artificial intelligence on the commodore 64

make your micro think

keith & steven brain

artificial intelligence on the commodore 64

make your micro think

keith & steven brain

First published 1984 by:
Sunshine Books (an imprint of Scot Press Ltd.)
12–13 Little Newport Street,
London WC2R 3LD

British Library Cataloguing in Publication Data
Brain, Keith
Artificial intelligence on the Commodore 64.
1. Artificial intelligence 2. Commodore 64
(Computer)
I. Title II. Brain, Steven
001.53'5'0285404 Q335

ISBN 0-946408-29-7

Cover design by Graphic Design Ltd.
Illustration by Stuart Hughes.
Typeset by V & M Graphics Ltd, Aylesbury, Bucks.
Printed in England by Short Run Press Ltd, Exeter.

CONTENTS

Contents in detail

Introduction

Artificial Intelligence is undoubtedly an increasingly important area in computer development which will have profound effects on all our lives in the next few decades. The main aim of this book is to introduce the reader to some of the concepts involved in Artificial Intelligence and to show them how to develop 'intelligent' routines in BASIC which they can then incorporate into their own particular programs. Only a superficial knowledge of BASIC is assumed, and the book works from first principles as we believe that this is essential if you are really to understand the problems involved in producing intelligence, and how to set about overcoming them.

The basic format of the book is that ideas are taken and suitable routines built up step by step, exploring and comparing alternative possibilities wherever feasible. Rather than simply giving you a series of completed programs, we encourage you to experiment with different approaches to let you see the results for yourself. Detailed flowcharts of most of the routines are included. The main emphasis in the routines is placed on the AI aspects and we have therefore avoided 'tarting up' the screen display as this tends to obscure the significance of the program. In places you may notice that odd lines are redundant, but these have been deliberately included in the interests of clarity of program flow. As far as possible, retyping of lines is strenuously avoided but modification of lines is commonplace. All listings in the book are formatted so that they appear as you will see them on the screen. In most cases, spaces and brackets have been used liberally to make listings easier to read but be warned that some spaces and brackets are essential so do not be tempted to remove them all. All routines have been rigorously tested and the listings have been checked very thoroughly so we hope that you will not find any bugs. It is a sad fact of life that most bugs arise as a result of 'tryping mitsakes' by the user. Semi-colons and commas may look very insignificant but their absence can have very profound effects!

Artificial Intelligence is increasing in importance every day and we hope that this book will give you a useful insight into the area. Who knows — if you really work at the subject you might be able to persuade your machine to read our next book for itself!

Keith and Steven Brain
Groeswen, January 1984

CHAPTER 1
Artificial Intelligence

Fantasy

For generations, science fiction writers have envisaged the development of intelligent machines which could carry out many of the functions of man himself (or even surpass him in some areas), and the public image of Artificial Intelligence has undoubtedly been coloured by these images. The most common view of a robot is that it is an intelligent machine of generally anthropomorphic (human) form which is capable of independently carrying out instructions which are given to it in only a very general manner.

Of course, most people have ingrained Luddite tendencies when it comes to technology so in the early stories these robots tended to have a very bad press, being cast in the traditional role of the 'bad guys' but with near-invincibility and lack of conscience built in. The far-sighted Isaac Asimov wove a lengthy series of stories around his concept of 'positronic robots' and was probably the first author really to get to grips with the realities of the situation. He laid down his famous 'Three Laws of Robotics' which specified the basic ground rules which must be built into any machine which is capable of independent action — but it is interesting to note that he could not foresee the time when the human race would accept the presence of such robots on the earth itself.

'Star Wars' introduced the specialised robots R2D2 and C3PO, but we feel that many of their design features were a little strange. Perhaps there is an Interplanetary Union of Robots, and a demarcation dispute prevented direct communication between humans and R2D2. In 'The Stepford Wives', the local husbands got together and had the (good?) idea of converting their wives into androids who automatically did exactly what was expected of them, but the sequel revealed the dangers of the necessity to continuously reinforce with an external stimulus! Perhaps one hope for mankind is that any aliens who chance upon us will not have watched 'Battlestar Galactica', and will therefore build robots of the Cylon type who, rather like the old Space Invaders, are always eventually defeated because they are totally predictable.

Of course intelligent computers also appear in boxes without arms and legs, although flashing lights seem obligatory. Input/output must obviously be vocal but the old metallic voice has clearly gone out of fashion in favour of some more definite personality. If all the boxes look the same then this must be a good idea, but please don't make yours all sound like Sergeant-Major Zero from 'Terrahawks'! Michael Knight's KITT sounds like a reasonable sort of machine to converse with, and it is certainly preferable to the oily SLAVE and obnoxious ORAC from 'Blake's Seven'. ORAC seemed to pack an enormous amount of scorn into that little perspex box, but other writers have appreciated the difficulties which may be produced if you make the personality of the machine too close to that of man himself.

In Arthur C. Clarke's '2001: A Space Odyssey', the ultimately-intelligent computer HAL eventually had a nervous breakdown when he faced too many responsibilities; but in 'Dark Star' the intelligent bomb was quite happy to discuss Existentialism with Captain Doolittle but was unwilling to deviate from his planned detonation time, although still stuck in the bomb bay. In 'The Restaurant At The End of The Universe', the value of the Sirius Cybernetics Corporation Happy Vertical People Transporter was reduced significantly when it refused to go up as it could see into the future and realised that if it did so it was likely to get zapped; and the Nutri-Matic Drinks Synthesiser was obviously designed by British Rail Catering as it always produced a drink that was 'almost, but not quite, entirely unlike tea'.

More worrying themes have also recently appeared. The most significant feature of 'Wargames' was not that someone tapped into JOSHUA (the US Defence Computer), but that once the machine started playing thermonuclear war it wouldn't stop until someone had won the game. And in 'The Forbin Project' the US and Russian computers got together and decided that humans are pretty irrelevant anyway. Of course, if you are Marvin the Paranoid Android and have a brain the size of a planet and a Genuine People Personality, you can succeed without weapons by confusing the enemy machine into shooting the floor from under itself whilst discussing your personal problems.

Reality

The definition and recognition of machine intelligence is the subject of fast and furious debate amongst the experts in the subject. The most generally-accepted definition is that first proposed by Alan Turing way back in the late 1940s when computers were the size of houses and even rarer than a slide-rule is today. Rather than trying to lay down a series of criteria which must be satisfied, he took a much broader view of the problem. He reasoned that most human beings accept that most other human beings are intelligent and that therefore if a man cannot determine whether he is dealing with another man (or woman), or only with a computer, then he must accept that such a machine is intelligent. This forms the basis of the famous 'Turing Test', in which an operator has to hold a two-way conversation with another entity via a keyboard and try to get the other party to reveal whether it is actually a machine or just another human being — very awkward!

Many fictional stories circulate about this test, but our favourite is the one where a job applicant is set down in front of a keyboard and left to carry on by himself. Of course he realises the importance of this test to his career prospects and so he struggles valiantly to find the secret, apparently without success. However after some time the interviewer returns, shakes him by the hand, and congratulates him with the words 'Well done, old man, the machine couldn't tell if you were human so you are just what we need as one of Her Majesty's Tax Inspectors!'

Everyone has seen from TV advertisements that the use of computer-aided design techniques is now very common, and that industrial robots are almost the sole inhabitants of car production lines (leading to the car window sticker which claims 'Designed by a computer, built by a robot, and driven by an idiot'). In fact, most of these industrial robots are really of minimal intelligence as they simply follow a pre-defined pathway without making very much in the way of actual decisions. Even the impressive paint-spraying robot which faithfully follows the pattern it learns when a human operator manually moves its arm cannot learn to deal with a new object without further human intervention.

On the other hand, the coming generation of robots have more-sophisticated sensors and software, which allow them to determine the shape, colour, and texture of objects, and to make more rational decisions. Anyone who has seen reports of the legendary 'Micromouse' contests, where definitely non-furry electric vermin scurry independently and purposefully (?) to the centre of a maze, will not be aMAZEd by our faith in the future of the intelligent robot, although there seems little point in giving it two arms and two legs.

Another important area where Artificial Intelligence is currently being exploited is in the field of expert systems, many of which can do as well (or even better) than human experts, especially if you are thinking about weather forecasting. These systems can be experts on any number of things but, in particular, they are of increasing importance in medical diagnosis and treatment — although the medical profession doesn't have to worry too much as there will always be a place for them since 'computers can't cuddle'.

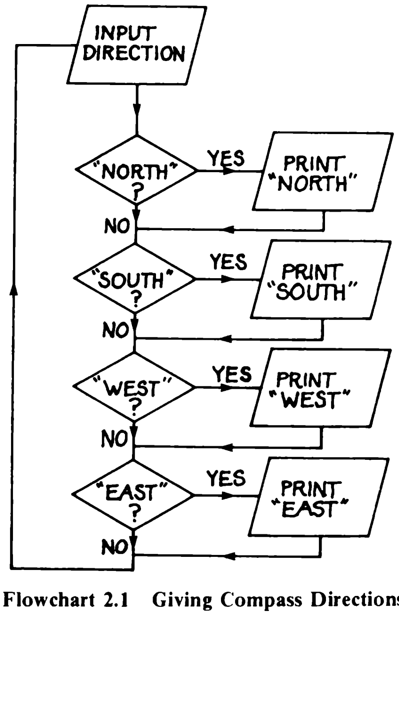
A major barrier to the wider use of computers is the ignorance and pig-headedness of the users, who will only read the instructions as a last resort, and who expect the machine to be able to understand all their little pecularities. Processing of 'natural language' is therefore a major growth area and the 'fifth generation' of computers will be much more user-friendly.

Most of the serious work on Artificial Intelligence uses more suitable (but exotic) languages than BASIC, such as LISP and PROLOG, which are pretty unintelligible to the average user and are probably not available for your home micro in any case. The BASIC routines which follow cannot therefore be expected to give you the key to world domination, although they should give you a reasonable appreciation of the possibilities and problems which Artificial Intelligence brings.

