and creating data structures. These topics are covered in programming books.

More BASIC

Your AMSTRAD BASIC interpreter offers a fairly "standardized" set of facilities, plus many extensions. The facilities offered by AMSTRAD BASIC include everything we have studied so far, plus six types of statements. We will present a brief overview of

each of these additional types. You will want to study them on your own or with a more advanced book. They are:

1. **Functions:** Functions are either *built-in* or *user-defined* expressions that operate on a given variable and perform a specific computation or action. AMSTRAD BASIC provides many built-in functions (such as ABS, COS, EXP, INT, RND, SGN, SIN, SQR, or TAN). These functions will automatically perform common tasks.



Study more

Additional functions can be defined by the user. Let's examine a few of the typical built-in functions:

- The INT function computes the integer part of a decimal number by dropping the fractional part of the number. For example, INT(1.234) yields the value 1.
- Similarly, ABS computes the absolute value. For example, ABS(−5.2) is 5.2.
- The SQR function computes the square root of a number. For example, SQR(4) yields the value 2.

Functions may also be defined by the user. A user-defined function is essentially a formula written by the user that has a name, operates on a variable, and can be used in the program many times. Then every time the name of the function is used, the formula is computed with the current value of the variable. This is very convenient shorthand notation.

Here is an example of a user-defined function that computes 2% of X:

```
10 DEF FNA(X) = X * 2 / 100
```

And here is one way of using this function:

```
20 PRINT FNA(50)
```

This will print the value 1.

Let's look at these statements more closely. FN means function. FNA is function A, i.e., the name of the function. X is the "dummy" variable. X does not have a value when the DEFinition is written. The actual value or variable is substituted for X at the time the function is used, for example by writing FNA(50).

2. **Subroutines:** A subroutine is a group of instructions within a BASIC program that can be used repeatedly. Every time the subroutine is called (using the GOSUB statement) in the body of the program, all the instructions within the subroutine are executed. Subroutines are convenient for executing a program segment repeatedly, without having to repeat the instructions in the program every time they are required.

Here is an example of a subroutine:

```
500 REM THIS IS THE SUBROUTINE AVERAGE
510 PRINT "I WILL COMPUTE THE AVERAGE OF A AND B"
520 PRINT "A = "; A, "B = "; B
530 AVERAGE = (A + B) / 2
54O PRINT "THE AVERAGE IS "; AVERAGE
550 RETURN
```

This subroutine can be called by an instruction like:

```
50 GOSUB 500
```

It can then be used again later in the program with an instruction such as:

```
170 GOSUB 500
```

Using subroutines will clarify your program, shorten it, and save you time. You can also build a library of common subroutines and use them in various programs. However, be careful with variable names and statement numbers.

3. **String Operators:** String operators allow you to manipulate text conveniently by operating on strings of characters. Operations on strings may include modifying them, measuring them, connecting them together, cutting them, inserting text to the left, the right, or at any given position, substituting characters, or comparing strings of characters. This is useful for word and text processing applications.

4. **Data Structures** or **Arrays:** AMSTRAD BASIC allows subscripted variables with several subscripts. These variables correspond to mathematical vectors and matrices, or arrays. The use of subscripted variables allows you to set up structures like lists, and then conveniently refer to any element within the list. For example, the elements of a list named CLIENT can be referred to as: CLIENT(1), CLIENT(2), etc.; and you can print all ten values with:

```
50 FOR N = 1 TO 10
60 PRINT CLIENT(N)
70 NEXT N
```

You can compare two consecutive elements by writing:

```
100 IF CLIENT(K) < CLIENT(K + 1) THEN 500
```

This is a great convenience. The number of subscripts is limited by the memory size of your AMSTRAD, not by the BASIC interpreter. For example, your clients may be grouped by borough within county. Where the boroughs and counties are numbered, you can display a particular client's name with the statement:

```
220 PRINT CLIENT(5,2,12)
```

5. **Files:** AMSTRAD BASIC provides facilities for storing and/or retrieving data from cassette files. These facilities are specific to your computer and your interpreter and are described in the manufacturer's documentation.

6. **Additional Resources:** Additional resources that are available with the AMSTRAD BASIC interpreter include the following statements: ON . . . GOTO, ON . . . GOSUB, READ . . . DATA,

and PRINT USING. You should explore these facilities with your interpreter's manual or a book, as they will make programming easier.

Also, shorthand facilities are available; for example, you are allowed to type:

```
?25 * 32
```

thus providing a convenient feature for using your computer as a pocket calculator.

Finally, remember that there are commands to generate graphics, produce sound effects and facilitate editing—for example, there are commands that allow you to modify your program and the files it creates—as well as aid in the running and debugging of a program. Such facilities include screen control, and tracing the instructions that are executed during execution.

Conclusion

This purpose of this book has been to teach you quickly and effectively how to write your first AMSTRAD BASIC programs. If you have become truly interested in BASIC, you will now want more. Your next step should be to work through all the exercises and develop some of your own programs.

As you progress, you will want to learn more advanced techniques. A list of programming books is presented at the end of this book.

I hope that you agree that learning BASIC can be easy and enjoyable, and I hope that your desire now is to do and learn more. I would appreciate hearing from you if you have any suggestions for possible improvements to this book.

Appendix A

Answers to Selected Exercises

2

2-2:

```
10 PRINT "AAAAA"
20 PRINT "BBBB"
30 PRINT "CCC"
40 PRINT "DD"
50 PRINT "E"
```

2-4: **a.** In BASIC, a label is a line number, and it must precede each statement that is part of a program.

b. The deferred execution mode is the normal mode in which a program is entered. BASIC statements are typed with line numbers, and memorized by the computer for later execution.

c. Direct execution is the mode in which a statement is typed without a line number and executed immediately. This is also called immediate execution or calculator mode.

d. An empty statement is a statement that does nothing. Typically it is a line number or label that appears alone—without anything else on the same line —or with "REM."

e. A cursor is a special visual symbol on the screen (a square) that shows your current position.

f. The control or CTRL key, when pressed at the same time as an alphabetic key, will issue a specific command. Control keys make it easier to issue frequently-used commands to the computer, since only two keys have to be pressed.

g. A keyword is a name that has a specific meaning to BASIC. It may not be used as the beginning of a variable name.

h. A prompt is a character or message generated by a program that indicates that the program expects the user to enter informa-

tion. AMSTRAD BASIC uses the prompt "Ready", to mean "type your next instruction"; the prompt "?" means "give me an input".

2-6: Yes, but this seems a cumbersome way to do it, since if you desire to run the program again, you must type in the statements again. Also, conditional branches (to be covered later) will not work properly when typed in this way.

2-8: Yes, BASIC will insert each statement in its proper place within the program so that all the labels are in numeric order.

