The ZX PROGRAMMERS' COMPANION

John and Catherine Grant

The ZX PROGRAMMERS' COMPANION

John and Catherine Grant

The right of the University of Cambridge to print and sell all manner of books was granted by Henry VIII in 1534. The University has printed and published continuously since 1584.

Cambridge University Press
Cambridge
London New York New Rochelle
Melbourne Sydney

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
32 East 57th Street, New York, NY 10022, USA
296 Beaconsfield Parade, Middle Park, Melbourne 3206, Australia

First published 1984

Printed in Great Britain at the University Press, Cambridge

Library of Congress catalogue card number: 83-23967

British Library cataloguing in publication data
Grant, John, 1984
The ZX Programmers' companion.
1. Sinclair ZX81 (Computer)—Programming
2. Sinclair ZX Spectrum (Computer)—Programming
I. Title II. Grant, Catherine
001.64'2 QA76.8.S625
ISBN 0 521 27044 8

D.S.

CONTENTS

AUTHORS' NOTE

The programs in this book are written for the ZX Spectrum or the ZX81 computers. Each program is written for one of the computers, and an indication given of how to convert it into the form required for the other. In many cases only small changes are needed to convert the program for other microcomputers that support BASIC.

The Timex/Sinclair TS1000 is the North American version of the ZX81, and the TS2000 is the North American version of the Spectrum. References to the ZX81 or Spectrum should be understood to apply also to the corresponding North American version.

The differences between the ZX81 and the Spectrum are firstly that the TV picture is made up in rather different ways in the two machines, and secondly that the Spectrum's BASIC is an 'extended' version of the ZX81's. On the Spectrum the state of each point on the screen is stored separately, so that high-resolution graphics pictures can be drawn; on the ZX81 only text and low-resolution graphics are available. Also, of course, the ZX81's picture is in black and white whereas the Spectrum produces a colour picture (although the colour information is to a rather lower resolution than the graphics) and even on a monochrome TV set will produce several different shades of grey.

Apart from the additional commands etc. to support its graphics facilities the Spectrum's BASIC provides the additional commands BEEP, DATA, DEF FN, MERGE, OUT, READ, RESTORE, and VERIFY, and functions FN, IN, and VAL$ as well as lower case letters, the 'colon' separator which allows several commands to be put on one line, and extra facilities in commands CLEAR, INPUT, LOAD, and SAVE. The ZX81 commands FAST, SCROLL, SLOW and UNPLOT are not included in the Spectrum's BASIC because it does not have separate 'fast' and 'slow' modes and scrolling and unplotting are done in a different way.

There are some differences in notation between the two BASICs; the same notation has been used throughout the book regardless of which version of BASIC is being used. Lower case letters have been used for the names of variables (to distinguish them from the 'tokens' which are in capitals) although on the ZX81 only capitals are available. The following tokens have different spellings on the two machines:

ZX81	Spectrum	Used here
CONT	CONTINUE	CONTINUE
GOSUB	GO SUB	GOSUB
GOTO	GO TO	GOTO
RAND	RANDOMIZE	RANDOMIZE
**	↑	↑

PART I

WHAT IS A COMPUTER?

A computer is a machine which
is used to store and process information,
and which is controlled by a 'program'
stored in the machine along with the other
information. In Part I we look at just
what this means in practice.

1

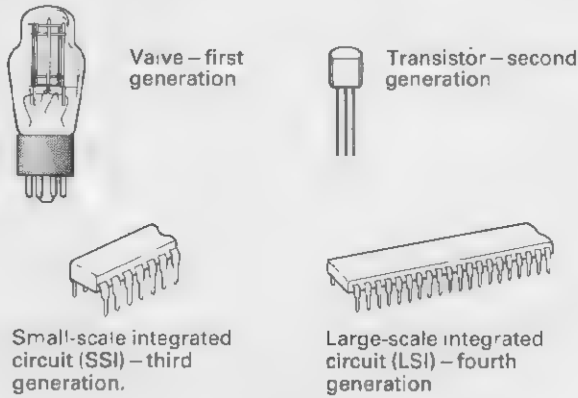
AN HISTORICAL INTRODUCTION

The most direct l ne of descent to present-day m crocomputers probably starts with the mechanical calcu ators first developed by Pascal in the 17th century. The important difference between computers and mechanica ca culators lies n the ability of a computer to be 'programmed' to carry out a sequence of calculations. The calcu ator has to be made to perform each operation separately (by pressing a key or turning a hand e), wa ting until one operat on is done before going on to the next, but once the 'program of operations is stored in the computer's memory it can work through the sequence over and over again at its own speed.

n the 19th century, Charles Babbage tried to bu ld a programmable mechanical calcu ator which he called the analytical engine but the task proved to be mpossible with the mechanica engineering technology avai able at the time (t is thought that using modern materials and more accurately machined parts a working analyt cal engine could now be built.)

The first working computers were built in the 1940s using thermion c valves (called 'tubes' in North America); these machines are now referred to as 'first generation' computers (see Fig 1 1). The 'second generation' computers of the ear y 1960s used transistors, and the third generat on used 'integrated circuits' in which a dozen or so transistors were comb ned on a single si icon 'chip' Each chip was a build ng block performing a s mple function (such as amplifying a signal or combining several signals into one) which previously had to be done by a circuit using several separate transistors and other components. The fourth generation, which includes microcomputers, uses 'large scale' integrated circuits, LSI for short, in which the chip contains tens of thousands of transistors. The pattern of

FIG 1.1



transistors and the connections between them on an integrated circuit is created in a similar way to the pattern of connections on a printed circuit board, but on a much smaller scale.

The invention of LSI circuits can be likened to the invention of the printing press. Before there were printing presses, every copy of a book had to be written out by hand, but with the printing press a whole page (once it had been typeset) could be printed in a single, quick operation.

The difference between soldering individual components onto a circuit board and producing an LSI chip is very similar.

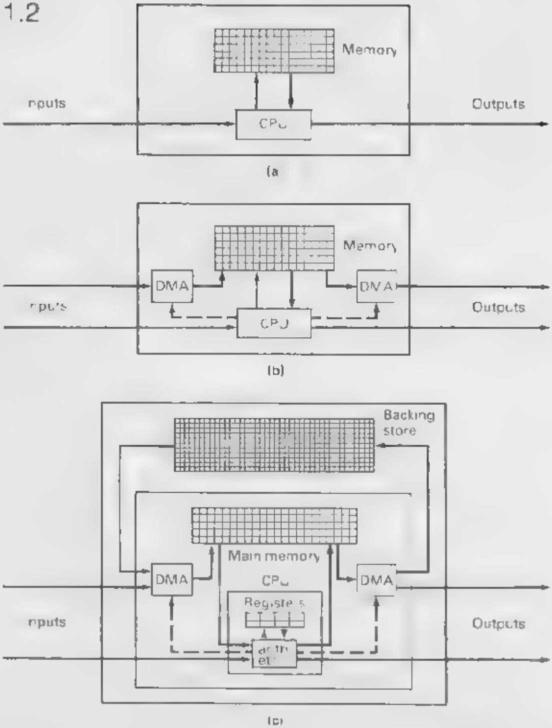
All the first four generations of computers have the same basic structure shown in Fig. 1.2(a), which was first described by John von Neumann in the early 1940s. The memory can be thought of as a huge bank of switches, each of which can be either on or off. the switches are grouped into 'words', and each word in the memory contains the same number of switches. Each word has its own 'address' which is simply a number that identifies that particular word, rather in the way that each house in a street has its own number.

The 'central processing unit', or CPU, is able to look at, or 'read', any word in the memory; it can also attempt to 'write' any word (i.e. to set up a new pattern of ons and offs on the switches) but with some kinds of memory, called 'read-only' memory or ROM for short, this attempt will not succeed.

When we wish to store numbers in the memory, the on/off state of each switch is used to represent either 0 (in one state) or 1 (in the other); these are called 'binary digits' or 'bits' for short. On most present-day computers a word contains 8 bits and is called a 'byte'. There are 256 different

BIT – something that can be in one of two states, usually called '0' and '1'. Can refer either to a piece of hardware or to the abstract '0' or '1' state stored in it.
BYTE – a bit string (qv) 8 bits long. Most computers nowadays process data in units of 1, 2, or 4 bytes at a time.

FIG 1.2



combinations of 1s and 0s (called 'bit strings') that can be stored in a byte 00000000, 00000001, 00000010, 00000011, 00000100 and so on up to 11111111. A pair of bytes (containing a total of 16 bits) will hold one of 65 536 different bit strings We talk further about representing data by bit strings in Chapter 3.

The CPU used in the ZX computers is the Z80 which uses 8-bit words stored in separate memory chips with a total of up to 65 536 different addresses. There are a further eighteen 8-bit words and four 16-bit words that are stored in the CPU chip itself and are called 'registers'. The CPU sometimes uses two 8-bit words to make up a 16-bit word As with all CPUs, the apparently complicated tasks it performs are composed of large numbers of simple steps called 'machine cycles' It performs about a million machine cycles every second

The first machine cycle that the CPU performs reads a byte from the memory, this byte is called an operation code, or 'op-code', and represents one of 256 possible operations that the CPU can perform The CPU then performs the indicated operation if it is something simple like copying the bit string stored in one register to another (so that, as it were, the switches that form the

BIT STRING – A number of bits (qv) considered together as a unit. In a bit string *n* bits long, there are 2 ↑ *n* (*n* twos multiplied together) different possible patterns of 1s and 0s.
REGISTER – hardware for storing a bit string of a fixed length, usually within a central processor. (In main memory they tend to be called 'locations' instead.)
MACHINE CYCLE – a distinct operation performed by the CPU, such as fetching (from memory) and interpreting an instruction code, or storing data in memory. A simple instruction may require just one machine cycle; a more complex one maybe half a dozen

second register are set to the same states as those for the first) it is completed during the same machine cycle and the next machine cycle reads the next op-code from memory reading it from the next address in sequence. Other possible operations include reading data from the memory writing to the memory, deriving a new bit string from existing ones and storing it in a register or in the memory, and 'jump' operations that change the address from which the next op-code will be read.

A computer is not much use unless it can communicate with the outside world. The ZX computers do this chiefly through the keyboard and the TV picture, but inputs can be taken from anything that produces a measurable electrical signal and outputs can go to anything that can be electrically controlled. The CPU reads the inputs in much the same way that it reads the memory however the bit string it receives is not something that was previously stored but rather an indication of the present state of something outside the computer The Z80 uses a completely separate set of addresses for input/output from that used for memory, but some other CPUs just have one set of addresses

In the ZX computers there is an address such that five of the bits in the byte read from it correspond to five of the keys on the keyboard, each being a 0 if the corresponding key is pressed and a 1 if it is not, seven other addresses similarly sense the states of the rest of the keys The CPU reads these eight addresses in turn to discover which of the 40 keys on the keyboard are pressed, and is thus able to see when the user presses a key and react accordingly. Another bit enables it to see whether the signal from the cassette tape is at a high or low voltage, this bit being used during LOAD.

The CPU also writes the outputs in much the same way that it writes the memory but the bit string, instead of being simply stored so that it can be read back, is used to control something outside the computer. In the ZX computers this includes the signal recorded on the cassette tape during SAVE and the electrical signals that control the printer.

Sometimes bit strings are copied directly from an input to the memory, or from the memory to an output as in Fig. 1.2(*b*). This is called 'direct memory access' or DMA. The Spectrum uses DMA for the TV picture; a part of the memory is set aside for it, into which the CPU writes the appropriate bit strings to produce the required effect on the screen. The DMA circuitry copies the data to the part of the electronics that generates the video signal. The ZX81 uses its CPU to output the video signal, and the CPU cannot do this at the same time as running the BASIC program, the user is therefore given the choice of SLOW mode, in which

CLOCK CYCLE – one complete cycle of the square wave signal that controls how fast the computer runs and keeps the various parts of the computer in synchrony. The Z80 requires between three and six clock cycles for each machine cycle (qv), other processors require different numbers, e.g. just one for the 6502, 15 for the 8048.

DMA – direct memory access: hardware that allows data to be transferred directly between a peripheral and main memory, without involving the CPU. This means the CPU can be doing something else while the transfer is taking place.

the BASIC only runs during that part of the TV signal that does not contain any data, and FAST mode, in which no TV picture is generated while the BASIC is running

Other CPU types differ from the Z80 in the operations they can perform the way these are represented by op-codes, the wordlength of the memory, the number of different addresses in the memory, and other details. But all have the basic cycle of

read an instruction from the memory
obey it
read the next instruction
obey it

and so on Because the instructions are obeyed in a sequence one after the other conventional computers are called 'sequential' machines

Japan, the UK, and the EEC have announced projects to develop 'fifth generation' computers which would be able to perform a large number of operations at the same time These computers are expected to be much better at tasks that involve looking at a lot of data simultaneously than the present sequential computers, this is covered more fully in Chapter 2. Whether the computer industry can break out of the strait-jacket of the von Neumann design remains to be seen

WHAT DO COMPUTERS DO?

Electronic computers (and calculators) work much faster than mechanical calculators, so that the typical present-day computer spends much of its time waiting for a human operator to give it a command (usually by typing on a keyboard) even though each command will probably cause the computer to perform several thousand individual operations. For example entering the command

```
PRINT 417/23
```

into a ZX computer requires the computer to identify that it is to print something out, to convert the digits 417 into the bit string that represents the number 417 in the memory, to identify that one number has to be divided by another, to convert 23 into the relevant bit string, to do the division (which is itself built up from addition and subtraction operations) convert the result back into decimal digits, and arrange for these digits to be added to the picture on the TV screen. The speed with which this is done, however, makes it appear almost instantaneous.

This example raises a number of important points:
a even apparently 'primitive' operations such as division are built up from simpler operations, because the computer can only do very simple operations, but it does them so quickly that a large number of them can be used,

SEQUENTIAL MACHINE – a machine controlled by a 'clock' signal (see 'clock cycle'). Each time the clock ticks, the machine does a single operation.

FIFTH GENERATION – the next generation of computers which are intended to have many of the skills that people have, such as being able to understand spoken English. Such a computer would not be a single sequential machine (qv), but probably a large number of very simple machines working together.

b much of the work done by the computer is concerned with putting things into a convenient form for the person using it, rather than actually carrying out the calculations he (or she) requests,
c if all these operations had to be done one by one by the human operator, it would be quicker and easier to do the division sum by hand

When electronic computers were first invented, it was thought that about twenty of them would be sufficient to do all the calculations that needed to be done in the world This estimate was based on the number of calculations that were done by mathematicians at the time, and there was a total failure to realise that *because* computers could do large numbers of calculations quickly and reliably, it would become viable to do many tasks on a computer that had previously been done by other means as for instance in the example above.

WHAT ARE THEY USED FOR?

If only twenty (or even two hundred) computers are needed for 'number-crunching' what do all the others do?

The invention of the punched card is attributed to Hermann Hollerith who was one of the people given the task of analysing the data collected in the US census of 1880. He must have found the work rather tedious, because for the 1890 census he invented a system in which the answers from each census form were recorded by means of holes in a card.

One column (or group of columns) on the card would be used for each question on the census form, and there were a number of places within the column where a hole could be punched according to the answer to the question. The cards were run through a machine (a 'sorter') which sensed electrically whether there was a hole punched in a particular column on each card the card was dropped into one hopper or another according to where the hole was punched. The machine also counted how many cards went into each hopper.

Using the machine the work was completed in one third of the time it had taken to do it by hand ten years previously (Clearly the authorities had not caught on to the potential of automated data processing – if they had, the job could have taken twice as long but produced six times as many analyses.)

Punched cards were subsequently used for many data processing applications, particularly centralised accounting functions in large organisations. Computers quickly began to be used in punched card systems where they provided the ability to perform more complicated analyses than could be done with card sorters and tabulators – operations such as taking a number punched on a card adding it to a number punched on a second card, and punching a new card including the number just calculated. For instance, a computer could be fed with a stack of cards containing details of customers' accounts and a second stack containing details of payments made (the stacks having of course already been sorted into an appropriate order), it would then be able to produce a new stack of cards containing the updated account details

As an alternative to individual pieces of cardboard, the data were often recorded on magnetic tape in a similar format to that used on the cards Reading a 'record' from the tape produces the same electrical signals to the computer that the card reader would have produced on reading a card containing the same data and such records are often referred to as 'card images'. Tapes could however work rather faster than card readers and were not restricted to a particular size of record, also a tape was rather easier to handle than a box of cards containing the same amount of data.

Present-day computers increasingly use magnetic discs instead of tape. The data are stored in the same way as on a tape, that is to say in the form of small areas of magnetism in a coating made of a suitable magnetic material, and there are no grooves of the kind used on gramophone records. The advantage of using discs is that any of the records on the disc can be read in a fraction of a second, whereas to read a record on a tape may require several minutes to wind the tape to the appropriate place.

Magnetic discs are also used as 'backing store' to extend the amount of memory to which the CPU has access There are thus several levels of memory as depicted in Fig 1.2 (*c*), from registers, which offer a limited amount of storage that is very easily accessible, to backing store, which offers a large amount of storage that takes a comparatively long time to access.

In spite of the many advances made in data processing technology over the last thirty years, a very large proportion of modern 'fourth generation' computers are used to store card images and to do on them the same kind of operations – sorting cards into a particular order, counting them, printing out (or 'listing') the data from them, extracting cards with particular data values – that were done on pre-computer punched card equipment and indeed by Hollerith in 1890.

PROGRAMMING LANGUAGES

To use a computer for a particular job, it is first necessary to 'program' it by storing in its memory the required sequence of operations.

The first computers were programmed by writing down the operations to be performed, then writing down the string of numbers that corresponds to the relevant bit string, and finally loading this string of numbers into the computer's memory. This process was tedious and errors were often made so that the program loaded into the computer did not

RECORD (noun) – a bit string (qv) containing a number of data items all related in some way: the characters in a line of text perhaps, or the catalogue number, stock level, and price of a particular product.
CARD IMAGE – a record in a format that corresponds exactly to a punched card, normally 80 bytes long with the individual bytes representing the character code punched in each column of the card.
BACKING STORE – memory in which blocks of data from main memory can be stored and later retrieved; the CPU cannot read individual bytes directly from backing store but must first cause the whole block to be copied into main memory.

do what it was supposed to. (These errors are called 'bugs', and more will be said of them anon.)

It was quickly realised that converting the program into this string of numbers from a form in which it was meaningful to a person reading it was just the kind of job for which you should use a computer. Accordingly, programming 'languages' were developed – formal notations in which programs could be written (as 'source code') for subsequent translation by special programs called 'compilers' into the bit string (or 'machine code') that represented the appropriate sequence of operations. The computer would then 'run' the program by performing these operations.

Different machines have different repertoires of operations that they can perform, and different ways of representing them in the computer's memory. We talk of them as having different 'instruction sets'. At first, each machine also had its own programming language, or 'autocode', but it soon became apparent that it would be helpful if the same language could be used on all machines - then programmers would not need to learn a new language when they moved from one machine to another, and programs written to run on one machine could be run on another without needing to be rewritten in the second machine's programming language.

The first such language, developed in the mid 1950s was Fortran. The name is short for 'formula translator' because (as was appropriate for the days in which computers were used mainly for number-crunching) it was chiefly concerned with calculating the values of mathematical formulae. For instance the formula

```
A/B + C * 5
```

was translated into a sequence of machine code operations that would divide the number represented by A by that represented by B, and multiply by 5 the number represented by C, and add the two results together. (The asterisk was used because the equipment on which programs were typed did not have a multiplication sign; we shall see how the computer finds what numbers A, B, and C represent in Chapter 3.)

A Fortran program is made up of 'statements', written with one statement on each line. A statement represents a single action, such as storing the value of a formula in the computer's memory, although this usually corresponds to a sequence of several machine operations as in the example in the previous paragraph. The term 'statement' is rather misleading, because for instance

```
A = B + C
```

BUG – a mistake in a program or in the design of a piece of hardware, as a result of which it behaves in a way that is different to that intended. Very occasionally you can pretend you actually intended it to behave that way all the time, in which case the bug becomes a 'facility'.

COMPILER – a program which translates text in a high-level language into a sequence of instructions in machine code or in an 'intermediate code'. In the latter case another program called a 'code generator' may translate it into machine code or it may be 'interpreted' directly.

does not *state* that the value of A is equal to the sum of the values of B and C, but *commands* the computer to do the necessary operations to store B+C as a new value for A (This is covered more fully in Chapter 3.)

Fortran is still much used on large computers, but is not one of the most popular languages for microcomputers.

Another language first defined in the 1950s was called 'Algol', short for 'algorithmic language'. 'Algorithm' originally meant the Arabic system of numbering (as distinct from say, Roman numerals) and arithmetic based on it, so Algol was simply a language oriented towards arithmetic, an 'algorithm' has now come to mean a step-by-step specification of how a calculation is to be carried out, and Algol is of course a language in which such specifications can be written.

The designers of Algol had three objectives the language should be as close as possible to standard mathematical notation, it should be suitable for describing algorithms in journals, and it should be possible for a computer to translate it into machine code. It is noteworthy that the designers put communication of algorithms between people before communication from a person to a computer

All the versions of Algol use a special symbol, made up from a colon and an equals sign and called 'becomes', to indicate the action of storing a number in the memory. Thus

```
a := b + c
```

is different from

```
a = b + c
```

and the latter expresses that the two values are (or happen to be) equal, and has no connotation of commanding the computer to change anything in order to make them equal.

The third important early language is Cobol, the 'common business-oriented language'. It is quite different from the number-crunching languages, being aimed at the kind of task that is appropriate to punched-card equipment and to the manipulation of 'files' of card-image records on magnetic tapes and discs; indeed it has been said that there are only four Cobol programs — one to read in new cards, one to check that the data punched on them are valid, one to sort them and 'merge' them with an existing file, and one to print them (or the new file) out. In contrast with the Algol aim of using standard mathematical notation, Cobol uses English words as in

```
ADD B TO C GIVING A
```

in an attempt to make programs comprehensible to people who are not familiar with computers, and who are not mathematicians. One effect of this is to make even simple programs rather long; some abbreviations are allowed, but a program written in the abbreviated form is not likely to make much sense to an uninitiated reader. Moreover some Cobol constructions are not all obvious even in their

ALGORITHM — a specification of [illegible] in carrying out a calculation, etc., in the computer.

unabbreviated form: for example in the part of the program that describes the heirarchy of data structures used, certain 'levels' in the heirarchy (66, 77, and 88) behave very differently from the others.

All three of the above languages are intended to be read by people as well as translated into machine code by computers. However, the features that make programs easier for people to read also tend to make the language more verbose, and hence make programs longer to write and to type. These languages were also intended for an environment in which the programmer would first think out in detail how to do the calculation he required, then write it down in the relevant language, then punch it (or have it punched) on cards or paper tape. Finally the cards or tape were read into the computer which translated the program into machine code and 'ran' it. If the program did not work some of the cards were replaced or an amended copy of the paper tape was made, and the new version was tried on the computer. Often there was a long wait for access to the computer, so the programmer had plenty of time to reflect on whether the program was likely to work and a strong incentive not to be too careless in the writing of it.

During the 1960s there was a move, particularly in universities, towards 'multi-access' (or 'on-line') computers which allowed programmers to type their programs directly into the computer's memory instead of using cards or paper tape (which are referred to as 'off-line' because typing and editing of the program are done on equipment not directly connected to the computer). This made possible 'interactive' use of the computer, in which the programmer could type in a command, have the computer obey it, and look at the result before going on to the next command. Programming was thus able to become more of a trial-and-error process than before.

APL (which stands simply for 'A Programming Language') was, like Algol 60, not originally intended as a language in which programs would actually be input to a computer; indeed, it was about eight years after the invention of the language that it was first used on a computer. Unlike Algol, it was not intended for communication of algorithms from one person to another but rather as a notation which a person would use when designing an algorithm, so it had to be designed in such a way that any particular calculation would require the minimum of writing. APL is still used very much in this way, although using a terminal on-line to a computer rather than a pencil and paper, so the requirement for terseness remains.

For this reason, APL uses mathematical notation (a minus sign, for instance needs only one keystroke, whereas the word SUBTRACT needs eight, and a ninth for the space that separates it from the next word) and special symbols were introduced for various commands for which no suitable mathematical notation was available. Most of the

ON-LINE – connected directly to the computer, so that data can be conveyed by electronic signals rather than transported to and from it on media such as magnetic tape or disc, paper tape, punched cards, etc.
INTERACTIVE – involving a two-way conversation with the program, rather than simply providing a set of data before the program runs and getting back a set of results after the program has finished.

symbols are used for several different things, rather in the way that some words in English are, and the particular meaning of a symbol used in a command is determined by the context.

Because APL is used chiefly for person-to-computer communication, and because commands are often ephemeral (that is, once the command has been typed and the computer has obeyed it – a process which often takes only a couple of seconds in all – the command is no longer needed and can be forgotten) the form of a typical APL program is such that it is extremely difficult for someone other than the author of the program to see what is going on (It is usually equally difficult for the author once he has had a few weeks to forget how the program was written.)

There is a more or less inevitable trade-off to make it easier for a person to understand, the program must have extra information added to it, and one way or another this will require extra typing. APL is sometimes called a 'write-only' language because you can write programs in it but you cannot read them afterwards Because of the large number of special symbols, and the powerful facilities they make available, it takes some time to learn to use APL effectively

BASIC (an acronym for Beginner's All-purpose Symbolic Instruction Code) dates from 1964 but only became really widespread with the advent of microcomputers in the late 1970s. Like APL it is intended for interactive use, but as its name suggests it is intended for people who are new to programming. Therefore it does not have the powerful facilities of APL, nor the special symbols that invoke them, commands are introduced by English words such as PRINT Some implementations, including those on the ZX computers, reduce the amount of typing by using a single key for each of these words.

HIGH AND LOW LEVELS

Languages are often described as 'high-level' or 'low level'. A low-level language is one that specifies the individual machine operations that are to appear in the machine code although (as with an autocode) the form in which it is written is more helpful to the human reader than a string of numbers. These languages are often called 'assembly codes' and are used where it is important to control exactly which operations the computer performs, where the program has to be particularly efficient in its use of the computer, and where no suitable high-level language is available on the computer in question.

The program that controls each of the ZX series computers (checking when keys are pressed on the keyboard, arranging for the appropriate picture to be shown on the TV, obeying BASIC commands, etc.) is written in assembly code for all three of these reasons. For instance the frequency of the signals recorded on the cassette tape by the SAVE command depends on the exact sequence of operations done during the SAVE process. Careful design allows more facilities to be fitted into the available memory and allows commonly used parts of the program to be made as fast as possible.

High-level languages are supposed to concentrate on expressing *what* task needs to be done, and to relieve the programmer of the need to decide just *how* the computer will do it. This is necessary if high-level languages are to be 'machine-independent', i.e. if the same program is to be able to be run on any computer. In practice the most common languages concern themselves a great deal with the 'how', to the extent that it is rather easy to lose sight of the 'what'. This is in some measure inevitable in a general-purpose language because the only thing the tasks have in common is that they can be done on a computer: the language provides a way of describing what the computer is to do, but cannot do this in a way that is related to the needs of any particular application.

There are some special-purpose languages that are used for particular kinds of task, and more general languages that are more truly high-level are now beginning to appear; these are called 'very high-level' to distinguish them from the older languages.

Many languages (such as Fortran and Cobol) have survived far longer than one would expect given the rate at which other aspects of computer technology are developing. New computers use existing languages so that programs written for earlier computers can be run on them and so that programmers who are familiar with the languages can use the new computer with the minimum of retraining, and it is usually easier to use an existing language (however inconvenient) than to create a new one.

BLOCK STRUCTURED LANGUAGE – a language in which a number of statements (or commands) can be grouped together as a 'block' which has the status (as far as the syntax is concerned) of a single statement or value. Usually a block can have its own 'private' variables, to prevent data being overwritten when recursion is used.

2

WHAT CAN A PERSONAL COMPUTER DO?

In the early days of computing there was much publicity on the subject of how many years it would take a team of mathematicians to do a set of calculations that a computer could do in an hour or two. Many people therefore got the impression that computers could do anything that mathematicians could do, only faster and more accurately. But by the mid-1960s researchers into artificial intelligence had shown that there were other tasks that people (including mathematicians) could do in a fraction of a second but which took a computer a quarter of an hour.

In general, computers are good at tasks that involve simple operations on numbers: copying them from one place to another, comparing two numbers to see which is the larger, adding and subtracting them. Except in some very large computers called 'array processors', these calculations are done one after the other ('serially' or 'in series') although they are done so quickly that it may look as if many of them have been done at the same time. At any instant the computer is only able to consider two or three of the thousands of numbers it has available in its memory.

Computers are good at converting small amounts of data into large amounts. This does not simply mean that they are good at producing enormous quantities of paper covered with numbers, although they have been much used in this way in the past. Consider for example a page of teletext displayed on a TV screen. (A teletext page consists of 24 lines of text, each line containing 40 'characters', characters are things like letters of the alphabet digits, punctuation marks, and the spaces between words. UK readers who do not have teletext sets can look at the BBC s 'Ceefax in vision' programmes to see some typical teletext pages) This kind of text is stored in a computer using one number for each character, so it needs a total of 960 numbers When the page is displayed on a TV screen it consists of 57600 separate dots, and the TV picture can be stored in a form that uses a number to show the colour and brightness of each dot, requiring 57600 numbers

It is a simple calculation to generate the 57 600 numbers that represent the TV picture from the 960 numbers that represent the text assuming the computer has available a table giving the pattern of dots that represents each character, and in fact the ZX computers all use essentially this method of generating the TV picture although the details differ somewhat

People, by contrast, are good at tasks that involve reducing large amounts of data to smaller amounts; this is called pattern recognition' Thus looking at the TV picture of the page of teletext we recognise patterns formed by the 57 600 points of light on the screen as letters of the alphabet etc; we do not remember the colour and position of individual dots, nor even the letters nor the words formed by them, but the overall appearance of the page and the sense of the message conveyed by the words

When we look at a page in this way, we are looking at the whole page at once, considering the 57600 pieces of information 'in parallel' It is true that if we read through all the text on the page we read it serially, starting at the top left but if one part of the page is in a brighter colour, or flashing on and off, our eye is drawn immediately to this area A computer that scanned serially through the picture would not be aware of a brighter area at the bottom of the screen until it came to it, on the other hand it would not be distracted by it while processing the other parts of the picture.

It is comparatively easy for a computer to convert the TV picture of a teletext page back into the text form, by simply looking at each of the 60 dots that make up each 'character position' and comparing them in turn with the dot patterns of each of the available characters, if an exact match is found then the character in that position has been identified if none of them matches exactly then the picture must have become corrupted in some way The computer could even deal with this case by choosing the character that is 'nearest' to the dot pattern, using some simple measure of 'nearness' that can be calculated from the dot pattern on the screen and the dot pattern for the character (such as counting how many of the 60 match exactly)

Now suppose that we have the same text printed on a piece of paper and we hold it in front of a TV camera Suppose even that it is handwritten rather than printed, or that it is being held upside-down. To a person looking at a TV screen it will appear obviously similar to the teletext form and there is not much difficulty in, say, locating the fourth letter on the third line and identifying it as an 'e' But someone trying to program a computer to examine the picture, which it has to do a dot at a time, has a real problem, because the program must first decide where the individual letters come and then identify each one from a dot pattern which is probably rather different from the 10 by 6 pattern in the teletext.

If the picture is not of a page of text but of, say, a street, then the problems for the computer are correspondingly greater. A person does not have any trouble in recognising, say, a car, but it is not easy to specify in terms of patterns of light on the screen how a computer could distinguish the image of a car from any other part of the picture. Remember that the camera may be seeing the car from the front or the back or the side, and the car may be anything from a small red sports car to a large black saloon.

This example shows that when performing everyday tasks people use a great deal of 'cultural' information (such as just what is, and what is not, a car) which it is impractical (at least with present-day technology) to store inside the computer. Consider, for example, the amount of information that would need to be stored in a computer controlling a robot for it to be able to go to the fridge and take out a bottle of milk. This requires

the concept of what a fridge is;

the ability to locate the fridge (including knowing which room it is in, and being able to distinguish it from the oven or the dishwasher);

the concept of what a bottle of milk is;

the inference that it is necessary to open the fridge door, and the knowledge that it is desirable not to leave it open too long; and

the ability to locate the door handle and open the door, and to locate the bottle of milk and pick it up.

We should avoid the temptation to think about computers (and robots) as being rather stupid people, and think of them instead as being rather sophisticated machines. Asking someone to make you some tea is a very different act from pressing a button on a machine which causes it to dispense a cup of tea, and will remain so no matter how sophisticated tea machines become.

HOME COMPUTERS

Personal computers are normally concerned only with processing data, and do not directly control or manipulate physical objects (except for the printer on which results etc. are printed out, and disc or tape drives on which data are stored for later use).

There has been some talk in recent years of 'home computers' which can control various things around the house such as the heating and lighting. While you are away

TELETEXT – text transmitted with a television signal during the interval between one 'frame' of the picture and the next, normally for display on the screen by a suitably equipped television set but also able to be read into the memory of a computer. Not to be confused with teletex [illegible]

TELETEX – a telecommunications service similar to telex but offering a much larger set of characters including lower case letters, accents, subscripts and superscripts; also not to be confused with teletext, q[illegible]

VIDEOTEX – an interactive service providing text in a similar format to teletext (qv) but transmitted over telephone lines. The user can ask to see pages from a very large 'data base' and can also type information in, e.g. to order air line tickets or to book theatre seats.

the computer can open and close the curtains and turn the lights on and off to make it look as if the house is occupied, and perhaps raise the alarm if an intruder is detected, you can phone it up and (using a device similar to the remote control for a TV set) tell it when you will be back so that the heating and perhaps the cooker can be turned on at the right time. It can be used to enable the electricity company to control when your water heater is switched on, so that they can spread the load on the local electricity supply more evenly.

Although interfaces are available for many personal computers that would allow them to become home computers, the other apparatus required is not freely nor cheeply available. For instance you are very unlikely to find that your cooker has anywhere for you to plug in a computer that would control it the motors etc required to open and close curtains of just one window would cost nearly as much as the computer; the 'auto answer' device that would allow the computer to respond to telephone calls is likely to be fairly expensive, and you may find you need a separate phone line for it.