CHAPTER 2
Just Following Orders

As your computer is actually totally unintelligent, you can only converse with it in very simple terms. The first step, used in many simple adventure games, is to have a series of preset orders to which there are fixed responses. Let's start by taking a look at giving compass directions for which way to move. At first sight, the simplest way to program this appears to be to ask for an INPUT from the user and to write a separate IF–THEN line for each possibility (see **Flowchart 2.1**).

```
100 PRINT"DIRECTION?";
120 INPUT IN$
200 IF IN$="NORTH" THEN PRINT "NORTH"
210 IF IN$="SOUTH" THEN PRINT "SOUTH"
220 IF IN$="WEST" THEN PRINT "WEST"
230 IF IN$="EAST" THEN PRINT "EAST"
250 GOTO 100
```

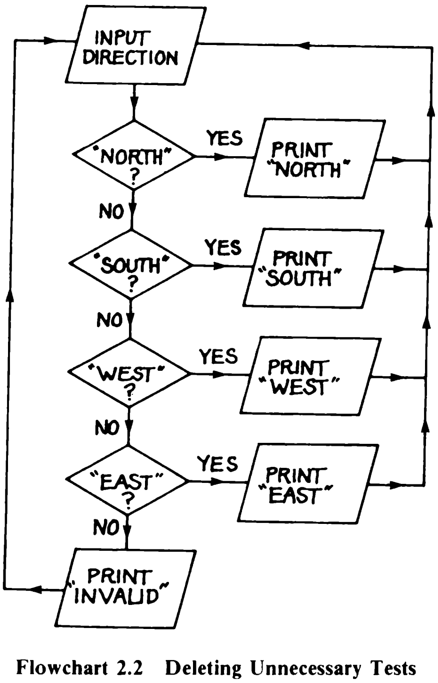


Flowchart 2.1 Giving Compass Directions

If you type in anything other than the four key command words, nothing will be printed except for another input request. It would be more user-friendly if the computer indicated more clearly that this command was not valid. You could do that by including a test which shows that none of the command words has been found, but this becomes very long-winded, and effectively impossible when you have a long list of valid words. (Note that this line is so long that you can only enter it if you use the keyword abbreviations given in the back of your Commodore 64 manual.)

```
240 IFIN$<>"NORTH"ANDIN$<>"SOUTH"ANDIN$<
>"WEST"ANDIN$<>"EAST"THENPRINT"INVALID R
EQUEST"
```

On the other hand, adding GOTO 100 to the end of each IF-THEN line will force a direct jump back to the INPUT when a valid command is detected. If all the IF tests are not true then the program falls through to line 240 which prints a warning. Making direct jumps back when a valid word is found is a good idea anyway, as it saves the system making unnecessary tests when the answer has already been found (see **Flowchart 2.2**).



Flowchart 2.2 Deleting Unnecessary Tests

```
200 IF IN$="NORTH" THEN PRINT "NORTH":GO
TO 100
```

```
210 IF IN$="SOUTH" THEN PRINT "SOUTH":GO
TO 100
220 IF IN$="WEST" THEN PRINT "WEST":GOTO
 100
230 IF IN$="EAST" THEN PRINT "EAST":GOTO
 100
240 PRINT"INVALID REQUEST"
```

That will echo the command given on the screen but of course it does not actually DO anything. As a model to work with, we will start at a position defined as X%=0 and Y%=0 and indicate movement as plus and minus in relation to this point. Notice that integer variables are used wherever possible, as they are processed faster than real numbers, and this also removes the possibility of clashing with reserved variables.

```
10 X%=0:Y%=0
```

We now need to add the real response to the command, as well as the message indicating that it has been understood (see **Flowchart 2.3**).

```
200 IF IN$="NORTH" THEN PRINT "NORTH":Y%
=Y%-1:GOTO 100
210 IF IN$="SOUTH" THEN PRINT "SOUTH":Y%
=Y%+1:GOTO 100
220 IF IN$="WEST" THEN PRINT "WEST":X%=X
%-1:GOTO 100
230 IF IN$="EAST" THEN PRINT "EAST":X%=X
%+1:GOTO 100
```

That modification actually shows your position appropriately, relative to the origin. So that you can see what is happening, and where you are, add a printout of your current position:

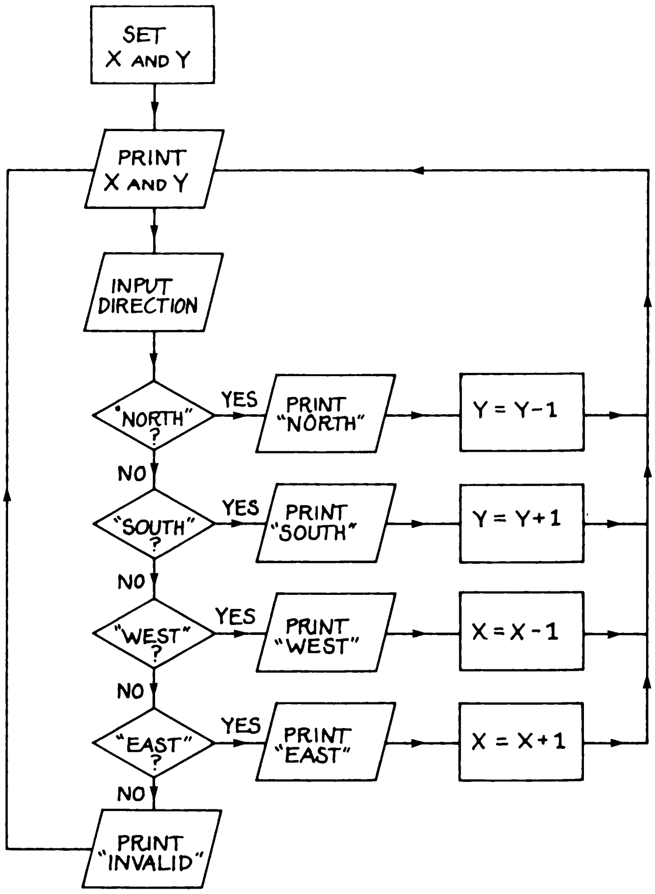
```
110 PRINT"X";X%,"Y";Y%
```

Using subroutines

Of course, that was a very simple example and, particularly where the results of your actions are more complicated, it is usually better to put the responses into subroutines.

```
200 IF IN$="NORTH" THEN GOSUB 2000:GOTO
100
```

```
210 IF IN$="SOUTH" THEN GOSUB 2100:GOTO
100
220 IF IN$="WEST" THEN GOSUB 2200:GOTO 1
00
230 IF IN$="EAST" THEN GOSUB 2300:GOTO 1
00

2000 PRINT "GOING NORTH":Y%=Y%-1:RETURN
2100 PRINT "GOING SOUTH":Y%=Y%+1:RETURN
2200 PRINT "GOING WEST":X%=X%-1:RETURN
2300 PRINT "GOING EAST":X%=X%+1:RETURN
```



Flowchart 2.3 Adding a Response

More versatility

You could extend this use of IF–THEN tests ad infinitum (or rather ad memoriam finitum!), but it is really a rather crude way of doing things which creates problems when you want to make your programs more sophisticated. A more versatile way to deal with command words and responses is to enter them as DATA and then store them in string arrays. First you must DIMension arrays of suitable length for command words (C$) and responses (R$). As variable-length strings are allowed (up to 255 characters) the actual text can be of almost any length.

```
30 DIM C$(3),R$(3)
```

If you put the commands and responses in pairs in the DATA statement, then it is more difficult to get them jumbled up and easier to read them in turn into the equivalent element in each array (see **Table 2.1**).

```
10000 DATA NORTH,GOING NORTH,SOUTH,GOING
 SOUTH,WEST,GOING WEST,EAST,GOING EAST
11000 FOR N=0 TO 3
11010 READ C$(N),R$(N)
11020 NEXT N
```

ELEMENT NUMBER	COMMAND WORD C$(n)	RESPONSE R$(n)
1	NORTH	GOING NORTH
2	SOUTH	GOING SOUTH
3	WEST	GOING WEST
4	EAST	GOING EAST

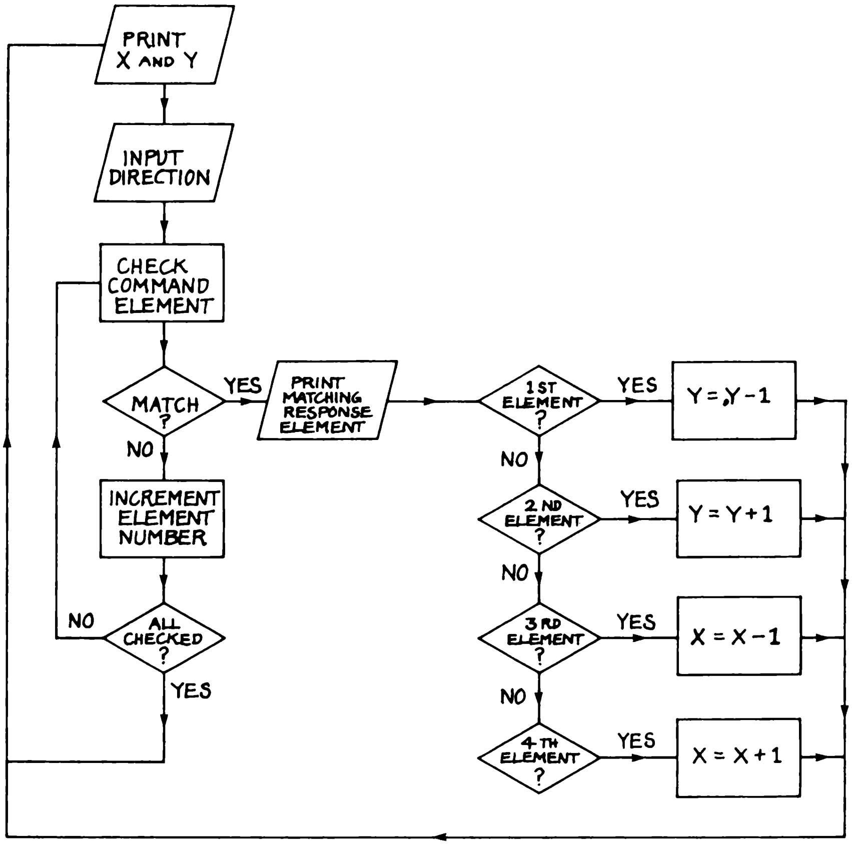
Table 2.1 Content of Command and Response Arrays

To initialise the arrays (fill them with your words), when you RUN add a GOSUB and RETURN.

```
40 GOSUB 10000
11030 RETURN
```

All those IF-THEN tests can now be replaced by a single loop which compares your INPUT with each element of the array containing the command words (C$) in turn (see **Flowchart 2.4**). Lines 200-220 need to be replaced by the following lines but notice also that line 230 must be deleted.

```
200 FOR N=0 TO 3
210 IF IN$=C$(N) THEN PRINT R$(N):GOTO 1
00
220 NEXT N
```



Flowchart 2.4 More Versatility

Now, IF your input, IN$, corresponds to any of the command words, the program jumps out of the loop after printing the appropriate response, R$(N).