2-10: No: you must type:

```
PRINT "EXAMPLE"
```

2-12: To erase statement 20 in a program, type an empty statement with the label 20.

3

3-1:
```
PRINT (5 + 6) / (1 + 2 / 3)
```

3-2:
```
PRINT (1 + 1 / 2) * (1 / (1 + 1 / 2))
```
or
```
PRINT (1 + .5) / (1 + .5)
```

3-4:
```
PRINT 100 / 1.6
```

3-6:
```
PRINT 250 / 45
```

4

4-1:
```
10 INPUT A, B, C, D
20 SUM = A + B + C + D
30 AVG = SUM/4
40 PROD = A * B * C * D
50 PRINT SUM, AVG, PROD
60 END
```

4-2:

a = no	e = yes	i = yes
b = yes	f = yes	j = no
c = no	g = no	k = yes
d = yes	h = yes	l = yes

4-4:
```
10 PRINT "GIVE THE NAME OF AN OBJECT";
20 INPUT O$
30 PRINT "GIVE ME THE NAME OF A PIECE OF FURNITURE";
40 INPUT F$
50 PRINT "GIVE ME THE NAME OF A FRIEND";
60 INPUT P$
70 PRINT
80 PRINT "DOES YOUR FRIEND "; P$; " HAVE A "; O$; " ON A ";
F$; "?"
90 END
```

4-6: b, c, f are valid.

5

5-2: a = yes d = yes
b = yes e = no
c = yes f = no

5-4:
```
INPUT "YOUR NAME: ", NAME$
INPUT "WHAT IS 2 + 3? ", C
INPUT "THE LAST TWO NUMBERS ARE . . ."; A, B
```

5-6: A = 3

6

6-1: The IF statement allows the program to make decisions, thus changing its behavior according to variations in input data or computed values.

6-3: a = yes e = yes, but not very useful
b = yes f = yes
c = yes g = yes
d = yes

6-4: In AMSTRAD BASIC, yes.

6-5: A program loop repeatedly executes a part of a program. In order to prevent an infinite loop (a loop that will not stop repeating) a test should always be made within the loop, which, when successful, allows the program to jump out of the loop.

7

7-1:
```
10 FOR I = 1 TO 15
20 PRINT I;
30 NEXT I
40 END
```

7-2:
```
10 INPUT "HOURS,MINUTES: ", HOURS, MINUTES
20 REM VALIDATE INPUT
30 IF HOURS >= 0 AND HOURS < 72 AND MINUTES >= 0
     AND MINUTES < 60 THEN 70
40 PRINT "INCORRECT, TRY AGAIN"
50 GOTO 10
60 REM PRINT LINE OF H'S
70 IF HOURS = 0 THEN 110
80 FOR A = 1 TO HOURS
90 PRINT "H";
100 NEXT A
110 PRINT
120 REM PRINT LINE OF M'S
130 IF MINUTES = 0 THEN 170
140 FOR A = 1 TO MINUTES
150 PRINT "M";
160 NEXT A
170 PRINT
180 END
```

7-4: A jump can be made into a loop not run by FOR . . . NEXT statements.

7-6:
```
10 PRINT "PROGRAM TO SUM ALL ODD INTEGERS"
20 PRINT "TO A USER-ENTERED VALUE"
30 INPUT "HIGHEST ODD INTEGER TO SUM: ", NUM
40 REM TEST FOR VALID INPUT
50 REM INT(NUM/2) <> NUM/2 CHECKS FOR ODD NUMBER -
     SEE CH. 10
60 IF NUM > 0 AND NUM < 10000 AND INT(NUM/2) < NUM/2
     THEN 90
70 PRINT "BAD NUMBER . . . TRY AGAIN"
80 GOTO 30
90 REM DO TABLE
100 PRINT "NUMBER", "SUM"
110 PRINT
120 FOR N = 1 TO NUM STEP 2
130 SUM = SUM + N
140 PRINT N, SUM
150 NEXT N
160 END
```

7-8:
```
10 INPUT "TAX RATE IN PERCENT: ", TAX
20 IF TAX < 1 OR TAX > 100 THEN 10
30 PRINT "PRICE", "TAX", "TOTAL"
40 FOR PRICE = 1 TO 100
50 PRINT PRICE, PRICE * TAX / 100, PRICE + PRICE * TAX / 100
60 NEXT PRICE
70 END
```

8

8-2: Coding is one step of programming. Programming involves algorithm design, flowcharting, coding, debugging, and documenting.

8-4: A variable is traced by inserting PRINT statements to show the value of the variable at critical points in the program. The TRON and TROFF commands (trace on and trace off) are provided by the interpreter to show the path of execution of a program.

8-6: A flowchart is a symbolic diagram showing the sequence of events occurring during the execution of a program.

8-8: Clearly written programs are easy to understand, and thus easy to modify. This allows the programmer who created the program, as well as other programmers, to easily change it.

Appendix B:

AMSTRAD BASIC Keywords

This list will help you avoid using illegal names for variables.

ABS	AFTER	AND	ASC	ATN	AUTO
BIN$	BORDER	CALL	CAT	CHAIN	CHAIN MERGE
CHR$	CINT	CLEAR	CLG	CLOSEIN	CLOSEOUT
CLS	CONT	COS	CREAL	DATA	DEF
DEFINT	DEFREAL	DEFSTR	DEG	DELETE	DI
DIM	DRAW	DRAWR	EDIT	EI	ELSE
END	ENT	ENV	EOF	ERASE	ERL
ERR	ERROR	EVERY	EXP	FIX	FN
FOR	FRE	GOSUB	GOTO	HEX$	HIMEN
IF	INK	INKEY	INKEY$	INP	INPUT
INSTR	INT	JOY	KEY	KEY DEF	LEFT$
LEN	LET	LINE	LINE INPUT	LIST	LOAD
LOCATE	LOG	LOG10	LOWER$	MAX	MEMORY
MERGE	MID$	MIN	MOD	MODE	MOVE
MOVER	NEW	NEXT	NOT	ON	ON BREAK
ON ERROR	ON GOTO	ON GOSUB	ON ERROR	ON SQ	OPENIN
OPENOUT	OR	ORIGIN	OUT	PAPER	PEEK
PEN	RAD	RANDOMIZE	READ	RELEASE	REM
REMAIN	RENUM	RESTORE	RESUME	RETURN	RIGHT$
RND	ROUND	RUN	SAVE	SGN	SN
SOUND	SPACE$	SPC	SPEED	SQ	SQR
STEP	STOP	STR$	STRING$	SWAP	SYMBOL
TAB	TAG	TAGOFF	TAN	TEST	TESTR
THEN	TIME	TO	TRON	TROFF	UNIT
UPPER$	USING	VAL	VPOS	WAIT	WEND
WHILE	WIDTH	WINDOW	WRITE	XOR	XPOS
YPOS	ZONE				

APPENDIX C:

BASIC GLOSSARY

algorithm A sequence of steps that specify the solution to a given problem.

alphanumeric The set of alphabetic and numeric characters.

assignment The operation of giving a value to a variable, which is indicated by the " = " symbol in BASIC.