It is quite possible that within a decade the electricity companies will begin to supply home computers to their customers, the principle use of these home computers will be for the control of power consumption (as suggested above) and for automatically reading the meter. Communication (of control commands to the computer and of the meter readings from the computer) would probably be along the power cables rather than through the telephone system, the same computer would probably also be able to carry out similar functions for the gas and water supplies.

WORD PROCESSING

We saw in Chapter 1 that all data are stored in the computer in a form which we think of as a sequence of numbers (Chapter 3 will consider this further.) Therefore the computer can store anything that can be represented as a sequence of numbers and do any processing that can be defined in terms of arithmetic operations on those numbers (including simply copying them from one place to another in the computer's memory).

A common use of computers, particularly personal computers is the storage and manipulation of text – reports, letters, and other documents Suppose you are writing a report on some topic or other; you (or your secretary) might type a draft on your typewriter and send copies to some of your colleagues for their comments As a result of their comments, and any further thoughts you might have had, various alterations are made to the draft and a complete new copy is typed If any mistakes are made in the typing, they must be rubbed out or painted over, it has been estimated that typists spend about 30% of their time correcting mistakes – it only takes a fraction of a second to press a key to print a character but it takes much longer to rub the character out again if it is wrong Also, typists tend to

INTERFACE – the connection between one part of a system and another. Bugs (qv) often arise because an interface is not well-defined, so the part of the system on one side of the interface misunderstands signals sent across the interface from the other side.

type less quickly than they could, for fear of making mistakes.

A 'word processor' is a special purpose personal computer for storing and manipulating text; there are also programs available that run on general-purpose personal computers and provide much the same facilities. With a word processor, you would not need to type the whole document a second time, just to 'edit' the draft stored in the computer. Typing mistakes are corrected as quickly as they are made, by pressing a 'delete' key which simply means 'remove that last character from the document'; because at that stage nothing has been printed on paper, there is no rubbing out to do.

The text is stored by using a number to represent each character. Essentially this means there is a different number for each key on the typewriter keyboard; for most keys there are in fact two numbers, one for the shifted (or upper case) character and one for the unshifted (or lower case) character. Note that there are also codes for the keys that do not actually print anything, such as 'space', 'new line', 'tab', and 'backspace'. The sequence of key presses is stored in the computer as a sequence of these codes.

For example, in the code that is used on almost all types of personal computer (including the Spectrum but not the ZX81) the lower case letters have codes a=97, b=98, c=99, and so on up to y=121 and z=122, the code for a capital letter is 32 less than the code for the corresponding lower case letter, and the codes for space and exclamation mark are 32 and 33 respectively. Thus the sentence

```
Go away!
```

would be encoded as

```
71, 111, 32, 97, 119, 97, 121, 33
```

To change it to

```
Gone away!
```

we require the computer to copy all but the first two codes two places further up the memory and put the codes 110 and 101 (for 'n' and 'e') in the gap: this is an example of the 'editing' referred to earlier. In practice the codes would be copied up one place at a time: once when you typed the 'n' and again when you typed the 'e'. Suppose you missed the 'e' key and hit 'w' instead: the text would read

```
Gonw away!
```

encoded in the computer as

```
71, 111, 110, 119, 32, 97, 119,
97, 121, 33
```

and you would hit the 'delete' key to tell the computer to remove the 119 that has just been inserted and close up the gap again by copying the remaining six numbers one place back. All this copying back and forth may appear tedious,

WORD PROCESSOR — a computer doing the job of a typewriter; the text is stored in the computer and can be viewed on a screen so that the operator can check that it is correct before printing it. The text can be kept on backing store [illegible] and retrieved, updated, and reprinted as required.

but it is the kind of thing that a computer can do extremely quickly, and is much easier for the user than having to tell the computer in advance how many codes are to be inserted

The facilities provided by word processors, therefore, are to type in text, to edit it, to print it out, and to store it away for future use. For this you need a keyboard to type on and a screen of some kind to see what you have typed (all personal computers have these, of course); some kind of storage device that will preserve the data even when the computer is switched off (disc is ideal but cassette tape is acceptable); and a typewriter or other printer of adequate quality to type the documents out on.

It is this last requirement that makes the ZX computers unsuitable for word processing (except as noted in the next paragraph). The basic computer does not have any means of producing printed output, the add-on ZX printer uses special paper and the characters it prints do not approach typewriter quality You cannot print on your own letterhead or invoice forms, for example, the best you can do is to print on the special aluminised paper that the printer uses and then cut out the text and stick it onto your own paper – a tedious process which results in an appearance that will be unacceptable in many situations

However, it is possible to attach a typewriter-quality printer to a ZX computer although it requires some special-purpose electronic circuitry to be constructed. You can expect that anyone who sells a device of this nature will also sell the necessary word-processing programs to go with it.

There are some more advanced facilities that word processors provide. One example is arithmetic on figures in tables converting figures from one basis to another in a report, perhaps, or calculating totals on an invoice. An example of the first would be a table which listed income and expenditure for the various different divisions of an organisation for the last five years. The figures might be typed in in thousands of dollars and the table might also show the expenditure in each case as a percentage of the corresponding income figure, and profit both in thousands of dollars and as a percentage of the profit made by the whole organisation in that year.

The character codes are almost always chosen so that the value of a digit is equal to the difference between its code and the code for the digit zero. Thus in the Spectrum the code for zero is 48, '1' is 49, '2' is 50, and so on. When you subtract 48 from a character code, if the result is 6, say, then the character is '6'. If the result is less than zero or more than 9, the character is not a digit. The word processor can easily identify where the number begins (usually it will start at a 'tab' character or even a special code inserted by the user to mark it) and work out its value and then perform whatever arithmetic operations the user has requested

Another common facility is checking for typing errors by verifying that each word in the document is a correct spelt word. The words can easily be identified – a word is simply a group of letters preceded and followed by codes that are not letters, such as space, new line, and punctuation and compared against words held in a 'dictionary' file. Allowances can easily be made for letters being in upper or lower case (for instance by converting everything to lower case before doing the comparison).

Some care is needed in organising the words in the 'dictionary' so that they can be found quickly enough, but this, too, is not a particularly difficult problem. The system needs to be able to add words to its dictionary: when a word is found that is not in the dictionary, the user is asked if this word is a typing error (in which case the document can be edited to correct it) or a new word to be added to the dictionary.

Thus if you mistype 'ti' instead of 'to' the spelling checker will tell you that the word 'ti' is not in its dictionary and you can make the necessary correction, but if you type 'too' in mistake for 'to' the program does not say anything about it because 'too' is also a word in its dictionary.

For the program to detect this kind of error would require it to be able to parse and, in many cases, in some sense 'understand' sentences in English. This is something that cannot be reduced to a sequence of simple arithmetic operations, and although a complicated program running on a powerful computer would be able to do it sufficiently well to detect a fair proportion of errors of this kind it is outside the scope of present-day personal computers.

LIMITATIONS

We have seen that computers in general can do any jobs that can be described as a sequence of simple arithmetic operations. This includes not only obviously numerical processes such as those involved in keeping accounts, but also storing and editing text and pictures, which are represented inside the computer by sequences of numbers. However it stops short of being able to deal with the kind of idea or concept that people learn by example rather than from rigorous definitions: we know what a dog is because from a very early age we have been shown dogs and pictures of dogs and told 'this is a dog', and have learnt to distinguish a dog from a cat by various features such as the shape of its head and the texture of its fur, but how can these criteria be translated into numbers?

Although computers are able to do calculations very quickly, it is quite easy to write a program that involves a huge number of calculations and thus takes a very long time to run. The effect of increasing the number of calculations tends to be imperceptible up to a certain point and then becomes apparent quite suddenly: if the computer will do 100 000 calculations per second then anything that takes less than 10 000 calculations will appear to take almost no time at all, increasing the number to 50 000 introduces a slight hesitation, 100 000 a much more noticeable one, and by 300 000 there is a significant delay while the program runs. Thus the difference between 50 and 500 is not noticeable, but the difference between 50 000 and 500 000 is quite dramatic.

The number of calculations needed to do a given job can also become very large because of the way in which more complex operations are defined in terms of simpler operations. For instance, we may define simple operations

[illegible]
[illegible]
interpreted in the correct context, for instance, in English, to
[illegible]

that consist of 100 calculations each (not at all a large number, particularly if any repetition is involved) and more complex operations that consist of 100 of the simpler operations. Then a program that consisted of 100 of the more complex operations – not at all a large program – would do a total of 1 000 000 calculations when it was run.

With ZX BASIC we have very much this kind of structure: There are simple operations such as fetching a number that has been stored away in the memory which includes finding where it has been stored, or multiplying two numbers together, which the computer has to do by a kind of 'long multiplication' because it cannot deal with the whole number in one go. The more complex operations, which are the BASIC commands, are defined in terms of these simpler operations, and the BASIC program is built up from BASIC commands.

Because the computer does so many individual calculations for each BASIC command, it cannot manage more than a few hundred commands each second, and a single command that makes heavy use of certain of the 'simple' operations, such as converting a number into character form (particularly on the ZX81) and calculating trigonometric (etc.) functions (and particularly the to-the-power operation), can take a second or more to obey

More powerful computers are able to do more calculations each second, and often require fewer individual calculations to perform a particular operation They often (but by no means always) have a more sophisticated means of translating commands into sequences of individual calculations than is possible with the resources available to ZX BASIC, so that fewer such calculations need to be performed to obey a particular command.

Apart from the speed with which the computer obeys commands, the aspect that is usually most important is the amount, and type, of memory available. The only type of memory available to the program in a ZX computer is in the RAM chips that are inside the computer or (for the ZX81's add-on memory) plugged into the back. This is 'volatile' memory: when you switch the computer off all the data stored in it are lost.

The most common type of 'non-volatile' memory, which will retain the data when you switch the computer off, is magnetic disc. Data stored on disc have to be read into RAM before the computer can use them, but the computer can fetch any part of the data when the program needs it and store updated records back on the disc when required; this is covered more fully in Chapter 10.

The only kind of nonvolatile memory available on the ZX computers at the time of writing is the cassette tape. This not controlled by the computer, so its use is limited to the user-controlled operations of storing a complete copy of a program on the tape (by the SAVE command) and retrieving it (by the LOAD command). Fortunately, when the program is saved the data it is keeping in memory are saved with it, so that when it is loaded again it can continue

VOLATILE – when applied to memory, means it loses the data stored in it when the computer is turned off. Semiconductor RAM is volatile, although CMOS RAM (which draws very little current) can be made to appear nonvolatile by supplying power from a small battery while the main power is off.

where it left off (there are examples of this in Chapters 9 and 10) but while the program is running it must have all the data it needs in RAM. A computer with a disc on the other hand, does not need to fit all its data into RAM because when processing a series of data records it only needs to have in RAM the records it is actually working on – a record is transferred back to the disc when it has been finished with and another one is read into the part of the RAM vacated by it

In practice this means that in 'record keeping' applications the maximum total size of the information (or 'data base' to use the jargon word) that can be kept is much smaller in the ZX computers than in a computer with a disc Most of the jobs of this kind that are done on a computer are of a commercial nature, such as stock control payroll accounts, and mailing lists, this is because companies are better able to justify the expenditure on a computer system and they also tend to have large amounts of data to keep up-to date. There are however several personal 'data base' applications that could be done on a computer including addresses, phone numbers, birthdays, recipes, and bank (etc) accounts as well as membership lists for clubs or societies.

Fortunately many of these only involve comparatively small amounts of data and so can be viable on a ZX computer, but because only small amounts of data are involved it is likely to be just as easy to use a pencil and notebook as it is to use a computer The computer is more likely to be a benefit where some arithmetic is to be done on the data (as in the example in Chapter 10) than where the data consists simply of text, such as lists of addresses Also, you can fill up the space in the computer's memory more quickly with text than with numbers. If you have the ZX81 with the add-on RAM, for instance, you might have about 13 K of RAM available for your data in which you could fit some 2500 numbers but only about 130 names and addresses or about a dozen recipes

DATA BASE [illegible] information about a particular topic, such as the properties of different chemicals or a company's stock records and accounts. If the records are not all [illegible] 'distributed' data base.

K – 2↑10 or 1024 [illegible] computer is described as having 16 K of memory, it means 16 K bytes (which is equal to 16 384 bytes). Just to be confusing, when referring to the individual memory chips 16 K usually means 16 K bits [illegible] 2 K (or 2048) bytes.

PART II

WRITING PROGRAMS

3

STORING AND USING DATA

If you wr te on a piece of paper We have 17 widgets in stock' this conveys information to anyone looking at it who can read and understand English The written words do not however, ook at all ike 17 widgets, or indeed 17 of anything. A Roman might have scratched on a clay tablet 'XV widget habemus', which looks quite different again.

We are, therefore, quite used to storing information in forms that do not bear any resemblance to the things described We also often leave a arge part of the informat on to be deduced from the context, for instance a card in a drawer label ed stock records' may ust say 'widgets . . 17

Computers also store informat on n forms that are convenient for them (not because the computers prefer t that way, but because t makes life easier for the people who make them) Computers, as we saw n Chapter 2 are not good at reading words written on pieces of paper, so the format of the informat on s such that it can easily be sensed and manipulated electronically.

Information written on paper is made up from a repertoire of shapes 10 dig ts, 26 capital letters, 26 lower case etters, and a number of other symbols including punctuation marks and accents and s gns such as '+' and '% . We use the letters to make up words, and the words (together with punctuation marks) to make up sentences We use the digits to represent numbers because there are ten different digits, this is called 'decimal' numbering (from the Latin for 'ten'). The Romans did not have separate digits but used letters (I, V, X, L, C, D, M) in their notation for numbers.

Sometimes quite different ways of representing the letters etc. are used, for instance patterns of dots in Braille or long and short pulses of sound in Morse code. And of course there are alternative ways of conveying whole words: most notably in spoken form but also sign language and shorthand and pictograms.

We saw in Chapter 1 that information inside a computer is represented in 'binary' form as a string of 'bits' each of which can take one of two values. The bits may be stored on magnetic media such as tape and discs (from which they are read by moving the media past a 'read head' that converts the magnetic signal into an electrical one; cassette recorders are an example of this), in the form of holes in paper tape or cards, and in the form of magnetic or electrical signals in the computer's 'main memory' (from which any bit can be read directly without having to move any mechanical parts such as a tape drive mechanism).

There are a number of different ways in which bits are represented on magnetic media, but only the engineers who design computers need to know the details of them. The ZX81 uses a system of long and short 'tone bursts' very much like the dots and dashes of Morse code but about a hundred times faster; a dot represents a 0 and a dash a 1. On paper tape and cards, a hole represents a 1 and the absence of a hole represents a 0. Main memory in the 1960s consisted of magnetic 'cores' magnetised one way round to represent a 0 and the other way round to represent a 1

Microcomputers use silicon chip memories, these have a number of ways of storing the bits internally, but at the pins which form the electrical connection to the rest of the computer they all use a 'high' voltage (above about 2 V) to represent a 1 and a 'low' voltage (below about 1 V) to represent a 0

In the same way that letters are grouped together to form words and sentences, bits are grouped together into 'bit strings'. A single bit can only be one of two things, a 1 or a 0; sometimes there are only two possible values for the data we need to store (true or false, present or absent, male or female) and in this case a single bit is sufficient. A pair of bits can have one of four values (00, 01, 10, or 11) and can thus represent data where there are up to four possibilities. Each bit added to a bit string doubles the number of possible values, so three bits have eight values, four bits have 16 and so on. A group of eight bits, called a 'byte', can thus make one of 256 values.

It should be emphasised that not only can a bit string be represented in many different forms (electrical, magnetic etc.) but it can in turn represent many different things. The 256 different values in a byte can for instance represent the whole numbers 0 to 255, or the whole numbers −128 to +127, or the fractional numbers 0 to 255/256 (in steps of 1/256), or the various characters that can be produced by a printer or shown on a TV screen, or the different operations that the computer can carry out on the data, or indeed the members of any set of not more than 256 things. There is nothing in the bit string to show which set of things is being represented, so a particular byte value could mean 186, or −70, or (on the ZX81) a white-on-black letter

L, or (on many machines including the ZX Spectrum) a colon, or an instruction to compare the values stored in two certain places in the computer, or (on the ZX Spectrum) that a character should be in red on white and flashing; or any other meaning you care to give it. Only by the context can you decide which meaning is appropriate.

Almost all high-level languages use 'data types' to distinguish between different kinds of 'values' that can be represented. The data type does a number of jobs; it defines which values are included, which value each possible bit string corresponds to, how long the bit string is (i.e. how many bits it contains: there is usually no explicit indication of where a bit string ends in the way that the end of a word is shown by a space or punctuation mark) and which arithmetic 'operations' are available.

In ZX BASIC there are just four data types: numbers, strings, arrays of numbers, and arrays of characters. Numbers, for instance, occupy 40 bits and cannot be larger than about 10^{38}, they are accurate to about nine decimal digits except that all numbers less than about 10^{-38} are stored as zero. The various parts of the program that do arithmetic on numbers, including those that convert them to and from the decimal representation, need to agree on the exact way in which each number is represented by a 40-bit bit string, but the user of the computer will not normally need to be bothered with such details.

When a program is translated into machine code, most of the information that the data type conveys is left behind. In the same way, a program in a traditional general purpose language such as BASIC only conveys a limited amount of information about the way in which the rather limited repertoire of the data types is used to represent values encountered in the real world. The programmer may for instance want to store the amount of one-inch hexagonal steel bar which a company has in its warehouse; the computer has no knowledge of steel, bars, hexagons, or inches, and the data type will probably just indicate that it is a number. But the number could be the number of pieces, the total length in feet, the total length in metres, or the weight in imperial or metric or US short tons; it is up to the programmer to ensure that the correct units are assumed at each place in the program that the number is used.

DATA TYPE – [illegible] the computer can find out how long it is, what operations can be done on it, and [illegible] the program.
VALUE – the thing [illegible] in a variable or being processed by the program; the information required to discover the [illegible] included in its data type (q.v.)

VARIABLES

Most computer languages store data in 'variables'. Just as in algebra, a variable is something that has a value that can change, but there the similarity ends. In algebra, we have things like

$$y=ax^2+bx+c$$

in which *a*, *b* and *c* are 'constants', which take particular values, and *x* and *y* are 'variables' which can take a range of values. We can draw a graph (as in Chapter 7) showing which values of *y* correspond to which values of *x*. Having drawn the graph, we can see all the values of *x* and *y* at the same time.

The computer, on the other hand, is essentially a serial device, and so variables in a computer have to have their values one at a time. At any one time a variable in a computer can only have one value, but it may have different values at different times and this is indeed the essence of how computers work. In BASIC we say

```
LET variable = value
```

and the computer *first* works out the value and *then* 'assigns' the value to the variable: the variable has this value until another value is assigned to it.

Something which is particularly confusing at first to anyone used to the algebraic sort of variable is

```
LET x = x/2
```

which looks as of we are wanting *x* to have a value which is equal to half of itself (i.e. zero). But look at what the computer does: it takes the value of *x* (which must have been assigned to it previously) and divides it by 2, then assigns this new value to x. To put it another way, x is assigned a new value which is half its old one. Similarly

```
LET x = x+1
```

(which would be nonsense as an algebraic equation) assigns to x a new value which is one more than its old one.

Although this describes adequately what a variable does in terms of the notations in the high-level language, it is worth looking at how a variable is actually stored in the computer.

Most of the early languages were 'compiled' into machine code; BASIC, as we shall see shortly, is an exception to this, but the way in which it uses variables is similar. In a compiled language a variable has

a a name,
b a data type, and
c a representation of the value.

The name is used in the program to identify which particular variable is being talked about (and is for this reason sometimes called an 'identifier') just as the name 'Joe

VARIABLE – a place in the computer's memory where a bit string is stored. In most high-level languages, the programmer's only direct access to the memory is to store data in, and retrieve it from, variables: see 'data type' and 'identifier'.
IDENTIFIER – the name given to a variable or other entity in a program. In most programming languages (though not in ZX BASIC) the identifier is only used in the text (or 'source code') form of the program, a more direct way of establishing the variable's type and location being used in the machine code.

Bloggs' would be used to identify a particular person. The representation is a bit string representing the variable's value at any time, which is the value most recently assigned to it. This bit string is often of a fixed length (a fixed number of bits) so that a particular group of bit cells (i.e. places in the memory in which bits can be stored) can be set aside to contain it, and any new value will occupy exactly the same amount of memory as the old one it replaces.

The data type describes all those things the compiler needs to know about the value in order to translate ('compile') the program into machine code such as how many bits it requires and what machine instructions to use when doing arithmetic on it.

The name and data type are thus used when the program is compiled (at 'compile time') while the value is of course only present when the program is run (at 'run time'). The name and data type are not usually available at run time, although some aspects of the data type will be implicit in the machine code operations that use the variable.

In languages of the sort just described there are two distinct phases – compile and run – and the compile phase is completed before the run phase begins. In interactive BASIC, as on the ZX computers, the user is able to type in part of a program and run it and then type in some more and run that. Obviously the second part is not compiled until after the first part has run, so it is not possible to throw away the 'name' and 'type' information at the end of compilation in case more compilation is required later.

In fact most BASICs, including ZX BASIC, differ from the earlier languages more radically than this in that they are not compiled at all but *interpreted*. Instead of translating the instructions in the program into machine instructions which will be obeyed later, the computer obeys them as it goes along. This means that it is not necessary to find space in the computer's memory for all the machine instructions, and it does not have to process any of the instructions in the program that are not going to be obeyed (i.e. those that are there to deal with situations that do not happen to arise in this particular run). On the other hand, an instruction that is to be obeyed many times has to be translated many times whereas in a compiled language it only gets translated once.

(The ZX BASICs are actually reasonably efficient in this respect. Keywords such as PRINT are stored as single codes so that the computer can immediately look up in a table what kind of action to take rather than having to identify them from the separate letters P,R,I,N,T. The codes are only translated into letters when the program is listed on the screen or printer. Numbers are translated into binary when they are typed in and the binary is stored alongside the character form in the program.)

In the ZX computers, all three parts of the information about a variable are therefore stored together and are available all the time. String variables are distinguished from numeric variables in the program by the dollar sign at the end of the name, and arrays (see below) are distinguished by the parentheses that follow the name, so the computer can always tell the data type of a variable without needing any 'context' information. In many languages, each name must be 'declared' before it can be used: the declaration specifies the data type. Thus in Algol 68

```
BEGIN REAL x, STRING s
```

declares that will the variable *x* will be of type 'real'. (a floating point number) and the variable *s* will be a string We can then say

```
x := 5;  s := "Hello"
```

but not

```
s := 5
```

which would attempt to assign to s a value which it cannot represent In BASIC there are no declarations (except insofar as the DIM command declares the size of an array as mentioned below) but the data type is deduced from the form of the name if it ends in a dollar sign it is a string, otherwise it is a number, so we can say

```
LET x = 5
LET s = 5
LET s$ = "Hello"
```

but not

```
LET s$ = 5
LET s = "Hello"
```

Note, by the way, that *s* and *s$* are different variables even though their names are similar.

In the ZX BASICs, and in most other BASICs, the variables are kept jumbled together in one part of memory, and if a variable that has not been mentioned before is used it is added to the top of the heap. There are some restrictions to this process, as follows.

If you try to use the value of (as distinct from assign to) a variable that does not exist, the ZX BASICs signal an error Th s is because t is assumed that you have made some mistake such as mistyping the name or forgetting to put in the command that should have assigned a va ue to it Therefore, the first time the computer comes across each name (other than an array) must be to the left of the equals s gn in a LET or FOR command.

Some BAS Cs simply assume the value is zero if the offending var able is a number or the 'empty' string (which contains no characters) if it is a string. This is helpful if it is what you intended but it can cause programs to do some very strange things if it is not.

An array is a group of var ables of the same type that are all collected together under one name; arrays are dea t with more fully n Chapter 5 The whole group is created at the same t me by a DIM command, which specifies how many variables are required Because it does not also specify a value, all the variables have zero (if numeric) or the 'space' character (otherwise) assigned to them The first time the computer comes across each array name must therefore be in a DIM command but once it has obeyed the DIM command you can use the 'elements' (the

CHARACTER STRING – a number of characters considered together (and in a particular order) as a single object (cf. 'bit string'). BASIC allows character strings to be manipulated as a whole or dissected into their individual characters..

ARRAY – a collection of variables all of the same type, with a single identifier (qv). The individual variables (or 'elements') are selected by one or more numbers called 'subscripts' because they correspond to numbers in algebra which are written as subscripts.

individual variables from which the array is made up) without having to assign to them with a LET command first.

Some languages allow you to treat the whole array as a single value, for instance to assign it to another array. Its representation is after all simply a (usually) rather large bit string which happens to consist of the values of the elements one after the other. It is also quite common to be able to group variables of different types together into a 'record' which again can then be treated as a single value, the individual variables of which it is made up being the 'fields' of the record. These facilities are, however, not available in BASIC.

ASSIGNMENTS

It was rather glibly stated above that assignment commands take the form

```
LET variable = value
```

without too much being said about how the value was expressed. The simplest kinds of value are variables and 'literals'. The current value of a variable is indicated by simply writing the name of the variable as in

```
LET x = y
```

which assigns to variable *x* a copy of the value in variable *y*.

(In BASIC this will always result in the values of *x* and *y* being the same bit string, but in other languages it is possible for *x* and *y* to be of different types – perhaps two different ways of representing numbers, in which case the computer would translate the value from one representation to the other.)

The value in a variable is thus represented indirectly by giving the name of the variable; the same name may correspond to different values at different times if the variable has been altered in the meantime.

A literal is a value that is represented directly as itself in the text of the program, and is thus always the same value whenever the command containing it is obeyed. In BASIC there are just two types of literal: number and string, as in

```
LET x = 42
LET q$ = "What?"
```

Numbers are of course in decimal notation, and can be a whole number as in the example above, or with a decimal point as in 3.162, or in what is often called 'scientific notation' in which a letter E is used to mean 'times ten to the power'. Ten to the power *n* is the number which is written as a 1 with *n* noughts after it, and ten to the power $-n$ is written as a 1 with *n* noughts before it and a decimal point after the first nought. Thus 5.4E3 is 5.4×1000 or 5400, and 47E-6 is 47×0.000001 or 0.000047. Table 3.1 shows the correspondence between powers of ten and the prefixes used in metric and SI units, so for instance 1.25 centimetres is 1.25E-2 metres and 94 MHz is 94E6 Hz.

Table 3.1 SI prefixes

d	deci-	E−1	da	deka-	E1
c	centi-	E−2	h	hecto-	E2
m	milli-	E−3	k	kilo-	E3
				myria-	E4
μ	micro-	E−6	M	mega-	E6
n	nano-	E−9	G	giga-	E9
p	pico-	E−12	T	tera-	E12
f	femto-	E−15	P	peta-	E15
a	atto-	E−18	E	exa-	E18

Literal strings are always enclosed in quotes. These notations are used in almost all computer languages.

So far, then, we have seen how a variable can store numbers and strings that we have typed in and copies of things that have already been stored in other variables. But a computer ought to be able to compute, or derive, new values from existing ones, and the notation used for this is called an 'expression'.

Expressions are built up from arithmetic and similar operations; the symbols that show what operations to do are called 'operators' and the values on which the operations are done are called 'operands'. For instance in the expression $x+3$ the operator is the plus sign and the operands are the value of the variable x and the literal value 3. The operation to be done is adding these two values together, and the value of the whole expression is the result of this operation, so if, say, the value currently stored in x is 5 then the value of the expression is 5+3, or 8.

An expression can have more than one operator, as in

```
2 + 3 * 5
```

but we have to decide just what the operands are. Is 2+3 an operand of the multiply operator, so that the value is 5×5 or 25 or is 3*5 an operand of the addition operator so that the value is 2+15 or 17?

Some languages do all the operations in the order in which they appear, from left to right, so that in the example the addition is done first and the result, 5, is an operand of the multiplication. One or two do them in order from right to left, but most (including BASIC) use the concept of 'priority'. Each operator has a 'priority', and the operators with the highest priority are done first. Operators with equal priority are done from left to right. Multiplication has a higher priority than addition, so 3*5 is worked out first and the result, 15, is then an operand of the plus sign and the value of the expression is 17.

Just as in ordinary arithmetic or algebra, parentheses can be used to group things together into a single operand, as in

```
( 2 + 3 ) * 5
```

OPERATOR — a symbol or identifier indicating an arithmetic or other operation, similar to a function as far as the computer is concerned. The parameters are called 'operands' and do not have to be written in parentheses.

where the expression in parentheses is an operand of the multiplication and the value of the whole expression is 25. Obviously

```
3 ) * 5
```

is not a valid expression and so cannot be an operand of '+'.

The priorities are mostly arranged so that an expression without any parentheses is evaluated in a way that will appear sensible to the programmer. Multiplication and division have a higher priority than addition and subtraction so that an expression such as

```
a * x + b - c / x
```

is evaluated in the same way as the corresponding algebraic expression

$$ax + b - c/x$$

There are a number of other operators, mostly to do with comparing values to see whether one is bigger than the other etc. and with combining the results of several such comparisons, and their priorities are listed near the end of the manual that comes with the computer.

There is never any harm in using parentheses to show in what order the operations should be done even where they are not strictly needed (except, perhaps, if the computer is so full that there is not room for them). Therefore, when in doubt (and where someone reading the program might be in doubt) parenthesise.

A few languages (notably POP-2 and Forth) and some calculators use 'reverse Polish notation' for expressions. Polish notation was invented for use particularly in formal logic and put the operator first, as in

```
+ 2 * 3 5      or      * + 2 3 5
```

meaning respectively 'add 2 to the result of multiplying 3 by 5' and 'multiply the result of adding 2 to 3 by 5'. Reverse Polish puts the operator last as in

```
2 3 5 * +      or      2 3 + 5 *
```

which is more convenient for computers. Both forward Polish and reverse Polish have the advantage that neither brackets nor priorities are needed, the order in which operators and operands appear defines uniquely in what order the operations are to be done and what their operands are.

Operators such as plus and multiply are called 'binary' operators because they have two operands. There are also 'unary' operators that have just one operand; they are also called 'prefix' operators if they are written before the operand and 'postfix' operators if they are written after it. Binary operators (except in Polish notation) are also called

PREFIX OPERATOR – an operator which precedes its operand, such as '–' or 'NOT'. In ZX BASIC, 'LOG', 'COS', etc., are prefix operators whereas in most BASICs they are functions.
POSTFIX OPERATOR – an operator which follows its operand or operands. In reversed Polish notation, all operators are postfix operators.

infix' operators because they are written between their operands.

BASIC does not have any postfix operators; an example of a postfix operator in ordinary mathematics is the exclamation mark used to indicate 'factorial' as in '4!' which means 'factorial 4' or 4×3×2×1. In a programming language that recognised measurements such as length and weight as data types in their own right (in which case a computer would be able to be more helpful in the example involving steel bar earlier in this chapter) postfix operators such as 'ft' and 'lb' might be defined to convert ordinary numbers into a measure of length etc. However, no such language is available at the present time.

ZX BASIC has two prefix operators of the ordinary mathematical kind: minus and NOT. (The latter is used in Boolean arithmetic, described in Chapter 4.) There are also many prefix operators which are 'functions' such as SIN and COS and LOG: in most BASICs these are written with their operand in parentheses rather like array elements, as in

```
SIN(x)          or      LOG(COS(x))
```

but in ZX BASIC they have the status of operators so that the rather more natural form

```
SIN x           or      LOG COS x
```

can be used.

Prefix operators need to have a priority just as infix operators do, because

```
SIN x + y   could mean   SIN (x+y)

or   (SIN x) + y
```

and in fact most of the prefix operators have a priority higher than any infix operator so that it is the latter meaning that is used. One exception is NOT, which although it has a higher priority than the infix Boolean operators comes below the other infix operators for a reason that will become clear in Chapter 4, and the other is minus, which comes below 'to the power' so that for instance $-x \uparrow 2$ means $-(x \uparrow 2)$ or $-x^2$ and not $(-x) \uparrow 2$ which would be $(-x)\times(-x)$ or $+x^2$

GETTING THE ANSWERS OUT

We have now seen how the computer can have values specified to it, calculate other values from them, and store all the values away in variables. To use the computer as a simple calculator we clearly need one more facility namely the ability to display the values so that the user can read them, and this is done by the PRINT command.

The PRINT command simply consists of the word PRINT (to see the results on the TV screen, LPRINT to see them on the printer) followed by the values you want printed, separated by semicolons. String values are displayed by sending to the screen (or printer) the characters they contain. Each character can be thought of as representing a key being pressed on an electric typewriter, most are 'printing characters' and cause the relevant character to be printed and the 'print position' (where the next character will

INFIX OPERATOR = an operator which has one operand before it and another after it. The 'four functions' add, subtract, multiply, and divide in normal arithmetic are examples of infix operators.

go if it toos a printing character) to be moved on one place, but others include 'space' (which moves the print position on without printing anything) and, on the Spectrum, codes that alter the colour etc. of following printing characters.

Numbers are converted into strings, and the resulting string values are treated as above (There is also a prefix operator STR$ that does the same conversion if you need it within a program.) Unlike some BASICs ZX BASIC does not include any 'space' characters in the string into which a number is converted, so for instance

```
LET x = 4
LET y = 2
PRINT x;y
```

and

```
LET z = 42
PRINT z
```

both print the same thing. If a comma is used instead of the semicolon, at least one space will be inserted between the numbers, in fact sufficient spaces to bring the print position to the centre of the line or the beginning of the next line (whichever comes first). Often there will be text to insert between two numbers anyway, as in

```
PRINT n;" eggs at ";c;"p/doz cost ";
n*c/12;"p"
```

which raises a number of points worthy of remark Spaces are included in the pieces of text to separate the numbers from the words, so that if, say *n* is 6 and *c* is 78 the output reads

```
6 eggs at 78p/doz cost 39p
```

and not (for instance)

```
6eggs at78p/doz cost39p
```

Although this may be obvious looking at the example outputs here, it is surprising how often it is forgotten when a PRINT command is being written The reason ZX BASIC does not insert spaces in numbers is that otherwise the output might look like the following

```
6 eggs at          78 p/doz cost        39 p
```

and inserting spaces in PRINT output is easier than taking them out All the above formats are preferable to

```
number of eggs       =   6
cost (p/doz)         =  78
total cost (pence)   =  39
```

which may have been appropriate in the days of the punched card tabulator and is far too prevalent today. Finally, note that (unlike Fortran, for instance) BASIC allows calculated values as well as simply values from variables in a PRINT command, and the above example is more efficient (in space to store the program in time to obey it, and in reducing the amount of 'clutter' the human reader has to contend with) than

```
LET total = n*c/12
PRINT n;" eggs at ";c;"p/doz cost ";total;"p"
```

which uses two commands instead of one and introduces an extra variable.