Of course we are now back in our original position of actually doing nothing, so we need to be able to call those action subroutines. First of all let's arrange to jump out of the loop, if a match is found, to a new routine at line 300.

```
210 IF IN$=C$(N) THEN PRINT R$(N):GOTO 3
00
```

We still have a pointer to indicate which word matched the input, as N (the number of array elements checked) holds this value. We can use this in an ON-GOSUB line to move to appropriate routines which are similar to the ones we wrote earlier, except that there is no need to define the particular message: this has already been printed as R$(N).

```
300 ON (N+1) GOSUB 2000,2100,2200,2300:G
OTO 100
2000 Y%=Y%-1:RETURN
2100 Y%=Y%+1:RETURN
2200 X%=X%-1:RETURN
2300 X%=X%+1:RETURN
```

Expanding the vocabulary

The arrays can easily be expanded to contain more words. It would be better if we defined the number of words as a variable WD%, which we would then use to DIMension the arrays and for both the filling and scanning loops. This produces a general routine which is easily modified.

```
20 WD%=3
30 DIM C$(WD%),R$(WD%)
200 FOR N=0 TO WD%
11000 FOR N=0 TO WD%
```

For example we can add intermediate compass directions which change both X and Y axes.

```
20 WD%=7
10010 DATA NORTH EAST,GOING NORTH EAST,S
OUTH EAST,GOING SOUTH EAST
10020 DATA SOUTH WEST,GOING SOUTH WEST,N
ORTH WEST,GOING NORTH WEST
```

and add some more subroutines:

```
300 ON (N+1) GOSUB 2000,2100,2200,2300,2
400,2500,2600,2700:GOTO 100
```

```
2400 Y%=Y%-1:X%=X%+1:RETURN
2500 Y%=Y%+1:X%=X%+1:RETURN
2600 Y%=Y%+1:X%=X%-1:RETURN
2700 Y%=Y%-1:X%=X%-1:RETURN
```

Removing redundancy

All the responses so far have included the word 'GOING' and this word has actually been typed into each DATA statement. Now typing practice is very good for the soul but it would be much more sensible to define this common word as a string variable. Notice that a space is included at the end to space it from the following word.

```
10100 G$="GOING "
```

You can then delete all occurrences of this word in the DATA and combine G$ with each key word in the response instead.

```
210 IF IN$=C$(N) THEN PRINT G$;R$(N):GOT
O 300
10000 DATA NORTH,NORTH,SOUTH,SOUTH,WEST,
WEST,EAST,EAST
10010 DATA NORTH EAST,NORTH EAST,SOUTH E
AST,SOUTH EAST
10020 DATA SOUTH WEST,SOUTH WEST,NORTH W
EST,NORTH WEST
```

Now that is starting to look rather silly as both arrays now contain exactly the same words, so why not get rid of the response array, R$, and simply print C$(N)? Well, in this case you could do that without any problem, but of course where the responses are not simply a repetition of the input (as is very often the case) the second array is essential.

If you look hard at all those subroutines you will realise that they all do only one thing — update the values of X% and Y%. Now we could include that information in the original DATA and get rid of them altogether! We need to add two more arrays to hold the X and Y coordinates, add the appropriate values into the DATA lines after each response, and READ in this information in blocks of four (INPUT, RESPONSE, X-MOVE, Y-MOVE—see **Table 2.2**).

```
30 DIM C$(WD%),R$(WD%),X(WD%),Y(WD%)
10000 DATA NORTH,NORTH,0,-1,SOUTH,SOUTH,
0,1,WEST,WEST,-1,0,EAST,EAST,1,0
10010 DATA NORTH EAST,NORTH EAST,1,-1,SO
```

```
UTH EAST,SOUTH EAST,1,1
10020 DATA SOUTH WEST,SOUTH WEST,-1,1,NO
RTH WEST,NORTH WEST,-1,-1
11010 READ C$(N),R$(N),X(N),Y(N)
```

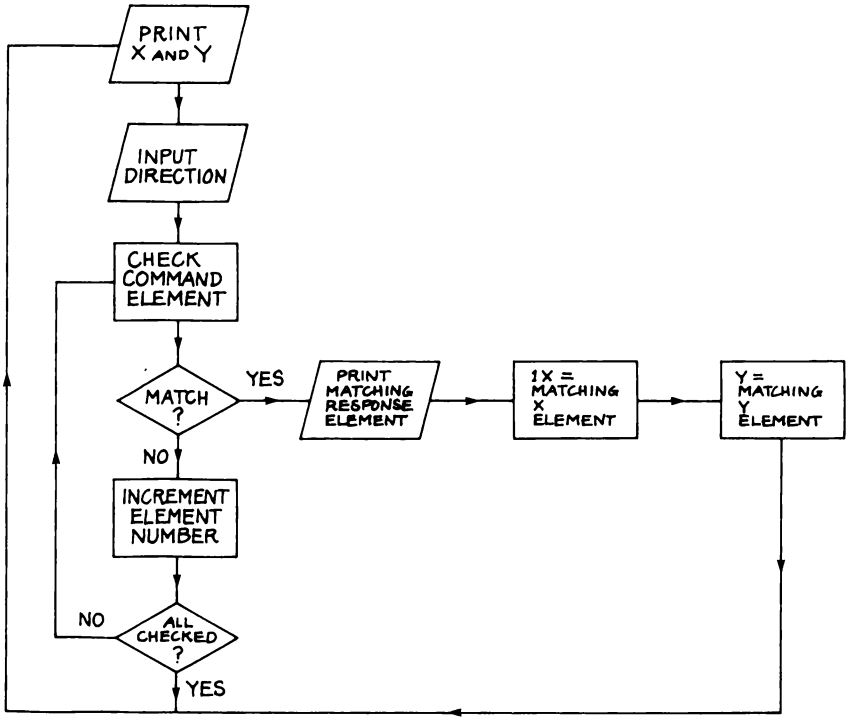
ELEMENT NUMBER	COMMAND WORD C$(n)	RESPONSE R$(n)	X-MOVE X(n)	Y-MOVE Y(n)
1	NORTH	NORTH	0	−1
2	SOUTH	SOUTH	0	1
3	WEST	WEST	−1	0
4	EAST	EAST	1	0
5	NORTH-EAST	NORTH-EAST	1	−1
6	SOUTH-EAST	SOUTH-EAST	1	1
7	SOUTH-WEST	SOUTH-WEST	−1	1
8	NORTH-WEST	NORTH-WEST	−1	−1

Table 2.2 X and Y Moves Incorporated into Arrays

Now we can delete lines 300 to 2700 and modify line 210 so that X% and Y% are updated here (see **Flowchart 2.5**).

```
210 IF IN$=C$(N) THEN PRINT G$;R$(N):X%=
X%+X(N):Y%=Y%+Y(N):GOTO 100
```

This overall pattern of putting all the information into a series of linked arrays is a very common feature which is used in several of the later programs in this book.



Flowchart 2.5 Using Linked Arrays

Abbreviated commands

So far we have always used complete words as commands, but that means that you have to do a lot of typing to give the machine your instructions. If you are feeling lazy you might think of changing the command words to the first letter of the words only, and then INPUT a single letter. However, unless you start using random letters that will only work as long as no two words start with the same letter! To code all the eight compass directions used above, we will have to use up to two letters: N, NE, E, SE, S, SW, W, NW.

```
10000 DATA N,NORTH,0,-1,S,SOUTH,0,1,W,WE
ST,-1,0,E,EAST,1,0
10010 DATA NE,NORTH EAST,1,-1,SE,SOUTH E
AST,1,1
10020 DATA SW,SOUTH WEST,-1,1,NW,NORTH W
EST,-1,-1
```

Notice that it is only the actual command words which have changed and that the computer gives a full description of the direction, as we are still using that second array which holds the response.

Partial matching

In all the programs above we have always checked that the input matched a word in the command array *exactly*. However, it would be useful if we could allow a number of similar words to be acceptable as meaning the same thing. For example, you could check whether the first letter of the input word matched the abbreviated keyword by only comparing the first character (taking LEFT$(IN$,1)).

```
190 IN$=LEFT$(IN$,1)
```

That will work with NORTH, SOUTH, EAST and WEST, but there are obvious problems in dealing with the intermediate positions. In addition there are lots of words beginning with the letters N, S, E and W — all of which would be equally acceptable to the machine as a valid direction.

For example:

NOT NORTH

would produce:

GOING NORTH

A more selective process is to match a number of letters instead of just one. In this example the first three letters of the four main directions are quite characteristic.