BASIC *Beginners All-Purpose Symbolic Instruction Code.* A high-level programming language designed for ease in learning.

binary A numbering system that uses only two digits: 0 and 1.

bit A contraction of the words binary digit. A bit may take the value 0 or 1.

bug Program error. Bugs should be prevented rather than cured.

byte A group of eight bits.

chip An integrated circuit that resides on a small silicon square mounted on a plastic or ceramic package.

coding The act of converting an algorithm or a flowchart into a sequence of program statements. This is one of the stages of the programming process.

command A keyword used to perform a specific housekeeping chore, such as clearing the screen, starting a program, or accessing files. Specifying a command activates a specialized program inside the computer to perform the chore.

computer An enclosure containing at least a central processing unit, a memory, basic interfaces that allow it to communicate with the outside world, and a power supply. The enclosure may also include a keyboard, a screen, and disk drives. A computer is capable of storing programs and executing them. It communicates with the outside world by means of input/output devices. The usual input device is a keyboard. The usual output device is a screen and may also be a printer. Additional memory is usually provided in the form of disk units or cassette recorders.

CPU *Central Processing Unit.* An electronic module in charge of fetching, decoding and executing in the proper sequence instructions stored in the memory. On most small computers, the CPU and the memory reside on a single circuit board or "card." The CPU normally uses a microprocessor chip and a few other components.

CRT A *Cathode Ray Tube.* A television-like screen.

cursor A symbol used to indicate the current position at which a character will be displayed on the screen. Often a blinking square or underline. Specialized keys are generally available to conveniently position the cursor on the screen.

data Any text or numbers on which a program can operate.

debugging Action of removing the bugs from a program. Debugging may be arduous and all efforts should be used to design a correct program so that it has as few bugs as possible.

disk Magnetic medium on which data and programs can be stored. On disk, information is organized in files that can be retrieved by name. Disks can store a large amount of data and are a common mass memory device used with small computers.

disk drive The mechanism used to read and write from a disk.

diskette A floppy disk, i.e., an 8" or 5-1/4" soft disk designed to provide inexpensive storage for programs and data.

double precision Number that has twice as many digits as in the normal representation. Each interpreter represents numbers with a set number of digits. Double precision is usually required for scientific computations or business calculations that involve large numbers or extensive computations.

editor Program designed to facilitate the entering and modifying of text. Used to remove errors or make changes while typing in a program.

empty statement A statement that does nothing. For example, the statement:

```
10 REM
```

can be used to provide a space in the program and enhance readability.

endless loop A loop that has no exit point and executes forever. This is a common error made by programmers when no test condition is included in the loop or when the test always succeeds (or always fails). Breaking out of an endless loop requires the use of a special escape character—ESCAPE on the AMSTRAD, often CTRL C on other computers.

expression A combination of operands or variables separated by operators. An expression represents a formula and performs a specific calculation. When an expression is evaluated by the interpreter at execution time, it results in a value.

file A collection of information that has been given a name. A program is usually stored on a disk as a file.

fixed point number An integer, i.e., whole number, where the decimal point is in a fixed position to the right of the last digit.

floating point number A decimal number. A fixed number of digits is used internally to represent any number; and as operations are performed on the number, the position of the decimal point floats to the left or the right.

flowchart Symbolic visual representation of an algorithm.

graphics Pictures, figures, or drawings displayed on the screen, generally using a pattern of small adjacent dots. The use of color enhances the appearance of graphics.

hardware The equipment, including the computer, the disks, and any other items, that makeup a computer system. *Contrast with:* software (i.e., the instruction or the programs).

high-level language A programming language designed to facilitate giving instructions to the computer. BASIC is a high-level language.

IC An *Integrated Circuit.*

initialization The phase (in a program) during which initial values are assigned to the variables. Every loop requires an initialization phase.

instruction A valid order given to the interpreter that will affect the data being operated on. Each instruction preceded by a line number is part of a program. (Commands do not affect the values of the data. They perform housekeeping chores or facilitate designing or using a program.)

integrated circuit Also called IC. An electronic circuit with many transistors and logic functions realized on a small piece of silicon.

interface Electronic circuits that allow the connection of a specific device to the computer. A disk, a printer and a recorder require specific interfaces.

interpreter The program in charge of translating the instructions of a programming language (say BASIC) into the binary language of the computer and executing them. Once the interpreter has been installed on the computer, the computer appears to understand BASIC instructions.

I/O Input/Output, i.e., the communications to and from the computer.

jump Branching to, i.e., executing an instruction out of sequence.

K 1024. Read as "K" or kilo. K is used to denote memory size, usually in bytes.

keyword A word that has a predefined meaning for the BASIC interpreter. It may not be used by the programmer as a variable name.

label A line number in BASIC.

loading Transferring data or a program into the computer's internal memory.

logical A variable or expression that takes the value true or false.

logical expression A combination of operands or variables separated by relational operators (=, >, etc.). The value of a logical expression is either true or false.

loop A sequence of program instructions that executes repeatedly until a specified event occurs, usually until a variable reaches a given value.

machine language The binary language that the computer understands directly, i.e., a limited set of instructions that manipulate binary information inside the CPU and the memory.

memory Medium that stores information. The internal memory usually resides on the same board as the CPU and stores programs and data as bytes. Mass memory units are cassettes and disks.

microcomputer A computer that uses a microprocessor as its central processing unit.

microprocessor An integrated circuit that implements most of the function of a CPU on a single chip. A contemporary microprocessor incorporates sometimes tens of thousands of transistors within a single chip and may even incorporate the memory.

monitor A minimal program required to operate the computer system. The monitor reads characters from the keyboard, displays them on the screen, and performs the basic data transfers between the keyboard, screen and common peripherals.

MPU *Microprocessor unit.* A chip.

nested loop Loop embedded within another loop.

non-resident A program that is normally not in the permanent memory (the ROM) of the computer. A non-resident program may be stored on a cassette or diskette.

operating system A housekeeping program that provides extensive facilities for performing all common data transfers and data processing required to conveniently use the resources of a computer system. The operating system manages disk files, performs format conversions, and starts and stops programs.

operator A symbol representing any valid operation on values (i.e., + (add), * (multiply), etc.).

peripheral Device connected to a computer, such as a printer, a disk drive or a terminal.

program Sequence of instructions to be executed by the computer. Each program is written in a specific computer language and must be loaded into the computer's memory in order to be executed.