4

INPUT AND DECISIONS

In Chapter 3 we considered the commands LET and PRINT which allow the computer to be used in much the same way as a desk calculator. That is, calculations (which are typed in in the form of expressions) can be performed and the results displayed on the TV screen (with PRINT) or on the printer (with LPRINT, which is exactly like PRINT) or stored away for later use (with LET).

Programs consist of sequences of commands; there are of course commands other than LET and PRINT and LPRINT, many of which are introduced later in this section and all of which are described fully in the manual that comes with the computer. The program is built up from 'lines', on the ZX81 there is one command to each line, but on the Spectrum several commands can be written on one line, separated by colons. Each line has a *line number* which shows where it goes in the program; if you type in a line without a line number it is obeyed immediately and then thrown away, but a line that begins with a number is added to the program and is not obeyed at this stage.

The manual describes the details of how programs are typed in. The screen is divided into a top part, which is a 'window' through which the program can be seen, and an area at the bottom in which the line currently being typed is displayed. Therefore, when the program is changed (by inserting or removing or replacing a line) the result of the change is immediately apparent. This contrasts with many BASICs in which the screen merely shows a record of the lines that have been typed recently, although a 'listing' of the program can be displayed in response to a command. Some computers permit this listing to be altered without necessarily doing the corresponding alterations to the program.

A further feature of ZX BASIC is that lines are entered into the program only if their syntax is correct, whereas other BASICs will accept anything and only check it when the program is run. Syntax is concerned with the way words etc. are put together to form commands, semantics are concerned with the meaning of the command and semantic correctness usually depends on the context within which the command is obeyed. Thus

```
LET x$ = SQR y   and   LET > THEN IF
```

are syntactically incorrect and would be rejected, while

```
LET x = SQR y
```

is syntactically correct and is accepted into the program, but may fail at run time if the value of y is negative or if y does not exist at all. This still applies even if the semantics do not in practice happen to depend on the context, so that

```
LET x = SQR -1
```

is accepted as syntactically correct even though it will always fail at run time.

Natural languages such as English also have syntax and semantics. For instance 'The fat cat on the mat' is syntactically not a sentence because it does not include a verb. 'The green sky is sleeping furiously' is syntactically correct but semantically does not appear to make sense

The computer does not reject incorrect commands it does not understand because it gets a pedantic satisfaction from making you do it again properly, but because any sense that it might make of the command might not be what you intend, for instance

```
LET x$ = SQR y
```

could have been typed in mistake for

```
LET x$ = STR$ y   or   LET x$ = STR$ SQR y
or  LET x = SQR y
```

and the computer is not intelligent enough to be able to decide that any particular one is more likely.

In the same way that the text of this book is divided into paragraphs, it is helpful if a program can be divided visually into separate sections. The REM command (short for 'remark') is provided for this purpose. It instructs the computer to ignore the rest of the line, which can then be used for a description of what the following piece of code does, for the benefit of the human reader. (On the Spectrum,

SYNTAX – the structure of a language, which tells you the context within which to interpret the various words that appear in a statement (cf. 'parse'). For instance, the syntax of 'The cat ate a mouse on the mat' shows which of the things appearing in the sentence (the cat, the mouse, the mat) did the eating and which it was that got eaten.
SEMANTICS – the meanings of individual words and phrases in a language, within the framework defined by the syntax, qv. The syntax defines that the cat ate the mouse, and the semantics defines what a cat is, and what a mouse is.

note that it ignores the whole of the rest of the line, even though if you type a colon as part of the remark it goes into K mode as if it was expecting another command.)

On the Spectrum, a blank line can be inserted into the program by typing a line consisting of the line number and a space. (There has to be something following the line number because otherwise the line with that number (if any) will be deleted from the program and nothing will be inserted.) On the ZX81 the nearest thing to a blank line is a line containing the keyword REM and nothing else.

Unfortunately, even by being careful to lay out the program in a 'well-structured' way we cannot entirely overcome BASIC's essentially 'write-only' nature (see Chapter 1). But if we cannot make it easy for other people to understand how a program works we can still make it easy for them to use it without needing to see the command lines from which it is made up at all.

Often two quite separate people are concerned with a program, the *programmer* who writes it and the *user* who will run it. The user simply wants his data processed, and is concerned only with the data values and not at all with the variables in which the programmer chooses to store them. Moreover he will not want to type any more than is necessary.

The INPUT command is the means whereby the user enters data into the computer. With the LET command, the programmer is able to assign a value to any variable at any time; the programmer controls the order in which the assignments are done, and must be assumed to understand the effects of them. With the INPUT command, it is still the programmer who decides what variables are to be assigned to and at what stage, it is however the user who provides the values to be assigned.

An INPUT command consists of the keyword INPUT and a variable (although on the Spectrum more elaborate forms are also possible), and the effect is just the same as a LET command except that the value is typed in by the user instead of being included as part of the command. In most BASICs (and indeed in most other computer languages) the input value must be a literal, but in the ZX BASIC it can be anything that could appear to the right of the equals sign in a LET command. This is useful for typing in values such as PI/2 and in circumstances where the user knows the names of some of the variables available at that point in the program but it also permits cheating in programs that give practice in arithmetic by asking the answers to sums (This problem can be avoided on the Spectrum by using INPUT LINE, checking that the line contains nothing but digits and then using VAL to convert it to a number; ordinary string input will not do because the user can rub out the quotes and use STR$. Of course in many circumstances where this kind of program is used the pupil will have nothing to gain by such cheating anyway.)

When the INPUT command is obeyed, the user is presented with the 'L' cursor at the bottom left of the screen, enclosed in quotes if a string is required or on its own if a number is required It is easy for the programmer, who is familiar with the way the program works, to forget that the user will often not have any idea what the computer is asking and hence not know what to reply. It is therefore important to get into the habit of *always* putting a message on the screen indicating what value is being requested On the Spectrum this can be done as part of the INPUT command but on the ZX81 it must be done by preceding the INPUT command with a PRINT command. It is important to word the message in terms that the user will understand. Suppose a variable *r* holds the percentage rate of interest payable on an investment a suitable message when INPUTting *r* is 'Type in the percentage rate of interest per annum' or 'Interest rate (percent per annum)?' and definitely not 'Value of *r*?' even though this is how the programmer thinks of it.

DECISIONS

When the computer is being used as a calculator, commands are typed in one by one and each command is obeyed before the next one is typed. There is no clear distinction between the roles of the programmer who decides what commands are to be obeyed by the computer, and the user, who supplies the data for those commands. In particular, results from earlier commands, or data not directly used in the calculations, may influence the choice of which commands are given subsequently. For instance, income tax might be deducted at different rates depending on the total amount of income, interest on different kinds of bank account may be calculated in different ways.

Where the action to be taken by the computer depends on things that will not be known until the program is run, the programmer has to allow for all the possible circumstances. (Programs often fail because a combination of circumstances arises that the programmer did not think of.) The IF command is used to switch the computer to one course of action or another depending on a value which it calculates. This is only useful if the calculated value is not known when the program is written: one example of this is a value which is input by the user when the program is run (or is calculated from one that is). Another very common example occurs in a 'loop', which is a sequence of commands that is obeyed repeatedly until some condition is fulfilled: at the end of the sequence the computer tests to see whether it should repeat it or go on to the next part of the program. This is a little different from the case where the programmer specifies alternative courses of action, one or the other of which will be taken depending on the circumstances, because both courses are taken when the program is run, but at different times.

The IF command takes the form

```
IF condition THEN line
```

in which 'line' represents anything that could be a command line (though without a line number, of course). If the condition is satisfied, the line following THEN is obeyed, otherwise it is ignored, just as a line starting with REM is ignored. For instance

```
IF x=y THEN LET x=5
```

tests if the values of *x* and *y* are equal: if so, 5 is assigned to *x* but if not the LET command is ignored and *x* retains its former value. On the Spectrum, where several commands can be put on one line, note that the whole line is ignored if the condition is not satisfied; most of the BASICs that allow several commands on a line do this, but there are a few which only ignore the first command after the THEN, so that

```
IF x=y THEN LET x=5: LET y=4
```

would assign 4 to *y* whether or not *x* and *y* were equal. This is rather like the priority of operators in expressions – on the Spectrum the colon has a higher priority than THEN, but in some BASICs it has a lower priority.

Some BASICs, and many other languages, allow an ELSE part which is obeyed only if the condition is not satisfied, as in

```
IF x>0 THEN LET y=x ELSE LET y=-x
```

but ZX BASIC does not.

The condition usually takes the form

```
value  comparison  value
```

in which the two values are of the same type (numbers or strings) and the comparison operator is one of

```
=    >    <    >=    <=    <>
```

meaning respectively 'equal to', 'greater than', 'less than', 'greater than or equal to', 'less than or equal to' and 'greater than or less than.' The last three may also be thought of as 'not less than', 'not greater than', and 'not equal to'.

It is convenient to think of the comparison operators as yielding a 'Boolean' value – TRUE or FALSE – the former if the condition is satisfied, the latter if it is not. (More is said about Boolean arithmetic later in this chapter.) Thus $x=5$ is TRUE if *x* holds the value 5 and FALSE if *x* holds any other value, $x>3$ is TRUE if *x* holds a value which is greater than 3 (such as 4 or 597.6 or 3.000001), and FALSE if it holds the value 3 or any value less than 3 (such as 0 or

2.9 or −47). If the condition is TRUE the part after THEN is obeyed, if the condition is FALSE it is ignored.

The comparison operators can appear in expressions in the same way as ordinary arithmetic operators; the comparison operators have a lower priority so that expressions such as

```
x*y > n+5
```

have the obvious meaning.

In some languages a separate data type is used for Boolean values, but in ZX BASIC they are stored as numbers, with zero representing FALSE and any other value representing TRUE. Any Boolean value generated by the computer uses 1 to represent TRUE.

Boolean values can be stored in numeric variables just like any other numbers, so we can write

```
LET smaller = x<y
LET p = x=3
LET p=x=3
```

to assign 1 to the variable *smaller* if the value in *x* is less than that in *y* and 0 otherwise, and to assign 1 to *p* if *x* holds the value 3 and 0 otherwise. The third line of course means the same as the second as far as the computer is

BOOLEAN – Boolean arithmetic uses the two 'truth values' *true* and *false* instead of the numbers used in the algebra with which most people are more familiar. It is thus helpful for describing calculations on values that can be stored in a single bit.

concerned, but a person reading it is more likely to be misled into thinking that it sets both p and x to the value 3.

Using these variables we can write commands like

```
IF smaller THEN PRINT x;" is smaller"
IF p THEN LET x = y+5
```

and in fact between the IF and the THEN we can use anything that has a numeric value — if the value is zero the part to the right of the THEN is ignored, if the value is nonzero it is obeyed.

In ordinary mathematics, we are used to writing things like

$$1 \leqslant x \leqslant 5$$

to mean x is greater than or equal to 1 and less than or equal to 5, i.e. x is in the range 1 to 5 inclusive. In ZX BASIC we can write

```
1 <= x <= 5
```

and we might expect it to mean the same, but on closer consideration this turns out not to be so. The two operators are of the same priority so are done from left to right. Therfore the value of '1 <=x' is worked out as either TRUE (1) or FALSE (0), which then goes on to be the left operand of the second '<=' operator. The expression thus reduces to either '0 <=5' or '1 <= 5', and in either case yields TRUE! So

```
IF 1 <= x <= 5 THEN
     PRINT "Value is in range"
```

always prints the message 'Value is in range' whether x is in the range 1 to 5 or not.

A final caveat on the comparison of numbers concerns values that have been derived from calculations in which some of the numbers used were not stored exactly in the computer. The only numbers that are stored exactly are whole numbers less than about 4 000 000 000 and numbers (within the range the arithmetic allows) that are the result of multiplying or dividing these by powers of two, viz. by 2, 4, 8, 16, 32, 64, etc. Furthermore, literal numbers written with a decimal point or an E are always liable to be inaccurate because of the method the BASIC uses to convert them from the decimal form in which they are typed to the binary form used inside the computer.

The comparison operators work by subtracting one operand from the other and seeing whether the result is zero or positive or negative. The details of how this is done, in particular the 'rounding' of operands and results to 32 bits or about 9½ decimal digits, have some very curious effects. For instance the literal 0.25 is converted into binary by working out $(2+5\times0.1)\times0.1$ but the constant 0.1 cannot be represented exactly in binary and the final result is about 5×10^{-11} less than one-quarter. The division sum ¼ is however worked out exactly. It happens that when one of these numbers is subtracted from the other the result is rounded up by the same amount of about 5×10^{-11}, so that ¼−0.25 yields about 10^{-10} while 0.25−¼ yields 0. It follows that 0.25=¼ is TRUE but ¼=0.25 is FALSE!

In any programming language it is always wise to allow for the possibility that numbers (other than whole numbers below a certain size) may be inexact, and for instance to replace

```
IF x=y THEN ...
```

by something like

```
IF ABS (x-y) < 1E-8 THEN ...
```

if x and y are known to be between about 0.5 and 10, or

```
IF ABS (x-y) < 1E-8 * ABS x THEN ...
```

if we have little idea what size they will be.

As well as using Boolean values in IF commands we can use them in Boolean arithmetic, which was invented by the French mathematician George Boole. Remember that we have just two values, zero representing FALSE and any value greater than zero representing TRUE; we will assume for the moment that none of the values we encounter will be negative.

If x and y are two such values then $x \times y$ is zero if either x or y is zero; if both are nonzero then $x \times y$ is nonzero also. Similarly $x+y$ is zero if x and y are both zero but if either is greater than zero then (because neither of them is negative) the sum is greater than zero also.

In Boolean arithmetic the multiply operator is also called AND, so that x AND y is TRUE only if x and y are both TRUE; the addition operator is also called OR because x OR y is TRUE if x is TRUE or y is TRUE, or both are.

Although Boolean arithmetic would work satisfactorily in most cases using the ordinary arithmetic add and multiply operators, as in

```
IF (y>1) + smaller * (p<3) THEN ... ,
```

the operators AND and OR as in

```
IF y>1 OR smaller AND p<3 THEN ...
```

are also provided. These have the correct effect even if their operands are negative or so large that adding or multiplying would result in a value too large to store in the computer.

The third Boolean operator that is available in ZX BASIC is NOT, which is a prefix operator with only one operand. NOT *x* has the value 1 if *x* is 0, and 0 otherwise, so NOT *x* is TRUE if, and only if, *x* is not TRUE.

The priorities of the Boolean operators are chosen as follows. AND has a higher priority than OR in the same way that multiply has a higher priority than add, so that the operations are grouped in the same way in each of the two examples above. NOT, being a prefix operator, has a higher priority than either of these, but all the Boolean operators have a lower priority than the comparison operators so that the parentheses can be omitted in the second example above.

A somewhat devious use of AND allows the equivalent of the 'conditional expressions' that are provided in some other languages. The value of *x* AND *y* is in fact the same as *x* if *y* is TRUE, and zero if *y* is FALSE. Thus for instance the value of

```
(a+b AND q) + (b*c AND NOT q)
```

is $(a+b)+0$ if *q* is TRUE, and $0+(b^*c)$ if *q* is false, so that the whole expression has the effect of

```
IF q THEN a+b ELSE b*c
```

The left operand of AND (though not of OR) may also be a string, in which case if the right operand is FALSE the value of the expression is the empty string (a string with no characters in it). The '+' operator between two strings means 'concatenate', and concatenating the empty string (like adding zero to a number) has no effect. Therefore exactly the same construction may be used as with numbers; for example

```
(a$ AND x=y) + (b$ AND x<>y)
```

has the effect of

```
IF x=y THEN a$ ELSE b$
```

A slightly more elaborate example is

```
("greater" AND x>y) + ("less"
AND x<y) + ("equal" AND x=y)
```

Unlike the conditional expressions in most languages, it is not necessary to have the ELSE part, so we can write

```
PRINT x;" is ";"not " AND x<>y;"equal to ";y
```

and

```
PRINT x;" inch";"es" AND x<>1
```

in ZX BASIC.

5

ARRAYS AND STRINGS

In Chapter 3 we saw briefly that an 'array' is a group of variables all of the same type, collected together under one name. The invidual variables are called 'elements' of the array and are selected by a number called an 'index' or 'subscript'; it is called a subscript because in ordinary mathematical notation a subscript would be used, as in v_3 or b_n, but on the computers that run BASIC we cannot write subscripts in this form so it is put in parentheses instead, as in $v(3)$ or $b(n)$.

Each array is said to have a particular number of 'dimensions'. For a one-dimensional array we imagine all the elements being laid out along a line, for a two-dimensional array they form a rectangle, for a three-dimensional array they form a cuboid. For four or more dimensions we need to go into hyperspace, but the idea is the same. The 'dimensions' of a three-dimensional array, for example, are the length and width and height of the cuboid, i.e. the number of elements in each direction.

The DIM command is used to create (i.e. reserve space for) an array, and specifies its dimensions. Thus

```
DIM v(10)
```

sets up a one-dimensional array v with elements $v(1)$, $v(2)$, . . . $v(10)$ and

```
DIM c(4,3,2)
```

sets up a three-dimensional array with elements $c(1,1,1)$, $c(1,1,2)$, $c(1,2,1)$, $c(1,2,2)$, $c(1,3,1)$, $c(1,3,2)$, $c(2,1,1)$, ..., $c(4,3,2)$. Elements are always numbered from 1 upwards in each dimension.

If the name of the array ends in a dollar sign, the elements of the array are characters instead of numbers, so

```
DIM c$(4,3,2)
```

sets up a three-dimensional array of characters. It can also however, be used as a 4×3 (and hence two-dimensional) array of strings, each string being exactly two characters long (because in this case the last dimension of the character array is 2).

The maximum size of an array depends on the amount of storage available in the computer. An array of numbers takes four bytes for the name etc plus two for each dimension and five for each element, an array of characters is similar except that only one byte is taken for each element. For example:

DIM a(10)	4 + 2×1 + 5×10 =	56 bytes
DIM a$(10)	4 + 2×1 + 1×10 =	16 bytes
DIM a$(6,100)	4 + 2×2 + 1×(6×100) =	608 bytes
DIM a(3,3,3,6)	4 + 2×4 + 5×(3×3×3×6) =	822 bytes
DIM a$(6,6,6,6,6)	4 + 2×5 + 1×(6×6×6×6×6) =	7790 bytes
DIM a(9,9,9,9)	4 + 2×4 + 5×(9×9×9×9) =	32817 bytes

If the other things competing for space (the program, other variables display file, etc.) are all quite small, then the approximate amount of space available is as shown in Table 5 1 As an indication of what can be stored in it, the table also shows the dimensions of a two dimensional numeric array of approximately this size.

Table 5.1. Storage space

Computer	Total RAM	Bytes available	2-D array
ZX81 (European)	1K	800	(12, 13)
TS1000	2K	1800	(12, 30)
ZX81 + 16K RAM pack	16K	15400	(30, 100)
Spectrum	16K	8700	(30, 58)
Spectrum	48K	41500	(83 100)

As an example of how an array might be used in a program, suppose you want to store the pattern of colours on a Rubik's cube.

```
DIM c(6,3,3)
```

creates an array *c* in which you can store the 3×3 pattern of colours on each of the six faces: *c*(1,1 1) might be at the top left of the front face *c*(1,1,3) at the top right, *c*(1,3 2) centre bottom, and so on. How you choose the numbers and orientations of the other faces may well make a substantial difference to how easy it is to write routines to do the various rotations that are possible with the real cube in the worst case you would have to write a separate piece of code to deal with each face.

It is sometimes convenient to use an array as a 'look-up table', where the translation from one value to another cannot be calculated easily using the normal arithmetic operations and functions For instance, suppose you decide to number the cube as follows:

top (3,1,3) (3,2,3) (3,3,3)
(3,1,2) (3,2,2) (3,3,2)
left side (3,1,1) (3,2,1) (3,3,1) right side

(2,3,1) (2,2,1) (2,1,1) (1,1,1) (1,1,2) (1,1,3) (5,3,3) (5,3,2) (5,3,1)
(2,3,2) (2,2,2) (2,1,2) (1,2,1) (1,2,2) (1,2,3) (5,2,3) (5,2,2) (5,2,1)
(2,3,3) (2,2,3) (2,1,3) (1,3,1) (1,3,2) (1,3,3) (5,1,3) (5,1,2) (5,1,1)

(4,3,3) (4,3,2) (4,3,1)
bottom (4,2,3) (4,2,2) (4,2,1)
(4,1,3) (4,1,2) (4,1,1)

(6,3,1) (6,2,1) (6,1,1)
back (6,3,2) (6,2,2) (6,1,2)
(6,3,3) (6,2,3) (6,1,3)

Then if you look at any face *f* with (*f*,1,1) in the top lefthand corner the colours on that and the adjacent faces are stored in the following elements of *c*:

(*t*,1,1) (*t*,2,1) (*t*,3,1)

(*l*,1,1) (*f*,1,1) (*f*,1,2) (*f*,1,3) (*r*,3,3)
(*l*,1,2) (*f*,2,1) (*f*,2,2) (*f*,2,3) (*r*,2,3)
(*l*,1,3) (*f*,3,1) (*f*,3,2) (*f*,3,3) (*r*,1,3)

(*b*,3,3) (*b*,3,2) (*b*,3,1)

The values of *t*, *l*, *b*, and *r* are different for each *f*. For instance if *f*=1, indicating that you are looking at the front face, then *t*=3, *l*=2, *b*=4, and *r*=5. If *f*=2, so you are looking at the left side, then *t*=1, *l*=3, *b*=6, and *r*=4. Note that you had to look at it sideways to get the (2,1,1) element in the top lefthand corner. If we store the values of *t*, *l*, *b* and *r* for each *f* in four 6-element arrays, we can find them easily when required for operations such as rotating one face, and all the faces can be dealt with by the same piece of code

On the Spectrum we can write

```
10  DIM t(6): DIM l(6): DIM b(6): DIM r(6)
20  FOR i=1 TO 6
30  READ t(i), l(i), b(i), r(i)
40  NEXT i
50  DATA 3,2,4,5, 1,3,6,4, 2,1,5,6, 6,5,1,2,
         4,6,3,1, 5,4,2,3
```

but on the ZX81 the DIM commands have to be on separate lines and READ and DATA are not available. (Chapter 22 of the ZX81 manual shows a way round this)

Once these arrays are set up we know that the colour above $(f, 1, j)$ is $c(t(f), j, 1)$, for instance, and that to the right of $(f, i, 3)$ is $c(r(f), 4-i, 3)$

YOU DO NOT ALWAYS NEED ARRAYS

There are, however, situations in which an array is not the right way to deal with repetitive data

Suppose we have to find the mean of a list of numbers x_1 to x_n, which is the result of adding all the numbers together and then dividing by n A common approach to this is to divide the program up as

read in the numbers
calculate the mean
write out the answer

and the program might go something like

```
 10  PRINT "How many numbers? "
 20  INPUT n
 30  DIM x(n)
 40  PRINT "Now type in the numbers"
 50  FOR i=1 TO n
 60   INPUT x(i)
 70  NEXT n
100  LET sum = 0
110  FOR i=1 TO n
120   LET sum = sum+x(i)
130  NEXT i
140  PRINT "Mean is ";sum/n
```

This program has two FOR-loops in it, but actually there is no reason why we cannot amalgamate them and replace lines 50 to 130 by

```
 50  LET sum=0
 60  FOR i=1 TO n
 70   INPUT x(i)
120   LET sum = sum+x(i)
130  NEXT i
```

But now look at what is happening: the program reads the first number into *x*(1) and then adds it on to *sum*. It never uses *x*(1) again but goes on to read the next number into *x*(2), and so on. Each time round it uses a new element of *x* which it then never uses again, so it could instead have just one variable and use it over and over again, as in

```
 50  LET sum=0
 60  FOR i=1 TO n
 70   INPUT x
120   LET sum = sum+x
130  NEXT i
```

Now we do not need line 30 either; if n is at all large, a great deal of space has been saved. This version of the program does a small amount of calculation between reading one number and asking for the next but the time it takes to do this is not likely to be noticeable. The original version did much of its calculation after the last number has been input, and if n is large there may well be a noticeable pause between typing in the last number and seeing the answer on the screen.

Sometimes a bit of algebraic manipulation of the original problem can make the programming more efficient. If we also want to work out the standard deviation s, using the formula

$$s^2 = sumsqdev / (n-1)$$

where *sumsqdev* is the sum from 1 to n of $(x_n - mean)^2$, then we can only begin to calculate it after we have worked out the mean, and we therefore need to store all the numbers (in the array x) in order to work out *sumsqdev* at the end of the program. But note that

$$(x_n - mean)^2 = x_n^2 - 2 \times x_n \times mean + mean^2$$

so the formula can be rewritten as

$$s^2 = (sumsq - 2 \times sum \times mean + n \times mean^2)/(n-1)$$

where *sumsq* is the sum of x_n^2, which can of course be calculated as the numbers are being read in. Replacing *mean* by *sum/n* we get

$$s^2 = (sumsq - 2 \times sum^2 / n + n \times sum^2 / n^2) / (n-1)$$

which reduces to

$$s^2 = (sumsq - sum^2/n) / (n-1)$$

Here is the program modified to work out the standard deviation as well. It's also modified to avoid asking the user how many numbers there will be, and to echo the numbers on the screen so that the user can always see the last 20 or so numbers he typed. It is written for the Spectrum; the main modification required for the ZX81 is to insert a SCROLL command in front of each PRINT (except for those before line 100) and to change ENTER to NEWLINE in lines 40 and 50

```
 10 PRINT AT 10,0; "Mean and standard deviation"
 20 PRINT
 30 PRINT "Type each number terminated"
 40 PRINT "by ENTER."
 50 PRINT "Then type ENTER again."
 60 PRINT
 70 LET sum=0
 80 LET sumsq=0
 90 LET n=0
100 INPUT x$
110 IF x$="" THEN GOTO 190
120 LET x=VAL x$
130 LET n=n+1
140 PRINT n;"  ";x
150 LET sum=sum+x
160 LET sumsq=sumsq+x*x
170 GOTO 100
180    REM
181    REM here to print mean and s.d.
182    REM
190 IF n=0 THEN GOTO 280
```

```
200 PRINT
210 PRINT "         Mean "; sum/n
220 IF n=1 THEN GOTO 280
230 LET v = (sumsq - (sum*sum/n)) / (n-1)
240 PRINT "     Variance "; v
250 PRINT "Standard devn "; SQR v
260    REM now check this is really the end
270 PRINT
280 PRINT "Any more numbers? (reply Y or N)"
290 INPUT x$
300 IF x$="Y" OR x$="y" THEN GOTO 100
310 IF x$<>"N" AND x$<>"n" THEN GOTO 280
```

PROCESSING TEXT

The operations that are typically done on arrays of numbers are fairly familiar to most people: ordinary arithmetic on individual elements, and totalling rows and columns. The computer does these operations in very much the same way that people do them, but rather faster.

Operations on character arrays and strings are unfamiliar because the computer's view of a character string is quite different from a person's. The character string is converted into a string of numbers on which simple arithmetic operations are then done; this is a rather laborious method, but it is the only one available, and the computer's speed at doing the arithmetic that is involved makes it much less laborious than it would be for a person.

The numbers (or 'codes') into which the various characters are translated are listed in Appendix A of the ZX81 and Spectrum manuals. Because of the way the TV picture is made, the codes used in the 'display file' on the

ZX81 had to be the numbers 0 to 63 for ordinary characters and 118 for 'newline'. (The display file is the representation stored in the memory of the picture on the TV screen, on the ZX81 this consists of a list of character codes with a 'newline' code to mark the end of each line.) The codes were chosen in a way that seemed convenient. for instance the digit *n* has code 28 + *n* and the nth letter of the alphabet has code 37 + *n*. Again because of the way the hardware makes the TV picture, adding 128 to the code for a character produces that same character but in white-on-black instead of black-on-white.

The same codes that are used in the display file are used in the other places that characters are stored, for instance in the text of the program and in variables, and some of the codes that cannot be used in the display file are used to represent 'tokens' such as the keywords LET and PRINT and THEN, in particular, adding 192 to the code for a letter gives the code of the keyword that shares a key with that letter on the keyboard (for instance letter G is code 44, 44 + 192 = 236, and code 236 is GOTO which is on the same key as G).

The Spectrum display file does not store character codes directly and therefore does not restrict the choice of character codes. However, the Spectrum was always intended to support the serial interface add-on which allows data to be exchanged with other data processing equipment, and the Spectrum character codes have therefore been chosen to be, as far as possible compatible with devices using various international standard codes: ASCII and ISO-7 and the newer codes for videotex (also called viewdata), teletex (a kind of super-telex for word processors), and teletext (which is broadcast along with television pictures).

We have already seen how a string can be stored, printed out, and joined onto another string, but BASIC also allows you to dissect a string and look at the individual characters or groups of characters it contains. ZX BASIC's method for doing this is called 'slicing' in the manual; this is the name used for a similar facility for dissecting arrays in Algol 68, but it can be used on strings as well as on character arrays. Like the functions LEFT$, MID$, and RIGHT$ in other BASICs, it lets you select a part of the string starting in a specified place and of a specified length. However, a part of the string is specified purely in terms of the number of characters from the start of the string: if the string contains a sentence in English, say, then to select the second word you must first find where it is. The following piece of program assigns to *w$* the *n*th word of the sentence in *s$*

```
100  LET k=0
110  FOR i=1 TO n
120  LET k=k+1
130  IF s$(k)=" " THEN GOTO 120
140  LET j=k
150  LET k=k+1
160  IF s$(k)<>" " THEN GOTO 150
170  NEXT n
180  LET w$ = s$(j TO k-1)
```

Lines 120 and 130 move *k* on to the start of the word, which we remember as *j*, then lines 150 to 160 move it to the character after the end of the word. Starting from there, we search for the next word, and repeat the process until the nth word is found If we run off the end of the string, the program will stop with error code 3.

The operator LEN gives the length of a string, so line 160 above could have read

```
160  IF LEN s$>=k THEN IF s$(k)<>" " THEN GOTO 150
```

to prevent error 3 happening on the last word The operator VAL interprets the contents of a string as a numeric expression and yields its value and on the Spectrum VAL$ is also available, which does the same job for an expression that yields a string, but there is no facility to obey a whole command contained in a string.

(Note, by the way, that

```
160  IF LEN s$>=k AND s$(k)<>" " THEN GOTO 150
```

would still give error 3 at the end of the string because *s$(k)* would still get evaluated whatever the value of the left operand of AND.)

As was indicated in Chapter 2 examining the individual characters in a string is a long way from the kind of processing people do when looking at a piece of text; it is not possible to stand back and look at the whole string at one go The piece of program above dissects a string into individual words, but using a rather simple-minded definition of 'word' the kind of thing a user might type in real life is

```
"Fred,Joe and Jim."
```

so we need to extend the program to recognise "Fred" and "Joe" as two words (the program as written reckons the first word is "Fred,Joe") and to separate "Jim" from the full stop. On the Spectrum, we also have to cope with the fact that letters can be in upper or lower case: "THE" and "The" and "the" must be recognised as the same word even though they are different strings. If the program is to make any attempt to interpret a whole sentence written in English, it needs to have some kind of 'dictionary' to tell it the meanings of all the words the user might possibly type.

Because of the difficulty of writing a program that can interpret English sentences correctly, it is often better to present choices to the user in the form of a 'menu' rather than ask a question and attempt to interpret the answer. For instance, the program

```
10  PRINT "Where is the Vatican?"
20  INPUT c$
30  IF c$ <> "Rome" THEN PRINT "Wrong!"
```

will print 'Wrong!' if the user types any of the following

```
"ROME"   "Rome."   " Rome"   "It is in Rome."
```

although none of them could be considered to be a wrong answer. The program could be modified to make some attempt at picking out the word 'Rome' from these answers perhaps as follows (lines 30 to 50 are not needed on the ZX81 which does not have lower case letters)

```
10  PRINT "Where is the Vatican?"
20  INPUT c$
30  FOR i=1 TO LEN c$:
       REM convert to upper case
```

```
40  IF c$(i) >= "a" AND c$(i) <= "z" THEN LET
    c$(i) = CHR$ (CODE c$(i) - CODE "a" + CODE "A")
50  NEXT i
60  FOR i=1 TO LEN c$ - 3
70  IF c$(i TO i+3) = "ROME" THEN GOTO 100
80  NEXT i
90  PRINT "Wrong!"
```

This won't print 'Wrong!' if the reply contains the letters R,O M,E together anywhere in it, which is somewhat over-generous as it means that 'Cromer' would be taken to be a correct answer. We can allow for this by adding, perhaps

```
 95  GOTO 130
100    REM check "ROME" isn't part of a longer
       word
110  IF i>1 THEN IF c$(i-1) >= "A" AND
       c$(i-1) <= "Z" THEN GOTO 80
120  IF i < LEN c$ - 3 THEN IF c$(i+4) >= "A"
       AND c$(i+4) <= "Z" THEN GOTO 80
```

but this will still not trap

```
"50 miles north of Rome."
```

as a wrong answer. To be sure there is no confusion, the 'menu' approach is preferable, as in

```
10  PRINT "Where is the Vatican?"
20  PRINT
30  PRINT "1.  Florence"
40  PRINT "2.  Monte Carlo"
50  PRINT "3.  Norwich"
```

```
 60  PRINT "4.  Rome"
 70  PRINT "5.  Naples"
 80  PRINT
 90  PRINT "Type the number corresponding"
100  PRINT "to the correct answer."
110  INPUT city
120  IF city <> INT city THEN GOTO 80 :
        REM not a whole number
130  IF city < 0.9 OR city > 5.1 THEN GOTO 80 :
        REM not in range 1 to 5
140  IF city <> 4 THEN PRINT "Wrong!"
```

Restricting the user's choice is perhaps undesirable in this kind of quiz game (because if he just guesses he has one chance in five of being right), but in more typical situations where the program is asking the user what he wants it to do next it is usually helpful

6

PROGRAMS FOR OTHERS TO USE

So long as you are using your personal computer as a glorified calculator, or to produce pictures on the TV screen, or even for it to play simple games with you, it does not matter too much how your programs are written provided they fit in the amount of memory you have available and produce the required results. But if you are writing a large or complicated program, or one that other people will use, or one that someone else will later need to modify, or one in which it is important to be reasonably sure that the results correspond correctly to the input data, then there are a number of guidelines that should be followed.