NOR
SOU
EAS
WES

If you use these as command words, then, for example:

NOR
NORTH
NORTHERN
and NORTHERLY

will all be equally acceptable, but:

NOT
NEARLY
NOWHERE
and NONSENSE

will all be rejected.

All we need to do is to take the first three letters of the input, LEFT$(IN$,3), and compare them with a revised DATA list. Lines 10010 and 10020 can be deleted and the word number variable WD% must then be amended to 4.

```
20 WD%=3
190 IN$=LEFT$(IN$,3)
10000 DATA NOR,NORTH,0,-1,SOU,SOUTH,0,1,
WES,WEST,-1,0,EAS,EAST,1,0
```

Sequential commands

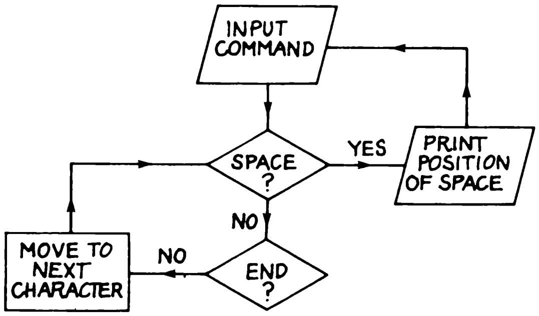
In the routines above we have dealt with the intermediate compass positions as separate entities, but if we could give a sequence of commands at the same time we would not need to do this. There is always more than one way to get to any point, and if more than one command word could be understood at the same time we would not have to worry about checking for directions such as 'NORTH EAST' as they could be dealt with by the combination of 'NORTH' and 'EAST'.

This brings us to the very significant question of how to split an input into words. First you must ask yourself how you recognise that a series of characters make up a separate word. The answer, of course, is that you see a SPACE between them. Now if we look for spaces we can break the input into separate words which we can look at individually.

The easiest way to look for spaces is with the INSTR command which searches the whole of a designated search string for a match with a second target string. Unfortunately this command is not provided in standard Commodore BASIC , so we will have to use a series of BASIC commands to emulate this. These will be placed in a subroutine at line 5000, which we will refer to for the rest of this book as simply the INSTR routine.

```
5000 FOR N=1 TO LEN(IN$)
5010 IF MID$(IN$,N,1)=" " THEN SP%=N:RET
URN
5020 NEXT N
5030 SP%=0
5040 RETURN
```

This routine will check whether the first character in IN$ is a space. If it is not a space then it will automatically continue checking until the end of IN$ is reached. If no space is found in the whole of IN$ then SP% will be set to zero. If a space is found then the value of SP% will be the number of characters along IN$ that the space is located (see **Flowchart 2.6**).



Flowchart 2.6 Locating the Position of a Space

We need to call this from the main routine and we will print out the result when we RETURN so that you can see what is happening.

```
130 GOSUB 5000
140 PRINT SP%:GOTO 100
```

Try this out with:

NOR WES

SP% 4

NORTH WEST

SP% 6

NOR NOR WEST

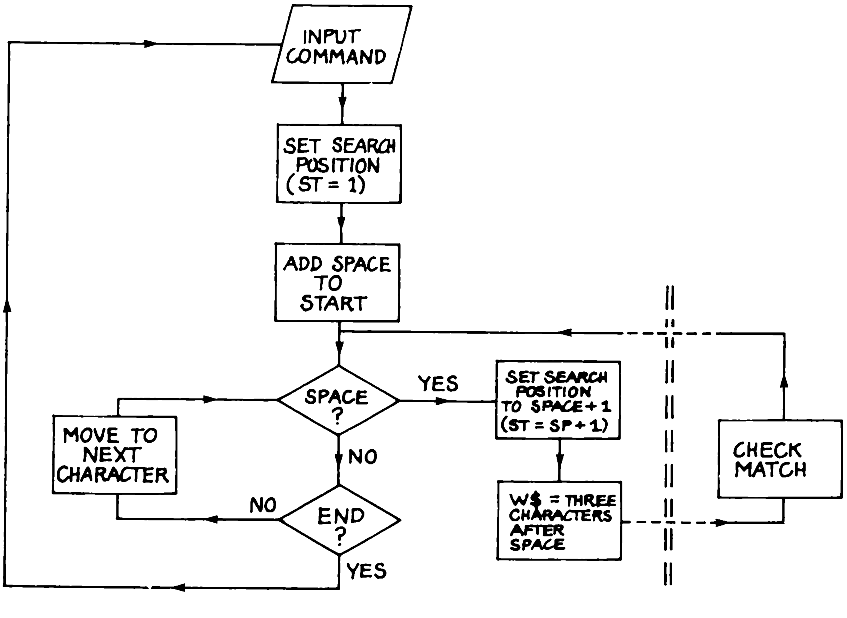
SP% 4

Notice that the length of the word is accounted for by SP% but that only the first space is found. To find all the spaces we are going to have to work harder. First delete that temporary line 140.

Let's look at the input logically from the start (lefthand side). We will replace the LEFT$(IN$,3) with MID$(IN$,ST%,3) so that we can look at any three-letter combination in the whole of IN$. To make it more sensible we will call the result of this W$ as it shows the position of a word. To start with we must set the search start position ST% equal to one and add a space to the front of IN$ so that the first word is also found (see **Flowchart 2.7**).

```
125 ST%=1:IN$=" "+IN$
130 GOSUB 5000
190 W$=MID$(IN$,ST%,3)
210 IF W$=C$(N) THEN PRINT G$;R$(N):X%=X
%+X(N):Y%=Y%+Y(N):GOTO 100
5000 FOR N=ST% TO LEN(IN$)
```

If you run this as it stands then you will still only find the first word as we have GOTO 100 on the end of line 210. However simply sending the program back to the INSTR check in line 130 instead does not help either, as it will always start checking from the beginning of IN$ and will always find the same first space. Once we have found this first space we need to


INPUT COMMAND
SET SEARCH POSITION (ST = 1)
ADD SPACE TO START
SPACE ?
YES
NO
SET SEARCH POSITION TO SPACE+1 (ST = SP+1)
W$ = THREE CHARACTERS AFTER SPACE
CHECK MATCH
MOVE TO NEXT CHARACTER
END ?
NO
YES

Flowchart 2.7 Searching for a Keyword

move the start position ST% for the next search on to the character after that space, SP%+1. When no more spaces can be found then the end of the input has been reached and we can GOTO 100 again.

```
140 IF SP%>0 THEN ST%=SP%+1:GOTO 190
150 GOTO 100
```

```
210 IF W$=C$(N) THEN PRINT G$;R$(N):X%=X
%+X(N):Y%=Y%+Y(N):GOTO 130
```

Now typing:

NORTH WEST

produces:

GOING NORTH
GOING WEST

and even:

NOR NOR EAST

is decoded as:

GOING NORTH
GOING NORTH
GOING EAST

It would be a lot neater if we deleted all those redundant 'GOINGs' and put all the reported directions on the same line. We need to PRINT G$ once, immediately before the INSTR check. Now each time we go through the loop comparing the current word with those stored, we PRINT R$(N); if there is a match. As there is a semi-colon after this, the words will be printed on the same line but we also need to add spaces between them. Finally we add a simple PRINT just before we go back for a new input, to move the cursor position on to the next line.

```
126 PRINT G$;
145 PRINT
210 IF W$=C$(N) THEN PRINT R$(N);" ";:X%
=X%+X(N):Y%=Y%+Y(N):GOTO 130
```

Now:

NORTH EASTERLY SOUTH WEST

sends you neatly round in circles:

GOING NORTH EAST SOUTH WEST

CHAPTER 3
Understanding Natural Language

So far we have only communicated with the computer in a very restricted way, as it has only been programmed to understand a very few words or letters and it only recognises these if they are entered in exactly the right way. For example, if you put a space before or after your command as you INPUT it then it will be rejected. This is because we are comparing whether the two strings match exactly.

On the other hand in the real world everyone uses what is known as 'natural' language which is a very sophisticated and extremely variable thing which only the human brain can cope with effectively. Even if we forget for the moment the difference between 'English' and 'American' or even regional dialects of either (can 'Ow bist old but' really mean 'How are you old friend'?) dealing with language has an infinite number of problems.

Even the most sophisticated systems in the world cannot cope with everything. There is an old story which illustrates this point very well. The CIA developed a superb translation program which could instantly convert English into Russian and vice versa. In the hope of impressing the President they laid on a demonstration of its capabilities, in which it converted everything he said into Russian, spoke that, and then retranslated the Russian back into English. He was most impressed and was totally absorbed until one of his aides reminded him that he had forgotten that the First Lady was waiting for him outside. When he ruefully commented 'out of sight, out of mind' he was amazed to hear the machine come back with 'invisible maniac'!

Dealing with sentences

Everyone knows that real language is made up of sentences, but what exactly do we mean by a sentence? Well, the most obvious way we recognise a sentence is that we see a full stop! However if we are going to be able to deal with sentences, we are going to have to think a lot harder than that.

The Oxford Dictionary definition includes 'a series of words in connected speech or writing, forming grammatically complete expression of single thought, and usually containing subject and predicate, and conveying statement, question, command or request' but also concedes that it is used loosely to mean 'part of writing or speech between two full stops'. Phew! Can somebody translate that into everyday English, please?

The intricacies and illogicalities of the English language are infamous so how can we expect a computer to cope?

Well, let's start by looking at some simple examples of sentences.

I WANT.

consists of a subject I and a verb WANT

I WANT BISCUITS.

also has an object BISCUITS

I WANT CHOCOLATE BISCUITS.

qualifies the object with an adjective CHOCOLATES

I SOMETIMES WANT CHOCOLATE BISCUITS.

qualifies the verb with an adverb SOMETIMES.

The most important word in all the above examples was 'WANT' as it conveyed the main idea. The second example was more informative as it indicated that only one particular type of object, BISCUITS, was wanted. The addition of an adjective, CHOCOLATE, gave further information on the type of object wanted, but life became more uncertain again when the adverb SOMETIMES was included.