RAM *Random Access Memory.* The read/write (modifiable) portion of the computer's memory (the rest being ROM).

relational operator A logical operator that establishes a size relationship between values.

resident A program that is stored permanently in the computer's memory, i.e., in ROM.

ROM *Read-Only Memory.* This memory may not be changed by the programmer. It generally contains part or all of the monitor and sometimes a simple BASIC interpreter.

RUN Command that starts execution of a BASIC program.

software The programs.

statement A BASIC instruction or a command, that is, part of a program.

string A sequence of characters (*contrast with:* a number). In BASIC, the name of a variable is different, depending on whether it contains a string or a number. For example, WORD$ is a string variable, while NUMBER is a numeric variable.

syntax A set of rules that defines the acceptable instructions of a computer language. BASIC has a simple syntax. The interpreter always checks for, and indicates, syntax errors.

terminal Combination of a screen and a keyboard, or a printer and a keyboard, used to communicate conveniently with a computer.

variable Memory location that has a name and may take successive values over time. BASIC distinguishes between string variables and numeric variables. The name of a string variable must end with a $.

Index

Algorithm 139–143, 146
Alphabet keys 34
AMSTRAD BASIC 25
APL 24
Arithmetic operations 58–60
Arrow 147
Assignment statement 77–81

BASIC 24
BASIC, versions of
  extended 23
  floating point 23
  full 22
  integer 23
  mini 22
  tiny 22
BASIC expression 60
Binary 20–21
Bit 20
Boldface, type 70
BREAK 37
Bug 154
Built-in function 184
Byte 20

Calculator mode 42
CAPS LOCK 34, 36
Carriage return 34–35
Celsius 63
Central Processing Unit (CPU) 28
Chip 28
CLR 34–38
COBOL 24
Coding 139, 169
Comma 61
Command 46
  END 46, 147
  FOR...NEXT 127, 129, 132–134, 175
  GOTO 112, 173
  IF...THEN 100, 104, 107, 114
  INPUT 68, 80, 90, 92
  LIST 43, 46–49, 96
  NEW 46–48
  PRINT 39, 52, 53
  RUN 42, 43, 47
Computer 26
Control character 36
CONTROL (CTRL) key 34, 36

COPY 34, 37–38
Counter variable 128
Counting technique 83, 115
CPU 28
CTRL keys, examples of 34, 36
Cursor 37, 38

Dashes 89
Data 33
Data bank 30
Data structure 186
Debugging 139
Decision box 147
Deferred execution 45
Deferred instruction 43
Deferred mode 42, 45
DEL 34, 37
Design error 154
Diagnostic 163
Digit 34
Disk drive 30
Disk unit 28, 30
Documentation 156–157
Documenting 139

Empty PRINT statement 92, 172
Empty statement 50
END 46, 147
ENTER 35, 39
Equal sign 81
Error 20, 140
Error message 40
ESCAPE 34, 37
Executable statement 107
Execution 45
Explicit value 80
Exponentiation 58
Expression 61
Extended BASIC 25

Fahrenheit 63
False 100, 105
File 186
Floating-point BASIC 25, 57
Floating-point number 57
Flowchart 126, 139, 144, 145, 148–149, 157, 167, 169, 173
Flowcharting 144, 150, 162
FOR...NEXT 127, 129, 134, 176
FORTRAN 24
Full BASIC 22
Function 184
  built-in 184
  user-defined 184

Gallon 63
GE225 24
GOTO 112, 173
Graphic 187

Hand testing 150
High-level programming language 21, 22, 24

IF 100, 103, 104, 107, 114
IF/GOTO technique 124
IF...THEN 100, 104, 107, 115
Immediate instruction 44
Immediate mode 42, 45, 56, 83, 156
Initial value 124
Initialization 84, 116, 124, 128
Information 20
Inner loop 135
INPUT 68, 80, 171
Input device 26
Instruction 20
Integer BASIC 23
Interactive 24
Interface 30
Intermediate variable 78
Interpreter 22, 29, 30, 57

Jump 102, 173

K 25
Kemeny, John 24
Keyboard 26–29, 34
Keys, examples of
  BREAK 37
  CAPS LOCK 36
  CLR 37
  CONTROL 36
  COPY 37
  DEL 37
  ENTER 35
  ESCAPE 37
  SHIFT 36
Keyword 40, 73
Kilometer 63
Kurtz, Thomas 24

Label 42
Label number 48, 49
Language 20–23
Languages, high-level
  APL 24
  BASIC 24
  COBOL 24
  FORTRAN 24
  Pascal 24
Line number 44, 94, 95
LIST 44, 46–49, 51, 96
Literal string 76
Loading 28, 30
Logical expression 100, 105
Logical operator 106
Long name 74
Loop 113, 125, 129, 132
Looping 132
Loops, types of
  Inner 135
  Nested 134, 136
  Outer 135
Lowercase 36

Machine language 20–22
Memory 28, 29
Menu 108, 109
Microprocessor 28
Mileage 63
mini-BASIC 22
Modem 30
Monitor 29
Multiple statement 89

Name 72
Nested loop 134, 136
NEW 46–48
Numeric Keypad 38
Numeric variable 71, 76
Numerical processing 20

Operation 60
Operator 58
Outer loop 135
Output device 30

Parentheses 59
Pascal 24
PRINT 40, 56, 61, 155
Printer 30
Printing numbers 56
Program 20–21, 41
Programming 20
Programming language 21
Prompt 40

Quotes 61

RAM 28–30, 45
Read-only memory 29
Read/write memory 28
Relational operator 107
REM 88, 90, 94
Resident BASIC 22
RETURN 34, 35
ROM 28–30
RUN 41, 42, 45, 46

SAVE 45, 46
Scientific notation 57
Scientific representation 57
Screen 27
Semicolon 61, 74
SHIFT 34, 36
Shortcut input 92
Space requirements 22
Spaces 90
Standard 25
START 147, 165
Statement 21
Statements, types of
  assignment 77–81
  empty 50
  empty PRINT 92, 172
  executable 107
  INPUT 80
  multiple 89

String 56, 73
String operator 186
String variable 71, 73
Subroutine 185
Sum of the first N integers 129
Symbol 21, 147
Syntax 21, 31, 80, 82
Syntax error 31, 154, 155

Tab 61
Tables of values 130
Testing 154
Text 71
Text processing 20
Tiny BASIC 22
Tracing 155
True 100, 105
Truncated 57
Types of variable 71

Uppercase 36
User-defined function 184

Validating the inpupt 117
Validity test 170
Variable 69
Variable counter technique 83
Variable name 70, 73, 94
Variable step 132, 133

The SYBEX Library

Buyer's Guide

THE BEST OF TI 99/4A™ CARTRIDGES

by Thomas Blackadar

150 pp., illustr., Ref. 0-137

Save yourself time and frustration when buying TI 99/4A software. This buyer's guide gives an overview of the best available programs, with information on how to set up the computer to run them.