Actually, if you are writing 'serious' programs you probably should not be using BASIC at all. Several languages are available on cassette for the ZX computers. If you buy one, make sure you know whether it implements the whole language or only parts of it; also check how much of the computer's memory it takes up. Preferably read the reviews in the microcomputer magazines. Remember that each time you switch off the computer you lose what is in the memory, so you will have to read the cassette in afresh each time you want to use it: only BASIC is in the machine when you switch on.

As the name, *Beginner's* All-purpose Symbolic Instruction Code, implies, BASIC is intended as a way of introducing people to computers and programming, in the expectation that they will later graduate to programming in other languages. But, as many people in the software industry have noted (not least the members of the Alvey committee which reported recently to the UK government on certain aspects of computing in the 1980s), BASIC can get you into some very bad habits

STRUCTURED PROGRAMMING

The larger a program is, the more difficult it becomes to keep track of the effect of any particular commands on the rest of the program, or the state of play as regards things that other parts of the program should be updating. Questions arise such as: Is it all right to use variable *j* or is there some other part of the program that has left something stored there which it is expecting to be able to retrieve later on? Has *nextvalue* (which is supposed to hold the value that the program will look at next) been updated yet, or are there circumstances in which it still holds a value that has already been dealt with? Can we print a message here without obliterating, or otherwise interfering with, something written by another part of the program?

Techniques for limiting these kinds of problem have two main components:

1 Break the program up into pieces of manageable size

It is not possible to be very precise as to how big is 'manageable': up to perhaps 40 or 50 commands in average circumstances, but a straightforward process which happens to require a lot of commands can take more; a complicated one should be limited to rather fewer

2 Use comments to show what each piece does, what resources it uses, etc.

There are usually things that are true throughout the program. and can be documented in a comment at the top of the program (if that does not take up too much memory) or outside the computer altogether (if you can be sure it will not get lost); for example in a program to play a board game such as chess there will probably be an array which holds the current position and a variable which shows whose move it is the documentation should define how the information is encoded in them Then when writing the part of the program that displays the board on the screen you need refer only to this documentation; it is not necessary to look at the part of the program that sets up the initial position nor at the part that updates the position when a move is made. Similarly, the comments on individual parts of the program can assume that you already know the information that is in this 'global' documentation and need not repeat it

Incidentally the design of just how the information is to be represented in the memory (called the 'data structure') is the most important part of most non-trivial programs Once this has been done, the program usually falls automatically into a number of sections each of which is concerned with updating a part of the data structure to take account of a change in the thing represented such as making a move in a chess game or adding a new transaction to a bank balance. If the data structure has been well designed these operations should be fairly easy. When designing the data structure you should always try to use a representation that will be convenient for the program In particular consider how you can avoid making the number of different operations (and hence the number of different sections in the program) unnecessarily large. In the chess game, for instance, do you just need one routine for 'make a move', or do you need separate routines for 'white's move' and 'black's move'? If you favour having just one, however, it is going to be significantly more complicated than either of the separate ones? If so, it might be better to use two after all.

During the 1970s the term 'structured programming' became current. This is a technique whereby you describe the task your program has to do in terms of lower-level tasks; the description should be of manageable size, i.e. less than a page. For the chess game this might be

1. set up initial position, set 'white to move', ask whether computer is to play black or white or both or (if two people are using it as a kind of electronic chess board) neither;
2. display board on TV screen, indicate whose move it is;
3. if the player cannot move, indicate 'checkmate' or 'stalemate' and go to 7;
4. if it is the computer's move, work out what the move should be; if the user's move, ask for the move to be input;
5. make the move, or if it's 'resigns' go to 7;
6. swap from 'white to move' to 'black to move' or vice versa, and go to 2;
7. show (on the screen) which player has lost, and ask whether another game is to be played; if so go to 1.

Each of these seven tasks is in turn described in terms of lower-level tasks, and so on until all the tasks have been defined as sequences of commands in BASIC (or whatever programming language is being used — but for the purpose of this book we will assume it is BASIC).

A feature of BASIC which very few other languages share is that each line of the program has a number, and you may as well make use of this to assist understanding of the program by using line numbers from 1000 up for step 1, 2000 up for step 2, and so on. If step 2 consists of six lower-level steps, these should start at 2100, 2200, . . ., 2600; line 2000 should contain a REMark indicating what step 2 does.

Some structured programming purists would insist on a more direct manifestation of the top level in the program, as in

```
10  GOSUB 1000
20  GOSUB 2000
30  GOSUB 3000: IF done THEN GOTO 70
40  GOSUB 4000
50  GOSUB 5000: IF done THEN GOTO 70
60  GOSUB 6000
62  GOTO 20
65
70  GOSUB 7000: IF another THEN GOTO 10
72  GOTO 9999
```

They would also insist on eliminating all GOTOs from the program. In the 'block structured' languages, particularly the newer ones, such as Algol 68, Pascal, BCPL, and C facilities are provided which can replace most uses of GOTO. In the example above, lines 20 to 65 would be bracketed together as a 'block' in some way (which depends on the language), the GOTOs on lines 30 and 50 would take the form of 'exit from the block' commands and that on line 62 would be shown as 'repeat the block'. Lines 10 to 70 would be another block (with the first block 'nested' inside it) with the IF . . . GOTO replaced by a command of the form 'repeat while (another)'. We can do some of these things in BASIC, as in

```
 10  FOR a = 0 TO 0 STEP -1
 15  GOSUB 1000
 20  GOSUB 110
 70  GOSUB 7000
 72  LET a = another:  REM go round again if TRUE
 75  NEXT a
 78  STOP
100
110  FOR b = 0 TO 1 STEP 0
120  GOSUB 2000
130  GOSUB 3000:  IF done THEN RETURN
140  GOSUB 4000
150  GOSUB 5000:  IF done THEN RETURN
160  GOSUB 6000
170  NEXT b
```

but this does not really seem to make what the program is doing any clearer.

Many of the advocates of structured programming concentrate on the elimination of GOTOs with almost religious fervour (sometimes actually quoting Genesis chapter 11 verse 7 in which, in the Authorised Version, God says 'Go to, let us go down, and there confound their language, that they may not understand one another's speech' as evidence that it is GOTOs that make programs incomprehensible). However, it is possible to find very clear and comprehensible programs that use GOTOs, and very obscure and muddled ones that do not.

But if you are writing in ZX BASIC you do not really have any sensible alternative to using GOTOs, and you should be aware of how to use them and how not use them.

In most languages, if you wish to GOTO a command that command must have a 'label'. A label is very much like the name of a variable, in that it identifies the part of the memory in which the line is stored; usually it has the same form as a variable name (letter followed by letters and/or digits) though in Fortran it is a number. Many compilers have the ability to generate a 'cross-reference' table which shows where each label is used. When you are looking at a piece of the program, either to check whether it is correct or to see whether a change you are proposing to make will upset something else, you can be sure (1) that the only ways into the piece of program are at the top and at each label, and (2) that each place from which it is entered can be found in the cross-reference table. You can therefore be certain of being able to check every context in which the piece of program can be used.

In BASIC, every line has a line number and is therefore potentially the target of a GOTO. No cross-referencer is provided in the standard firmware, although it would not be too difficult to write a crude one in BASIC. (Look in Chapters 27 and 28 of the manual in the case of the ZX81, and Chapters 24 and 25 in the case of the Spectrum, to find where to PEEK and what to look for there; in each case Appendix A tells you that the code for GOTO is 236.) The problem is made worse by the availability of commands like

```
GOTO (j+5)*100
```

which is liable to GOTO anywhere — remember that *j* is not necessarily a whole number nor is it necessarily positive; it could for instance be −3.53 in which case the command reduces to GOTO 147, and if there is no line 147 it will

GOTO the next highest line number.

It is therefore most important to put a REMark at any place that is liable to be GOne TO from any other part of the program. This also applies (indeed rather more so) to places that are the target of a GOSUB; in this case the REMark should make clear what the end effect of the GOSUB will be (i.e. what will have been done by the time the RETURN is reached).

Another criticism often levelled at GOTOs is that indiscriminate use leads to a program which, if you trace all the paths the computer can follow through it, looks like a plate of spaghetti. (It has also been said that the more extreme forms of 'structured programming', with their many separate layers of program, resemble a dish of lasagne, which is just as difficult to see through.) We have found that a good rule that helps avoid this kind of problem is:
Backwards jumps should only be used for loops

A backwards jump is one that GOes TO a command that precedes it in the program, such as

```
120  IF n<>0 THEN GOTO 100
```

(100 being before 120). A loop is a sequence of commands that is obeyed several times, for example

```
100  INPUT n
110  LET total = total+n
120  IF n<>0 THEN GOTO 100
```

in which lines 100 to 120 are obeyed repeatedly until a zero value is input. (We assume that a message such as 'type in the numbers, terminated by a zero' is printed before the loop is entered.) Clearly there has to be a backwards jump at some point in the loop (unless FOR ... NEXT is used, which is not very appropriate here). But the program should be arranged such that there are no backwards jumps except those from the middle or end of a loop back to the beginning. For example, suppose you have to take different action at a certain point if the variable *x* contains the value zero.

```
 240  IF x=0 THEN GOTO 2500
 260  [action if x is nonzero]
 280  [next part of program]
 ...
 ...
2500  REM here from 240 if x=0
2520  [action if x is zero]
2540  GOTO 280
```

This contains a backwards jump on line 2540, and we can see that if there are many sections like lines 2500 to 2540 scattered around the program it will have the spaghetti-like structure alluded to earlier. However, we can rearrange it as

```
240  IF x<>0 THEN GOTO 270
250  [action if x is zero]
260  GOTO 280
270  [action if x is nonzero]
280  [next part of program]
```

LOOP — a part of a program that is obeyed over and over again; also used as a verb, meaning to obey a sequence of commands repeatedly. A common consequence of a bug (q.v.) is that the program stays in a loop forever.

which has no backwards jumps, and has a much cleaner structure (similar in fact to the form the program would take in a 'GOTO-less' language) so that we can see what it does without having to keep track of odd bits of program in other parts of the listing.

RELIABILITY

The novice programmer is usually surprised the first time he lets someone else try out a program he has just written, to discover just how easily it can be made to fail and indeed how difficult it is for the guinea-pig user to get it to work at all

Usually the problem is that the user's input is not quite in the form that the programmer expected. Perhaps when asked to type a list of words he puts commas between them when the programmer expected spaces, or several spaces where the programmer expected just one. Perhaps he was not told that none of the words may be more than ten letters long. Perhaps he was asked for a number but not told that it must be a integer (i.e. a 'whole number') or less than a hundred, or greater than zero To the programmer, knowing how the program works such restrictions might be obvious, and it is sometimes difficult for him to remember that the user does not have this information. Often the reason that the program does not cater for a particular form of input is that the programmer would himself never think of using it anyway it would never occur to him that adjacent words in a list should be separated by anything other than a single space, so the program does not allow for several spaces, or a comma.

The programmer's defence against this problem is a kind of 'belt and braces' approach:

(1) Make sure that the user has been told exactly what the program expects.

(2) Make sure that the program can cope with any kinds of input, even those that are not in accord with the instructions given in (1).

There is a limit to the effectiveness of (1): users are often too eager to get on with trying out the program to take the necessary time to read the instructions carefully; indeed, in the computing trade it is generally believed that users only look at the instructions as a last resort, if all other attempts to get the program to work have failed. The user's understanding of some of the words you use may not be the same as yours. If you build all the instructions into the program and display them on the screen at appropriate times, you may have to abbreviate them for lack of space.

With (2) we attempt to trap wrong inputs which are the result of mistyping or of the user's misunderstanding of what is required. The program should check whatever assumptions it makes about the input data, preferably immediately after they are input. For example, suppose we want the user to choose an integer in the range 1 to 999

```
100  PRINT "Think of a number less than 1000"
110  PRINT "What is your number?"
120  INPUT number
130  IF number<1000 THEN GOTO 160
140  PRINT "Your number was too big"
150  GOTO 100
160  IF number>0 THEN GOTO 190
170  PRINT 'We need a number greater than 0"
```

```
180  GOTO 210
190  IF number=INT number THEN GOTO 230
200  PRINT "We need a whole number"
210  PRINT "Think of another number"
220  GOTO 110
230  REM NUMBER is an integer, 1 to 999
```

The INPUT command ensures that what we get is a number and we then check that it obeys any restrictions we have assumed later in the program. Here we have told the user that it should be less that 1000 and have assumed that most users will not think of choosing a negative or fractional number. Each of the conditions is checked, and the user is told if his number is rejected, including the reason for the rejection. This last is most important; there few things more infuriating than a computer which refuses to process the things you give it without giving some indication of what is wrong. (This is why when ZX BASIC rejects a command line because of a syntax error it positions the 'S' cursor at the place where it thinks the error is. It might perhaps have been more helpful if it also told you the nature of the error, but often the nature of the error is fairly obvious once its position has been pointed out, and in many cases the computer would have difficulty deciding just what the cause of the error was.)

The above piece of program does not give the user a long message listing all the restrictions on the input, it just gives the important details. It then checks all the assumptions, including those the user has been told about. If you are systematic, you should be able to make certain that the input to a program conforms to whatever assumptions the rest of the program makes about it. You can then be sure that the program should perform correctly whatever inputs the user gives it.

SPEED

The ZX computers do not run programs particularly quickly, and so it can be important for the programmer to be aware of how to avoid making the program slower than it has to be. Some aspects of the design of the BASIC are inherited from the ZX80, in which the requirement to fit the whole system into a very small amount of memory was paramount. To keep the internal design simple (and thus to minimise the space used by the machine code program that interprets the BASIC) the various things that have to been kept in memory (the BASIC program, variables, strings, etc.) are simply stacked one after the other so that if a particular item is required the computer searches through from the beginning until it finds it, if one needs to be inserted the others are moved up to make room, and if one needs to be removed the others are moved back to close up the space. This saves keeping (and keeping up-to-date) the multitude of pointers which would be needed to find things more quickly at the expense of a fair amount of searching and (when things have to be moved) copying; but since there cannot be very much to search through or to copy anyway (because there is so little room) this does not take very long.

The name of a variable, for instance, is stored in the program without any additional information as to where it is stored, so every time a variable is used when the program is run the computer searches through the part of the memory where the variables are kept, looking for the variable with the required name. In a ZX81 with only 1K of

RAM this will not take very long, as there cannot be very many variables to search through, but in one with 16K, or in a Spectrum (especially one with 48K) there is room for a big program with lots of variables and if the computer has to keep finding the one that happens to be at the end of the list this will slow the program down noticeably

Whichever language and computer are used, in the typical program only about 20% of the commands are obeyed often enough for it to matter at all how long they take. Except where animated displays are being generated what usually matters to the user is the time between hitting ENTER, at the end of a command or piece of input, and seeing either results or an invitation to supply more input, so any command that is obeyed only once or twice during this time is not likely to make a significant difference.

Some of the techniques for reducing the run time of a program apply to most languages on most computers, and are largely common-sense measures such as not doing inside a loop (and hence once each time round) a calculation that could be done outside it. But there are a few peculiarities of the ZX BASIC that deserve special mention in this context.

GOTO searches the program from the beginning for the line you want, so a line near the beginning of the program can be found more quickly than one near the end The natural way to write a program is with the initialisation (which is just done once) first, then data input, then processing and output. But this would put the part of the program most likely to benefit from faster GOTOs in the place where GOTOs are slowest so a better order would be

GOTO initialisation

processing & output (often-used loops)
processing & output (rest of)
STOP (or GOTO end)
initialisation
data input
GOTO processing

This includes a backwards jump that is not part of a loop, and indeed it is already less clear just what the program is doing, so we see that we have to choose between a faster program and a well-structured one. The extra GOTOs are obeyed just once each, so the time they take does not matter.

NEXT and RETURN are also jumps and use the same mechanism as GOTO to find the FOR or GOSUB instruction to return to. NEXT is particularly important because it is always part of a loop, and therefore obeyed many times.

You may be able to reduce the number of lines in your program by, for example, replacing

```
140  PRINT "Value is ";
150  PRINT x
```

by

```
140  PRINT "Value is ";x
```

or

```
270  LET q = x + LOG y
280  LET q = q * EXP z
```

by

```
270  LET q = (x + LOG y) * EXP z
```

On the Spectrum you can put several commands on one line separated by colons, and since it is the number of lines rather than the number of commands that matters this can speed things up considerably. Also, there is a little extra processing to be done at the end of a line, so putting your commands on fewer lines will speed the program up a little anyway. But it is still likely to be worth starting a new line for a FOR or GOSUB command, because the special kind of jump done by NEXT and RETURN searches for the line containing the FOR or GOSUB (skipping down just looking at the line numbers) and then scans through the line counting the commands in it until it comes to the one after the FOR or GOSUB.

In this case, the format that will run faster is also likely to be fairly good from the point of view of readability, as in

```
2100  LET a=5: LET b=0: LET c=7
2110  GOSUB 1000
2120  FOR n=1 TO 50: LET q(n)=q(n)+r(n): NEXT n
2130  GOSUB 1200
etc
```

The way in which variables are found is in many ways similar to the way in which program lines are found, and it is usually done rather more often. Each variable that has been assigned to is described by a record which specifies its type, name, and value. (Trying to use a variable for which no record exists causes error 2.) Assuming the program is started by RUN, there are no records present when the program starts, new records are added at the end by DIM, LET, FOR, and INPUT commands, and the records are searched from the beginning so the oldest one is always looked at first.

In detail: DIM adds a new record describing an array FOR and also LET assigning to a numeric variable, will use the existing record for that variable if there is one, otherwise it will add a new one. LET assigning to a string variable always adds a new record, if there is an existing one, it is removed and all the later records are moved back to close up the gap LET will never create a new record when assigning to an array element.

As a general rule, therefore, the arrays and numeric variables that are going to be used frequently should be DIMensioned or assigned to before anything else even if they are not going to be used until later String variables will usually gravitate to the end in any case

Short names should be used for numeric variables (arrays and strings are restricted to one character names anyway – another hang-over from the ZX80). A variable with a name six or seven characters long takes twice as long to search past as one with a one-character name The characters that are the value of a string variable take a similar amount of time to search past.

Note that the computer searches for a variable each time it appears in the program Thus in

```
100  FOR j=n+1 TO n+10
110  LET a(j) = a(j)*j
120  NEXT j
```

in which line 100 is obeyed once and lines 110 and 120 are obeyed 10 times, it searches for *n* twice (both on line 100), *a* 20 times (all on line 110, twice each time round), and *j* 41 times (once on line 100, three times each time round on line 110 and once each time round on line 120). In all, these three lines therefore contain 63 searches for variables and 9 jumps.

It takes about the same time to search past a variable as a program line so the extra time to find the twenty-first variable (say) instead of the first is about the same as the extra time to GOTO the twenty-first program line instead of the first This is a little more than the time it takes to do a floating point addition or subtraction, but somewhat less than the time required for a multiplication or division.

The other trap for the unwary is in some of the operations on numbers The 'to the power' operator is always worked out using the formula

```
x ↑ n   =   EXP (n * LOG x)
```

except when $x=0$. This means that $x \uparrow 2$ takes about twenty times as long to calculate as $x*x$ does, and may give a less precise answer Similarly $x \uparrow 3$ takes about ten times as long as $x*x*x$. Also $x \uparrow n$ causes an error A if x is negative because you cannot take the LOG of a negative number so if $x=-3$ then $x*x$ is $+9$ but $x \uparrow 2$ stops the program with error A Moral: use multiplication instead of 'to the power' whenever possible.

SQR x is worked out as $x \uparrow 0.5$, and thus also takes rather a long time to calculate, but there is not really any viable alternative Some of the trigonometric functions are slower than others: TAN x is worked out as SIN x / COS x; ASN uses SQR and ATN, as does ACS

PART III

EXAMPLE PROGRAMS

7

GRAPHICAL PRESENTATION OF DATA

The programs in this chapter are concerned with displaying numeric data on the screen in pictorial form. We saw in Chapter 2 that this is the kind of task to which computers are well suited, and a pictorial display often gives a much better over-all impression of trends etc. in the data than a column of figures would.

The most literally 'graphical' presentation is by drawing a graph, as of a value *y* which depends on another value *x*. Mathematicians say *y* is a 'function' of *x* and show this by writing $y=f(x)$. The graph is drawn by considering each possible value of *x* in turn, working out the corresponding *y*, and (starting from a fixed point called the 'origin') measuring *x* units along the paper and *y* units up the paper and marking the place.

In ZX BASIC, the PLOT command does most of the work of this for us. Having worked out the values *x* and *y*, we need only say

```
PLOT x,y
```

to get the relevant point blacked in on the screen. Thus:

```
10  FOR x=0 TO 255
20  LET y=SIN x
30  PLOT x,y
40  NEXT x
```

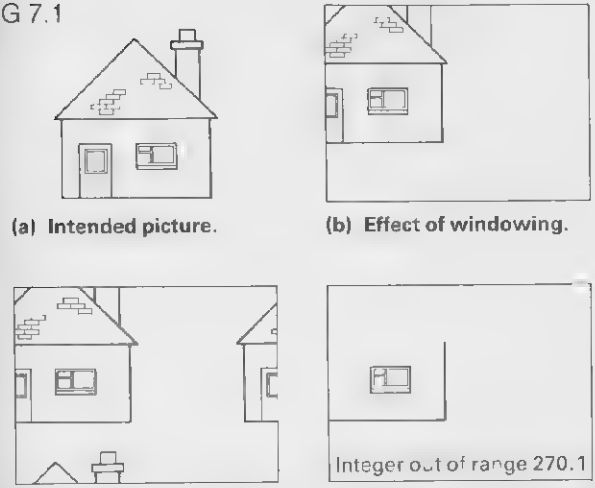
But if you try the program in this form you will find it does not plot a sine wave, in fact it stops with error B (indicating that the graph does not fit on the screen) before getting very far at all. We need to make sure the graph is large enough to see properly without being too big to fit on the screen.

As the manual (Chapter 18 for the ZX81, Chapter 17 for the Spectrum) describes, the screen is divided into a rectangular array of 'picture elements', called 'pixels' for short. The rows and columns are identified by whole numbers (which we will call 'co-ordinates') starting with zero in the bottom lefthand corner; on the ZX81 there are 44 rows and 64 columns, so the top righthand corner is column 63 and row 43. We will find it more convenient to think of the top righthand corner as ($x=63$, $y=43$); this is why the rows are numbered upwards. The pixels on the Spectrum are much smaller than those on the ZX81, and there is room for four times as many in each direction, so the co-ordinates go up to ($x=255$, $y=175$).

Different computers behave in different ways when a picture does not fit on the screen. Consider the simple line drawing in Fig. 7.1(*a*) positioned on a screen as in Fig. 7.1(*b*). Part of the picture is off screen and does not appear; this technique is called 'windowing' because it is as if the screen is a window through which you are looking at the picture, and you see only those parts of the picture that are opposite the window.

Another technique is called 'wraparound', here the parts that fall off one edge appear at the opposite edge, as in Fig. 7.1(*c*). This is rather as if you had drawn the picture on a car tyre inner tube (a shape mathematicians call a 'torus') and then cut the tube open and flattened it out. Wraparound was much used in the early days of graphical

FIG 7.1



(a) **Intended picture.**

(b) **Effect of windowing.**

(c) **Effect of wraparound**

(d) **Appearance in ZX BASIC.**

PIXEL – the smallest part of a picture for which the computer can define the colour, brightness, etc. The size of a pixel defines the maximum resolution available.

WINDOWING – using the TV screen (or part of it) as a window through which you see a part of a picture; the rest of the picture is stored in the computer's memory but the only way to see it is by moving the position of the window.

WRAPAROUND – when referred to computer graphics, is an alternative to windowing (qv): when a line goes off one side of the screen, it comes on again at the opposite side, so that the whole picture is visible although possibly in a rather jumbled form.

displays because, with the technology then available, it was much easier to implement, but windowing is normally more convenient for the user.

ZX BASIC does not use either of these techniques, but simply signals error B whenever PLOT etc. finds that a point does not fit on the screen. However, it is not too difficult to check inside your program that you are not plotting points off-screen, and even to do your own windowing. In the above example we can change line 30 to

```
30  IF x>=0 AND x<=63 AND y>=0 AND y<=43
       THEN PLOT x,y
```

On the ZX81 this plots the point only if it fits on the screen; on the Spectrum it plots it only if it is in the lower lefthand corner of the screen, so that your picture will not obliterate whatever is already in the rest of the screen. By varying the four numbers against which the values of *x* and *y* are tested, you can have a rectangular window of any size anywhere in the screen. By using different tests you can have windows of other shapes; for instance

```
30  IF x>=0 AND x<=y-8 AND y<=40 THEN PLOT x,y
```

defines a triangular window and

```
30  IF (x-128)↑2+(y-88)↑2 < 1600 THEN PLOT x,y
```

defines a circular one in the middle of the screen on the Spectrum. (On the ZX81 you would have to use smaller numbers to keep it on the screen.)

As indicated at the end of Chapter 6, it would be better to use $(x-128)*(x-128)$ instead of $(x-128)\uparrow 2$ to reduce the time taken to do the test (Even though the computer has to find x twice and do the subtraction twice, this is still a lot quicker than using 'to the power' operator.)

Whatever kind of window we use, we still do not get anything that looks much like a sine wave; we need to choose the right scale at which to draw the picture For any x, the value of $\sin x$ is in the range -1 to $+1$ This is why we have had so little success so far. all the points plotted were either in the bottom two rows of pixels or just off the bottom of the screen

We therefore need to 'scale' and 'translate' the picture Scaling is done by multiplying all the coordinates by a certain number (so the picture gets bigger or smaller but is still centered on the same origin), translation is done by adding the same number to all the y co-ordinates so that it moves up or down or to the x co-ordinates so that it moves sideways Considering only the y co-ordinates for the moment, we can do

```
25  LET y = y*20+22
```

on the ZX81 or

```
25  LET y = y*80+88
```

SCALE (verb) – to change the size of a picture by multiplying the coordinates of all the points in the picture by the same number.
TRANSLATE – to move a picture by adding the same pair of numbers (one for the x direction, another for the y direction) to the coordinates of all the points in the picture

on the Spectrum to get the *y* values into the range 2 to 42 on the ZX81 or 8 to 168 on the Spectrum which fits comfortably on the screen.

What about the *x* direction? One complete cycle of the sine wave takes from zero to 2π or about 6.3 so if we divide the pixel co-ordinate by 10 we will get one complete cycle on the ZX81 or four on the Spectrum We may as well start at $x=0$, so no translation is needed Note, by the way, that in the *x* direction we are starting from the pixel co-ordinate and calculating the value whereas in the *y* direction it is the other way round we start with the value (derived from the *x* value) and calculate the pixel co-ordinate

Let us now rewrite the program in the ZX81 version Note that we are now assured that each point (x,y) is on-screen so there is no need for any windowing

```
10  FOR x=0 TO 63
20  LET y = SIN (x/10)
25  LET y = y*20+22
30  PLOT x,y
40  NEXT x
```

This calculates the *y* co-ordinate in two stages (lines 20 and 25) before using it on line 30. We can make the program slicker by calculating it all in one go and putting the expression in the PLOT command instead of putting the result in the variable *y* and taking it out again:

```
10  FOR x=0 TO 63
30  PLOT x, 20 * SIN (x/10) + 22
40  NEXT x
```

This will work on the Spectrum too, but produce a rather small picture in the bottom lefthand corner of the screen. To fill the screen we should do

```
10  FOR x=0 TO 255
30  PLOT x, 80 * SIN (x/10) + 88
40  NEXT x
```

You should try one of these out on a ZX81 or Spectrum, and experiment with changing the various constants (two in line 10, three in line 30) to see what happens.

A MORE GENERAL VERSION

We can adapt the program to print the value of any expression as follows. This shows the power of the VAL operator in ZX BASIC, which allows the string to contain any expression rather than just a literal number. The version given is for the spectrum.

```
 10  LET i=0
 20  DIM y(255)
 30  LET ymin = 0
 40  LET ymax = 0
 50  LET x = 0
 55  LET xstep = 0
 60  PRINT "Graph plotting"
 70  PRINT
 80  PRINT "Type the value of y as an"
 90  PRINT "    expression involving x."
100  PRINT
110  PRINT "Be careful to use the single"
120  PRINT "    keys for SIN, LOG, etc"
130  PRINT "    instead of spelling them"
```

```
140  PRINT "    out letter by letter."
150  INPUT f$
160  CLS
170  PRINT "y = ";f$
180  PRINT
190  PRINT "x start at?"
200  INPUT x
210  PRINT "x finish at?"
220  INPUT xmax
230  LET xstep = (xmax-x)/255
240  CLS
250  PRINT "y = ";f$
260  LET y0 = VAL f$
270  LET ymin = y0
280  LET ymax = y0
290    REM now get the rest of the y's and
          find the max and min
300  FOR i = 1 TO 255
310  LET x = x + xstep
320  LET y(i) = VAL f$
330  IF y(i) < ymin THEN LET ymin = y(i)
340  IF y(i) > ymax THEN LET ymax = y(i)
350  NEXT i
360    REM now we know the range that y covers
370  IF ymin = ymax THEN LET ymax = ymin+1
380  LET yscale = 168 / (ymax-ymin)
390  PLOT 0, (y0-ymin) * yscale
400  FOR i = 1 TO 255
410  PLOT i, (y(i)-ymin) * yscale
420  NEXT i
```

For the ZX81, change 255 to 63 wherever it occurs and change 168 to 41 on line 380. On the Spectrum you can delete lines 180 to 210 and 240 and 250 if you replace line 220 with

```
220  INPUT "x start at ";x;", finish at ";xmax
```

On lines 10 to 50 we make sure that the variables we will be using most often are mentioned before f$ as explained at the end of Chapter 6, this helps keep as short as possible the pause between when the user enters the last inputs and when the results start to appear. We keep the values in an array to save having to calculate them twice, if each one takes quite a long time to work out this speeds up the second loop (lines 400 to 420) quite a lot, while if they take only a short time the program will run quite quickly anyway and there is no significant penalty On the other hand, we could simply work out all the values twice over, as in

```
260  LET ymin = VAL f$
270  LET ymax = ymin
280  LET xmin = x
290    REM now find the max and min y values
300  FOR i=1 TO 255
310  LET x = x + xstep
320  LET y = VAL f$
330  IF y < ymin THEN LET ymin = y
340  IF y > ymax THEN LET ymax = y
350  NEXT i
360    REM now we know the range that y covers
370  IF ymin = ymax THEN LET ymax = ymin + 1
380  LET yscale = 168 / (ymax-ymin)
390  LET x = xmin
400  FOR i = 0 TO 255
410  PLOT i, (VAL f$ - ymin) * yscale
415  LET x = x + xstep
420  NEXT i
```

(Lines 10 to 250 are the same as before except that line 20 is a LET rather than a DIM.) This will start to draw the graph a little earlier (because the loop on lines 300 to 350 is a little quicker) but will usually take longer to draw it; psychologically this might be better as the user can see that the program is doing something and indeed can see how far it has got.

Note that if *yscale* was set up as (*ymax*−*ymin*) /168 we would have to divide rather than multiply in line 410, and division takes longer than multiplication. Similarly in line 310 we find the next *x* from the old one rather than deriving each one afresh from *i* as in

```
310  LET x = xmin + i * xstep
```

because this would involve an unnecessary multiplication.

HISTOGRAMS

As well as drawing graphs where the pairs (*x*,*y*) are associated by some mathematical formula such as the one stored in *f*$ in the program above, we can draw graphs in which the pairs (*x*,*y*) represent experimental or other data from the 'real world'. The graphical form may well reveal trends or periodic variations that are not nearly so apparent from the raw figures. A program to draw such a graph is

```
10  PRINT "Caption for graph?"
20  INPUT c$
30  PRINT c$
40  PRINT "Minimum y value?"
50  INPUT ymin
60  PRINT ymin
```

```
 70  PRINT "Maximum y value?"
 80  INPUT ymax
 90  IF ymax>ymin THEN GOTO 130
100  PRINT "Maximum must be greater than"
110  PRINT "   minimum!"
120  GOTO 70
130  LET yscale = 168 / (ymax-ymin)
140  PRINT "Now input the y values in"
150  PRINT "   order from left to right"
160  INPUT y
170  CLS
180  PRINT c$
190  LET x=0
200  IF y>=ymin AND y<=ymax THEN
        PLOT x, (y-ymin) * yscale
210  LET x = x+1
220  IF x>255 THEN GOTO 9999
230  INPUT y
240  GOTO 200
```

We should probably print a further message between lines 150 and 160 telling the user that STOP can be used if there are less than 256 numbers (see Chapter 9 of the ZX81 manual or Chapter 2 of the Spectrum manual).

Note that we check (on line 90) that the maximum and minimum values supplied by the user are sensible and (on line 200) we do not assume that the *y* values will in fact necessarily come within these limits. No effort is made to maximise the speed with which the program runs because it has very little to do between being given one input value and asking for the next.

Another kind of display that is often used is the histogram We can make the program plot a histogram by simply replacing line 200 with

```
200  IF y>ymax THEN LET y=ymax
202  FOR j=0 TO (y-ymin) * yscale
204  PLOT x,j
206  NEXT j
```

and making a suitable alteration to the message printed by line 10. On the Spectrum it will be quicker to use DRAW keep the new line 200 but instead of 202 to 206 use

```
204  IF y>=ymin THEN PLOT x,0: DRAW 0,
       (y-ymin)*yscale
```

Histograms are often drawn with the individual columns separated; all we need to do is to replace line 210 with

```
210  LET x = x+2
```

On the Spectrum we may want to make them rather chunkier; to do this replace lines 200 to 220 in the original program by

```
200  IF y>ymax THEN LET y=ymax
202  IF y<ymin THEN GOTO 210
204  LET y = (y-ymin) * yscale
206  FOR j = x TO x+3: PLOT j,0: DRAW 0,y: NEXT j
210  LET x = x+6
220  IF x>252 THEN GOTO 9999
```

Sometimes it is helpful to have histograms in several colours The ZX81 allows grey as well as black and white, using the characters available in G mode on keys ASDFGH when SHIFT is held down (see Chapter 11 of the ZX81 manual) They are not supported by PLOT, so you have to either use PRINT AT or else accumulate the picture in a character array and then copy it to the screen when it is complete. An example program for the ZX81 using PRINT AT can be made by replacing lines 200 to 220 of the original program by

```
192    REM define character codes for block
          and half-height block
194  LET block = 8
196  LET half = 9
198    REM draw column upwards
200  LET y = (y-ymin) * yscale
202  FOR v=21 TO 1 STEP -1
204  IF y>1.5 THEN GOTO 210
206  IF y>0.5 THEN PRINT AT v,x; CHR$ half;
208  GOTO 216
210  PRINT AT v,x; CHR$ block;
212  LET y = y-2
214  NEXT v
216  LET x = x+1
218  IF x>31 THEN GOTO 9999
220    REM swap colours for next column
223  LET block = 136-block
226  LET half = 140-half
```

HISTOGRAM – a graphical representation of data in which each number is shown as a rectangle the area of which is proportional to the value depicted. Usually all the rectangles are the same width, so that the height is also proportional to the value.