Now how could a computer program decode such sentences? The answer must be to find some logical structure in the sentence, so what 'rules' could we lay down for this example?

1) All started with a subject I and ended with a full stop.
2) The last word was always the object BISCUITS (unless there was no object and only two words).
3) If the word before the object was not the verb WANT it was an adjective CHOCOLATE.
4) If the word before the verb was not the subject I it was an adverb SOMETIMES.

Let's write a program in which we give the computer sentences and ask it to break them up into their component parts.

To start off, we need to give it a vocabulary of objects, adjectives and adverbs to work with. We will READ these from DATA and store them in arrays OB, AJ and AV, according to type.

```
10 GOSUB 10000
10000 DIM OB$(5),AJ$(5),AV$(2)
10999 REM OBJECTS
11000 DATA BISCUITS,BUNS,CAKE
11010 DATA COFFEE,TEA,WATER
11019 REM ADJECTIVES
11020 DATA CHOCOLATE,GINGER,JAM
11030 DATA COLD,HOT,LUKEWARM
11039 REM ADVERBS
11040 DATA ALWAYS,OFTEN,SOMETIMES
11100 FOR N=0 TO 5
11110 READ OB$(N)
11120 NEXT N
11130 FOR N=0 TO 5
11140 READ AJ$(N)
11150 NEXT N
11160 FOR N=0 TO 2
11170 READ AV$(N)
11180 NEXT N
11190 RETURN
```

Now we need to break the sentence into words (see **Flowchart 3.1**). Once again we will do that with an INSTR search for spaces, and to make life easier we will add a space on to the end of IN$ so that the format of the last word looks just like that of other words.

```
100 INPUT IN$
120 IN$=IN$+" "
130 GOSUB 5000
190 GOTO 130
```

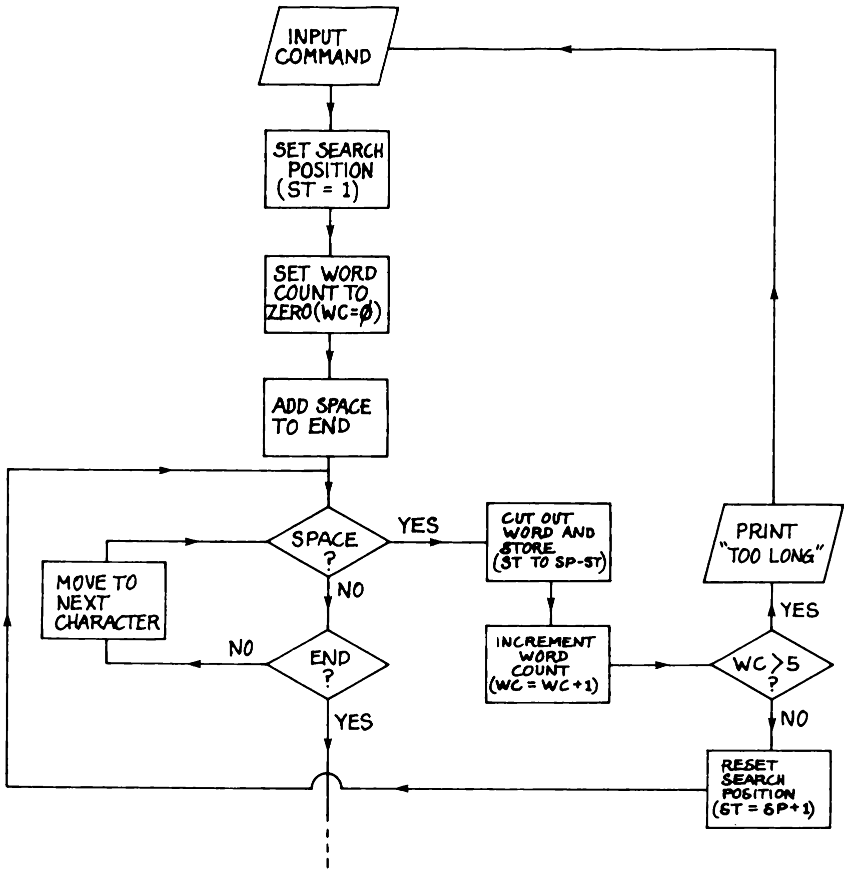
The end of the sentence has been reached when no more spaces can be found.

```
140 IF SP%=0 THEN 200
```

If a space is found then the section of IN$ from ST% (current search start) to SP%—ST% (current space—current start=length of word) is cut out and stored in a word store array W$(WC%).

```
150 W$(WC%)=MID$(IN$,ST%,SP%-ST%)
10010 DIM W$(4)
```

To begin with ST%=1 so that the search starts at the first character in the


INPUT COMMAND
SET SEARCH POSITION (ST = 1)
SET WORD COUNT TO ZERO(WC=Ø)
ADD SPACE TO END
SPACE ?
YES
NO
CUT OUT WORD AND STORE (ST TO SP-ST)
INCREMENT WORD COUNT (WC = WC+1)
WC>5 ?
YES
PRINT "TOO LONG"
NO
RESET SEARCH POSITION (ST = SP+1)
MOVE TO NEXT CHARACTER
END ?
NO
YES

Flowchart 3.1 Cutting Out Words

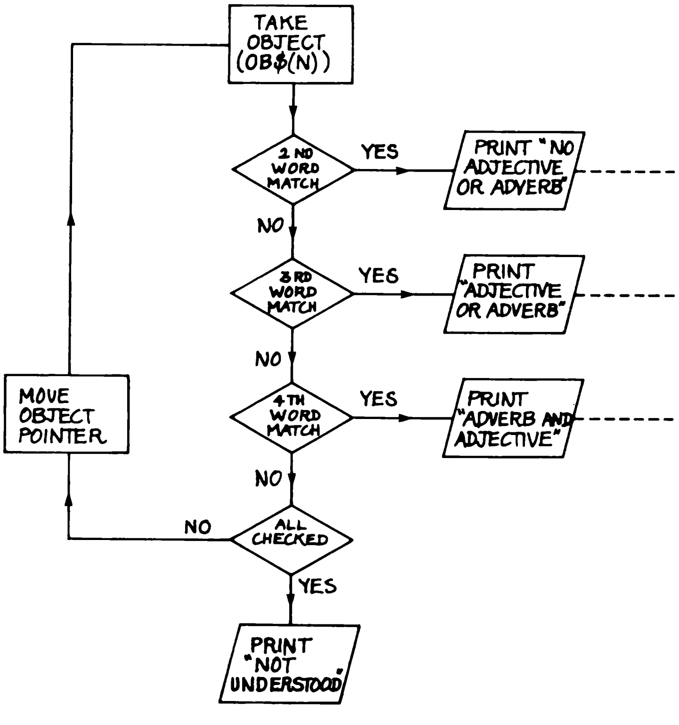
input string. The word count variable WC% is set to zero so that the first word found is stored in the zero element of the word store array.

```
110 ST%=1:WC%=0
```

The word count is incremented (so that the next element of the array W$ is used next time) and a check made that there are not more than five words in the sentence. The start position for the next search is then set to one more than the position of the last space and the search is continued.

```
160 WC%=WC%+1
170 IF WC%>5 THEN PRINT "SENTENCE TOO LO
NG":GOTO 100
180 ST%=SP%+1
```

A test is now made to see whether there is a match between the key words in the sentence and the objects in the vocabulary array OB$(N) (see **Flowchart 3.2**). Only words 2, 3 and 4 are checked as these are the only possible



Flowchart 3.2 Looking for a Match

positions for the object in our restricted sentence format. Three different routines are jumped to according to the position of the matching word in the sentence. If no match is found a message is printed and a new input requested.

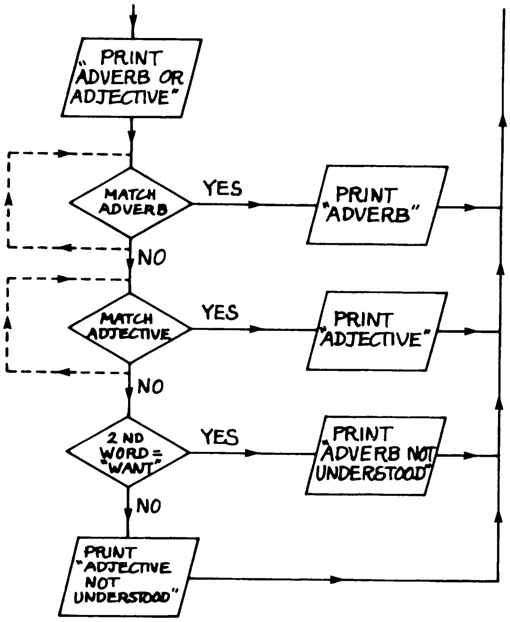
```
200 FOR N=0 TO 5
210 IF W$(2)=OB$(N) THEN 500
220 IF W$(3)=OB$(N) THEN 600
230 IF W$(4)=OB$(N) THEN 700
240 NEXT N
250 PRINT "OBJECT NOT FOUND"
260 GOTO 100
```

If the object was found as word three then there was neither adjective or adverb.

```
500 PRINT "NO ADJECTIVE OR ADVERB"
510 GOTO 100
```

If the object was found as word four then there could have been either an adjective or an adverb in the sentence (see **Flowchart 3.3**).

```
600 PRINT "EITHER ADJECTIVE OR ADVERB"
```



Flowchart 3.3 Adverb *or* Adjective

First we check for a match between the second word and the contents of the adverb array.

```
610 FOR N=0 TO 2
620 IF W$(1)=AV$(N) THEN 900
630 NEXT N
```

If no match is found then we check the third word against the adjective list

```
640 FOR N=0 TO 5
650 IF W$(2)=AJ$(N) THEN 1000
660 NEXT N
```

If a match is not found in either of these lists, then it would be useful to indicate which word was not understood. The simplest answer is to check whether the second word was not the verb 'WANT', as in that case the second word must have been an adverb. On the other hand, if the second word was the verb then the third word must have been an adjective. Notice that the actual word which did not match is now included in the message.

```
670 IF W$(1)<>"WANT" THEN PRINT "ADVERB
";W$(1);" NOT UNDERSTOOD":GOTO 100
680 PRINT "ADJECTIVE ";W$(2);" NOT UNDER
STOOD":GOTO 100
```

If a match is found in either test then a success message is printed. Note that these possibilities are exclusive and that in four words we can only have one or the other.

```
900 PRINT "ADVERB"
910 GOTO 100
1000 PRINT "ADJECTIVE"
1010 GOTO 100
```

Where both adverb and adjective are present we must check for both, and therefore a match in the first test also jumps on to the second test (see **Flowchart 3.4**).