FAMILY COMPUTERS UNDER $200

by Doug Mosher

160 pp., illustr., Ref. 0-149

Find out what these inexpensive machines can do for you and your family. "If you're just getting started . . . this is the book to read before you buy."—Richard O'Reilly, Los Angeles newspaper columnist

PORTABLE COMPUTERS

by Sheldon Crop and Doug Mosher

128 pp., illustr., Ref. 0-144

"This book provides a clear and concise introduction to the expanding new world of personal computers."—Mark Powelson, Editor, San Francisco Focus Magazine

THE BEST OF VIC-20™ SOFTWARE

by Thomas Blackadar

150 pp., illustr., Ref. 0-139

Save yourself time and frustration with this buyer's guide to VIC-20 software. Find the best game, music, education, and home management programs on the market today.

SELECTING THE RIGHT DATA BASE SOFTWARE

SELECTING THE RIGHT WORD PROCESSING SOFTWARE

SELECTING THE RIGHT SPREADSHEET SOFTWARE

by Kathy McHugh and Veronica Corchado

80 pp., illustr., Ref. 0-174, 0-177, 0-178

This series on selecting the right business software offers the busy professional concise, informative reviews of the best available software packages.

Introduction to Computers

OVERCOMING COMPUTER FEAR

by Jeff Berner

112 pp., illustr., Ref. 0-145

This easy-going introduction to computers helps you separate the facts from the myths.

COMPUTER ABC'S

by Daniel Le Noury and Rodnay Zaks

64 pp., illustr., Ref. 0-167

This beautifully illustrated, colorful book for parents and children takes you alphabetically through the world of computers, explaining each concept in simple language.

PARENTS, KIDS, AND COMPUTERS

by Lynn Alpers and Meg Holmberg

208 pp., illustr., Ref. 0-151

This book answers your questions about the educational possibilities of home computers.

THE COLLEGE STUDENT'S COMPUTER HANDBOOK

by Bryan Pfaffenberger

350 pp., illustr., Ref. 0-170

This friendly guide will aid students in selecting a computer system for college study, managing information in a college course, and writing research papers.

COMPUTER CRAZY

by Daniel Le Noury

100 pp., illustr., Ref. 0-173

No matter how you feel about computers, these cartoons will have you laughing about them.

DON'T! (or How to Care for Your Computer)

by Rodnay Zaks

214pp., 100 illustr., Ref. 0-065

The correct way to handle and care for all elements of a computer system, including what to do when something doesn't work.

YOUR FIRST COMPUTER

by Rodnay Zaks

258 pp., 150 illustr., Ref. 0-045

The most popular introduction to small computers and their peripherals: what they do and how to buy one.

INTERNATIONAL MICROCOMPUTER DICTIONARY

120 pp., Ref. 0-067

All the definitions and acronyms of microcomputer jargon defined in a handy pocket-sized edition. Includes translations of the most popular terms into ten languages.

FROM CHIPS TO SYSTEMS: AN INTRODUCTION TO MICROPROCESSORS

by Rodnay Zaks

552 pp., 400 illustr., Ref. 0-063

A simple and comprehensive introduction to microprocessors from both a hardware and software standpoint: what they are, how they operate, how to assemble them into a complete system.

Personal Computers

ATARI

YOUR FIRST ATARI® PROGRAM

by Rodnay Zaks

150 pp., illustr., Ref. 0-130

A fully illustrated, easy-to-use introduction to ATARI BASIC programming. Will have the reader programming in a matter of hours.

BASIC EXERCISES FOR THE ATARI®

by J.P. Lamoitier

251 pp., illustr., Ref. 0-101

Teaches ATARI BASIC through actual practice using graduated exercises drawn from everyday applications.

THE EASY GUIDE TO YOUR ATARI® 600XL/800XL

by Thomas Blackadar

175 pp., illustr., Ref. 0-125

This jargon-free companion will help you get started on the right foot with your new 600XL or 800XL ATARI computer.

ATARI® BASIC PROGRAMS IN MINUTES

by Stanley R. Trost

170 pp., illustr., Ref. 0-143

You can use this practical set of programs without any prior knowledge of BASIC! Application examples are taken from a wide variety of fields, including business, home management, and real estate.

Commodore 64/VIC-20

THE COMMODORE 64™/VIC-20™ BASIC HANDBOOK

by Douglas Hergert

144 pp., illustr., Ref. 0-116

A complete listing with descriptions and instructive examples of each of the Commodore 64 BASIC keywords and functions. A handy reference guide, organized like a dictionary.

THE EASY GUIDE TO YOUR COMMODORE 64™

by Joseph Kascmer

160 pp., illustr., Ref. 0-129

A friendly introduction to using the Commodore 64.

YOUR FIRST VIC-20™ PROGRAM

by Rodnay Zaks

150 pp., illustr., Ref. 0-129

A fully illustrated, easy-to-use introduction to VIC-20 BASIC programming. Will have the reader programming in a matter of hours.

THE VIC-20™ CONNECTION

by James W. Coffron

260 pp., 120 illustr., Ref. 0-128

Teaches elementary interfacing and BASIC programming of the VIC-20 for connection to external devices and household appliances.

YOUR FIRST COMMODORE 64™ PROGRAM

by Rodnay Zaks

182 pp., illustr., Ref. 0-172

You can learn to write simple programs without any prior knowledge of mathematics or computers! Guided by colorful illustrations and step-by-step instructions, you'll be constructing programs within an hour or two.

COMMODORE 64™ BASIC PROGRAMS IN MINUTES

by Stanley R. Trost

170 pp., illustr., Ref. 0-154

Here is a practical set of programs for business, finance, real estate, data analysis, record keeping and educational applications.

GRAPHICS GUIDE TO THE COMMODORE 64™

by Charles Platt

192 pp., illustr., Ref. 0-138

This easy-to-understand book will appeal to anyone who wants to master the Commodore 64's powerful graphics features.

IBM

THE ABC'S OF THE IBM® PC

by Joan Lasselle and Carol Ramsay

100 pp., illustr., Ref. 0-102

This is the book that will take you through the first crucial steps in learning to use the IBM PC.

THE BEST OF IBM® PC SOFTWARE

by Stanley R. Trost

144 pp., illustr., Ref. 0-104

Separates the wheat from the chaff in the world of IBM PC software. Tells you what to expect from the best available IBM PC programs.