Line 223 swaps *block* between 8 the code for a grey square, and 128, the code for a black one. Line 226 swaps *half* between 9, for a grey half-square, and 131 for a black one. If you change these lines to

```
223  LET block = 13-block
226  LET half = 13-half
```

then the black columns will be half-width, leaving a gap before the next grey one. There are no characters available which would allow you to make narrower grey columns unless you turn the histogram on its side (cf. Exercise 3 in Chapter 11 of the ZX81 manual).

On the Spectrum, you have eight colours available including black and white, and you also have two brightness levels. You can draw the columns as wide as you like, and have gaps of any width between them; the version of the program on page 130 had columns four pixels wide all in the same colour with a gap two pixels wide, but if we add

```
195  LET colour = 2
206  FOR j = x to x+4: PLOT INK colour; j,0:
        DRAW INK colour; 0,y: NEXT j
210  LET x = x+8
225  LET colour = 7-colour
```

(which will cause the existing lines 206 and 210 to be deleted) then the columns will be alternately red and pale blue, five pixels wide with a three-pixel gap.

Although the Spectrum graphics have high resolution when used for monochrome pictures (with the same 'paper' and 'ink' colours over the whole screen), for multicoloured pictures the resolution with which you can specify the colours is much lower. The screen is divided into 'character positions' each consisting of an 8×8 array of pixels, so there are 64 pixels in each character position When you are using the screen for the text (as in the PRINT command) each character printed occupies a character position For each character position you can specify the 'paper' and 'ink' colours (in each case one of eight colours, if you count black and white as colours), and also whether the character is highlighted (by being brighter than normal) and whether it is to flash In the case of text this allows you complete freedom to specify the colour of each character independently.

When drawing pictures, however you need to be aware that all the 64 pixels that make up a character position share the same colour specification If any of them is to be highlighted, or to flash, then all must do so You only have two colours – paper colour and ink colour available

Suppose you want to draw a histogram with adjoining red, blue, and green columns on a white background Suppose you start by drawing the red column in the righthand half of a column of character squares in red ink on white paper (say with the *x* value going from 4 up to 7) in the manner of line 206 in the program just above You can go on to draw the blue column next to it (with *x* from 8 to 11) in blue ink on white paper If you now try to add the green column, in green ink which we will suppose is half as high as the blue column, you will find that the bottom half of the blue column (which shares character squares with it) changes to green because you are changing the ink colour in those character squares.

You can create the green column by changing the paper colour to green, but this also is done a whole character position at a time: when you change the bottom row from white to green the next seven rows change as well You can thus only get one-eighth the resolution that is otherwise available on the Spectrum and indeed only one-half the resolution that is available on the ZX81. Now suppose the blue column is (say) 50 pixels high and the green column is higher: you will be changing the paper colour above the top of the blue column to green To be sure of avoiding this problem the height of the blue column has to be a multiple of eight pixels also.

One way round this restriction is to use half-tones to generate intermediate colours in the same way that the ZX81 generates grey. (This is mentioned in Chapter 17 of the Spectrum manual) For instance, keeping lines 10 to 130 from the program above, we can have

```
140    REM set up user-defined characters b to i
          as half-tone
150   FOR i=USR "b" TO USR "b"+62 STEP 2
160   POKE i, BIN 01010000: POKE i+1, BIN 10100000
170   NEXT i
180    REM now make a to h into 0 to 7 rows
          (out of 8) of half-tone
190   FOR i=0 TO 7
200   FOR j=USR "a" + 8*i TO USR "a" + 7*i + 7
210   POKE j,0
220   NEXT j
230   NEXT i
240   PAPER 7: INK 0: CLS: PRINT c$
250   FOR i=1 TO 16
```

```
 260  INK 2: LET c$="red"
 270  GOSUB 1000
 280  FOR j=i*16-12 TO i*16-9: PLOT j,0:
         DRAW 0,y: NEXT j
 290  INK 1: LET c$="pale blue"
 300  GOSUB 1000
 310  FOR j=21 TO 1 STEP -1
 320  IF y<7 THEN PRINT AT j,i*2-1:
         CHR$(145+y);: GOTO 350
 330  LET y=y-8: PRINT AT j,i*2-1; CHR$(152);
 340  NEXT j
 350  LET c$="dark blue"
 360  GOSUB 1000
 370  FOR j=i*16-4 TO i*16-1: PLOT j,0:
         DRAW 0,y: NEXT j
 380  NEXT i
 390  GOTO 9999
 400
1000  INPUT (i); "st " AND i=1; "nd " AND i=2;
          "th " AND i>2; (c$); ": "; y
1010  LET y = (y-ymin) * yscale
1020  RETURN
```

The program draws the red column in one set of character squares and the blue and half-tone blue in another. The half-tone blue is done before the full-colour blue because PRINT writes the whole character square, but it can be done afterwards by

```
PRINT OVER 1; AT j,i*2-1; CHR$(145+y);
```

which will not disturb the parts that have already been written by DRAW.

A number of variations on the above themes are possible and should be done as exercises They include drawing a line between adjacent points on a graph so that the graph forms a continuous line even where one *y* value differs from the next by more than 1 drawing axes on the graph and labelling them; and histograms in which each column has more than one colour An example of the last is a histogram of sales showing home-market sales in full colour and export sales in half-tone.

SCATTER DIAGRAMS

Another way of presenting 'real-world' data is in the form of a 'scatter diagram' in which we simply plot (*x*,*y*) pairs This lets us see whether there is any correlation between *x* and *y* if there is, the points will be grouped together around a line or curve but if there is none the points will be randomly positioned all over the screen

A suitable program to make a scatter diagram on the ZX Spectrum is

```
 10   REM x is max, n is min, v is value
 20  DIM x(2)
 30  DIM n(2)
 40  DIM v(2)
 50  PRINT "Caption? ";
 60  INPUT c$
 70  PRINT c$
 80  FOR i=1 TO 2
 90  PRINT "Minimum ";"xy"(i);" value? ";
100  INPUT n(i)
110  PRINT n(i)
120  PRINT "Maximum ";"xy"(i);" value? ";
130  INPUT x(i)
140  IF x(i)>n(i) THEN GOTO 170
```

```
150  PRINT "Maximum must be greater than",
        "   minimum!"
160  GOTO 90
170  PRINT x(i)
180  NEXT i
190  PRINT "Now type in (x,y) pairs"
200  PRINT "STOP terminates"
210  INPUT v(1)
220  CLS
230  PRINT c$
240  INPUT v(2)
250  FOR i=1 TO 2
260  IF v(i)<n(i) OR v(i)>x(i) THEN GOTO 300
270  LET v(i) = (v(i)-n(i)) / (x(i)-n(i))
280  NEXT i
290  PLOT v(1)*255,v(2)*168
300  INPUT v(1)
310  GOTO 240
```

On the ZX81 line 290 becomes

```
290  PLOT v(1)*63,v(2)*41
```

because of the lower-resolution graphics

The expression

```
"xy"(i)
```

SCATTER DIAGRAM — used for results of experiments (or similar data) that consist of a pair of numbers, where we are looking for a relationship between the two numbers in the pair. For each pair a point is plotted at the corresponding (x,y) coordinates.

used in lines 90 and 120 is a 'slice' in which we select the *i*th character of the string "*xy*". Thus *x* is printed when $i=1$, and *y* when $i=2$. In fact *x* will be printed if i is anywhere in the range 0.5 to 1.5, and *y* if it is anywhere in the range 1.5 to 2.5; if it is outside the range 0.5 to 2.5 the program stops with error code 3 (subscript out of range) or B (integer out of range). Contrast this with

```
("x" AND i=1) + ("y" AND i=2)
```

which still yields *x* when $i=1$ and *y* when $i=2$ but yields the empty string (and does not cause an error) if i has any other value including, for instance, 1.0001 or -42.

To see how a scatter diagram might look without needing to have any real data, we can do the following. Firstly for a completely random pattern:

```
10  FOR i=1 TO 200
20  PLOT RND*255, RND*168
30  NEXT i
```

The regular lines of dots that appear are an artefact of RND, which produces numbers that are not quite as random as they should be. As usual, for the ZX81 the multipliers in line 20 should be 63 and 41 (or 43 as there is no caption). Also you should have only 30 or 40 points rather than 200, otherwise most of the screen will be black. For the kind of distribution more likely to occur in nature, again with no correlation between *x* and *y*:

```
10  FOR i=1 TO 200
20  GOSUB 1000
30  LET x=v
40  GOSUB 1000
```

```
  50  LET y=v
  60  IF x<0 OR x>1 OR y<0 OR y>1 THEN GOTO 20
  70  PLOT x*255, y*168
  80  NEXT i
  90  STOP
1000    REM Set v to a weighted random number
1010  LET v = RND*1.9999 + .00005
1020  LET v = 0.1 * LN (v/(2-v)) + 0.5
1030  RETURN
```

As before, any apparently regular patterns are an artefact of RND. For a scatter diagram in which the two values are linearly related, replace line 50 by

```
50  LET y = 0.2 + x*0.6 + v*0.2
```

An example of a non-linear relationship is produced by

```
50  LET y = 3*(x-0.4)*(x-0.4) + v*0.2
```

8

STATISTICS

The purpose of a scientific experiment is to test a hypothesis (or theory) by seeing if an outcome predicted by the hypothesis occurs in practice. Sometimes the experiment is such that its outcome is unequivocal – there is no doubt whether the predicted event has occurred – but often, particularly in the life sciences, statistical methods must be used to show whether the result of the experiment accords with the hypothesis. More precisely, we need to know how likely the observed event (or set of events) would be if the hypothesis is correct.

For example, in an experiment in which a number of plants are grown from seed the hypothesis being tested might predict that half of them would have yellow flowers, a quarter of them red flowers, an eighth blue flowers, and an eighth purple flowers. Suppose 404 plants survive and flower: the hypothesis does not predict that you will get 202 yellow ones, 101 red, 50 blue, 50 purple and one with blue and purple stripes. Rather it is predicting that each one of the 404 plants has an even chance of having yellow flowers, 3–1 against red, and 7–1 against each of blue and purple. If the hypothesis is true it is still possible for all 404 to turn out to be blue, but it is so extremely unlikely that you would think that either the hypothesis was false or there was something very wrong with your experiment. In practice you will get a result such as 190 yellow, 126 red, 47 blue, and 41 purple and you will want to know just how likely this result is if the probabilities are as stated above.

This is calculated using χ^2, which measures how different the observed results (here 190, 126, 47, and 41) are, overall, from the expected results (here 202, 101, 50 5, and 50.5). The likelihood of results deviating from the expected by as much as the observed results do is calculated from χ^2 and the number of 'degrees of freedom'.

The number of degrees of freedom is always one less than the number of possible outcomes of each event in the experiment. In the example here, we have 404 events each of which has four possible outcomes, yellow red, blue, or purple, so that the observed results consist of four numbers which must add up to 404 There are three degrees of freedom because any three of those numbers can vary independently; we may get any number with yellow flowers and any number of the rest may have red flowers, and any number of what remains may have blue flowers, but then all that are left over must have purple flowers

The following program works out the probability according to equations derived from M. Abramowitz and I.A Segun, *Handbook of Mathematical Functions*, Dover Publications Inc , New York, 1965, equations (26.4.4), (26.4.5) (26 2.1), (26.2.5), and (26 2 17). The version for the Spectrum is given first then the alterations for the ZX81 (which only affect the INPUT prompts and the presentation of the results on the TV screen) are given

Lines 10 to 80 set up a table on the screen so that several sets of results can be processed. Lines 100 to 160 set up coefficients that are used in the calculation of $q(x)$ which is needed if the number of degrees of freedom is odd Using this method instead of writing the coefficients directly into line 520 as literals gives a rather tidier layout; it does make the program run more slowly but this is unlikely to be perceptible. Lines 310 to 350 accumulate χ^2 as the sum of

$$(x-e)^2 / e$$

where e is the expected and x the observed value, avoiding use of the 'to-the-power' operator which is very slow.

If v, the number of degrees of freedom, is an even number, we require to calculate

$$s = \sum_{r=1}^{(v-2)/2} \chi^{2r} / (2\times4\times6\times \ldots \times 2r)$$

or, to put it another way,

$$s = \chi^2/2 + \chi^4/(2\times4) + \chi^6/(2\times4\times6) + \ldots$$
$$\ldots + \chi^{v-2}/(2\times4\times6\times\ldots (v-2))$$

which the program works out as

$$s = \chi^2/2 \times (1 + \chi^2/4 \times (1 + \chi^2/6 \times \ldots$$
$$\ldots \times (1 + \chi^2/(v-2) \times (1 + 0)) \ldots))$$

The loop at lines 420 to 440 starts at the righthand end of this expression, keeping the intermediate result in q each time; the final step is done at line 460 which calculates the probability as

$$\exp(-\chi^2/2) \times (1 + s)$$

If v is an odd number, we have

$$s = \chi + \chi^3/3 + \chi^5/(3\times5) + \ldots$$
$$\ldots + \chi^{v-2}/(3\times5\times7\times\ldots\times(v-2))$$

which is worked out as

$$s = \chi \times (1 + \chi^2/3 \times (1 + \chi^2/5 \times ..$$
$$\ldots \times (1 + \chi^2/(v-2) \times (1 + 0)) ..))$$

As before, the loop at lines 420 to 440 does most of the work, this time the last step is in line 530 which calculates the probability as

```
EXP (-chi2/2) * SQR (2/PI) * (1-p(chi) + s)
```

using the value of $1-p(chi)$ calculated on the previous line.

```
 10   REM chi-squared test
 15   REM set up screen
 20  CLS
 30  PRINT "Degrees of       2     Proba-"
 40  PRINT " freedom      chi        bility"
 50  PRINT
 60  PLOT 0,156: DRAW 255,0
 70  PLOT  85,0: DRAW 0,175
 80  PLOT 170,0: DRAW 0,175
100   REM coefficients in polynomial
110  LET b1 =  0.31938153
120  LET b2 = -0.356563782
130  LET b3 =  1.781477937
140  LET b4 = -1.821255978
150  LET b5 =  1.330274429
160  LET p  =  0.2316419
200   REM input data
210  INPUT "Degrees of freedom: ";v
300   REM calculate chi-squared
310  LET chi2=0
320  FOR n=1 TO v+1
```

```
330  INPUT "Observed value: "; x,
        "Expected value: "; e
340  LET chi2 = chi2 + (x-e)*(x-e)/e
350  NEXT n
400    REM work out probability value
410  LET q=0
420  FOR n=v TO 3 STEP -2
430  LET q = 1 + (chi2/n) * q
440  NEXT n
450  IF n=1 THEN GOTO 500
455    REM here if v is even
460  LET q = EXP (-chi2/2) * (1 + (chi2/2) * q)
470  GOTO 600
500    REM here if v is odd
510  LET chi = SQR chi2: LET t = 1/(1+p*chi)
520  LET b = ((((b5*t+b4)*t+b3)*t+b2)*t+b1)*t
530  LET q = EXP (-chi2/2) * SQR (2/PI) *
        (b + chi*q)
600    REM display results
610  LET x=v: GOSUB 1000
620  PRINT x$;
630  LET x=chi2: LET n=4: GOSUB 2000
640  PRINT AT 23-PEEK 23689,11; x$
650  LET x=q: LET n=6: GOSUB 2000
660  PRINT AT 23-PEEK 23689,22; x$
700  INPUT "Another set of data? (Y/N) ";a$
710  IF a$ = "Y" OR a$ = "y" THEN GOTO 200
720  IF a$ <> "N" AND a$ <> "n" THEN GOTO 700
900  GOTO 9999
1000   REM Set x$ to value of x rounded to an
            integer and right-aligned in 10
            characters
```

```
1010  LET x$ = "         " + STR$ INT (x+0.5)
1020  LET x$ = x$(LEN x$ - 9 TO )
1030  RETURN
2000    REM Set x$ to value of x rounded to n
              decimals and right aligned in 10
              characters
2002    REM Assumes 0<x<10^(9-n) and 0<n<9
2010  LET x$ = STR$ INT (x * 10↑n + 0.5)
2020  LET x$ = "00000000"(LEN x$ TO n) + x$
2030  LET x$ = "        "(LEN x$ TO 8) + x$
2040  LET x$ = x$(TO 9-n) + "." + x$(10-n TO)
2050  RETURN
```

In the routine on lines 2000 to 2050 line 2010 sets *x$* to the required string of digits without its decimal point, line 2020 adds zeroes to the left of *x$* if required to ensure there are at least $n+1$ digits, line 2030 adds spaces to bring it up to 9 characters, and line 2040 inserts the decimal point.

The main changes required for the ZX81 are to replace lines 50 to 80 by

```
50  FOR i=0 TO 43
55  PLOT 20,i
60  PLOT 42,i
65  NEXT i
70  FOR i=0 TO 63
75  PLOT i,39
80  NEXT i
```

and to replace 'AT 23−PEEK 23689,' by 'TAB' on lines 640 and 660. (On the Spectrum we cannot use TAB because it outputs spaces to the screen which would obliterate the vertical lines drawn by lines 70 and 80. On the ZX81, TAB skips over them in the same way that AT does. Incidentally, on both machines the 'comma' PRINT separator outputs spaces but new line at the end of a PRINT command does not.)

Where there is just one degree of freedom, the results given by the above program are a little misleading because the distribution of χ^2 is continuous whereas the observed values are discrete. That is to say the way in which we calculate the probability of χ^2 being within a certain value does not take into account the fact that only certain values of χ^2 are possible. Yates's correction allows for this, and gives a better value for the probability in the case where there is just one degree of freedom; the following lines should be added to the program to implement it.

```
220  IF v<>1 THEN GOTO 300
230  INPUT "With Yates's correction? (Y/N)";a$
240  IF a$="N" OR a$="n" THEN GOTO 300
245  IF a$<>"Y" AND a$<>"y" THEN GOTO 230
250  INPUT "Observed value: "; x,
        "Expected value: "; e
260  LET x = ABS (x-e) - 0.5: LET chi2 = x*x/e
270  INPUT "Observed value: "; x,
        "Expected value: "; e
280  LET x = ABS (x-e) - 0.5
285  LET chi2 = chi2 + x*x/e
290  GOTO 400
300    REM calculate chi-squared without
          correction
```

REGRESSION

At the end of Chapter 7 we looked briefly at scatter diagrams, which are appropriate for occasions on which two values are measured and we are looking for some correlation between them. In this section we will look at how the computer can provide some objective measurements of how the values are related.

Here, as with the χ^2 test, we are considering the outcome of a number of separate trials. The χ^2 test is used where each trial has only a finite (and often quite small) number of possible outcomes, as with the flower-growing example in which there were just four possible outcomes of each trial (yellow, red, blue, or purple; we discounted any trials in which we got no flowers at all). For each possible outcome the number of trials with that outcome is counted. In the χ^2 test we assumed that the differences between trials were due to chance (or to random variations in some factor we were not measuring) and investigated how likely it would be that the particular set of outcomes we observed would occur if the hypothesis was correct.

The techniques described in this section are appropriate where each trial produces two measurements (which will be called *x* and *y* here) and we are looking for a correlation between the two things measured, which would indicate that they are connected in some way. A relationship between the two measurements is sought, such that we can say that *y* consists of some function of *x* (i.e. a value worked out using an algebraic formula involving *x*) plus an additional component which is a random variation.

The implication is that there is a causal relationship between whatever is measured by *x* and whatever is measured by y, so that given the value for x in a particular trial we can predict the value of y more accurately than we could otherwise. This contrasts with the χ^2 example, in which we did not have any information that would (for example) allow us to select beforehand which plants were more likely to have red flowers.

For instance, we might enquire whether people's height is related to the height of their parents. We might ask a number of college students for their own height (for the variable y) and the average of the heights of their parents (for the variable x). We should obviously do this separately for male and female students, because men are on average taller than women. The kind of result we might expect is that tall parents will tend to have tall children and short parents will tend to have short children, and that there will be a certain amount of random variation in the heights of children of parents of a particular height, with the children tending to be slightly closer to the average height than their parents. We can express this as

$$y = a + bx + \text{random variation}$$

and we hope to take enough pairs of measurements to be able to identify the values of a and b with reasonable confidence.

As well as finding the numerical values for a and b we will plot a scatter diagram and draw on it the line $y=a+bx$.

```
10   REM x is max, n is min, v is value
20 DIM x(2)
30 DIM n(2)
40 DIM v(2)
```

```
 50  INPUT "Caption: "; c$
 60  PRINT c$
 70  FOR i=1 TO 2
 80  INPUT "Minimum ";"xy"(i);" value: ";
        n(i), "Maximum ";"xy"(i);"
        value: "; x(i)
 90  IF x(i) > n(i) THEN GOTO 120
100  PRINT "Maximum must be greater than",
       "   minimum!"
110  GOTO 80
120  NEXT i
130  PRINT "Now type in data as two numbers"
140  PRINT "   separated by a comma inside"
150  PRINT "   the quote marks, e.g."
160  PRINT
170  PRINT "        ""2.47,15.438"""
180  PRINT
190  PRINT "Use STOP (inside the quotes) to"
200  PRINT "   terminate."
210    REM read in data and accumulate sums
220  LET sx=0: LET sy=0: LET sxx=0:
        LET sxy=0: LET n=0
230  INPUT d$
240  FOR i=1 TO LEN d$
250  IF d$(i) = "," THEN GOTO 310
260  IF d$(i) = " STOP " THEN GOTO 500
270  IF i>3 THEN IF d$(i-3 TO i) = "STOP"
        THEN GOTO 500
280  NEXT i
290  INPUT "Your input must include either",
        "a comma or STOP ";d$
300  GOTO 240
310    REM here if i'th character of d$ is comma
320  LET v(1) = VAL d$( TO i-1)
330  LET v(2) = VAL d$(i+1 TO )
```

```
340  LET sx = sx + v(1):
        LET sxx = sxx + v(1)*v(1)
350  LET sy = sy + v(2):
        LET sxy = sxy + v(1)*v(2)
360  LET n = n + 1
370  IF n=1 THEN CLS: PRINT c$
380  FOR i=1 TO 2
390  IF v(i) < n(i) OR v(i) > x(i)
        THEN GOTO 230
400  LET v(i) = (v(i)-n(i)) / (x(i)-n(i))
410  NEXT i
420  PLOT v(1)*255, v(2)*151
430  GOTO 230
500    REM here at end of data
510  IF n=0 THEN STOP: REM no data at all
520  LET b = (sxy - sx*sy/n) / (sxx - sx*sx/n)
530  LET a = sy/n - b * sx/n
540  PRINT AT 1,0; "y = ";a;" + ";b;"x"
600    REM now draw line with this equation
610  LET x1 = n(1): LET y1 = a + b * x1
620  LET x2 = x(1): LET y2 = a + b * x2
630  IF y1 > x(2) THEN GOTO 700
640  IF y1 >= n(2) THEN GOTO 800
650    REM here if must start at bottom edge
660  LET y1 = n(2)
670  GOTO 720
700    REM here if must start at top edge
710  LET y1 = x(2)
720    REM here if not at lefthand edge
730    REM finish if not on screen at all
740  IF b=0 THEN GOTO 999:  REM horizontal
750  LET x1 = (y1-a) / b
760  IF x1 < n(1) OR x1 > x(1) THEN GOTO 999
800    REM line starts at (x1,y1)
810  IF y2 > x(2) THEN LET y2=x(2): GOTO 840
```

```
820  IF y2 >= n(2) THEN GOTO 900
830  LET y2 = n(2)
840    REM here if finishes at top or bottom
850    REM we now know it is on screen
860  LET x2 = (y2-a) / b
900    REM here when start and end points found
910  LET v(1) = 255 / (x(1)-n(1))
920  LET v(2) = 151 / (x(2)-n(2))
930  PLOT (x1-n(1)) * v(1), (y1-n(2)) * v(2)
940  DRAW (x2-x1) * v(1), (y2-y1) * v(2)
```

For the ZX81 the INPUT commands on lines 80 and 290 must be converted to use PRINT for the captions in a similar manner to the program in Chapter 7, if you restrict them to the first three lines of the screen they will not overwrite the scatter diagram. Lines containing several commands must be split up, for instance line 340 is replaced by

```
340  LET sx = sx + v(1)
345  LET sxx = sxx + v(1)*v(1)
```

Line 810 however must not be replaced by

```
810  IF y2 > x(2) THEN LET y2=x(2)
815  GOTO 840
```

because this would always GOTO line 840, even when *y2* was not greater than *x*(2). In fact the job done by lines 800 to 860 is exactly the same as that done by lines 630 to 760, except that the tests on lines 740 and 760 are not needed the second time. Lines 630 to 760 were written in a way that is compatible with the ZX81, lines 800 to 860 in a way that is more convenient on the Spectrum. Therefore for the ZX81 we just need to replace lines 810 to 830 with a copy of lines 630 to 710 with the line numbers suitably changed and $y1$ replaced by $y2$. Conversely, on the Spectrum we can make the program neater by replacing lines 630 to 710 with a similarly modified copy of lines 810 to 830.

(Make sure you understand why the tests are only needed once: first we find where the line, which we are drawing from left to right, comes onto the screen – top, bottom, or lefthand edge – then we find where it leaves it. The tests on lines 740 and 760 detect the cases in which the line does not come onto the screen at all; having come onto the screen it must then go off it again and, if we are drawing it from left to right, this must be at the top, bottom, or righthand edge.)

The multipliers 255 and 151 in line 420 and lines 910 and 920 need to be changed to 63 and 37 for the ZX81, and the DRAW command on line 940 must be replaced by the line drawing routine from Chapter 18, Exercise 6 of the ZX81 manual.

The word STOP in line 260 is the token STOP which is a shifted A on the keyboard (symbol shift in the case of the Spectrum, which has two shift keys) but in line 270 it is the letters S,T,O,P. We check for both because this is easier and also more helpful than explaining in detail to the user that one or the other must be used. (Note, by the way, that either will work in the program in Chapter 7: one gives code D, the other code 2.) The program should be refined to check the input more thoroughly, making sure that the substrings before and after the comma are valid numbers (consisting only of digits, point, letter E, and leading and trailing spaces) before applying VAL, so that the user will get a helpful error message and an opportunity to retype rather than have the program abort with code C. A further refinement of help to the user might be to display the last *x* and *y* co-ordinates, and the number of pairs so far, on the second and third lines of the screen, for instance by

```
375  PRINT AT 1,0; "Point number ";n;
         " was",v(1),v(2),
```

The comma at the end of the PRINT command ensures that the previous message is obliterated. Suppose the fifth *y* value is 12.7863 and the sixth is 103; if the comma was not there the sixth *y* value would appear as 1037863 (the 103 being the true value and the 7863 left over from last time). Line 540 needs to obliterate the last message printed by line 375. Because we do not know how long the two numbers will be we cannot be sure how many commas will be needed to clear the rest of the area without clearing any of the scatter diagram; therefore it is better to clear the area first by

```
535  PRINT AT 1,0,,,,
```

which clears two lines as required.

ZX BASIC restricts the names of arrays to one character. (This is another hangover from the ZX80.) Thus we cannot call the maximum and minimum values *max* and *min* (see lines 10 to 30) and they have been reduced to the rather less mnemonic letters *x* and *n*. Arguably *x* should not be used as *x*(1) and *x*(2) can be confused with the *x*1 and *x*2 used for the horizontal co-ordinate, perhaps the maximum should be called *m*. We should perhaps attempt to make things a little more readable by adding

```
45  LET x=1: LET y=2
```

and talking about $m(x)$, $m(y)$, $n(x)$, $n(y)$, $v(x)$, and $v(y)$ instead of using literal numbers for the subscripts. We should then also change lines 70 and 380 to

```
FOR i=x TO y STEP y-x
```

to make it clearer what the loops are doing.

CORRELATION

The line drawn by the above program is called a *regression line*. The program provides an objective way of finding this line, and a subjective assessment (by looking at the picture) of how closely the experimental results fit this line. But there is a quantity which we can calculate, called the *correlation coefficient*, which gives an objective measure of how well they fit.

If all the points lie exactly on the regression line, the correlation coefficient is +1 if the line slopes upwards from left to right, −1 if it slopes downwards. For any particular

REGRESSION LINE – a line drawn on a scatter diagram (q.v.) showing the relationship between the two numbers in question. The points do not normally lie exactly on the line because of the effect of other, random, influences on the results of the experiment.
CORRELATION COEFFICIENT – a number which shows how close the relationship is between two quantities (such as those represented by the *x* and *y* coordinates in a scatter diagram (q.v.)), assuming this relationship is linear.

regression line as the points move further away from the line the correlation coefficient gets nearer to zero. If the slope of the regression line decreases while the points maintain their distance from it the correlation coefficient again gets nearer to zero.

Thus if the correlation coefficient is +1 or −1 we can predict the value of *y* exactly, so long as we know the value of *x*. If the correlation coefficient is zero this shows that *x* has no influence on *y*: we cannot predict *y* any better knowing *x* than we could if we did not know *x*.

The following lines added to the program above enable it to calculate the correlation coefficient also.

```
225  LET syy=0

355  LET syy = syy + v(2)*v(2)

550  PRINT "Correlation coeff. "; (sxy - sx*sy/n)/
        SQR ((sxx - sx*sx/n) * (syy - sy*sy/n))
```

The values of the correlation coefficient that can be regarded as indicating a close relationship between *x* and *y* depend on the circumstances, but as a rough guide there is still a substantial random component in *y* for correlation coefficients as high as 0.85, as can be seen by running the program: you can use real data, data you invent yourself, or computer generated data (using RND) as in the programs at the end of Chapter 7.

9

ACCOUNTING

One of the problems with most pocket calculators is that there is no record of the numbers that were keyed in. Someone adding up a column of figures on (say) an invoice tends to look mostly at the invoice form and at the keyboard, and it is very easy to forget to look at the display during the brief time that each number appears on it (between keying the last digit and keying the '+' operator) to check that the number has been keyed correctly.

A program in a personal computer, on the other hand, can remember all the numbers that were keyed in (by storing them in an array) and display them on the screen so that they can be checked against the original data. If they are displayed in a column in a similar format to the original, the task of checking them becomes fairly easy.

Unfortunately, BASIC does not make it very easy to display figures properly aligned in a column. The following program (for the ZX81) shows the kind of thing that is required.

```
10  DIM p(201)
20  PRINT AT 10,0; "Type in the prices to be"
30  PRINT "   totalled;  use zero to"
40  PRINT "   terminate the list."
50  FOR i=1 TO 200
60  INPUT p(i)
70  LET p(i) = INT (p(i)*100 + 0.5) / 100
80  IF p(i)=0 THEN GOTO 110
90  NEXT i
```

```
100  SCROLL
110  PRINT "No room for any more."
120  SCROLL
130  PRINT "Now check the prices."
140  FOR i=20 TO 200 STEP 20
150  FOR j = i-19 TO i
160  IF p(j)=0 THEN GOTO 200
170  SCROLL
180  LET n$ = STR$ (p(j)*100)
190  PRINT j; AT 21,24-LEN n$; n$( TO LEN n$ - 2);
        "."; ("0"+n$)(LEN n$ TO LEN n$ + 1)
200  NEXT j
210  SCROLL
220  PRINT "All correct? (""Y"" or ""N"")"
230  INPUT r$
240  IF r$="Y" THEN GOTO 360
250  IF r$<>"N" THEN GOTO 230
260  PRINT AT 21,0; "Line number in error?"
270  INPUT j
280  IF j<=i AND j>i-20 THEN GOTO 310
290  PRINT AT 20,0; "Line ";j;" is not on screen."
300  GOTO 150
310  SCROLL
320  PRINT "Correct value?"
330  INPUT p(j)
340  LET p(j) = INT (p(j)*100 + 0.5) / 100
350  GOTO 150
360  IF p(j) <> 0 THEN NEXT i
370    REM here when all been checked
380  LET sum=0
390  FOR i=1 TO 200
400  LET sum = sum + p(i)
410  NEXT i
420  SCROLL
430  LET n$ = STR$ (p(j)*100)
```

```
440  PRINT "Total"; AT 21,24-LEN n$; n$( TO LEN n$
       - 2); "."; ("0"+n$)(LEN n$ TO LEN n$ + 1)
```

Lines 180 and 190, also lines 430 and 440, print the value of *p(j)* correctly aligned assuming it is already rounded to two decimal places (which is done by line 70 or 340), we assume that *p(j)* is in pounds (dollars, francs, marks etc.) so that the decimal places are the pence (cents, centimes, pfennigs, etc.). How would you modify it to deal with half-pence?

The only change required for the Spectrum is to remove the SCROLL commands. The PRINT commands on lines 220, 260, and 320 could be replaced by a caption in the INPUT command that follows in each case. For the version that works in half-pence, you could use one of the user-defined characters for the '½'.

Often, the price of each item on an invoice is broken down into *net + tax*, the *net* figure being in turn broken down into *quantity × unit cost*. The following program (written for the Spectrum) checks the figures on an invoice, perhaps one received from a supplier, on which the arithmetic has already been done. To keep it simple, we assume all items carry tax at 15%.