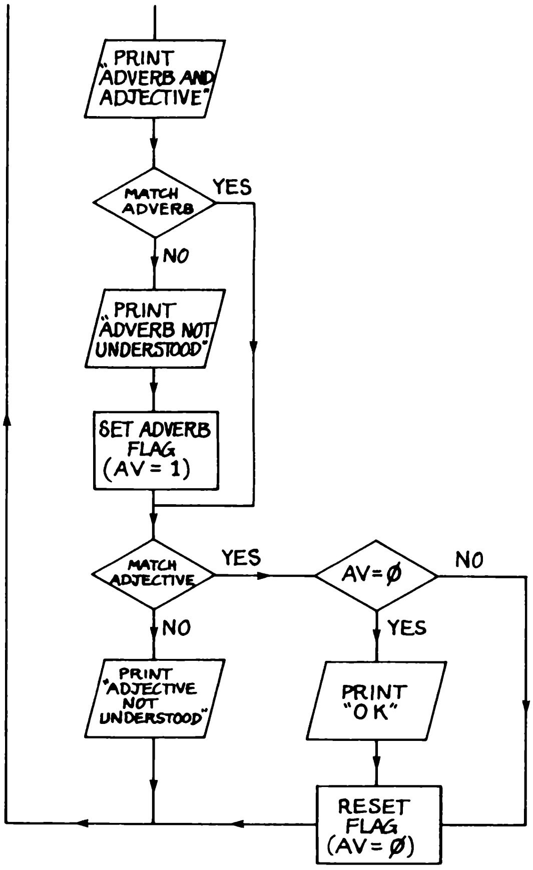
```
700 PRINT "ADVERB AND ADJECTIVE"
710 FOR N=0 TO 2
720 IF W$(1)=AV$(N) THEN 750
730 NEXT N
```

If no match is found for the adverb, then this fact is reported: a flag AV% is set to 1 to indicate failure at this point before the adjective is checked.

```
740 PRINT "ADVERB ";W$(1);" NOT UNDERSTO
OD":AV%=1
750 FOR N=0 TO 5
760 IF W$(3)=AJ$(N) THEN 800
770 NEXT N
```

If a successful match for the adjective is not found then the program loops back after a report.

```
780 PRINT "ADJECTIVE ";W$(3);" NOT UNDER
STOOD"
790 GOTO 100
```



Flowchart 3.4 Adverb *and* Adjective

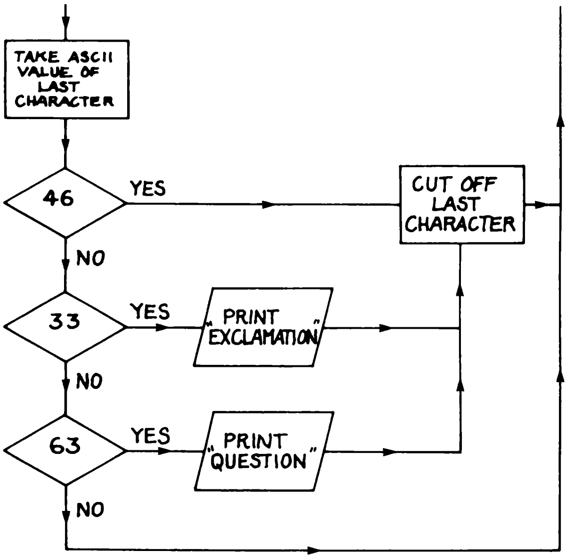
If the adjective was found then a test is made that the adverb flag AV% was not set before a match is reported. In any case, the flag is reset before the next input.

```
800 IF AV%=0 THEN PRINT "ADJECTIVE AND A
DVERB OK"
810 AV%=0
820 GOTO 100
```

What about punctuation?

As we have already said, you usually recognise the end of a sentence because it has a full stop, although when you type into a computer you usually forget all about such trivialities. But what will happen in the program so far if some 'clever' user puts in the correct punctuation? If you think for a moment, you will realise that the computer will start complaining as it will no longer recognise the last word, as this will actually be read as the word *plus* the full stop.

We therefore need to check if the last character in the input string IN$ is a full stop: this is simple as the ASCII code for this character is 46. The best place to check seems to be immediately after the INPUT. If the code of the last character is 46, then simply throw this character away and then continue as before.



Flowchart 3.5 Dealing with Punctuation

We will add this as a subroutine which is jumped to as soon as an input is made. Other punctuation marks may also appear at the end of the sentence, so we will read the last character as a variable LC% which we will also use later. This is stored as a simple variable by taking the ASCII code of the last character in IN$: using simple variables saves a lot of typing of string ($) indicators (see **Flowchart 3.5**).

```
105 GOSUB 2000

2000 LC%=ASC(RIGHT$(IN$,1))
2010 IF LC%=46 THEN 2100
2090 RETURN
2100 IN$=LEFT$(IN$,LEN(IN$)-1):RETURN
```

More useful sentence terminators are the question and exclamation marks which often indicate the context of the words. We can distinguish these in the same way by their ASCII codes and, for the moment, we will just report their presence.

```
2020 IF LC%=33 THEN PRINT"EXCLAMATION!":
GOTO 2100
2030 IF LC%=63 THEN PRINT"QUESTION":GOTO
 2100
```

The normal INPUT command will not accept anything after a comma, which it reads as data terminator. However we can produce a routine using GET which will accept any text including commas. First of all IN$ is set empty and a '<' printed as a cursor.

```
100 IN$="":PRINT "<"
```

Now a check is made for a key-press and if no key is pressed then the check is repeated.

```
101 GET I$:IF I$="" THEN 101
```

When a key is pressed a cursor left code ← is printed, followed by the character corresponding to the key pressed, I$. This character is then added on to IN$ and a jump made back to the keycheck. In this way the entry appears on the screen as in a normal INPUT, and any errors can be corrected with the backspace key.

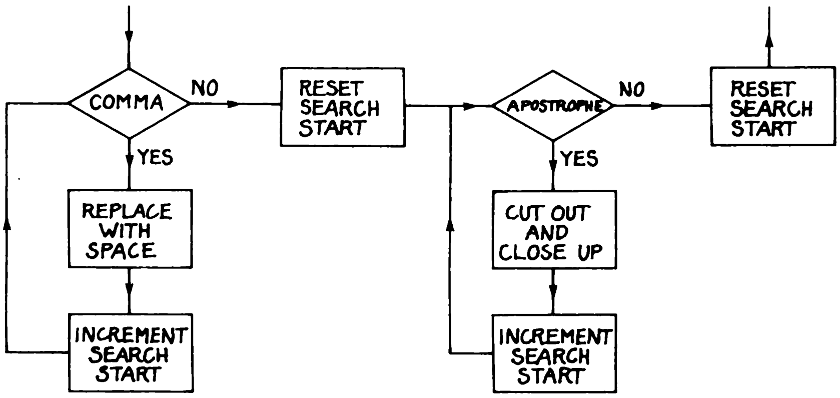
```
103 PRINT "[<--]";I$;"<";:IN$=IN$+I$:GOT
O 101
```

The end of the input is indicated by checking for the RETURN key, which has an ASCII code of 13. If the entry is complete, then the cursor is moved to the next line.

```
102 IF ASC(I$)=13 THEN PRINT:GOTO 105
```

Commas may be useful in indicating different parts of a sentence, which could be examined as 'sub-sentences' in their own right. However, in simple cases they are best deleted and replaced by spaces before the sentence is broken into words (see **Flowchart 3.6**). Note that this will only function totally correctly if there is no space after the comma, as any space following a replaced comma will be seen as a new word.

Rather than write a completely new INSTR routine, we will modify our



Flowchart 3.6 Replacing Commas and Apostrophes

existing one so that we can check IN$ for any predefined string TA$. To make things clearer in the long run, we will make the variable pointing to the position of the match in the string IS%, which can then be swapped with any number of different variables, such as SP%. First we must modify our space check to the new format.

```
130 TA$=" ":GOSUB 5000:SP%=IS%
5010 IF MID$(IN$,N,1)=TA$ THEN IS%=N:RET
URN
5030 IS%=0
```

Now the same method can be used to look for a comma, before replacing it with a space.

```
115 TA$=",":GOSUB 3000
3000 GOSUB 5000:CM%=IS%
3010 IF CM%=0 THEN ST%=1:RETURN
3020 IN$=LEFT$(IN$,CM%-1)+" "+RIGHT$(IN$
,LEN(IN$)-CM%)
3030 ST%=CM%+1
3040 GOTO 3000
```

If you add this line, you can see the punctuation being taken out of the string item.

```
3025 PRINT IN$
```

Apostrophes can be dealt with in the same way, except that we do not replace them with a space but simply close up the words.

```
115 TA$=",":GOSUB 3000:TA$="'":GOSUB 310
0

3100 GOSUB 5000:AP%=IS%
3110 IF AP%=0 THEN ST%=1:RETURN
3120 IN$=LEFT$(IN$,AP%-1)+RIGHT$(IN$,LEN
(IN$)-AP%)
3125 PRINT IN$
3130 ST%=AP%+1
3140 GOTO 3100
```

A sliding search approach

Although the method of examining a sentence described above will work, it has the disadvantage that it requires the sentence to be entered in a particular, restricted format. For example, if you enter:

I WANT HOT CAKES OFTEN

the computer will report:

OBJECT NOT FOUND

as it mistakenly takes the last word OFTEN as the object.