THE IBM® PC-DOS HANDBOOK

by Richard Allen King

144 pp., illustr., Ref. 0-103

Explains the PC disk operating system, giving the user better control over the system. Get the most out of your PC by adapting its capabilities to your specific needs.

BUSINESS GRAPHICS FOR THE IBM® PC

by Nelson Ford

200 pp., illustr., Ref. 0-124

Ready-to-run programs for creating line graphs, complex illustrative multiple bar graphs, picture graphs, and more. An ideal way to use your PC's business capabilities!

THE IBM® PC CONNECTION

by James W. Coffron

200 pp., illustr., Ref. 0-127

Teaches elementary interfacing and BASIC programming of the IBM PC for connection to external devices and household appliances.

BASIC EXERCISES FOR THE IBM® PERSONAL COMPUTER

by J.P. Lamoitier

252 pp., 90 illustr., Ref. 0-088

Teaches IBM BASIC through actual practice, using graduated exercises drawn from everyday applications.

USEFUL BASIC PROGRAMS FOR THE IBM® PC

by Stanley R. Trost

144 pp., Ref. 0-111

This collection of programs takes full advantage of the interactive capabilities of your IBM Personal Computer. Financial calculations, investment analysis, record keeping, and math practice—made easier on your IBM PC.

YOUR FIRST IBM® PC PROGRAM

by Rodnay Zaks

182 pp., illustr., Ref. 0-171

This well-illustrated book makes programming easy for children and adults.

YOUR IBM® PC JUNIOR

by Douglas Hergert

250 pp., illustr., Ref. 0-179

This comprehensive reference guide to IBM's most economical microcomputer offers many practical applications and all the helpful information you'll need to get started with your IBM PC Junior.

DATA FILE PROGRAMMING ON YOUR IBM® PC

by Alan Simpson

275 pp., illustr., Ref. 0-146

This book provides instructions and examples of managing data files in BASIC. Programming designs and developments are extensively discussed.

Apple

THE EASY GUIDE TO YOUR APPLE II®

by Joseph Kascmer

160 pp., illustr., Ref. 0-122

A friendly introduction to using the Apple II, II plus and the new IIe.

BASIC EXERCISES FOR THE APPLE®

by J.P. Lamoitier

250 pp., 90 illustr., Ref. 0-084

Teaches Apple BASIC through actual practice, using graduated exercises drawn from everyday applications.

APPLE II® BASIC HANDBOOK

by Douglas Hergert

144 pp., illustr., Ref. 0-155

A complete listing with descriptions and instructive examples of each of the Apple II BASIC keywords and functions. A handy reference guide, organized like a dictionary.

APPLE II® BASIC PROGRAMS IN MINUTES

by Stanley R. Trost

150 pp., illustr., Ref. 0-121

A collection of ready-to-run programs for financial calculations, investment analysis, record keeping, and many more home and office applications. These programs can be entered on your Apple II plus or IIe in minutes!

YOUR FIRST APPLE II® PROGRAM

by Rodnay Zaks

150 pp., illustr., Ref. 0-136

A fully illustrated, easy-to-use introduction to APPLE BASIC programming. Will have the reader programming in a matter of hours.

THE APPLE® CONNECTION

by James W. Coffron

264 pp., 120 illustr., Ref. 0-085

Teaches elementary interfacing and BASIC programming of the Apple for connection to external devices and household appliances.

TRS-80

YOUR COLOR COMPUTER

by Doug Mosher

350 pp., illustr., Ref. 0-097

Patience and humor guide the reader through purchasing, setting up, programming, and using the Radio Shack TRS-80/TDP Series 100 Color Computer. A complete introduction.

THE FOOLPROOF GUIDE TO SPRIPSIT™ WORD PROCESSING

by Jeff Berner

225 pp., illustr., Ref. 0-098

Everything you need to know about SCRIPSIT—from starting out, to mastering document editing. This user-friendly guide is written in plain English, with a touch of wit.

Timex/Sinclair 1000/ZX81

YOUR TIMEX/SINCLAIR 1000 AND ZX81™

by Douglas Hergert

159 pp., illustr., Ref. 0-099

This book explains the set-up, operation, and capabilities of the Timex/Sinclair 1000 and ZX81. Includes how to interface peripheral devices, and introduces BASIC programming.

THE TIMEX/SINCLAIR 1000™ BASIC HANDBOOK

by Douglas Hergert

170 pp., illustr., Ref. 0-113

A complete alphabetical listing with explanations and examples of each word in the T/S 1000 BASIC vocabulary; will allow you quick, error-free programming of your T/S 1000.

TIMEX/SINCLAIR 1000™ BASIC PROGRAMS IN MINUTES

by Stanley R. Trost

150 pp., illustr., Ref. 0-119

A collection of ready-to-run programs for financial calculations, investment analysis, record keeping, and many more home and office applications. These programs can be entered on your T/S 1000 in minutes!

MORE USES FOR YOUR TIMEX/SINCLAIR 1000™ Astronomy on Your Computer

by Eric Burgess

176 pp., illustr., Ref. 0-112

Ready-to-run programs that turn your TV into a planetarium.

Other Popular Computers

YOUR FIRST TI 99/4A™ PROGRAM

by Rodnay Zaks

182 pp., illustr., Ref. 0-157

Colorfully illustrated, this book concentrates on the essentials of programming in a clear, entertaining fashion.

THE RADIO SHACK® NOTEBOOK COMPUTER

by Orson Kellogg

128 pp., illustr., Ref. 0-150

Whether you already have the Radio Shack Model 100 notebook computer, or are interested in buying one, this book will clearly explain what it can do for you.

THE EASY GUIDE TO YOUR COLECO ADAM™

by Thomas Blackadar

175 pp., illustr., Ref. 0-181

This quick reference guide shows you how to get started on your Coleco Adam with a minimum of technical jargon.

YOUR KAYPRO II/4/10™

by Andrea Reid and Gary Deidrichs

250 pp., illustr., Ref. 0-166

This book is a non-technical introduction to the KAYPRO family of computers. You will find all you need to know about operating your KAYPRO within this one complete guide.

Software and Applications

Operating Systems

THE CP/M® HANDBOOK

by Rodnay Zaks

320 pp., 100 illustr., Ref 0-048

An indispensable reference and guide to CP/M—the most widely-used operating system for small computers.

MASTERING CP/M®

by Alan R. Miller

398 pp., illustr., Ref. 0-068

For advanced CP/M users or systems programmers who want maximum use of the CP/M operating system . . . takes up where our *CP/M Handbook* leaves off.

THE BEST OF CP/M® SOFTWARE

by John D. Halamka

250 pp., illustr., Ref. 0-100

This book reviews tried-and-tested, commercially available software for your CP/M system.