SCROLL – to move a picture up and down, or across, the screen, like winding the paper of a scroll from one roll to the other.

```
 10  LET runtotal=0
 15  LET netsum=0: LET taxsum=0
 20  CLS
 30  INPUT "Quantity (or zero if no more",
        "items): ";qty
 40  IF qty=0 THEN GOTO 500
 50  INPUT "Quantity: "; (qty), "Unit price: ";
        price, "Net: "; net, "Tax: "; tax,
        "Gross: "; gross
 60  IF ABS (qty*price - net) < 0.01
        THEN GOTO 110
 70  PRINT "Net price doesn't tally."
 80  PRINT qty;" x ";price;" = ";qty*price
 90  PRINT "given as ";net
100  BEEP 0.5,47: GOTO 30
110  IF ABS (net*0.15 - tax) < 0.01
        THEN GOTO 160
120  PRINT "Tax doesn't tally."
130  PRINT "15% of ";net;" is ";net*0.15
140  PRINT "given as ";tax
150  BEEP 0.5,47: GOTO 30
160  IF ABS (net+tax - gross) < 0.001
        THEN GOTO 210
170  PRINT "Gross price doesn't tally."
180  PRINT net;" + ";tax;" = ";net+tax
190  PRINT "given as ";gross
200  BEEP 0.5,47: GOTO 30
210  LET netsum = netsum + net
220  LET taxsum = taxsum + tax
230  GOTO 20
500    REM here when finished
510  INPUT "Total net (zero if not given):",net
520  INPUT "Total tax (zero if not given):",tax
530  INPUT "Invoice total: ",gross
540  IF net=0 THEN PRINT "Net total ";netsum:
        GOTO 590
```

```
550  IF ABS (net-netsum) < 0.001 THEN GOTO 590
560  PRINT "Net total doesn't tally."
570  PRINT "Calculated as ";netsum
580  PRINT "given as ";net
590  IF tax=0 THEN PRINT "Total tax ";taxsum:
        GOTO 640
600  IF ABS (tax-taxsum) < 0.001 THEN GOTO 640
610  PRINT "Total tax doesn't tally."
620  PRINT "Calculated as ";taxsum
630  PRINT "given as ";tax
640  IF ABS (netsum+taxsum-gross) < 0.001
        THEN GOTO 690
650  PRINT "Invoice total doesn't tally."
660  PRINT "Calculated as ";netsum+taxsum
670  PRINT "given as ";gross
680  INPUT "True total: ";gross
690  LET runtotal = runtotal + gross
700  INPUT "Another invoice to do? (Y/N)"; r$
710  IF r$="Y" OR r$="y" THEN GOTO 15
720  IF r$<>"N" AND r$<>"n" THEN GOTO 700
730  PRINT ,,,,"Total of all invoices this run",
        runtotal
```

The main change required to make the program work on the ZX81 is to separate out the captions from the INPUT commands and put them in PRINT commands as in the first program in this chapter. Also the BEEP commands must be removed, you should consider other ways of attracting the operator's attention such as making the screen flash.

The user is asked to type in all five figures for each item, and the program checks that net price, the tax, and the gross price have been calculated correctly. The first two only need to be correct to the nearest penny, because the true value may have been rounded up or down to a whole number of pence; the values given for the net price and tax should add up to the gross price exactly, but we have to allow for rounding errors in the computer and so we only insist on it being correct to the nearest tenth of a penny

If any of the figures in an item is wrong, the user has to type the whole item again. How would you modify the program so that only the incorrect figures had to be retyped? Remember that when the computer finds that three figures are inconsistent it does not know which of the three is at fault. Having found one error, the program does not check for any further errors but immediately asks for the item to be corrected. This is because the correction will probably affect the outcome of the remaining tests: if *net* does not tally with *qty* × *price*, it may well be that *net* has been mistyped, in which case the other two tests will fail also, or that *net* was wrongly calculated in the first place and the figures for *tax* and *gross* must now be recalculated

The program asks for all the figures to be typed in, rather than just asking for the quantity and unit price and working the others out. There are two reasons for this one is that the actual figures are used for *net* and *tax*, which may be up to a penny different from the calculated figures and thus (if there are a lot of items) make a noticeable difference to the total The other reason is that it would be quite easy for the user to fail to notice a discrepancy between the figures displayed on the screen and those on the invoice form, so it is better for the comparison to be done by the computer There is then no need for the kind of checking of input that was done in the first program in this chapter, because the numbers are checked against each other.

If we replace line 150 with

```
145 INPUT "Is tax figure correct? (Y/N)";r$
150 IF r$="N" OR r$="n" THEN GOTO 30
155 IF r$<>"Y" AND r$<>"y" THEN GOTO 145
```

then the program will cope with the occasional item at a rate of tax other than 15%. Consider how you would modify the program so that it always required the percentage rate of tax to be input, or (by inserting some commands between lines 110 and 120) so that it asked the rate of tax only if the input figures were not consistent with a rate of 15% Consider also how you would modify it to give a discount on the net price, so that *net* was not compared against

```
qty * price
```

but rather against

```
qty * price * (100-discount) / 100
```

Alternatively, by a similar modification at line 100 to that just described for line 150, the program could report the percentage discount (or surcharge) that appeared to have been applied and ask if this was correct. If *net* was smaller than *qty* × *price*, the discount would be calculated as

```
100 * (1 - net / (qty * price))
```

otherwise the surcharge would be calculated as

```
100 * (net / (qty * price) - 1)
```

The program might be written so that it misses this out if the rate of discount or surcharge is clearly absurd but it would not be easy to decide on a suitable criterion for 'absurd' adjustments to prices of finished goods caused by changing prices of certain metals, for instance, can be very arge – so it s probably better to leave t to the user to decide what appears reasonable.

The second part of the program checks the tota s on the invoice, it al ows for the possibility that the ind vidual totals of the *net* and *tax* figures may not be given on the or ginal invoice and it displays them on the screen in case they are needed for tax records. It a so keeps a grand tota of al the invoices; it could eas ly be altered to keep separate grand totals for *net* and *tax*.

INCOME TAX

Another kind of tax calculat on, at least in the UK, s PAYE (an acronym for 'pay as you earn') which is deducted from wages by employers when the wages are paid Readers who are not d rectly interested n PAYE should stil find the program instructive as an example of the kind of data process ng that is frequent y required in a commercia environment

Each week the total pay since the start of the tax year is ca culated and the 'free pay' (wh ch s the amount of pay the emp oyee can have without pay ng any tax) is deducted from it. The remainder, the 'taxab e pay' s then looked up in the Tax Tables (which are supplied to all emp oyers by the Inland Revenue) to find how much tax should have been pa d since the start of the tax year; the amount that had been paid by the previous week is subtracted to f nd how much is payable in the current week This amount is deducted from the employee s pay and rem tted to the Inland Revenue

The fo lowing program does most of the ar thmetic involved, as in the previous examp e, t is written for the

Spectrum but the only changes required for the ZX81 are to move the captions from INPUT commands into PRINT commands and to insert a CLS command before line 200. The program can very simply be adapted for monthly paid employees by substituting 'month' for 'week' throughout.

```
 10 LET all tax = 0: LET all pay = 0
100 INPUT "This week's pay: "; pay,
       "Previous total pay: "; prev pay,
       "Total free pay: "; free pay,
       "Previous total tax: "; prev tax
150 LET total pay = prev pay + pay
200 PRINT "        last week     this week"
210 PRINT
220 PRINT "week's pay"; TAB 22; pay
230 PRINT "total pay "; prev pay; TAB 22;
       total pay
240 PRINT "free pay"; TAB 22; free pay
250 PRINT "taxable"; TAB 22;
       total pay - free pay
260 INPUT "Total tax due: ";total tax
270 LET tax = total tax - prev tax
280 PRINT "total tax "; prev tax; TAB 22;
       total tax
290 PRINT "week's tax"; TAB 22; tax
300 PRINT
310 PRINT "net pay"; TAB 22; pay - tax
400 PRINT
410 PRINT "--------------------------------"
420 PRINT
500   REM accumulate totals of pay & tax
510   REM   for all employees
520 LET all tax = all tax + tax
530 LET all pay = all pay + pay - tax
540 INPUT "Any more employees? (Y/N) ";r$
```

```
550  IF r$="Y" OR r$="y" THEN GOTO 100
560  IF r$<>"N" AND r$<>"n" THEN GOTO 540
600  PRINT "Total to pay to employees"
610  PRINT "   this week       "; all pay
620  PRINT "to Inland Revenue "; all tax
```

The user types in the previous week's total pay and total tax from the Deduction Card, the current week's pay, and the 'total free pay' figure found by looking up the employee's tax code in the tax tables. The computer works out the new 'total taxable pay' figure, which the user has to look up in the tax tables to find the new 'total tax due' figure; the computer can then work out the remaining figures.

The display on the screen (generated by lines 200 to 290) shows the figures to be entered on, or checked against, the Deduction Card, in the order in which they appear on the Card. A better display, in which the figures are correctly aligned with their decimal points under each other and exactly two digits in the 'pence' column, would be produced by the method used in the first program in this chapter (lines 430 and 440). Note that this assumes that all the figures will be an exact number of pence; ideally we should hold them as pence rather than pounds, to avoid rounding errors in the calculations. (The computer can hold whole numbers up to about 4 000 000 000 exactly, but cannot hold numbers expressed as decimal fractions exactly. If your wages bill is more than £40m you will probably have a larger computer to do your payroll!) For example:

```
265  LET total tax = INT (total tax * 100 + 0.5)
273  LET m$ = STR$ prev tax: LET n$ = STR$
         total tax
```

```
275  IF LEN m$ = 1 THEN LET m$ = "0"+m$
277  IF LEN n$ = 1 THEN LET n$ = "0"+n$
280  PRINT "total tax"; TAB 21-LEN m$; m$(TO LEN
       m$-2); "."; m$(LEN m$ - 1 TO LEN m$); TAB 30-
       LEN n$; n$(TO LEN n$-2); "."; n$(LEN n$ - 1 TO
       LEN n$)
```

The version of the program given below uses a subroutine (at line 1000) to print a number correctly aligned.

If we store the various totals for each employee on cassette from one week to the next, the user will not need to enter these particular figures. The general shape of the program is

(1) set up arrays;
(2) perform one week's processing;
(3) save on tape;
(4) when reloaded from tape, repeat from 2.

The program below also includes National Insurance contributions; the contribution each week is based on the pay in that week, which is looked up in another of the Tax Tables. Unlike the version above, it copes correctly with the case where the employee's total pay is less than his free pay.

```
10  INPUT "Maximum number of employees: "; n
20  DIM n$(n,20): REM each employee's name
30  DIM p(n): REM total pay to date
40  DIM t(n): REM total tax to date
50  LET i=1:  GOSUB 2000
80  LET all tax = 0: LET all nic = 0
```

```
 90 LET all pay = 0
100 PRINT "Name: ";n$(i)
110 INPUT "This week's pay: "; pay,
      "Total free pay: "; free pay,
      "Total NI contr'n: "; tnic,
      "Employee's NI contr'n: "; enic
120 LET pay = INT (pay * 100 + 0.5)
130 LET free pay = INT (free pay * 100 + 0.5)
140 LET tnic = INT (tnic * 100 + 0.5)
150 LET enic = INT (enic * 100 + 0.5)
160 LET total pay = p(i) + pay
170 LET taxable pay = total pay - free pay
180 IF taxable pay < 0 THEN LET taxable pay = 0
200 PRINT TAB 11; "last week   this week"
210 PRINT
220 PRINT "total NIC";
225 LET v$ = STR$ tnic: GOSUB 1000
230 PRINT "employee's NIC";
235 LET v$ = STR$ enic: GOSUB 1000
240 PRINT "week's pay";
245 LET v$ = STR$ pay: GOSUB 1000
250 PRINT "total pay";
253 LET v$ = STR$ p(i): GOSUB 1100
256 LET v$ = STR$ total pay: GOSUB 1000
260 PRINT "free pay";
265 LET v$ = STR$ free pay: GOSUB 1000
270 PRINT "taxable pay";
275 LET v$ = STR$ (taxable pay): GOSUB 1000
280 INPUT "Total tax due: "; total tax
285 LET total tax = INT (total tax * 100 + 0.5)
290 LET tax = total tax - t(i)
300 PRINT "total tax";
303 LET v$ = STR$ t(i): GOSUB 1100
306 LET v$ = STR$ total tax: GOSUB 1000
310 PRINT "week's tax";
```

```
315  LET v$ = STR$ tax: GOSUB 1000
320  PRINT
330  PRINT "net pay";
335  LET v$ = STR$ (pay - tax - enic): GOSUB 1000
400  PRINT
410  PRINT "--------------------------------"
420  PRINT
430    REM check OK before updating arrays
440  INPUT "Figures OK? (Y/N) ";r$
450  IF r$="N" OR r$="n" THEN GOTO 100
460  IF r$<>"Y" AND r$<>"y" THEN GOTO 440
470  LET p(i) = total pay: LET t(i) = total tax
480  LET all tax = all tax + tax
485  LET all nic = all nic + tnic
490  LET all pay = all pay + pay - tax - enic
500  LET i=i+1
510  IF i <= n THEN IF n$(i,1) <> " "
        THEN GOTO 100
540  INPUT "Any more employees? (Y/N) "; r$
550  IF r$="Y" OR r$="y" THEN
        GOSUB 2000: GOTO 100
560  IF r$<>"N" AND r$<>"n" THEN GOTO 540
600  PRINT "Total pay to employees"
610  PRINT "   this week";
620  LET v$ = STR$ all pay: GOSUB 1000
630  PRINT
640  PRINT "To Inland Revenue:  tax"
650  LET v$ = STR$ all tax: GOSUB 1000
660  PRINT "        NI contributions"
670  LET v$ = STR$ all nic: GOSUB 1000
680  PRINT "                   total"
690  LET v$ = STR$ (all tax + all nic):
        GOSUB 1000
700  INPUT "Now write back to tape: start",
        "recording & then press ENTER."; r$
```

```
 710  SAVE "PAYE" LINE 760
 720  PRINT "Rewind & replay the tape"
 730  VERIFY "PAYE"
 740  PRINT "Tape checked successfully"
 750  GOTO 799
 760  LET i=1: REM restart here when reloaded
 770  GOTO 80
 799  GOTO 9999: REM skip over srtn at end of
         program
1000    REM Print number from v$ on righthand
              side of screen via GOSUB 1200
1010  LET tab = 30
1020  GOSUB 1200
1030  PRINT
1040  RETURN
1100    REM Print number from v$ in centre of
            screen
1110  LET tab = 21
1200    REM Print number of pence from V$ as
              pounds and pence, with last digit
              in column
1202    REM   number held in variable TAB
1210  IF LEN v$ = 1 THEN LET v$ = "0" + v$
1220  IF v$(1) = "-" THEN IF LEN v$ = 2 THEN LET v$
         "-0" + v$(2)
1230  PRINT TAB tab-LEN v$; v$( TO LEN v$ - 2);
         "."; v$(LEN v$ - 1 TO LEN v$);
1240  RETURN
2000    REM Input name for i'th employee
2010  INPUT "Employee's name: "; n$(i)
2020  IF n$(i,1) <> " " THEN RETURN
```

```
2030  INPUT "Name must not start with a",
        "space.   Employee's name:", n$(i)
2040  GOTO 2020
```

For the ZX81, there are the usual changes of splitting up lines with more than one command and separating the captions etc out of INPUT commands into separate PRINT commands. Lines 720 to 750 must be omitted because the ZX81 does not have a VERIFY facility and the text 'LINE 760' must be omitted from line 710 because the program will start there automatically when reloaded. It also carries on there after it has been saved, so the user will need to abort it with STOP (or by switching the computer off). The input on line 700 is ignored and is simply there to provide a convenient way of making the computer wait until the user has got the tape ready.

It is worth emphasising that the Spectrum version checks that the new data have been correctly stored on the tape but this is not possible with the ZX81. Unless you have a second ZX81 to LOAD it into, you will not discover that the tape will not LOAD until after the data have been cleared out of the computer's memory. It is safer to SAVE the data a second time (using GOTO 700), and listen to both copies on the tape to check they make the right noises, before throwing away the data in the computer, but even this is not a guarantee that it will load correctly next time.

There is still scope for improvement to the program. For instance, if the total amount of tax due is (say) £2.20 less than last week the program shows the 'tax due this week' as '−2.20' whereas it is entered on the Deduction Card as '2.20 R' (R for 'refund'). How would you make it print the latter format instead? The figure for 'this week's pay' may well be made up of a basic wage plus a bonus or commission and with superannuation payments etc. deducted; in which case the program should ask for these figures separately and calculate the week's pay from them. Possibly some of the figures (basic wage, superannuation contribution) are the same every week, and can be kept in arrays in the way that the employee's name is.

If you have the ZX printer, you can make the program print out a payslip for each employee.

The Inland Revenue publish algorithms whereby the figures for free pay, tax due, etc. can be calculated instead of asking the user to look them up. However, for a firm with only a few employees the effort of programming these algorithms and updating them when the tax rules change is probably greater than the effort of looking the figure up in the tables.

INTEREST

Another kind of financial calculation that the computer can do is to compare the returns on different kinds of investment. Perhaps you have an ordinary savings account with a building society that yields 7% interest, tax paid, and you think it unlikely you will need to withdraw your money in the near future. Should you consider transferring it to a different kind of account on which you get 8.25% interest, tax paid, but lose 28 days' interest if you make a withdrawal? The following program calculates the total amount of interest in each case.

```
10  PRINT AT 20,0; "Amount invested (£):"
20  INPUT amt
30  CLS
```

```
 40  PRINT "Interest on £"; amt
 50  PRINT
 60  PRINT "weeks    ordinary    high yield"
 70  PRINT
100  FOR n=1 TO 16
110  PRINT n*4; TAB 8;
120  PRINT INT (amt * 7 * n*28/365 + 0.5) / 100;
         TAB 20;
130  PRINT INT (amt * 8.25 * (n-1)*28/365 +
         0.5) / 100
140  NEXT n
```

Lines 120 and 130 print the interest at the relevant rates. The amount invested is multiplied by the percentage rate of interest to get the annual interest in pence, this is multiplied by the factor required to get the interest over the relevant period (4*n* weeks or 28*n* days) which is rounded to the nearest penny and converted to pounds. The interest rate is written into the program rather than being asked for as input because it is expected that a program of this nature will be fairly ephemeral – written for a particular task and then thrown away – and there is no point in making it more general than it needs to be for that task. Also, the user is probably the same person as the programmer and can quite easily alter line 120 or 130 if required; indeed it will usually be sufficient to replace lines 10 to 30 with a LET command setting *amt* to the sum you are thinking of investing (or reinvesting) and unless you are going to make a copy of the results on the printer the captions printed by lines 40 to 70 are not really necessary either.

In this example both forms of investment pay simple interest calculated daily, less tax at the standard rate (which you cannot claim back if you are not paying tax, but you do have to top up if you are paying tax at a higher rate). A bank deposit account might pay 10.5% on which you would then have to pay tax. So if the rate of tax is 30% you will only have 70% of the interest left after you have paid tax This is therefore calculated as

```
130  PRINT INT (amt * 10.5 * 0.7 * n*28/365
         + 0.5) / 100
```

You forfeit 7 days' interest if you do not give notice of a withdrawal, and you can see the effect of this by

```
130  PRINT INT (amt * 10.5 * 0.7 * (n*28-7)/365
         + 0.5) / 100
```

It is also important to know how often the interest is paid. Suppose that the ordinary account pays interest every three months and the high yield account once a year, and that in each case you have the interest paid into the account so that it is in effect compound rather than simple interest. The interest for the first three months on the ordinary account is therefore itself earning interest for the remaining three quarters of the year, while that for the high yield account is not The following program compares an account paying quarterly at the rate of 7% per annum against one paying annually at the rate of 8.25% per annum but with no interest being paid for the first 28 days (a very subtle difference from the earlier example in which it was the *last* 28 days for which no interest was paid). We assume that the quarters are respectively 90, 91, 92, and 92 days long, the first being 91 in a leap year, and that the investment is made 30 days after the start of the first quarter of a year; *amt* is the amount invested, *bal*1 the balance in the ordinary account *bal2* in the high yield account, and *int2* the interest accrued on the high yield account but not due to be paid until the end of the year, *days*1 is the number of days in the quarter for which interest will be paid on the ordinary account, *days*2 on the high yield account, and *days*3 the number of days in the year.

Lines 10 to 70 are as before except that 'weeks' on line 60 is replaced by 'qtr yr'. Lines 100 onwards are replaced by:

```
100  LET bal1 = amt
110  LET bal2 = amt
120  LET int2 = 0
130  LET qtr = 1
140  LET yr = 84
200  DIM d(4)
210  LET d(1) = 91
220  LET d(2) = 91
230  LET d(3) = 92
240  LET d(4) = 92
250  LET days1 = d(1)-30
260  LET days2 = days1-28
270  LET days3 = 366
300  LET bal1 = bal1 + INT (bal1 * 7 *
         days1/days3 + 0.5) / 100
310  LET int2 = int2 + INT (bal2 * 8.25 *
         days2/days3 + 0.5) / 100
320  PRINT "  "; qtr; " "; yr;
330  PRINT TAB 8; bal1-amt;
340  PRINT TAB 20; bal2+int2-amt
350  LET qtr = qtr+1
360  IF qtr <= 4 THEN GOTO 600
400    REM here at end of year
410  LET bal2 = bal2 + int2
```

```
420  LET int2 = 0
430  LET qtr = 1
440  LET days3 = 365
450  LET d(1) = 90
460  LET yr = yr+1
470  IF yr/4 <> INT (yr/4) THEN GOTO 600
500  LET days3 = 366
510  LET d(1) = 91
600    REM set up for next quarter
610  LET days1 = d(qtr)
620  LET days2 = days1
630  GOTO 300
```

On the ZX81 the program stops with report code 5 after 18 lines but you can get further output by using the CONTinue command. The Spectrum asks you whether to scroll the screen: use Y or ENTER to see the next screenful of output, N or SPACE (for BREAK) to stop.

In all the above you can get the figures more prettily aligned under each other by using the subroutines from the programs earlier in this chapter and in Chapter 10. You could also show the output pictorially as a graph or histogram, using the techniques introduced in Chapter 7.

The descriptions assumed you were the lender but the same principles apply if you are the borrower. You might, for instance, want to buy a bigger and better computer, and want to compare the cost of paying by credit card (interest free for the first few weeks, then interest compounded monthly) with that of increasing the mortgage on your house (much lower interest rate, but no interest free period and probably 'arrangement fees' to be paid). Another possibility is an overdraft at your bank: the amount of money you would normally keep in your current account reduces the amount of the overdraft but you may also have to allow for an increase in bank charges, and the program should take account of these factors

10

KEEPING RECORDS

Nearly all computers have the ability to keep data on 'backing store', which usually consists of magnetic disc or tape. There are three main reasons for using backing store: to increase the amount of memory available; to preserve data while the computer is switched off; and to allow data to be removed from the computer for safe keeping or so that it can be loaded into another computer.

For the computer to have free access to the data on a tape, it must be able to control the movement of the tape past the heads. On tape drives intended for use by computers, all the controls (record, rewind, playback, etc.) are operated electronically by the computer, so that the computer can search the tape for a particular record and read that record into its memory when required. Cassette tape recorders intended primarily for audio use have controls which are operated mechanically from pushbuttons on the recorder, although most have the facility to connect a 'remote pause' switch that will stop the motor.

Some personal computers (though not the ZX computers) make use of this 'remote pause' facility to provide a measure of control of the tape. The computer can read a record from the tape and then stop the tape until it is ready to read the next record, but it cannot rewind the tape nor can it change between the 'write' (or 'record') and 'read' (or 'playback') modes. If the computer can connect to two cassette recorders, it can read records from one, update them, and write the updated records out to the other.

The only form of backing store provided with the ZX81 and Spectrum is cassette tape without any 'remote pause' facility. It has been announced that there is an optional 'microdrive' for the Spectrum over which the computer will have full control, but at the time of writing few details are available.

Because the program does not have any control over the tape recorder, it cannot adopt the approach of 'read a record, process it, read another, process it, etc.' While it was processing the first record (which might take some time if it involved asking for input from the user) it could miss the second record. Therefore we have to adopt the approach of reading all the data into memory first, then processing it, then writing the new data out to tape again, as in the second version of the payroll program in Chapter 9. This means that any kind of 'data base' application has to be restricted to the amount of data that can be held in the computer's main memory.

The simplest way of discovering how much data will fit into the computer is to DIMension an appropriate set of arrays and see how big they can be made before error 4 occurs. As a rough guide, a ZX81 with the add-on RAM pack has about 15 000 bytes available for the BASIC program and data; without the RAM pack it has only about 700 or less, depending how much there is on the screen. The TS1000 without a RAM pack has about 1000 bytes more than the European ZX81. The smaller (16 K) Spectrum has about 9000 available, and the larger (48 K) Spectrum has about 41 000. A typical BASIC program takes very roughly 20 bytes per command, and the remainder of the space is available for data: characters take one byte each, numbers take five.

Taking as an example the 'bank account' program which follows, the program itself requires about 2500 bytes, and each record consists of 16 characters and 2 numbers, a total of 26 bytes. Therefore, the ZX81 with the add-on RAM pack can cope with up to about 480 records, the 16 K Spectrum about 250, and the 48 K Spectrum about 1480. The unextended ZX81 does not even have room for the program! The later version of the program (the one with the menu) is about 3000 bytes longer and thus leaves room for about 120 fewer records.

BANK ACCOUNTS

A fairly typical example of personal record-keeping is keeping track of a bank account. The computer record not only gives an up to date picture of your finances, it allows you to plan ahead by loading the data into the computer and adding entries for the transactions you expect to do in the next few weeks or months. (You should, of course, be careful not to save this fictitious future bank account on the tape in place of the real one.) You can also easily check it against the bank's version when your statement arrives.

This program is written for the Spectrum and

MICRODRIVE – an attachment for the Spectrum, similar to a cassette tape but an endless loop of tape instead of the more usual reel-to-reel cassette and using a rather faster data rate to provide some of the facilities that a floppy disc provides on other machines.

outputs your balance in red when you are overdrawn. To save space on the screen, a single column is used for the amount of all entries, credit entries being shown in black and debit entries in red. (Your bank statement uses separate columns for credit and debit entries.)

Many banks do not charge for transactions provided a certain minimum balance is maintained in the account. The program uses a yellow background for the balance if it is below this minimum (unless it is actually overdrawn), as a warning that you will have to pay charges. Some banks require a minimum cleared balance; you can get an approximate idea of what your cleared balance is likely to be by delaying all credit entries by four working days. For instance if you go into the bank on a Thursday and draw £50 cash and pay in a £75 cheque, you should make the £50 debit entry for that day but make the £75 credit entry for the following Wednesday.

```
 10   REM bank account
 20  LET next entry = 2
 30  LET max entry = 200
 40  DIM d$(max entry,5): REM dates
 50  DIM e$(max entry,11): REM details
 60  DIM a(max entry): REM amounts
 70  DIM b(max entry): REM balances
100  INPUT "Minimum for free banking: ";
        min free
110  LET min free = INT (min free * 100 + 0.5)
120  INPUT "Starting date (5 chs): ";d$(1)
130  LET e$(1) = "Balance fwd"
140  INPUT "Starting balance: "; b(1)
150  LET b(1) = INT (b(1) * 100 + 0.5)
200   REM draw "statement" form
```

```
210  INK 0: PAPER 7: CLS
220  POKE 23692,20
230  FOR n = 1 TO next entry - 1
240  GOSUB 1000
250  NEXT n
300    REM here to add an entry
310  INPUT "Make an entry? (Y/N) "; r$
320  IF r$="N" OR r$="n" THEN GOTO 620
330  IF r$<>"Y" AND r$<>"y" THEN GOTO 210
340  IF next entry >= max entry THEN GOTO 600
400    REM make next entry
410  INPUT "Date (5 chs): "; d$(next entry)
420  INPUT "Details (11 chs): "; e$(next entry)
430  INPUT "Amount: "; x: LET x = ABS x
440  INPUT "Credit or debit? (C/D): "; r$
450  IF r$="C" OR r$="c" THEN GOTO 480
460  IF r$<>"D" AND r$<>"d" THEN GOTO 440
470  LET x = -x
480  LET a(next entry) = INT (x * 100 + 0.5)
490  LET b(next entry) = b(next entry - 1)
        + a(next entry)
500  LET n = next entry: LET next entry = n+1
510  GOSUB 1000
520  GOTO 300
600    REM here if arrays full
610  PRINT "Sorry, no more room."
620    REM here when finished
630  INPUT "Save on tape? (Y/N) "; r$
640  IF r$="N" OR r$="n" THEN GOTO 9999
650  IF r$<>"Y" AND r$<>"y" THEN GOTO 630
660  SAVE "Bank" LINE 200
670  PRINT "Replay tape to verify"
680  VERIFY "Bank"
690  GOTO 9999
1000   REM add entry n to statement on screen
```

```
1010  IF PEEK 23692 > 20 THEN POKE 23692,14
1020  PRINT AT 21,0: PRINT
1030  PRINT AT 0,0;
         "Date  Details     Amount Balance",,
1040  PLOT 0,164: DRAW 255,0
1050  PRINT AT 20,0; d$(n); " "; e$(n);
1060  LET w=7: LET v=a(n): GOSUB 2000
1070  LET w=8: LET v=b(n)
1080  IF v >= 0 AND v < min free THEN PAPER 6
1090  GOSUB 2000: PAPER 7
1100  PLOT 44,8: DRAW 0,167
1110  PLOT 137,8: DRAW 0,167
1120  PLOT 193,8: DRAW 0,167
1130  RETURN
2000    REM write v pence in w characters (2<w<9)
2010  LET v$ = STR$ ABS v
2020  IF LEN v$ >= w THEN LET v$ = "  ******"
        (2 TO w)
2030  LET v$ = "      0"(9-w TO 7-LEN v$) + v$
2040    REM now v$ is w-1 chs long & last 2
           are digits
2050  PRINT INK 2 AND v<0; v$(1 TO w-3); ".";
        v$(w-2 TO w-1);
2060  RETURN
```

Lines 10 to 150 simply set up the arrays and fill in the first entry. Two character arrays hold the date and the description of each entry ('chq' and the cheque number for a debit entry which is a payment by cheque, 'salary' for a credit entry which is your monthly salary, etc.); two numeric arrays hold the amount of the entry (positive for a credit entry, negative for a debit entry) and the current balance. The latter is not strictly necessary, as it can be calculated by adding up all the 'amount' figures, but it can be useful if adjustments have to be made, to simply alter the 'balance' figures without actually inserting the extra entries. (When your bank statement arrives you might find that, because of paying bank charges say, or receiving dividends on shares, the two do not tally, and you may not wish to insert extra records to deal with them.) To eliminate the separate 'balance' figures delete lines 70 and 490 and add

```
 140  INPUT "Starting balance: "; a(1)
 150  LET a(1) = INT (a(1) * 100 + 0.5)

 225  LET balance = 0

1065  LET balance = balance + a(n)
1070  LET w=8: LET v = balance
```

Lines 200 to 250 write out the existing entries to the screen with the help of the subroutine on lines 1000 onwards, which scrolls the entries up the screen while maintaining the headings at the top of the screen and the lines marking the individual columns. Scrolling stops periodically, the user being asked to press a key when he has read what is on the screen; line 220 prevents this happening before anything has been written to the screen (try the program without it to appreciate this) and line 1010 arranges that the 'pages' that the user sees overlap by a few lines.

Lines 300 to 520 add a new entry to the records and to the screen, and lines 630 to 690 save the new state of affairs on tape.

It is important to get the right number of spaces in the character string literals: line 1030 has two spaces between the words 'Date' and 'Details', five between

'Details and 'Amount', and one between Amount' and 'Balance'. The two commas at the end ensure that the second line on the screen is cleared of anything that has been scrolled up into it The string on line 2020 has two spaces and six asterisks and that on line 2030 has five spaces before the zero.

The subroutine starting at line 2000 writes out the amount or balance correctly aligned (assuming it is in pence and is a whole number) or writes a row of asterisks if the figure is too large Line 2030 arranges for the string *v$* to be of the correct length and puts a zero in the tens-of-pence column if the figure is less than ten pence. Line 2050 prints it, complete with its decimal point, in red (colour 2) if *v* was negative and in black (colour 0) otherwise

To adapt the program for the ZX81, apart from the changes to INPUT and to lines with more than one command which should by now be familiar, we need to change the output format to use the less sophisticated facilities that the ZX81 provides The main differences are.
(*a*) We cannot use red for negative figures, probably the best compromise is to use white-on-black as in

```
2050  LET v$ = v$(1 TO w-3) + "." + v$ (w-2 TO w-1)
2052  IF v >= 0 THEN GOTO 2058
2054  FOR i=1 TO w
2055  LET v$(i) = CHR$ (128 + CODE v$(i))
2056  NEXT i
2058  PRINT v$;
```

(*b*) DRAW is not available for the lines separating the columns from each other We can use the pixel graphics to draw rather thicker lines, or the division can be made in some other way, for instance

```
1030  PRINT AT 0,0;
1035  PRINT "Date :Details    :Amount:Balance"
1040  PRINT "-----:-----------:------:-------"
1050  PRINT AT 20,0; d$(n); ":"; e$(n); ":";
1060  LET w=6
1062  LET v=a(n)
1064  GOSUB 2000
1070  LET w=7
1075  LET v=a(n)
1080  PRINT ":";
```

Note that, whatever method is used, *w* must be one less than in the Spectrum version.

(*c*) There is no direct equivalent of the yellow background used to show that the balance has fallen below the minimum for free banking. However, it can be marked by a character in the space between the 'Amount' and 'Balance' columns as in

```
1080  LET v$ = ":"
1082  IF v < 0 OR v >= min free THEN GOTO 1086
1084  LET v$ = "*"
1086  PRINT v$;
```

(*d*) Explicit scrolling (using the SCROLL command) must be used instead of that implied by the PRINT commands on line 1020, thus

```
1020  SCROLL
```

Also the commands on lines 220 and 1010 (which control the 'Scroll?' message on the Spectrum) must be replaced by something like

```
 220  LET scroll count = 19

1010  LET scroll count = scroll count - 1
1012  IF scroll count > 0 THEN GOTO 1020
1014  LET scroll count = 14
1016  PRINT AT 21,0; "Hit ""NEWLINE"" to scroll"
1018  INPUT v$
```

to allow the user to indicate when he has read one pageful and the program can go on to the next although it s arguable that the ZX81 prints numbers so slowly that he has plenty of time to read them anyway!