On the other hand using a sliding search of the whole sentence for each key word, without first breaking the sentence down into words, has the advantage that it allows a completely free input format. In this approach we take the first key word and try to match it against the same number of letters in IN$, starting at the first character. If this test fails then it is automatically repeated, starting from the second character, etc, until a match is found or the end of IN$ is reached. For example, if IN$ was 'I WANT CAKE' and the first key word was 'CAKE' the comparisons would be:

Pass 1	I WA
Pass 2	WAN
Pass 3	WANT
Pass 4	ANT
Pass 5	NT C
Pass 6	T CA
Pass 7	CAK
Pass 8	CAKE (match found)

So far our INSTR routine has only tried to match a single character, but we will have to modify line 5010 again, so that it takes into account the length of the target string LEN(TA$).

```
5010 IF MID$(IN$,N,LEN(TA$))=TA$ THEN IS
%=N:RETURN
```

Delete lines 105–1010 and add this line to check for the first object OB$(0).

```
210 TA$=OB$(M):GOSUB 5000:SP%=IS%:IF SP%
>0 THEN PRINT OB$(M);" ";
```

Each object can be compared in the same way by forming a loop. (Note that printing a semi-colon after OB$(M) ensures that each word is printed on the same line.)

```
200 FOR M=0 TO 5
220 NEXT M
```

Similar checks can be made for matching with words in the adverb and adjective arrays.

```
300 FOR M=0 TO 2
310 TA$=AV$(M):GOSUB 5000:SP%=IS%:IF SP%
>0 THEN PRINT AV$(M);" ";
320 NEXT M
400 FOR M=0 TO 5
410 TA$=AJ$(M):GOSUB 5000:SP%=IS%:IF SP%
>0 THEN PRINT AJ$(M);" ";
420 NEXT M
1000 GOTO 100
```

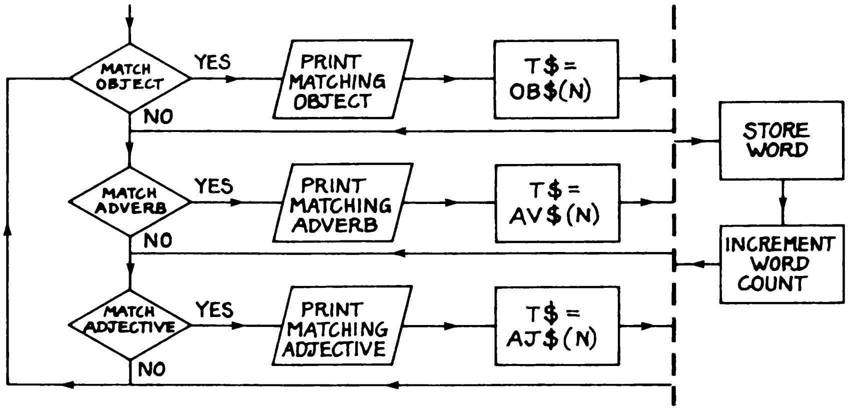
To report what has been found, and so that we can use the words discovered later, we will store each in an array as it is detected. We already have a word store array W$ but we will expand it to hold up to 20 words (which should be enough for even a very verbose sentence!).

```
10010 DIM W$(19)
```

If a match is found a temporary string T$ is set equal to the matched word, and a subroutine called at line 1500, which puts the word detected in the first array element (see **Flowchart 3.7**).

```
210 TA$=OB$(M):GOSUB 5000:SP%=IS%:IF SP%
>0 THEN T$=OB$(M):PRINT T$;" ";:GOSUB 15
```

```
00
1500 W$(WC%)=T$
```



Flowchart 3.7 Sliding Search

The word count WC% is then incremented, so that the next word is put in the next element, before returning.

```
1520 WC%=WC%+1
1530 RETURN
```

Using a temporary string T$ in the actual subroutine means that we can also use it in the tests for adverbs and adjectives in exactly the same way.

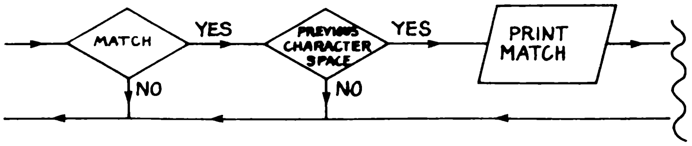
```
310 TA$=AV$(M):GOSUB 5000:SP%=IS%:IF SP%
>0 THEN T$=AV$(M):PRINT T$;" ";:GOSUB 15
00
410 TA$=AJ$(M):GOSUB 5000:SP%=IS%:IF SP%
>0 THEN T$=AJ$(M):PRINT T$;" ";:GOSUB 15
00
```

Partial matching

One advantage of the sliding search is that you can easily arrange to recognise a series of connected words by only looking for some key characters. This is obviously useful as it saves you having to put in both single and plural nouns such as BISCUIT and BISCUITS. If you amend the DATA in line 11000 as shown below than both will be recognised.

```
11000 DATA BISCUIT,BUN,CAKE
```

However life is not that simple as using BUN rather than BUNS can produce some unexpected results. On the plus side it will detect BUN, BUNS, and BUNFIGHT but unfortunately BUNCH, BUNDLE, BUNGALOW, BUNGLE, BUNK, BUNION, and BUNNY as well!



Flowchart 3.8 Checking That This is the Start of a Word

This problem is not restricted to prefixes as the computer will also not distinguish between HOT and SHOT. You could include a check that the character before the start of each match was a space (ie that this was the start of a word, see **Flowchart 3.8**). SP% gives the current start-of-word position so MID$(IN$,SP%−1,1) is the character before this.

```
210 TA$=OB$(M):GOSUB 5000:SP%=IS%:IF SP%
=0 THEN NEXT M:GOTO 230
211 IF MID$(IN$,SP%-1,1)<>" " THEN NEXT
M:GOTO 230
212 T$=OB$(M):PRINT T$;" ";:GOSUB 1500
310 TA$=AV$(M):GOSUB 5000:SP%=IS%:IF SP%
=0 THEN NEXT M:GOTO 330
311 IF MID$(IN$,SP%-1,1)<>" " THEN NEXT
M:GOTO 330
312 T$=AV$(M):PRINT T$;" ";:GOSUB 1500
410 TA$=AJ$(M):GOSUB 5000:SP%=IS%:IF SP%
=0 THEN NEXT M:GOTO 430
411 IF MID$(IN$,SP%-1,1)<>" " THEN NEXT
M:GOTO 430
412 T$=AJ$(M):PRINT T$;" ";:GOSUB 1500
```

For this to function correctly on the first word, just add a space to the start of IN$.

```
110 IN$=" "+IN$
```

In a similar way, you could use checks on the next letter after the match, or the length of the word, to restrict recognised words.

Putting things in order

Although we have now detected all the words in the sentence, regardless of their position or what else is present, they are found and stored in the order in which they appear in the DATA. This is because the comparison starts with the first item in the object array rather than the first word in the sentence. It would be useful if we could rearrange the word store array so that the words in it were in the order in which they appeared in the sentence.

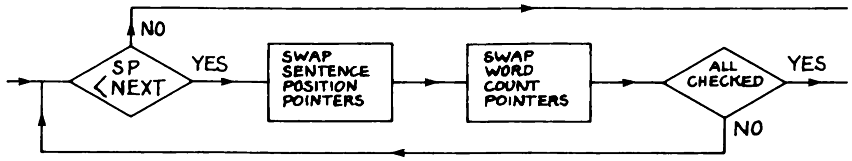
To do this, we must keep a record of the sentence position of the word SP% and word count WC%, as each word is matched in a new word position array WP%. This is a two-dimensional array with the sentence position kept in the first element, WP(WC%,0), and the word count, WP(WC%,1), in the second.

```
10020 DIM WP(19,1)
1510 WP(WC%,0)=SP%:WP(WC%,1)=WC%
```

The actual sorting subroutine which does the rearrangement is at line 4000. This must only be reached if a match is found.

```
440 IF WC%=0 THEN 470
450 GOSUB 4000
460 GOTO 100
470 PRINT"NO MATCH FOUND"
480 GOTO 100
```

The sort routine (see **Flowchart 3.9**) takes the sentence position of the first word found (first element in the first dimension WP(0,0)) and compares it with the sentence position of the second word found (second element in the first dimension WP(0+1,0)). If the position variable for the first word is of



Flowchart 3.9 Putting Words in Order

higher value than that for the second word then the first word found is farther along the sentence than the second word, and these therefore need to be swapped around. This will put the sentence-position pointers right but the word-count markers also need to be rearranged to the correct positions. This process is repeated until the word pointers are all in the correct order. Notice that the actual contents of the string array which holds the words are not altered but only the pointers (index) to them.

```
4000 FOR N=0 TO WC%-2
4010 IF WP(N,0)<WP(N+1,0) THEN NEXT N:GO
TO 4040
4020 D%=WP(N,0):WP(N,0)=WP(N+1,0):WP(N+1
,0)=D%
4030 D%=WP(N,1):WP(N,1)=WP(N+1,1):WP(N+1
,1)=D%:GOTO 4000
```

If the strings are now printed in revised word-count, WC%, order, they will be as they were in the original sentence, which should make it easier to understand them.

```
4040 PRINT:FOR N=0 TO WC%-1
4050 PRINT W$(WP(N,1));" ";
4060 NEXT N:PRINT
```

All elements in the sentence position array WP(N,0) and the word count WC% must be reset to 0 before the next input.

```
4070 FOR N=0 TO 19
4080 WP(N,0)=0
4090 NEXT N
4100 WC%=0
4110 RETURN
```

CHAPTER 4
Making Reply

More sensible replies

We have considered at length how to decode sentences which are typed into the computer, but the replies it has produced so far have been very limited and rigid. Although many of the original words in a sentence are often used in a reply, in a real conversation we look at the subject of the sentence and modify this word according to the context of the reply.