REAL WORLD UNIX™

by John D. Halamka

250 pp., illustr., Ref. 0-093

This book is written for the beginning and intermediate UNIX user in a practical, straightforward manner, with specific instructions given for many special applications.

THE CP/M PLUS™ HANDBOOK

by Alan R. Miller

250 pp., illustr., Ref. 0-158

This guide is easy for the beginner to understand, yet contains valuable information for advanced users of CP/M Plus (Version 3).

Business Software

INTRODUCTION TO WORDSTAR™

by Arthur Naiman

202 pp., 30 illustr., Ref. 0-077

Makes it easy to learn how to use WordStar, a powerful word processing program for personal computers.

PRACTICAL WORDSTAR™ USES

by Julie Anne Arca

200 pp., illustr., Ref. 0-107

Pick your most time-consuming office tasks and this book will show you how to streamline them with WordStar.

MASTERING VISICALC®

by Douglas Hergert

217 pp., 140 illustr., Ref. 0-090

Explains how to use the VisiCalc "electronic spreadsheet" functions and provides examples of each. Makes using this powerful program simple.

DOING BUSINESS WITH VISICALC®

by Stanley R. Trost

260 pp., Ref. 0-086

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

DOING BUSINESS WITH SUPERCALC™

by Stanley R. Trost

248 pp., illustr., Ref. 0-095

Presents accounting and management planning applications—from financial statements to master budgets; from pricing models to investment strategies.

VISICALC® FOR SCIENCE AND ENGINEERING

by Stanley R. Trost and Charles Pomernacki

225 pp., illustr., Ref. 0-096

More than 50 programs for solving technical problems in the science and engineering fields. Applications range from math and statistics to electrical and electronic engineering.

DOING BUSINESS WITH 1-2-3™

by Stanley R. Trost

250 pp., illustr., Ref. 0-159

If you are a business professional using the 1-2-3 software package, you will find the spreadsheet and graphics models provided in this book easy to use "as is" in everyday business situations.

THE ABC'S OF 1-2-3™

by Chris Gilbert

225 pp., illustr., Ref. 0-168

For those new to the LOTUS 1-2-3 program, this book offers step-by-step instructions in mastering its spreadsheet, data base, and graphing capabilities.

UNDERSTANDING dBASE II™

by Alan Simpson

220 pp., illustr., Ref. 0-147

Learn programming techniques for mailing label systems, bookkeeping and data base management, as well as ways to interface dBASE II with other software systems.

DOING BUSINESS WITH dBASE II™

by Stanley R. Trost

250 pp., illustr., Ref. 0-160

Learn to use dBASE II for accounts receivable, recording business income and expenses, keeping personal records and mailing lists, and much more.

DOING BUSINESS WITH MULTIPLAN™

by Richard Allen King and Stanley R. Trost

250 pp., illustr., Ref. 0-148

This book will show you how using Multiplan can be nearly as easy as learning to use a pocket calculator. It presents a collection of templates that can be applied "as is" to business situations.

DOING BUSINESS WITH PFS®

by Stanley R. Trost

250 pp., illustr., Ref. 0-161

This practical guide describes specific business and personal applications in detail. Learn to use PFS for accounting, data analysis, mailing lists and more.

INFOPOWER: PRACTICAL INFOSTAR™ USES

by Jule Anne Arca and Charles F. Pirro

275 pp., illustr., Ref. 0-108

This book gives you an overview of InfoStar, including DataStar and ReportStar, WordStar, MailMerge, and SuperSort. Hands on exercises take you step-by-step through real life business applications.

WRITING WITH EASYWRITER II™

by Douglas W. Topham

250 pp., illustr., Ref. 0-141

Friendly style, handy illustrations, and numerous sample exercises make it easy to learn the EasyWriter II word processing system.

Business Applications

INTRODUCTION TO WORD PROCESSING

by Hal Glatzer

205 pp., 140 illustr., Ref. 0-076

Explains in plain language what a word processor can do, how it improves productivity, how to use a word processor and how to buy one wisely.

COMPUTER POWER FOR YOUR LAW OFFICE

by Daniel Remer

225 pp., Ref. 0-109

How to use computers to reach peak productivity in your law office, simply and inexpensively.

OFFICE EFFICIENCY WITH PERSONAL COMPUTERS

by Sheldon Crop

175 pp., illustr., Ref. 0-165

Planning for computerization of your office? This book provides a simplified discussion of the challenges involved for everyone from business owner to clerical worker.

COMPUTER POWER FOR YOUR ACCOUNTING OFFICE

by James Morgan

250 pp., illustr., Ref. 0-164

This book is a convenient source of information about computerizing you accounting office, with an emphasis on hardware and software options.

Languages

C

UNDERSTANDING C

by Bruce Hunter

200 pp., Ref 0-123

Explains how to use the powerful C language for a variety of applications. Some programming experience assumed.

FIFTY C PROGRAMS

by Bruce Hunter

200 pp., illustr., Ref. 0-155

Beginning as well as intermediate C programmers will find this a useful guide to programming techniques and specific applications.

BUSINESS PROGRAMS IN C

by Leon Wortman and Thomas O. Sidebottom

200 pp., illustr., Ref. 0-153

This book provides source code listings of C programs for the business person or experienced programmer. Each easy-to-follow tutorial applies directly to a business situation.

BASIC

YOUR FIRST BASIC PROGRAM

by Rodnay Zaks

150pp. illustr. in color, Ref. 0-129

A "how-to-program" book for the first time computer user, aged 8 to 88.

FIFTY BASIC EXERCISES

by J. P. Lamoitier

232 pp., 90 illustr., Ref. 0-056

Teaches BASIC by actual practice, using graduated exercises drawn from everyday applications. All programs written in Microsoft BASIC.

INSIDE BASIC GAMES

by Richard Mateosian

348 pp., 120 illustr., Ref. 0-055

Teaches interactive BASIC programming through games. Games are written in Microsoft BASIC and can run on the TRS-80, Apple II and PET/CBM.

BASIC FOR BUSINESS

by Douglas Hergert

224 pp., 15 illustr., Ref. 0-080

A logically organized, no-nonsense introduction to BASIC programming for business applications. Includes many fully-explained accounting programs, and shows you how to write them.

EXECUTIVE PLANNING WITH BASIC

by X. T. Bui

196 pp., 19 illustr., Ref. 0-083

An important collection of business management decision models in BASIC, including Inventory Management (EOQ), Critical Path Analysis and PERT, Financial Ratio Analysis, Portfolio Management, and much more.

BASIC PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller

318 pp., 120 illustr., Ref. 0-073

This book from the "Programs for Scientists and Engineers" series provides a library of problem-solving programs while developing proficiency in BASIC.