(*e*) The same changes are needed to SAVE etc (lines 660 to 680) as in the payroll program in Chapter 9.

EXTRA FACILITIES

There are a number of facilities that it would be useful to add to this program, and which are indeed typical of this kind of database application for example the ability to alter records insert records, delete records list the records starting at a particular place, and scroll the listing down as well as up.

In all the programs so far, the program has followed a well-defined sequence of calculations and asked the user for input when it was required. The programmer, through the program was controlling the sequence of events, and the user's role was entirely passive. Of course, the user has ultimate control in that he can choose not to run the program and can abort the program at any time (Note that we do not say that the *computer* is controlling anything: the computer is not responsible for the way a program behaves any more than a tape recorder is responsible for the views expressed by a voice recorded on a tape.)

In the next example we have a somewhat different situation in which we require the user to choose what tasks the program shall perform and in what order. Once a particular function has been chosen, however, the programmer still controls how it is carried out.

The user could be offered the ability to type commands to the program much in the way the programmer types commands for the BASIC, but this would require the user to learn the 'language' in which the commands are expressed and in particular to learn just what commands are available; it would also require the program to interpret the commands (which would presumably be typed in as character strings) and, as we saw in the earlier chapters, this is likely to lead to either a rather complicated program (difficult for the programmer and possibly using up an embarrassingly large amount of the computer's memory) or else an over-rigid format for the commands (tedious for the user).

The usual method followed in this kind of situation is to offer the user a 'menu' of available facilities. The user selects one facility from this menu, and the program then asks the user for the necessary data in just the same way as in the earlier programs. In this way the user is shown exactly what facilities are available and the course taken by the program is directed by simple (mostly single-character) responses.

For the bank account program we might have the following, lines 10 to 150 being the same as before.

```
200  INK 0: PAPER 7: CLS
210  PRINT "1  View statement from start"
```

```
215  PRINT
220  PRINT "2  View last page of statement"
225  PRINT
230  PRINT "3  Add an entry to the end"
235  PRINT
240  PRINT "4  Add an entry in the middle"
245  PRINT
250  PRINT "5  Remove an entry"
255  PRINT
260  PRINT "6  Alter an entry"
265  PRINT
270  PRINT "7  Print statement out"
275  PRINT
280  PRINT "8  Save data on cassette tape"
285  PRINT
290  PRINT "9  Exit from program"
300  INPUT "Select item number:  "; n
310  LET n = INT (n+0.5)
320  IF n<1 OR n>9 THEN GOTO 300
330  CLS
340  GOTO 1000 * n
```

Line 340 jumps to line 1000 if item 1 is selected, 2000 if item 2, and so on. Line 310 ensures that the line jumped to is one of 1000 2000, 3000, etc: otherwise the user might type (say) 4 73 and cause the program to jump to line 4730 with consequences that are unlikely to be helpful. Alternatively line 310 could insist on a whole number, by doing

```
310  IF n <> INT n THEN PRINT AT 19,0;
       "Item number must not include a",
       "   fraction": GOTO 300
```

Similarly line 320 could give the user a message, which would indicate that a number in the range 1 to 9 is expected, before jumping back to line 300.

The part of the program that deals with each item must finish by jumping to line 200; alternatively we could replace line 340 with

```
340  GOSUB 1000 * n
350  GOTO 200
```

so that a RETURN command is used instead of GOTO 200, which at first sight appears to be rather neater and give a more 'well-structured' appearance. However, in the case of item 9 we do not wish to return to line 200 and so ought to remove the unwanted information from the GOSUB stack. In the Spectrum this can be done using CLEAR (which will also throw away all the arrays etc.) but in the ZX81 only NEW (which throws everything away) will do it. We should add

```
335  IF n=9 THEN GOTO 9999
```

so that item 9 is treated specially and avoids putting anything on the GOSUB stack in the first place.

Assuming that we retain the original line 340, the rest of the program can be as follows

```
500    REM add entry n to statement on screen
510  IF PEEK 23692 > 20 THEN POKE 23692,14
520  PRINT AT 21,0: PRINT
530  PRINT AT 0,0;
        "Date  Details     Amount Balance",,
540  PLOT 0,164: DRAW 255,0
550  PRINT AT 20,0; d$(n); " "; e$(n);
560  LET w=7: LET v=a(n): GOSUB 700
570  LET w=8: LET v=b(n)
580  IF v>=0 AND v < min free THEN PAPER 6
590  GOSUB 700: PAPER 7
```

```
 600  PLOT 44,8: DRAW 0,167
 610  PLOT 137,8: DRAW 0,167
 620  PLOT 193,8: DRAW 0,167
 630  RETURN
 700    REM write v pence in w characters (2<w<9)
 710  LET v$ = STR$ ABS v
 720  IF LEN v$ >= w THEN LET v$ = "  ******"
         (2 TO w)
 730  LET v$ = "      0"(9-w TO 7-LEN v$) + v$
 740    REM now v$ is w-1 chs long & last 2 chs
           are digits
 750  PRINT INK 2 AND v<0; v$(1 TO w-3); ".";
         v$(w-2 TO w-1);
 760  RETURN
1000    REM Display statement from start
1010  LET m=1
1020  GOTO 2030
2000    REM Display last page of statement
2010  LET m = next entry - 18
2020  IF m<1 THEN LET m=1
2030    REM Display statement from mth entry
2040  POKE 23692,20
2050  FOR n = 1 TO next entry - 1
2060  GOSUB 500
2070  NEXT n
2100    REM Wait for user then go to 200
2110  INPUT "Hit ENTER for main menu: "; r$
2120  GOTO 200
3000    REM Add entry at end
3010  IF next entry >= max entry THEN GOTO 4020
3020  LET n = next entry
3030  GOSUB 3500
3040  LET next entry = next entry + 1
3050  GOTO 2000: REM to display new entry
3500    REM Input entry number n
```

```
3510 INPUT "Date (5 chs): "; d$(n)
3520 INPUT "Details (11 chs): "; e$(n)
3530 INPUT "Amount: "; x
3540 LET x = INT (0.5 + 100 * ABS x)
3550 INPUT "Credit or debit? (C/D): "; r$
3560 IF r$="C" OR r$="c" THEN GOTO 3590
3570 IF r$<>"D" AND r$<>"d" THEN GOTO 3550
3580 LET x = -x
3590 LET a(n) = x
3600 LET b(n) = b(n-1) + x
3610 RETURN
4000   REM Insert entry
4010 IF next entry < max entry THEN GOTO 4100
4020   REM here if no room to insert
4030 PRINT "No room for any more records"
4040 GOTO 2100
4100   REM here if room to insert entry
4110 LET m$ = "Insert before": GOSUB 4500
4120 IF n=0 THEN GOTO 200
4130 FOR i = next entry TO n STEP -1
4140 LET d$(i+1) = d$(i): LET e$(i+1) = e$(i)
4150 LET a(i+1) = a(i): LET b(i+1) = b(i)
4160 NEXT i
4170 LET next entry = next entry + 1
4180 GOSUB 3500
4190 GOTO 6100
4500   REM Find record & set n to its number, or
4501   REM    to zero if not found
4502   REM Enter with m$ showing what to do
          with it
4510 PRINT "Note: you must give the date"
4520 PRINT "      EXACTLY as it is shown"
4530 PRINT "      on the statement"
4540 INPUT (m$+" entry dated: "); r$
4550 LET r$ = (r$+"     ")( TO 5)
```

```
4560  FOR n = 1 TO next entry - 1
4570  IF d$(n) = r$ THEN GOTO 4620
4580  NEXT n
4590  INPUT "Not found.  Try again? (Y/N) "; r$
4600  IF r$="Y" OR r$="y" THEN GOTO 4510
4610  LET n=0: RETURN
4620    REM Found record, n=number, see if another
4630  FOR i = n+1 TO next entry - 1
4640  IF d$(i) = r$ THEN GOTO 4700
4650  NEXT i
4660  RETURN: REM identified unambiguously
4700    REM here if more than one with the same date
4710  LET m=n
4720  POKE 23692,20
4730    REM write out relevant part of statement
4740  FOR n=m TO m+17
4750  IF n >= next entry THEN GOTO 4780
4760  GOSUB 500
4770  NEXT n
4780  INPUT "Entry number?  (1 = first on",
        "screen, 2 = second, etc) "; n
4790  LET n = m + n - 1
4800  RETURN
5000    REM Delete entry
5010  LET m$ = "Delete": GOSUB 4500
5020  IF n=0 THEN GOTO 200
5030  FOR i = n TO next entry - 2
5040  LET d$(i) = d$(i+1): LET e$(i) = e$(i+1)
5050  LET a(i) = a(i+1): LET b(i) = b(i+1)
5060  NEXT i
5070  LET next entry = next entry - 1
5080  GOTO 6100
6000    REM alter entry
6010  LET m$ = "Replace": GOSUB 4500
```

```
6020  IF n=0 THEN GOTO 200
6030  GOSUB 3500
6100    REM recalculate balances
6110  INPUT "Recalculate subsequent balances?";
         "   (Y/N) "; r$
6120  IF r$="N" OR r$="n" THEN GOTO 200
6130  IF r$<>"Y" AND r$<>"y" THEN GOTO 6110
6140  FOR i=n TO next entry - 1
6150  LET b(i) = b(i-1) + a(i)
6160  NEXT i
6170  GOTO 200
7000    REM output statement to printer
7010  IF IN 251 <> 255 THEN GOTO 7100
7020  PRINT "You must connect a ZX printer to"
7030  PRINT "   be able to use this facility."
7040  GOTO 2100
7100    REM here if we do have a printer
7101    REM set up characters for the ruled lines
7110  FOR i = 0 TO 15
7120  POKE USR "a" + i, BIN 00001000
7125  POKE USR "c" + i, BIN 01000000
7130  POKE USR "e" + i, BIN 00000000
7140  NEXT i
7150  POKE USR "b" + 3, BIN 11111111
7155  POKE USR "d" + 3, BIN 11111111
7160  POKE USR "f" + 3, BIN 11111111
7170  LPRINT "Date |Details     |Amount|Balance"
7180  LPRINT "-----+-----------+------+-------"
7200  FOR n = 1 TO next entry - 1
7210  LPRINT d$(n); "|"; e$(n); "|";
7220  LET w=6: LET v=a(n): GOSUB 7500
7230  LET r$ = "|": LET v=b(n)
7240  IF v >= 0 AND v < min free THEN LET r$="*"
7250  LPRINT r$;
7260  LET w=7: GOSUB 7500
```

```
7270  NEXT n
7280  GOTO 200
7500    REM write v pence in w characters (2<w<8)
7510  LET v$ = STR$ ABS v
7520  IF LEN v$ >= w THEN LET v$ = " ******"
         (2 TO w)
7530  LET v$ = "    0"(8-w TO 6-LEN v$) + v$
7540  LPRINT INVERSE v<0; v$(1 TO w-3); ".";
         v$(w-2 TO w-1);
7550  RETURN
8000    REM Save on tape
8010  SAVE "Bank" LINE 200
8020  PRINT "Replay tape to verify"
8030  VERIFY "Bank"
8040  GOTO 200
9000    REM exit
9010  PRINT "Finished."
9020  PRINT
9030  PRINT "To re-enter, do GOTO 200"
```

Note that the loop on lines 5030 to 5060, which copies the records back to close up the gap that would otherwise have been left by a deleted record works forwards through the records from the deleted record to the end, whereas the loop on lines 4130 to 4160, which copies them forwards to open up a gap into which a new record can be inserted, works backwards from the end to the point of insertion Consider what would happen if line 4130 was

```
FOR i = n TO next entry
```

so that the program worked forwards through the records: the first time round the loop record $n+1$ would be replaced by a copy of record n; the second time round record $n+2$ would be replaced by a copy of this new record $n+1$ which is the same as record n; the third time round record $n+3$ would be replaced by a copy of the new record $n+2$, which is the same as record n, and so on The end effect is that we have a large number of copies of record n and all the later records have been lost. It is always necessary when shifting blocks of data round in this way to take care that you move the data in the correct sequence and never re-use the space occupied by something until after you have moved it

The subroutine starting at line 4500 asks the user to identify a record by giving its date (which line 4550 ensures is exactly 5 characters long) if there are several records with the same date the user is asked to identify the required one in a portion of the data that is displayed on the screen for the purpose. This assumes that every record will have a date and that the records will be in date order; if the assumption is false the program will not actually crash but the user might not easily be able to identify the required record. An alternative would be to display the statement with a marker against the 'current record' rather like that against the 'current line' in the listing when a ZX BASIC program is being edited, this could be moved around by the cursor control keys (read via INKEY$) and cause the display to scroll when it gets near the top or bottom of the screen

The input and display routines could be modified so that if no date is input the date is assumed to be the same as that on the preceding entry, and when an entry is output its date is omitted if it is the same as that on the previous entry (unless it is the first one on the screen)

After making an alteration in the middle of the data, the user is offered the opportunity of having all the 'balance' figures after the point of alteration recalculated (lines 6100 to 6160). Further extensions to the program which you should consider are to allow the user to input an explicit 'balance' figure, and to allow a group of several entries to be deleted or amended.

Line 7010 tests whether the ZX printer is present. The ZX81 does not have the IN function, but a very simple machine code routine can be used instead: it consists of the six bytes

```
219, 251, 79, 6, 0, 201
```

which we can add to the front of the program in a REM command as

```
1  REM <= CLS ?▞ TAN
```

being careful to use exactly the right characters (looked up in Appendix A of the manual). There is a space character between the graphics character and TAN, but nowhere else. To get CLS, first type THEN (to get into K mode), then type CLS, then go back and rub the THEN out; alternatively type CLS immediately after the line number and then go back and type REM < = afterwards. The question mark represents the character with code 79, which you cannot type directly: use any character when first typing the line in, and after it has been added to the program do

```
POKE 16516,79
```

to insert the correct code. Now USR 16514 can be used in place of IN 251 in line 7010.

Lines 7110 to 7160 set up user-defined graphics for drawing lines on the printer similar to those used on the screen. The graphics-shift A is used for the line between the

'Date' and 'Details' columns C for the other two lines. When you type the program in, these characters will probably appear as capital A and capital C, but after the computer has obeyed lines 7110 to 7160 (which will only happen if you have a ZX printer and invoke item 7 unless you do a deliberate GOTO 7110) they will appear as the appropriate vertical bar symbols. The first plus-shaped symbol on line 7180 is graphics-shift *b*, the other two are *d*s, and the horizontal-line characters are *f*s.

The subroutine at lines 500 to 630 could be rearranged to use these graphics characters also, but it would still be necessary to ensure that the lines dividing the columns from each other extended from top to bottom of the screen, even when there was only one record to be displayed.

11

KEEPING SCORE AT GAMES

Many games involve a certain amount of arithmetic and record-keeping, and the computer can be used for this. The following program keeps score at darts; the total score for three darts is entered as a single number, although because ZX BASIC lets you enter any expression rather than restricting you to a literal number the user can type, for instance,

```
16+3*19+50
```

if the three darts are a 16, a treble 19, and a bull.

The program is written for the Spectrum, but the only changes required for the ZX81 are omission of the colour controls such as 'INK 4', and the alterations (to INPUT and to lines containing more than one command) that should by now be familiar from the earlier chapters.

```
 10   REM darts chalker
 20  DIM n$(2,15): DIM n(2): REM name & its length
 30  DIM s(2): REM score
 40  INPUT "Name of 1st team? "; s$
 50  LET n$(1) = s$: LET n(1) = LEN s$
 60  INPUT "Name of 2nd team? "; s$
 70  LET n$(2) = s$: LET n(2) = LEN s$
 80  INPUT "Starting score? "; start
 90  INK 7: PAPER 0: BORDER 0
100   REM start game
110  CLS: PRINT n$(1); TAB 16; n$(2)
120  PRINT
```

```
 130  FOR i = 1 TO 2
 140  LET s(i) = start
 150  LET s$ = "": GOSUB 1000
 160  NEXT i
 200    REM each side's throw
 210  FOR i = 1 TO 2
 220  INPUT (n$(i,1 TO n(i))+"'s score? "); score
 230  IF score=0 OR score > s(i) OR
          score = s(i)-1 THEN GOTO 270
 240  LET s(i) = s(i) - score
 250  LET s$ = STR$ score: GOSUB 1000
 260  IF s(i) = 0 THEN GOTO 300
 270  NEXT i
 280  GOTO 210
 300    REM here when i'th side has won
 310  PRINT
 320  PRINT FLASH 1; n$(i); " wins!"
 330  INPUT "Another game? (Y/N) "; r$
 340  IF r$<>"Y" AND r$<>"y" THEN GOTO 9999
 350  INPUT (n$(1)+" to start? (Y/N) "; r$
 360  IF r$="Y" OR r$="y" THEN GOTO 100
 370  IF r$<>"N" AND r$<>"n" THEN GOTO 350
 380  LET r$=n$(1): LET n$(1)=n$(2): LET n$(2)=r$
 390  GOTO 100
1000    REM print score s$ and new total for
1001    REM    side number i
1002    REM leaves s$ = total
1010  PRINT INK 4; TAB (16*i-12-LEN s$); s$;
1020  LET s$ = STR$ s(i)
1030  PRINT TAB (16*i-6-LEN s$); s$;
1040  RETURN
```

The program prints two columns for each side the lefthand column being the actual scores (in green chalk) and the righthand column being the amount left (in white) It does not print anything if the score is zero or 'bust'.

To keep a count of the games won by each side, and display it in red at the top of the screen the following lines can be added to the program. Note that when the array *g* is DIMensioned its elements *g*(1) and *g*(2) are set to zero

```
  25  DIM g(2): REM games won

 115  PRINT INK 2; g(1); TAB 16; g(2)

 305  LET g(i) = g(i) + 1

 385  LET score=g(1): LET g(1)=g(2):
         LET g(2)=score
```

If the two sides take it in turns to throw first you can miss out lines 350 to 370. If the loser of one game throws first for the next, you can put

```
        360  IF i=2 THEN GOTO 100
```

in their stead

If you replace line 1030 with

```
1030  PRINT TAB (16*i-6-LEN s$); PAPER s(i)=170 OR
         s(i)=167 OR s(i)=164 OR s(i)<162 AND
         s(i)<>159; s$
```

then the score is on a blue background instead of black if a three-dart finish is available How would you modify the program so that it listed all the possible three-dart finishes from the next player?

To reduce the number of occasions on which the players have to wait for the scorer to catch up, we need to make it as easy as possible to input the scores The following modification allows each dart to be entered separately, or two or three can be entered at once; all the input can be done without pressing any shift keys. The user is expected to use a single space to separate one dart from the next if more than one is entered at a time, and not to use spaces in any other circumstances, this must be clearly explained in documentation or in a message on the screen when the program is started. The reason for this apparently 'user-unfriendly' approach is to allow the input to be typed as quickly as possible.

The score is typed as a number which may be preceded by D or X to indicate a double or T for a treble, B (for bull) may be used instead of 25. Thus double top may be entered as 40 or D20 or X20, treble nineteen as 57 or T19, and a bullseye as 50 or D25 or X25 or DB or XB.

We replace line 220 with

```
220  GOSUB 2000
```

and the subroutine is

```
2000   REM input score for side i
2010  LET r$ = "": LET score=0
2020  POKE 23658,8: REM set caps lock
2030  LET d$="first": GOSUB 2100
2040  LET d$="second": GOSUB 2100
2050  LET d$="third"
2100   REM input one dart and add to SCORE
2101   REM leaves unused data in r$ for next time
2110  IF r$="" THEN GOTO  2140
2120  IF r$(1)=" " THEN LET r$=r$(2 TO): GOTO 2150
2130  BEEP .2,47: BEEP .35,31: BEEP .4,47
2140  INPUT (score;" ";n$(i);"'s ";d$;" dart: ");r$
```

```
2150  IF r$="" THEN RETURN
2160  IF r$(1)=" " THEN RETURN
2170  LET m=1
2180  IF r$(1)="D" OR r$(1)="X" THEN
          LET m=2: GOTO 2210
2190  IF r$(1)<>"T" THEN GOTO 2220
2200  LET m=3
2210  LET r$ = r$(2 TO)
2220  IF r$(1)="B" THEN LET n=25: GOTO 2290
2230  LET n = CODE r$ - 48
2240  IF n<0 OR n>9 THEN GOTO 2130
2250  LET r$ = r$(2 TO)
2260  LET n2 = CODE r$ - 48
2270  IF n2<0 OR n2>9 THEN GOTO 2300
2280  LET n = 10*n + n2
2290  LET r$ = r$(2 TO)
2300  IF r$ = "" THEN GOTO 2320
2310  if r$(1) <> " " THEN GOTO 2130
2320  LET score = score + m*n
2330  RETURN
```

Consider how the program can be enhanced to allow T (for 'top') to be typed instead of 20, so that DT or XT means double top or 40, and TT means treble top or 60. We can process T in the same way as B by adding

```
2225  IF r$(1)="T" THEN LET n=20: GOTO 2290
```

but this on its own is not enough: if the input string consists simply of "T" for 'top' the test at line 2190 will assume it means 'treble' and expect it to be followed by a number or B or T. At line 2240 the program finds that it is not, and 'complains'. This is an example of the adage 'an enhancement is a change to a program as a result of which it no longer works'. We must add

```
2195  IF CODE r$(2 TO ) < 48 THEN LET n=20: GOTO 2290
```

in which we assume T means 'top' if it is at the end of the string or followed by a space, 'treble' if it is followed by a letter or a digit.

The program as written here does not do an exhaustive check for errors; those that will slip by include further characters in the string *r*$ after the third dart has been read, and invalid scores such as 23 or 77 or T25 or X99.

Suggestions for further development are:

(*a*) Recognise when a side wins with its first or second dart,

(*b*) Recognise when a side goes bust (i.e. reduces the amount left to 1 or to a negative number) with its first or second dart, and do not ask for the remaining ones;

(*c*) Only allow a side to win if its last dart was a double (a single or treble that reduces the score to zero counts as 'bust');

(*d*) If the amount left is one of 50, 40, 38, 36, . . ., 2 show it as X25, X20, X19, X18, . . ., X1 to indicate that a one-dart finish is possible;

(*e*) Show the amount left after each dart, unless it is more than 170.

MAH-JONGG

Another game which requires a fair amount of arithmetic in the scoring is mah-jongg. It is a game for four players which is in many ways similar to canasta, but instead of cards it is played with 'tiles' made of bamboo and ivory or (increasingly nowadays) of plastic. Each hand ends either when one player wins or when all the tiles have been used up. In the latter case no scores are counted but otherwise all four players count up the values of the combinations of tiles they hold each player receives the value of his hand from each other player except that the winning player does not pay anything out. There is a further complication that the player who is 'East wind' pays and receives double

Suppose for example that player A wins with a score of 22 B scores 48 C is East wind and scores 4 and D scores 6 Then A receives 22 from B, 44 from C, and 22 from D, and (being the winner of this hand) pays nothing out, so he has a net gain of 88 B receives 96 from C and 48 from D and pays 22 to A 8 to C, and 6 to D, for a net gain of 108 C receives 8 from each of B and D, and pays 44 to A, 96 to B, and 12 to D, for a net loss of 136, and D receives 6 from B and 12 from C, and pays 22 to A, 48 to B, and 8 to C, for a net loss of 60. While the scorer is working all this out, the other three players are 'building the wall' i.e. getting the tiles ready for the next hand

The following program keeps account of the state of play. If the player who is East wind wins the hand, he is East wind again the next time, otherwise the next player becomes East wind The display shows which wind each player corresponds to, both by name and by number, and puts an asterisk by the one that is 'wind of the round' (The number is needed for scoring the flowers and seasons; if the flowers and seasons are not numbered in your set, you will want to put their names on the screen above the names of the winds. Which is wind of the round has a minor effect on the scoring) Below the players' names it shows the scores for the most recent hand (with an asterisk by the winning player's score) and the total points for each player. Whether you regard these as pence, pounds, or just numbers depends on the kind of stakes you like to play for.

```
  10   REM Mah-jongg scoring
  20  DIM n$(4,7): DIM n(4): REM players' names
  30  DIM s(4): REM score this hand
  40  DIM t(4): REM total score
  50  DIM w$(4,5): REM winds
  60  LET w$(1)=" East": LET w$(2)="South"
  70  LET w$(3)=" West": LET w$(4)="North"
 100  FOR i=1 TO 4
 110  INPUT (w$(i);" wind player's name? "); s$
 120  LET n$(i) = s$: LET n(i) = LEN s$
 130  NEXT i
 140  LET e wind = 1: REM Player who is East wind
 150  LET w round = 1: REM wind of the round
 160  LET winner = 0: REM winning player, 0 if none
 200   REM here at start of each hand
 210  GOSUB 2000: GOSUB 2300
 220  INPUT "Winner? (Give number of wind,",
        "or zero if no winner) "; winner
 230  LET winner = INT (winner+0.5)
 240  IF winner<0 OR winner>4 THEN GOTO 220
 250  IF winner=0 THEN GOTO 700
 260  LET winner = winner + e wind - 1
 270  IF winner>4 THEN LET winner = winner-4
 300  FOR i=1 TO 4
 310  INPUT (n$(i,TO n(i));"'s score? "); s(i)
 320  LET s(i) = INT (s(i) + 0.5)
 330  NEXT i
 400   REM Display scores to check
 410  GOSUB 2000: GOSUB 2200
```

```
 420  INPUT "Scores correct? (Y/N) "; r$
 430  IF r$="N" OR r$="n" THEN GOTO 220
 440  IF r$<>"Y" AND r$<>"y" THEN GOTO 420
 500    REM here if OK, work out new totals etc
 510  FOR i=1 TO 3
 520  FOR j=i+1 TO 4
 530    REM work out net amount j pays to i
 540  LET p = s(i)-s(j)
 550  IF i = winner THEN LET p = s(i)
 560  IF j = winner THEN LET p = -s(j)
 570  IF i = e wind OR j = e wind THEN LET p = 2*p
 580  LET t(i) = t(i) + p
 590  LET t(j) = t(j) - p
 600  NEXT j
 610  NEXT i
 700    REM see if winds change
 710  IF winner = e wind THEN GOTO 800
 720  LET e wind = e wind + 1
 730  IF e wind <= 4 THEN GOTO 800
 740    REM end of round
 750  LET e wind = 1
 760  LET w round = w round + 1
 770  IF w round > 4 THEN LET w round = 1
 800    REM display new position
 810  GOSUB 2000: GOSUB 2200: GOSUB 2300
 820  GOTO 220
2000    REM display headings of scoresheet
2010  CLS
2020  FOR i=1 TO 4
2030  LET j = i - e wind + 1: IF j<1 THEN
         LET j = j+4
2040  LET f$=" ": IF j = w round THEN LET f$="*"
2050  PRINT j; f$; w$(j); " ";
2060  NEXT i
2070  PRINT: PRINT
```

```
2100   REM names
2110 FOR i=1 TO 4
2120 PRINT n$(i); " ";
2130 NEXT i
2140 PRINT: PRINT
2150 RETURN
2200   REM display scores last hand
2210 FOR i=1 TO 4
2220 LET f$=" ": IF i=winner THEN LET f$="*"
2230 LET v$ = STR$ s(i)
2240 PRINT "     "(LEN v$ TO 5); v$; f$; " ";
2250 NEXT i
2260 PRINT: PRINT
2270 RETURN
2300   REM display total scores
2310 FOR i=1 TO 4
2320 LET v$ = STR$ t(i)
2330 IF v$(1) > "0" THEN LET v$ = "+" + v$
2340 PRINT "      "(LEN v$ TO 6); v$; " ";
2350 NEXT i
2360 PRINT: PRINT
2370 RETURN
```

Once the scores for the individual hands have been added up, all the scorer has to do is type them into the computer, and the computer does all the arithmetic. The scorer then has no excuse not to help with building the wall.

In the same way that the darts scoring program was enhanced to allow the individual dart scores to be input, this program could be adapted so that the various pungs, kongs, flowers, seasons, etc. are input separately. The amount of detail the scorer would need to type in to do the job thoroughly is so large that it is probably not worth the effort, especially considering that the ZX BASIC allows you to type in a number in a form such as

```
(4+4+8+16+20)*2*2
```

if you do not want to do the adding up and doubling yourself.

Similar techniques can be used to write a program for scoring at bridge. In rubber bridge, for instance, the program should ask for the contract (which side, suit number of tricks, whether doubled) and then for the number of tricks actually made, it is then quite simple to work out the scores, being careful to keep separately the scores above and below the line The program must also allow for extra scores to be added, e g for honours It should keep track of, and show on the screen, which side is vulnerable and when a rubber is won

12

BOARD GAMES ETC. FOR TWO PLAYERS

There are many games which involve two people who move alternately, with a board or some other means of showing the current state of the game. This chapter deals in particular with games in which the outcome depends solely on the players' choice of moves; these include not only board games such as chess and draughts (or checkers), but also games such as nim which do not require a special board or set of pieces. Games which include chance elements such as throwing dice or turning up cards from a pack are mentioned briefly at the end of the chapter.

A game which is simple enough to demonstrate the principles involved without requiring a very large program is noughts and crosses. The following program (written for the Spectrum but easily converted for the ZX81) allows two people to play noughts and crosses without using any paper or pencil.

```
500   REM noughts and crosses
550   REM define graphics shift A
          to be same as on ZX81
560  FOR i = USR "a" TO USR "a"+6 STEP 2
570  POKE i, BIN 10101010: POKE i+1,
          BIN 01010101: NEXT i
700   REM draw the "board"
710  CLS
720  FOR i = 5 TO 15
730  FOR j = 8 TO 12 STEP 4
740  PRINT AT i,j+5; "A"; AT j,i+5; "A"
750  NEXT j: NEXT i
```

```
760  PRINT AT 3,11;"1   2   3"
770  PRINT AT 6,8; "a"; AT 10,8; "b";
         AT 14,8; "c"
800  DIM b$(3,3): REM board matrix
810  LET moves = 0: REM number of moves so far
820  LET p$ = "X": REM player whose move it is
900    REM play game
910  INPUT (p$);"'s move? "; m$
920  IF LEN m$ <> 2 THEN GOTO 910
930  LET x=1: LET y=2
940  IF m$(1)<="3" THEN LET x=2: LET y=1
950  LET i = CODE m$(x) - CODE "A" - 1
960  IF i>3 THEN LET i = i - (CODE "a" - CODE "A")
970  LET j = CODE m$(y) - CODE "0"
980  IF i<1 OR i>3 THEN GOTO 910
990  IF j<1 OR j>3 THEN GOTO 910
1000 IF b$(i,j) <> " " THEN GOTO 910
1010 LET b$(i,j) = p$
1020 LET moves = moves+1
1400   REM display move on board
1410 PRINT AT 2+4*i,7+4*j; p$
1500   REM check for win
1505   REM check rows
1510 FOR n=1 TO 3: FOR m=1 TO 3
1520 IF b$(n,m) <> p$ THEN GOTO 1550
1530 NEXT m
1540 GOTO 1740
1550 NEXT n
1560   REM check columns
1570 FOR n=1 TO 3: FOR m=1 TO 3
1580 IF b$(m,n) <> p$ THEN GOTO 1610
1590 NEXT m
1600 GOTO 1730
1610 NEXT n
```

```
1620    REM check diagonals
1630  IF b$(1,1)=p$ AND b$(2,2)=p$ AND b$(3,3)=p$
          THEN GOTO 1720
1640  IF b$(1,3)=p$ AND b$(2,2)=p$ AND b$(3,1)=p$
          THEN GOTO 1700
1650  LET p$ = CHR$(CODE "X" + CODE "O" - CODE p$)
1660  IF moves<9 THEN GOTO 900
1670  PRINT AT 21,0; "Game drawn"
1680  GOTO 1820
1700    REM cross through winning line
1710  PLOT 84,52: DRAW 80,80: GOTO 1800
1720  PLOT 84,132: DRAW 80,-80: GOTO 1800
1730  PLOT 60+32*n,132: DRAW 0,-80: GOTO 1800
1740  PLOT 84,156-32*n: DRAW 80,0
1800    REM here when game won
1810  PRINT AT 21,0; p$;" wins"
1820  INPUT "Another game? (Y/N) "; a$
1830  IF a$="Y" OR a$="y" THEN GOTO 700
1840  IF a$<>"n" AND a$<>"N" THEN GOTO 1820
```

The As in line 740 are in graphics shift, after the program is first run they will be seen to have changed to grey squares The character string on line 760 has three spaces between the 1 and the 2, and three more between the 2 and the 3

The move is input in the form of a letter and a digit, being the 'map reference' of the square in which the symbol is to be placed The letter and digit may be in either order, and the letter may be in upper or lower case for example the righthand square in the middle row may be identified as 'B3' or 'b3' or '3B' or '3b' The program should explain this to the user if a wrong format is given, so that lines 920, 980, 990, and 1000 should not jump directly to 910 but first output a suitable message

The state of the game is stored in the array *b$*

which has one element for each of the nine squares in the 3×3 grid Each element holds either an X or an O or a space. Making a move consists of writing the appropriate symbol in the appropriate element of *b*$; having made a move we look to see if the player who made the move has won, and if he has not we offer the other player a move.

Although this may seem an obvious way of storing the position, it is by no means the only way Make the following changes to the program add

```
600   REM define value of each square
610  DATA 1344, 4160, 16449
620  DATA 1040, 4369, 16400
630  DATA 1029, 4100, 16644
640  DIM v(3,3)
650  FOR i=1 TO 3: FOR j=1 TO 3
660  READ v(i,j)
670  NEXT j: NEXT i
```

replace lines 800 to 910 by

```
800  DIM m(2,5):  REM moves played
810  LET move = 1:  REM current move number
890   REM each player moves, p=1 for X, p=2 for O
900  FOR p=1 TO 2
910  INPUT "XO"(p); "'s move? "; m$
```

and replace lines 1000 to 1810 by

```
1000  LET k = v(i,j)
1010  FOR n = 1 TO move
1020  IF m(1,n)=k OR m(2,n)=k THEN GOTO 910
1030  NEXT n
1040  LET m(p,move) = k
```

```
1400   REM add move to board
1410  PRINT AT 2+4*i,7+4*j; "XO"(p)
1500   REM check for win (3 in a row)
1510  LET s=0
1520  FOR i=1 TO move
1530  LET s = s + m(p,i)
1540  NEXT i
1550  FOR i=1 TO 8
1560  LET k = INT (s/4)
1570  IF s-k*4 = 3 THEN GOTO 1750
1580  LET s = k
1590  NEXT i
1600  IF move=5 THEN PRINT AT 21,0; "Game drawn":
          GOTO 1820
1610  NEXT p
1620  LET move = move + 1
1630  GOTO 900
1700   REM cross through winning line
1711  DATA  84,  52, 80,  80
1712  DATA  84,  60, 80,   0
1713  DATA  84,  92, 80,   0
1714  DATA  84, 124, 80,   0
1715  DATA  84, 132, 80, -80
1716  DATA  92, 132,  0, -80
1717  DATA 124, 132,  0, -80
1718  DATA 156, 132,  0, -80
1750  RESTORE 1710+i
1760  READ x,y,dx,dy
1770  PLOT x,y: DRAW dx,dy
1800   REM here when game won
1810  PRINT AT 21,0; "XO"(p); " wins"
```

Although the new program behaves in just the same way as far as the user is concerned, the way in which it stores the state of the game is very different. Instead of keeping a record of the symbol that is in each square, it keeps a record of the moves that have been made; moreover it uses a rather peculiar set of numbers to represent the nine possible squares in which each player can place his symbol.

There are eight 'lines' along which a winning set of three symbols can lie. three horizontal, three vertical and two diagonal. The number of symbols a player has on any given line can be 0, 1, 2 or 3. We give the lines values each of which is four times the last viz 1, 4, 16, 64, 256, 1024, 4096 and 16 384, we arbitrarily choose to allocate them in the order

1 diagonal bottom left to top right,
4 bottom horizontal line,
16 middle horizontal line;
64 top horizontal line;
256 diagonal top left to bottom right,
1024 lefthand vertical line;
4096 middle vertical line;
16384 righthand vertical line.

The value of each square is the sum of the values of all the lines it appears in thus the top lefthand square has the value 64+256+1024 and the square in the middle of the bottom row has the value 4+4096. The complete diagram is given below

	1024	4096	16384
256 ↘	↓	↓	↓
64 →	1344	4160	16449
16 →	1040	4369	16400
4 →	1029	4100	16644
1 ↗			

Note that instead of simply loading the values into *v* with the READ command we could have calculated them by

```
600   REM define value of each square
610  DIM v(3,3)
620  LET v(3,1)=1: LET v(2,2)=1: LET v(1,3)=1
630  LET j=4: FOR i=3 TO 1 STEP -1
640  FOR n=1 TO 3: LET v(i,n)=v(i,n)+j: NEXT n
650  LET j=j*4: NEXT i
660  FOR n=1 TO 3: LET v(n,n)=v(n,n)+j: NEXT n
670  FOR i=1 TO 3: LET j=j*4
680  FOR n=1 TO 3: LET v(n,i)=v(n,i)+j: NEXT n
690  NEXT i
```

At line 1000 in the new program we store in *k* the value of the square the player has chosen and then (lines 1010 to 1030) see if either player has used that square already. If not the move is valid and is added to the record of moves and to the picture.

In the earlier program, the code to see if a line of three has been made is very straightforward: for each row, column, and diagonal it looks to see if all three squares contain the symbol of the player who has just moved

The code in the new program is shorter but it is less easy to see what is going on. First we add up all the moves the player has made: lines 1510 to 1540 set *s* equal to the total. Then we repeatedly divide this total by four, *k* is the quotient, and $s-k*4$ is the remainder, so line 1570 looks each time to see if the remainder is 3 Remember that *s* is the total of the scores of all the squares in which the player's symbol has been placed, and each square scores 1 if it is in the bottom-left-to-top-right diagonal, 4 if it is in the bottom row, 16 if it is in the middle row, and so on All the scores except for the diagonal are multiples of 4, all except that and the bottom row are multiples of 16, and so on.

The first time the program divides *s* by 4, therefore, the remainder is the number of moves that were in the bottom-left-to-top-right diagonal If this is 3 then the player has occupied all three squares in the diagonal and has won, lines 1750 to 1770 read the co-ordinates of the line through this diagonal (from line 1711) and draw it Note that the player cannot play in more than three squares along the diagonal, and thus cannot score enough in 1s to 'carry over' into the 4s.

If the first remainder is not 3, *s* is set to one quarter of its previous value i.e the bottom row now scores 1, the middle row 4, the top row 16, and so on. The diagonal that we have already dealt with no longer scores at all On its second time through line 1570, the program looks at the remainder when this new *s* is divided by 4, i e at the number of squares occupied in the bottom row. As before, if it is 3 we jump to line 1750 to draw through the winning line

This time *i* is 2 so the DATA are taken from line 1712

Each time round the loop the count of squares occupied in another row is separated out until all eight have been considered. If one of them proves to have all three squares occupied by the player, the game is won and at line 1770 the program draws through the winning line.

PLAYING AGAINST THE COMPUTER

Having programmed the computer to make moves that are dictated to it, and to recognise when one side has won, the next step is to make the program able to play one side itself. For example we may add the following to the noughts and crosses program.

```
 590  LET auto = 2

 905  IF p=auto THEN GOTO 1100

1050  GOTO 1400
1100    REM computer's move
1110  LET i = INT (RND * 3)
1120  LET j = INT (RND * 3)
1130  GOTO 1000
```

and change line 1020 to jump to line 905 instead of 910 if the square is already occupied. Line 590 defines that the program will play second; you may prefer to set to 1 instead, so that the program will play first, or to ask the user whether the program should play first (*auto*=1) or second (*auto*=2) or not at all (*auto*=0).

In this version, the program's moves are purely random: it keeps choosing a random square until it finds one that is not already occupied, and does not make any effort to form a line of three. If you occupy two squares in a line of three and the third is free, your opponent should move in the third square, but the program is as likely to move in any of the other available squares and let you win. In short, the program does not make any attempt to win the game, nor even to defend itself when it is losing. This makes it a rather unsatisfactory opponent, as it is much too easy to beat.

You can use a similar technique to that on lines 1510 to 1590 to make the program look for two in a row (in which $s - k*4 = 2$), looking first for a row in which it can make a winning move (one in which it scores 2 and you score none) and then for one in which you will make a winning move if it does not get there first (one in which you score 2 and it scores none). Only if no such row is found will it move randomly. Having decided to move in a particular row, it must of course then discover which square to choose: after

```
LET k = INT (v(i,j) / n) / 4
```

the value of

```
k <> INT k
```

will be true if square (i,j) is in the row, column or diagonal that scores n and false if it is not, and it will not take long for the program to simply try all the squares that are not yet occupied until the correct one is found.

The program would not then play like a complete idiot but you would still be able to beat it fairly often, and it would only beat you if you were both careless and unlucky at the same time.

For the program to be able to play more competently it needs to be able to look ahead and see what further moves will be possible from each of the positions it can move to. For instance, consider the following position in which it is O's move; the squares are numbered in the same way as in the program:

```
              1      2      3

    a             |      |  0
                -----+-----+-----
    b             |  X   |
                -----+-----+-----
    c         X   |    . |
```

If O moves into square *b*1, X is then able to move into square *c*3 giving the position

```
                  |      |  0
                -----+-----+-----
              0   |  X   |
                -----+-----+-----
              X   |      |  X
```

from which X can win because he can complete a line by playing at either *a*1 or *c*2. if O blocks one of them by playing at *a*1, say, X can still play at *c*2 and win before O has a chance to complete the line in row *a*.

Thus although the move in *b*1 would not be seen as a losing move by the program just described, we can see that if O plays this move then he will lose unless his opponent is very careless. Indeed, of the six possible moves in this position four (*a*2, *b*1, *b*3, *c*2) are losing moves so that if you are X in this position you have a 2 to 1 chance of beating the program.

(If you make your first move in the centre, then the program will move either in a corner square such as *a*3 or in a centre-edge square such as *a*2. In the former case you can, as we have seen, win two games out of three by playing in the opposite corner; in the latter case you can always win by playing anywhere except directly opposite: if the program has played in *a2* you can win if you play in any square except *c*2. Because the program plays randomly, if you start in the centre every time then, on average, out of every six games the program will play in a corner square in three, two of which you will win, and in a centre-edge square in three, all three of which you will win. On average, then you should win five out of six games, so the odds are 5 to 1 in your favour.)

If the program is to find the 'best' move in any position then it must be able to ascribe a value to each possible move, and have objective criteria for calculating this value. In fact we tend to talk interchangeably about the value of a move and the value of a position: the value of a move is the same thing as the value of the position moved to. In many two-person games, including noughts and crosses, the possible values are simply 'win', 'draw', and 'lose', which are often represented as 1, 0, and −1. (In other games it can matter not just whether you win or lose but also by what margin: you may have a choice of three moves, all of which lose, but if one loses you £1 and the others lose you £5 you will choose the £1 move.)

The rule used by a program to find the value of a position (except one at the end of the game for which we know the value anyway) is the value of a position to the player whose turn it is to move is the greatest of the values of the moves he has available to him.

Thus if it is your move and you have a winning move available to you then you are in a winning position, even if you have a lot of other moves available which do not win. From your opponent's point of view, of course, the situation is reversed if there is one move available which will result in him losing then it is a losing position (although he can always hope that you will not spot the vital move)

The basic structure of a routine which finds the value of a position in this way is:

```
define "value of (p)" as:

     if end of game then [calculate value directly]

     else:  let v = worst possible value for the
                    current player

            now let q be, in turn, each position
                    we can move to
            for each q, let v2 = value of (q), and
                    if v2 is better than v then
                    let v = v2

            when all moves have been considered,
                    value is v.
```

This causes two main problems when we try to implement it in BASIC. First, the routine is 'recursive', which means that it is defined in terms of itself. Suppose we have a position *p*1 from which we can move to *p*2, *p*3, or *p*4, and suppose *p*2 is a drawn position (at the end of a game) but *p*3 is a position (not at the end of a game) which turns out to be lost. When *value of* (*p*3) is worked out, variable *v* is used to hold 'value of the best move from *p*3 so far found' (in this case 'lose'), but we must not overwrite the variable *v* which holds 'value of the best move from *p*1 so far found' (in this case 'draw'). Languages which are 'block structured', such as Algol and Pascal, take care of this kind of problem more or less automatically, but in BASIC we need to make provision for it in the program.

Secondly, this innocent-looking little routine can take an enormous amount of time to run. Suppose we are looking at the first move in a game of noughts and crosses: there are nine possible moves, so the routine is called for each of them. Within each of these nine calls, the routine is called again for each of the second player's eight possible moves, a total of 9×8 or 72. Within each of these 72 calls, the routine is called again for each of the first player's seven possible second moves, a total of 72×7 or 504 calls at this level. Within these we have 504×6=3024 calls to find the value of the second player's second move, which in turn involves 15 120 calls to find the value of the first player's third move. Of these, 2880 will be for 'end-of-game' positions in which the first player has won but the other 12 240 all give rise to further calls: it can be shown that for each one there will be at least 13 but less than 64 further calls.

Adding up all the calls at the different levels we can see that there will be at least 177 850 (but less than

802 090) calls, every one of which must at least check to see whether the game has been won. If we reckon that this will take around a hundredth of a second each time, it means that the program will take between half an hour and two hours to decide on its first move. In fact the time per call is likely to be nearer a tenth of a second than a hundredth, so the program could take anything up to 20 hours over its first move if it plays first, and 2 hours if it plays second.

There are two ways that this time can be reduced. It is clearly helpful if we can reduce the time taken to discover if a position is a winning position, but it is equally clear that we must make a significant reduction in the number of positions that the program considers.

The method of identifying winning positions used in the second version of the program in this chapter works very well in machine code and in some programming languages. This is because the number *s* which the program works out on lines 1510 to 1540 is held inside the computer as a bit string 16 bits long with 2 bits for each row, and the computer can in two or three operations (which work on a bit string as a whole) find out whether any row has the value 3

RECURSION – a technique whereby a function or routine is defined in terms of itself (e.g. factorial *n* defined as 'if *n*=0 then 1 else *n* • factorial *n*−1') so that during evaluation (say of factorial 5) the computer breaks off to go through the same code with different data (to evaluate factorial 4, 3, etc).
ITERATION – the alternative to recursion, in which the repetition of part of the code is explicit (as in the FOR loop) rather than implied by a call. An iterative definition of factorial *n* would be 'let *f*=1: for *i*=1 to *n*: let *f*=*f*•*i*: next *i*'

However, in BASIC we have to break s down into eight numbers, and moreover we have to do this using the 'divide' operator, which is one of the slower ones.

(The new program given below in fact retains the old method in the part that actually makes the moves. This is not strictly necessary although it does give a check that the part of the program that chooses the computer's move is not 'cheating'. It was done to avoid changing the program more than is necessary to add the new facility.)

The program below uses the following arrays:

p(9) holds the content of each of the squares as 1 for X, −1 for O, zero if empty. The squares are numbered 1, 2, ..., 9 rather than (1,1), (1,2) ... (3,3) to save time.

w(8) saves the value of *v* (which is essentially the *v* in the informal description of the routine above) at each 'level' of call.

n(9) similarly saves *i*, which keeps track of which move we are considering.

r(45) consists of five numbers for each square, being the number of each line (row, column or diagonal) the square is in, followed by enough zeros to make up five numbers (see lines 2010 to 2090). Again, one subscript rather than two is used for speed.

c(8) counts the symbols in each line: 3 for three Xs (so X has won), 2 for two Xs (so X can win if it is his move), 1 for one X or two Xs and an O, 0 for no symbols at all, −1 for one O or two Os and an X, −2 for two Os, −3 for three Os.

s(8,3) shows which squares make up each line (three squares to each of the eight lines: see lines 2100 to 2130 of the program). In this case there would be little advantage in making it use a single subscript.

The lines of squares are in the same order as before but numbered 1 to 8 so that 1 and 5 are the diagonals, 2 to 4 the rows, and 6 to 8 the columns.

The routine has been put at the top of the program to reduce the time taken for GOTO and GOSUB, as explained in an earlier chapter. It works as follows

GOSUB 100 calculates and stores in v, the value of the position after player q has moved in square i, q is 1 if the player is X, -1 if the player is O. The value is 1 if the position is a win for X, -1 if it is a win for O, zero if it is a draw.

After updating array p it sets about updating array c. Array r has already been loaded with the data from lines 2010 to 2090. Suppose for instance that $i=3$ so that $i \times 5-4 = 11$; j will be set to 11 so the program reads the first of the numbers loaded from line 2030, and we repeat lines 110 to 130 with $r(j)$ being in turn 1, 4, and 8 because square number 3, which is at the top right, is in lines 1 (diagonal), 4 (top row), and 8 (righthand column). Each time we update one of the elements of c, we check if the line now has two or three of the current player's symbols in it. On line 150 we recognise the position as a win if n is at least 2, which means that either there is now a row of three or else there is more than one row of 2, as in

```
  X  |  O  |  X
-----+-----+-----
     |  X  |
-----+-----+-----
     |  O  |
```

when X has just moved each of the diagonals has two Xs, and whichever of the bottom corners O plays in X can play in the other and win. Note that the top row only scores 1 and thus does not count as having two Xs

If there is just one line that scores 2, the opponent must play in the third square in that line; line 170 finds which square is still free. Thus after Xs move in

```
  X  |  X  |
-----+-----+-----
     |     |
-----+-----+-----
  0  |     |
```

O must play in the top righthand corner and the program does not waste time considering the other five squares.

Another test which reduces the number of positions considered is at the end of line 240, which looks to see if a winning move for the relevant player has been found. If it has, we have a winning position for that player and do not need to look at any further moves.

The code to be added to the earlier program is as follows. Only line 1040 replaces an existing line, the rest is additional to the previous code.

```
 10  GOTO 500
 90    REM set v = value of move i
100  LET p(i)=q: LET j=i*5-4: LET n=0
110  LET k=c(r(j)): LET c(r(j))=k+q:
          IF k=q THEN LET n=n+1: LET n2=j
120  IF k=q+q THEN LET n=2
130  LET j=j+1: IF r(j)<>0 THEN GOTO 110
140  IF n=0 THEN GOTO 210
150  IF n>1 THEN LET v=q: GOTO 320: REM win
```

```
160  LET n(m)=i: LET q=-q: LET m=m+1:
         REM opponent's move forced
170  LET i=s(r(n2),n): IF p(i)<>0 THEN
         LET n=n+1: GOTO 170
180  GOSUB 100: GOTO 310
200    REM look at all opponent's moves
210  IF m>7 THEN LET v=0: GOTO 320
220  LET n(m)=i: LET m=m+1: LET w(m)=q:
         LET q=-q: LET i=1
230  IF p(i) <> 0 THEN GOTO 260
240  GOSUB 100: IF v=q THEN GOTO 310
250  IF v=0 THEN LET w(m)=0
260  LET i=i+1: IF i<10 THEN GOTO 230
270  LET v=w(m)
300    REM now v = value;  undo move & exit
310  LET m=m-1: LET q=-q: LET i=n(m)
320  LET j=i*5-4
330  LET c(r(j))=c(r(j))-q: LET j=j+1:
         IF r(j)>0 THEN GOTO 330
340  LET p(i)=0: RETURN
500    REM
501    REM start of program proper
510  LET q=0: LET m=0: LET i=0: LET k=0:
         REM so they are first in v'bles area
520  DIM p(9): DIM w(8): DIM n(9): DIM r(45):
         DIM c(8): DIM s(8,3)
530  RESTORE 2000
540  FOR n=1 TO 45: READ r(n): NEXT n
550  FOR i=1 TO 3: FOR n=1 TO 8: READ s(n,i):
         NEXT n: NEXT i
560  FOR i=USR "a" TO USR "a"+6 STEP 2
570  POKE i, BIN 10101010: POKE i+1, BIN 01010101:
         NEXT i
590  LET auto=2
905  IF p=auto THEN GOTO 1110
```

```
1040  GOTO 1400
1100    REM computer's move
1110  LET m=2*move+p-2: LET q=3-p-p: LET i=1
1130  FOR n=1 TO 8: IF c(n)=q+q THEN GOTO 1230
1140  NEXT n
1150  FOR n=1 TO 8: IF c(n)=-q-q THEN GOTO 1230
1160  NEXT n
1170    REM evaluate each possible move
1180  IF p(i) <> 0 THEN GOTO 1210
1190  GOSUB 10: IF v=q THEN LET i2=i: GOTO 1370
1200  IF v=0 THEN LET i2=i
1210  LET i=i+1: IF i<10 THEN GOTO 1180
1220  GOTO 1370
1230  LET i2=n
1300    REM move in row i2
1310  IF p(s(i2),i)) <> 0 THEN LET i=i+1: GOTO 1310
1320  LET i2=s(i2,i)
1350    REM move in square i2
1370  LET i = INT ((i2+2)/3): LET j = i2 - i*3 + 3
1400    REM move in square (i,j)
1410  PRINT AT 2+4*i,7+4*j; "XO"(p)
1420  LET m(p,move) = v(i,j)
1430  LET q = 3-p-p
1440  LET p(3*i+j-3) = q
1450  LET n = 15*i + 5*j - 19
1460  LET k = c(r(n)): LET c(r(n)) = k+q
1470  LET n=n+1: IF r(n) <> 0 THEN GOTO 1460

2000    REM lines each square is in
2010  DATA 4,5,6,0,0
2020  DATA 4,7,0,0,0
2030  DATA 1,4,8,0,0
2040  DATA 3,6,0,0,0
```

```
2050  DATA 1,3,5,7,0
2060  DATA 3,8,0,0,0
2070  DATA 1,2,6,0,0
2080  DATA 2,7,0,0,0
2090  DATA 2,5,8,0,0
2100    REM squares in each line (read downwards)
2110  DATA 7, 7,4,1, 1, 1,2,3
2120  DATA 5, 8,5,2, 5, 4,5,6
2130  DATA 3, 9,6,3, 9, 7,8,9
```

This version (which is written so that the computer plays second) still looks at more positions than it needs to. It does not recognise symmetrical positions: for instance if your first move is in the centre, the program finds the value of moving in each of the eight remaining squares even though all the corner squares must have the same value, as must all the centre edge squares. By a more careful analysis of how the lines of squares are occupied, it could identify drawn positions, and winning moves, sooner.

(To win, you need two intersecting lines in each of which there are none of your opponent's symbols and just one of your own, which must not be in the square which is common to both lines. If this situation exists, the move in the square that is common to both lines is a winning move and no other moves need to be considered. A position in which neither side has two such lines available is drawn. If the current player does not have two such lines available and the first move we look at achieves a draw, the position is drawn and we do not need to look at the other moves as we know none of them can win.)

The program takes several minutes over its first move, and if it was changed so that the computer moved first rather than second it would take about nine times as long. However, we can add

```
1020  IF m<3 THEN LET i2=5: GOTO 1350

1360  IF p(i2) <> 0 THEN LET i2=9: REM if
             from 1020 with m=2 & sq 5 occ'd
```

so that if it moves first it always starts in the centre and if it moves second it moves in the centre unless you have already moved there in which case it goes in one of the corners. This dramatically reduces the time required for the program to decide on its first move. It also makes the question of symmetry much less important, as it is in the earliest stages of the game that the symmetrical positions mostly occur.

You might like to see all the various positions the program considers during its deliberations. this can be done by adding

```
  90  LET pos=0

 335  PRINT AT 20,pos+4;m;: FOR x=3 TO 9 STEP 3:
             PRINT AT x/3+18,pos;: FOR k=x-2 TO x:
             PRINT " " AND p(k)=0; "O" AND p(k)<0;
             "X" AND p(k)>0;: NEXT k: NEXT x:
             PRINT " ";v;:
             LET pos=pos+6: IF pos>28 THEN
             LET pos=0:
             PRINT '''': IF PEEK 23692>12 THEN
             POKE 23692,12
```

which prints out the position, move number, and value before returning from each call of the subroutine

Unfortunately it also destroys the display of the board and increases the time the program takes to run.

Although keeping extra data about the position can help reduce the amount of calculation, it is important to remember that although it saves work it also creates extra work maintaining the extra data structures. For instance, suppose that instead of the array *p*, which shows what is in each square, we kept a list of the squares that have not yet been occupied. Then for the eighth or ninth move in the game the program would not need to search through *p* looking for the one or two remaining squares. But the effort of maintaining the list would be more than the small amount of looping around lines 230 and 260 that would be saved.

In contrast to the earlier programs which hardly ever won and often lost, this program never loses. As an opponent you might find this even more unsatisfactory, but it is a fault of the game rather than of the program. Some possible improvements include choosing a move at random from among the available moves (excluding, of course, those moves that would lose) instead of the present regime of always taking the last one found, and perhaps making occasional random moves which might be losing moves (so the program behaves more like a human player). It could also take account of having a fallible opponent by preferring moves from which a win could occur if its opponent made a mistake, for instance all the possible second moves for X after

```
     |     |  O
-----+-----+-----
     |  X  |
-----+-----+-----
     |     |
```

are drawn, but whereas most of them more or less force a draw, playing in the bottom lefthand corner gives O plenty of opportunity to make a losing move.

OTHER GAMES

Other two-person games such as chess, go draughts, and othello, are programmed in essentially the same way The computer works out the value of the various available moves using the algorithm given earlier, and the number of positions to be considered has to be kept within reasonable bounds The main techniques we used here were to treat the opening moves of the game specially relying on past experience rather than analysing the position afresh every time; to recognise when a player's move is forced; and to recognise won and drawn positions as early as possible.

Except with very simple games like noughts and crosses it s also necessary to limit how many moves ahead the program looks. (In the program above, once we had eliminated the first two moves the length of the game limited it to looking about six moves ahead) When it reaches the limit, it has to use some other measure of the 'value' of a position, in chess this might take account of the number of pieces each side has on the board, which pieces are *en prise*, and the extent to which each side has control of the centre of the board The program needs to be somewhat flexible about where the limit comes not stopping in the middle of a sequence of captures or while a player is in check.

In general you should expect that to write a program that plays a game as complex as chess at a l competently you will need to use a language that is rather more efficient than ZX BASIC.

For games that involve a random element, the amount of looking ahead that can be done is severely constrained by not knowing what card will be turned up next or what number will come up at the next throw of the dice. The program can of course, consider all the possible outcomes of the random element but this is likely to cause a big increase in the number of positions to be looked at, and hence decrease the distance ahead that the program can look in a reasonable time. The program's performance therefore depends much more on considering the apparent worth of a position than on considering the moves that can be made from it how this 'apparent worth' can best be calculated will depend very much on the game and it is difficult to give any general rules for it except to reiterate that it must be able to be calculated from numerical properties of the position and cannot include any subjective criteria.

13

ANIMATION

The main limitation to providing moving pictures on the TV screen is the slowness of the ZX BASIC language. In most cases a fair amount of calculation is needed to produce each frame of a moving picture and it is unrealistic to expect to be able to do it in the 25th of a second or so that would be needed to produce a picture that appears to move smoothly.

For some kinds of moving display, you do not need to change the picture this often. To display a clock, for instance, you only need to change the picture once per second the manual contains a suitable program (at the start of Chapter 19 in the ZX81 manual Chapter 18 in the Spectrum manual) for one which contains only a second hand, and one of the exercises at the end of that chapter suggests you should extend the program to draw the hour and minute hands as well

To get an idea of how fast the Spectrum can draw and redraw things, use the following 'jiffy' program (A 'jiffy' program is a short program that is written quickly and is intended to be ephemeral Hence, for instance, we do not bother to put captions in the INPUT command)

```
10  OVER 1
20  INPUT k,s
30  LET n = INT (176/k) - 1
40  LET m = (n+1) * (k-1)
50  FOR i=s TO 255 STEP s
60  FOR j=0 TO m STEP n+1
```

```
 70  PLOT i,j: DRAW 0,n
 80  PLOT i-s,j: DRAW 0,n
 90  NEXT j
100  NEXT i
110  GOTO 20
```

This draws a line up the screen in *k* segments and moves it across the screen in steps of *s* pixel-widths at a time (It also, rather messily, leaves a line at each side of the screen this can be eliminated by adding

```
 43  FOR j=0 TO m STEP n+1
 45  PLOT 0,j: DRAW 0,n
 47  NEXT j

103  FOR j=0 TO m STEP n+1
105  PLOT i-s,j: DRAW 0,n
107  NEXT j
```

but you might not think it worth the effort)

By experimenting with different values of *k* and *s* you can see just how fast (or not so fast!) the computer can move things around on the screen When *k* is small it has very little effect on the speed because most of the time is taken up actually drawing the line, but as *k* gets larger the various overheads such as interpreting the DRAW and PLOT commands become more important. You could investigate the effect of line length by INPUTting a third parameter, *l* say, and replacing *n* with *l* in lines 60 and 70 (and 45 and 105 if you have them), or by simply replacing *n* in these lines by a constant.

For the ZX81 a similar program can be used

```
 10  SLOW
 20  INPUT k
```

```
 30  INPUT s
 40  LET n = INT (44/k)
 50  LET m = n * (k-1)
 60  FOR i = s TO 63 STEP s
 70  FOR j = 0 TO m STEP n
 80  PLOT i,j
 90  UNPLOT i-s,j
100  NEXT j
110  NEXT i
120  GOTO 20
```

Again, it shows how a very small amount of redrawing takes a noticeable amount of time It has to run in SLOW mode, because otherwise you do not see anything until it has finished

The following program for the Spectrum draws a matchstick figure which walks across the screen.

```
 10  DIM u(7): DIM v(7): REM params for current
        figure
 20  DIM t(7): DIM w(7): REM params for previous
        figure
 30  DIM q(7): REM preserves old u() in
        calculations
 40  DIM x(7): DIM y(7): REM params for first
        figure
100    REM parameters for start of stride
110  LET x(1) = 40: LET y(1) = 30
120  LET x(2) = -2.5 * SQR 3: LET y(2) = -2.5
130  LET x(3) = -20: LET y(3) = -8 * x(2)
140  LET x(4) = x(1) + x(3):
         LET y(4) = y(1) + y(3) + 1
150  LET x(5) = -10: LET y(5) = 4 * x(2)
160  LET x(6) = -10: LET y(6) = y(5)
```

```
170  LET x(7) = 5: LET y(7) = 0
200    REM initialise "previous figure"
210  FOR i=1 TO 7
220  LET t(i) = x(i): LET w(i) = y(i)
230  NEXT i
300    REM set up coefficients for rotations
301    REM the number following c or s is in
          degrees
310  LET c3 = COS (PI/60): LET s3 = SIN (PI/60)
320  LET c4 = COS (PI/45): LET s4 = SIN (PI/45)
330  LET c7 = COS (7*PI/180):
        LET s7 = SIN (7*PI/180)
340  LET c10 = COS (PI/18): LET s10 = SIN (PI/18)
400    REM now draw first figure
410  OVER 1
420  PLOT x(1)-x(2),y(1)-y(2): DRAW x(2),y(2):
          DRAW x(3),y(3)
430  DRAW x(5),y(5): DRAW x(6),y(6):
        DRAW x(7),y(7)
440  PLOT x(4),y(4): DRAW 0,40: DRAW 7,-22:
        DRAW 24,3
450  PLOT x(4),y(4)+40: DRAW 15,15: DRAW -15,15:
          DRAW -15,-15: DRAW 15,-15: DRAW -7,-25:
          DRAW 23,-7
500  FOR i=2 TO 7
510  LET u(i) = x(i): LET v(i) = y(i)
520  NEXT i
600    REM here to draw each subsequent figure
610  FOR i=1 TO 20
620  FOR j=2 TO 7: LET q(j) = u(j): NEXT j
700    REM update params for next figure
710  IF i<11 THEN LET u(2) = c3*u(2) + s3*v(2):
          LET v(2) = c3*v(2) - s3*q(2)
720  LET u(3)=c3*u(3)+s3*v(3):
        LET v(3)=c3*v(3)-s3*q(3)
```

```
730  LET u(4) = x(1)+u(3): LET v(4) = y(1)+v(3)
740  IF i<11 THEN LET u(5) = c10*u(5) - s10*v(5):
          LET v(5) = c10*v(5) + s10*q(5)
750  IF i>16 THEN LET u(5) = c10*u(5) + s10*v(5):
          LET v(5) = c10*v(5) - s10*q(5)
760  IF i<5 THEN LET u(6) = c10*u(6) + s10*v(6):
          LET v(6) = c10*v(6) - s10*q(6)
770  IF i>10 THEN LET u(6) = c10*u(6) - s10*v(6):
          LET v(6) = c10*v(6) + s10*q(6)
780  IF i<11 THEN LET u(7) = c7*u(7) + s7*v(7):
          LET v(7) = c7*v(7) - s7*q(7)
790  IF i>10 THEN LET u(7) = c10*u(7) - s10*v(7):
          LET v(7) = c10*v(7) + s10*q(7)
800    REM undraw old & draw new
810  PLOT t(1)-t(2), y(1)-w(2): DRAW t(2), w(2):
          DRAW t(3), w(3): DRAW t(5), w(5):
          DRAW t(6), w(6): DRAW t(7), w(7)
820  PLOT x(1)-u(2), y(1)-v(2): DRAW u(2), v(2):
          DRAW u(3), v(3): DRAW u(5), v(5):
          DRAW u(6), v(6): DRAW u(7), v(7)
830  PLOT t(4), w(4): DRAW 0,40: DRAW 7,-22:
       DRAW 24,3
840  PLOT u(4), v(4): DRAW 0,40: DRAW 7,-22:
       DRAW 24,3
850  PLOT t(4), w(4)+40: DRAW 15,15: DRAW -15,15:
          DRAW -15,-15: DRAW 15,-15: DRAW -7,-25:
          DRAW 23,-7
860  PLOT u(4), v(4)+40: DRAW 15,15: DRAW -15,15:
          DRAW -15,-15: DRAW 15,-15: DRAW -7,-25:
          DRAW 23,-7
900  FOR j=2 TO 7
910  LET t(j) = u(j): LET w(j) = v(j)
920  NEXT j
930  LET t(1) = x(1)
940  NEXT i
```

```
950  LET x(1) = x(1) + 40
960  GOTO 600
```

There are 20 separate 'frames' to each stride Lines 100 to 340 set up various parameters and calculate the sines and cosines of the angles that are going to be needed (3, 4, 7, and 10 degrees) to rotate the various parts of the legs from one frame to the next Lines 400 to 450 draw the first frame, and lines 600 to 940 draw each subsequent frame, erasing the previous frame as it goes. A diamond shape is used instead of a circle for the figure's head because circles take much longer to draw.

The picture moves very slowly and rather jerkily, the program can be altered so that it calculates all the parameters (*u, v t, w*) for each frame first and stores them in arrays but this does not make it run much faster as much of the time is taken up actually drawing the lines

Often the picture does not need to be moved smoothly In Space Invader type games, for instance, the effect is of the phalanx of aliens moving across the screen from left to right and back again. If it was done by moving this part of the picture smoothly back and forth across the screen, it would require a great deal of work on the part of the computer, but if you look closely you can see that what actually happens is that the individual aliens jump sideways one at a time This gives the effect of a smooth movement of the whole population, but in fact only a small amount of the screen is updated at a time, and that updating involves a jump of quite a large distance. You would draw the aliens using PRINT AT and graphics characters rather than with PLOT so the sideways jump would be the width of a character square.

The following ZX81 program generates a picture which, while not containing any movement as such, is constantly changing.

```
10  PRINT AT RND*21,RND*31; CHR$(RND*10)
20  GOTO 10
```

A similar thing can be done in colour on the Spectrum:

```
10  PRINT AT RND*21,RND*31; PAPER RND*7; " "
20  GOTO 10
```

Unlike this book, these programs never finish.

The ZX PROGRAMMERS' COMPANION

John and Catherine Grant

Grant & Grant

The ZX PROGRAMMERS' COMPANION

CAMBRIDGE

The ZX Programmers' Companion introduces the new programmer to the art and science of programming using the popular ZX machines and equivalent TS machines in the USA.

The instruction manual that comes with the ZX computer has to be an introduction to all the facilities provided on the machine and how they are used. It does not have the space to say much about how to write programs to do particular jobs. *The ZX Programmers' Companion* complements the manuals by explaining how to set about designing and writing programs for the ZX computers, and contains many examples of the kind of program that the ZX user might need. The steps in deciding the most appropriate way to tackle each problem are discussed and, finally, fully documented programs are given.

The authors' company, Nine Tiles Information Handling Ltd, was responsible for writing the instruction manuals and the built-in software for the ZX 81 and Spectrum machines, and this companion volume will be essential reading for all ZX users.

ISBN 0-521-27044-8