CELESTIAL BASIC

by Eric Burgess

300 pp., 65 illustr., Ref. 0-087

A collection of BASIC programs that rapidly complete the chores of typical astronomical computations. It's like having a planetarium in your own home! Displays apparent movement of stars, planets and meteor showers.

YOUR SECOND BASIC PROGRAM

by Gary Lippman

250 pp., illustr., Ref. 0-152

A sequel to *Your First BASIC Program*, this book follows the same patient, detailed approach and brings you to the next level of programming skill.

Pascal

INTRODUCTION TO PASCAL (Including UCSD Pascal™)

by Rodnay Zaks

420 pp., 130 illustr., Ref. 0-066

A step-by-step introduction for anyone wanting to learn the Pascal language. Describes UCSD and Standard Pascals. No technical background is assumed.

THE PASCAL HANDBOOK

by Jacques Tiberghien

486 pp., 270 illustr., Ref. 0-053

A dictionary of the Pascal language, defining every reserved word, operator, procedure and function found in all major versions of Pascal.

APPLE® PASCAL GAMES

by Douglas Hergert and Joseph T. Kalash

372 pp., 40 illustr., Ref. 0-074

A collection of the most popular computer games in Pascal, challenging the reader not only to play but to investigate how games are implemented on the computer.

INTRODUCTION TO THE UCSD p-SYSTEM™

by Charles W. Grant and Jon Butah

300 pp., 10 illustr., Ref. 0-061

A simple, clear introduction to the UCSD Pascal Operating System; for beginners through experienced programmers.

PASCAL PROGRAMS FOR SCIENTISTS AND ENGINEERS

by Alan R. Miller

374 pp., 120 illustr., Ref. 0-058

A comprehensive collection of frequently used algorithms for scientific and technical applications, programmed in Pascal. Includes such programs as curve-fitting, integrals and statistical techniques.

DOING BUSINESS WITH PASCAL

by Richard Hergert and Douglas Hergert

371 pp., illustr., Ref. 0-091

Practical tips for using Pascal in business programming. Includes design considerations, language extensions, and applications examples.

Assembly Language Programming

PROGRAMMING THE 6502

by Rodnay Zaks

386 pp., 160 illustr., Ref. 0-046

Assembly language programming for the 6502, from basic concepts to advanced data structures.

6502 APPLICATIONS

by Rodnay Zaks

278 pp., 200 illustr., Ref. 0-015

Real-life application techniques: the input/output book for the 6502.

ADVANCED 6502 PROGRAMMING

by Rodnay Zaks

292 pp., 140 illustr., Ref. 0-089

Third in the 6502 series. Teaches more advanced programming techniques, using games as a framework for learning.

PROGRAMMING THE Z80

by Rodnay Zaks

624 pp., 200 illustr., Ref. 0-069

A complete course in programming the Z80 microprocessor and a thorough introduction to assembly language.

Z80 APPLICATIONS

by James W. Coffron

288 pp., illustr., Ref. 0-094

Covers techniques and applications for using peripheral devices with a Z80 based system.

PROGRAMMING THE 6809

by Rodnay Zaks and William Labiak

362 pp., 150 illustr., Ref. 0-078

This book explains how to program the 6809 in assembly language. No prior programming knowledge required.

PROGRAMMING THE Z8000

by Richard Mateosian

298 pp., 124 illustr., Ref. 0-032

How to program the Z8000 16-bit microprocessor. Includes a description of the architecture and function of the Z8000 and its family of support chips.

PROGRAMMING THE 8086/8088

by James W. Coffron

300 pp., illustr., Ref. 0-120

This book explains how to program the 8086 and 8088 in assembly language. No prior programming knowledge required.



are different.

Here is why . . .

At SYBEX, each book is designed with you in mind. Every manuscript is carefully selected and supervised by our editors, who are themselves computer experts. We publish the best authors, whose technical expertise is matched by an ability to write clearly and to communicate effectively. Programs are thoroughly tested for accuracy by our technical staff. Our computerized production department goes to great lengths to make sure that each book is well-designed.

In the pursuit of timeliness, SYBEX has achieved many publishing firsts. SYBEX was among the first to integrate personal computers used by authors and staff into the publishing process. SYBEX was the first to publish books on the CP/M operating system, microprocessor interfacing techniques, word processing, and many more topics.

Expertise in computers and dedication to the highest quality product have made SYBEX a world leader in computer book publishing. Translated into fourteen languages, SYBEX books have helped millions of people around the world to get the most from their computers. We hope we have helped you, too.

For a complete catalog of our publications please contact:

U.K.
SYBEX Ldt., UNIT 4, Bourne Industrial Estate
Crayford, Kent DA1 4BU, Tel.: Crayford (0322) 57717, Telex: 897958

U.S.A.
SYBEX INC, 2344 Sixth Street, Berkeley, CA 94710, USA
Tel.: (415) 848-8233, Telex: 336311

GERMANY
SYBEX-VERLAG GmbH, Vogelsanger Weg 111, 4000 Düsseldorf 30
Tel.: (0211) 626441, Telex: 8588163

FRANCE
SYBEX, 6–8, Impasse du Curé, 75018 Paris
Tel.: 1/203-95-95, Telex: 211.801 f

"I've read all of the SYBEX books!"



SYBEX

In One Hour You'll Be Writing Your First Amstrad Program!

Your First Amstrad Program will teach you the basics of Amstrad programming.

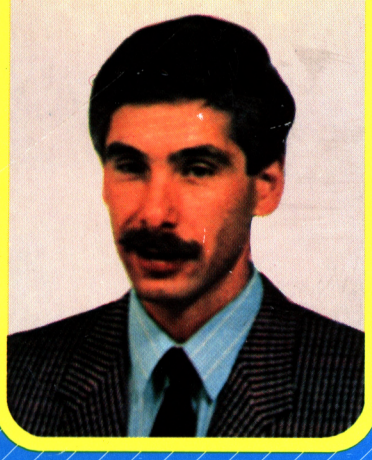
Your First Amstrad Program requires *no computer experience.* This book has been *written for everone.* If you're 8 years old or 88 and want to learn how to program your Amstrad, *this book is for you.*

Your First Amstrad Program entertains while it explains. In fact, you'll be writing your second Amstrad program in no time.

Your First Amstrad Program is a fun and easy introduction to programming computers. With loads of colourful illustrations and simple diagrams, **Your First Amstrad Program** should be your first Amstrad book.

ISBN 0-94788-800-4

The author, Dr. Rodnay Zaks, is widely known as a pioneer in computer education. He has lectured world-wide on all aspects of computers and programming. Dr. Zaks is the author of over 15 best-selling computer books, now available in ten languages. He is praised for his clear and effective writing style.



SYBEX U.K. £6.95



Document numérisé avec amour par

AMSTRAD CPC

MÉMOIRE ÉCRITE

https://acpc.me/