For example the input:

I NEED REST

might expect the confirmatory reply:

YOU NEED REST

and similarly:

YOU NEED REST

should generate:

I NEED REST

If you look at that situation logically, you will realise that for each input subject there is an equivalent output subject, and that we have simply chopped off the original subject and added the remainder of the sentence to the appropriate new subject.

'I' is only a single character so we could check LEFT$(IN$,1). If this was 'I' then PRINT "YOU" could be added to the front of the remainder of the input, RIGHT$(IN$,LEN(IN$)-1.

```
10 INPUT IN$
30 IF LEFT$(IN$,1)="I" THEN PRINT "YOU"+
RIGHT$(IN$,LEN(IN$)-1)
60 GOTO 10
```

In the same way, the first three characters LEFT$ (IN$,3) could be checked against 'YOU' and replaced when necessary by 'I'.

```
50 IF LEFT$(IN$,3)="YOU" THEN PRINT "I"+
RIGHT$(IN$,LEN(IN$)-3)
```

If you try that out with a series of sentences, you will see that it works OK until you type something like:

YOU ARE TIRED

which comes back as the rather unintelligent:

I ARE TIRED

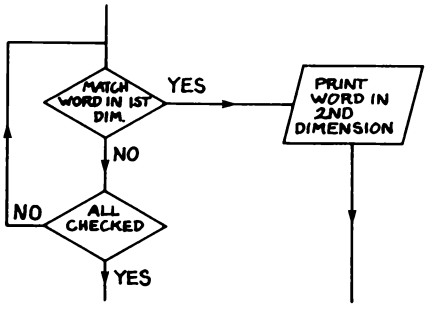
We could get around this by checking for the phrases 'I AM' and 'YOU ARE' as well as 'I' and 'YOU' on their own, but notice that you must test for these first and add GOTO 10 to the end of lines 20 and 40 to prevent a match also being found with 'I' and 'YOU' alone.

```
20 IF LEFT$(IN$,4)="I AM" THEN PRINT "YO
U ARE"+RIGHT$(IN$,LEN(IN$)-4):GOTO 10
40 IF LEFT$(IN$,7)="YOU ARE" THEN PRINT
"I AM"+RIGHT$(IN$,LEN(IN$)-7):GOTO 10
```

Although this method will work, the program soon gets very long-winded as a separate line is needed for each possibility as we must take into account the length of the matching word or phrase. Where many words are to be checked, it is therefore better to use a multidimensional string array which can be compared with the input by a loop.

A convenient format is to have a two-dimensional array IO$(n,m) where the first dimension of each element, IO$(n,0), is the input word or phrase and the second dimension, IO$(n,1), is the corresponding output word or phrase. It is easier to avoid errors if these are entered into DATA in matching pairs and READ in turn into the array. Start a new program with these lines which set up the array.

```
10 GOSUB 10000
10000 DIM IO$(3,1)
11000 DATA I,YOU,YOU,I,I AM,YOU
ARE,YOU ARE,I AM
12000 FOR N=0 TO 3
12010 READ IO$(N,0),IO$(N,1)
12030 NEXT N
13000 RETURN
```



Flowchart 4.1 Using a Corresponding Reply

We will use a looping sliding string search again, which for the moment will just print out the corresponding word or phrase to that matched, IO$(N,1) (see **Flowchart 4.1**). One advantage of the sliding string search here is that it will happily match embedded spaces in phrases as we have not broken IN$ into 'words' before matching.

```
100 INPUT IN$
110 ST%=1
200 FOR M=0 TO 3
210 TA$=IO$(M,0):GOSUB 5000:SP%=IS%:IF S
P%>0 THEN PRINT IO$(M,1)
220 NEXT M
250 GOTO 100
```

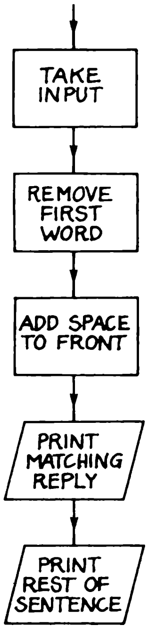
It is better to redefine the required response word as a new string which is the first part of the reply R1$, and then PRINT this when the loop is left.

```
210 TA$=IO$(M,0):GOSUB 5000:SP%=IS%:IF S
P%>0 THEN R1$=IO$(M,1)
230 PRINT R1$
```

To get a fuller reply, we need to add back on the rest of the original sentence R2$ (after inserting a space). It is not difficult to define the 'rest of the sentence'. We just need to subtract the end position of the word from the LENgth of the sentence and use this value in RIGHT$. SP% points to the start of the matched word: we have a record of the LENgth of this word in the first dimension of the array as IO$(N,0), so we just need to subtract SP%+LEN(IO$(N,0)).

```
210 TA$=IO$(M,0):GOSUB 5000:SP%=IS%:IF S
P%=0 THEN 220
```

```
215 R1$=IO$(M,1):R2$=" "+RIGHT$(IN$,LEN(
IN$)-(SP%+LEN(IO$(M,0))))
230 PRINT R1$;R2$
```



Flowchart 4.2 A Fuller Reply

Now when you try:

I AM CLEVER

the computer agrees:

YOU ARE CLEVER

But if you then press RETURN again it still tells you that you are clever — which is not true, as you have not emptied IN$, R1$ and R2$ before looping back to the next input!

```
100 IN$="":INPUT IN$
240 R1$="":R2$=""
```

Before you feel too clever try:

WE ARE STUPID

which may well surprise you when it gives the reply:

YOU

If you think for a few moments, you will see that one of our keywords is hiding inside another word in this particular sentence. If you cannot see it then try:

WE ARE INCOMPETENT

where the computer disagrees with you by returning:

YOU COMPETENT

Although each keyword is tested for in turn, each one is set to R1$ when a match is found so only the last match is reported. As the keyword is only checked for once in each sentence, embedded 'I' only causes problems when this is not the keyword in the sentence.

To get around this we must consider which keywords may cause problems. Although the letter 'I' is very common, it is very rarely the last letter in a word and so we could check that there is a space after the keyword. We must treat all keywords in the same way so add a space to the end of each. This could be done by changing all the DATA but it saves memory in the long run if we add the space as the array is set up. Note that there is no need to add spaces on to the end of the replies.

```
12020 IO$(N,0)=IO$(N,0)+" "
```

We also now need to subtract one less character from IN$ to give R1$, as the space has now become an integral part of the keyword.

```
215 R1$=IO$(M,1):R2$=" "+RIGHT$(IN$,LEN(
IN$)-(SP%+LEN(IO$(M,0)))+1)
```

The computer will now readily agree about your incompetence.

If the first keyword is not at the start of the sentence, then everything before it will be ignored in the reply.

For example the answer to:

WHAT IF I FALL?

will be:

YOU FALL?

Some strange results can still occur when two true keywords are present. For example:

WHAT IF YOU AND I FALL

gives

I AND I FALL

and

WHAT IF I AND YOU FALL

replies

I FALL

However, adding more suitable keywords is easy and some combinations will just not be acceptable. To make the routine more general, it is better to define the number of keywords as a variable KW% and use this in place of the actual number.

```
10 KW%=5:GOSUB 10000
200 FOR N=0 TO KW%
10000 DIM IO$(KW%,1)
11010 DATA WE,WE,THEY,THEY
12000 FOR N=0 TO KW%
```

Now the answer to:

WHAT IF WE FALL?

is the more logical:

WE FALL?

Pointing to replies

So far our computer has displayed only slightly more intelligence than a parrot, as it has merely regurgitated a slightly modified version of the input. The next stage, therefore, is to make it take some logical decisions on the basis of the input before it replies.

The numbers of subjects SU%, verbs VB% and replies RP% are defined as variables so that the program can be easily expanded, and three arrays using these are set up. (As we have a zero element in the array, these values are all one less than the number of words.) SU$(n,n) is a two-dimensional array which is concerned with the subjects in the input and output sentences. The first dimension (n,0) contains the recognised subject words and phrases allowed in the input, and the second dimension (n,1) contains the opposites which may be needed in the output. VB$(n) holds the legal verbs, and RP$(n) a series of corresponding replies.

```
10 GOSUB 10000
10000 SU%=26:VB%=6:RP%=6
10010 DIM SU$(SU%,1)
10020 DIM VB$(VB%)
10030 DIM RP$(RP%)
```

Table 4.1: Pairs of Subjects in SU$(n,n)

SU$(n,0)	SU$(n,1)
I HAVE	YOU HAVE
I'VE	YOU'VE
I AM	YOU ARE
I'M	YOU'RE
YOU HAVE	I HAVE
YOU'VE	I'VE
YOU ARE	I AM
YOU'RE	I'M
YOU	I
SHE HAS	SHE HAS
SHE IS	SHE IS
SHE'S	SHE'S
SHE	SHE
THEY'VE	THEY'VE
THEY ARE	THEY ARE
THEY'RE	THEY'RE
THEY	THEY
HE HAS	HE HAS
HE IS	HE IS
HE'S	HE'S
HE	HE
WE HAVE	WE HAVE
WE'VE	WE'VE
WE ARE	WE ARE
WE'RE	WE'RE
WE	WE
I	YOU

The first two lines of DATA contain paired input and output subjects (see **Table 4.1**) and these are READ into corresponding dimensioned elements in the SU$(n,n) array. As the pronouns ('I', 'YOU', etc) are frequently linked to other words to form phrases (such as 'I'VE'), these combined forms are also included in the DATA. Notice that these are arranged in such an order that the most complete phrase containing a keyword is always found first. A space is added on to the end of each element, so that some clashing of partial matches is avoided and a space is automatically formed in the reply.

```
11000 DATA I HAVE,YOU HAVE,I'VE,YOU'VE,I
 AM,YOU ARE,I'M,YOU'RE,YOU HAVE,I HAVE
11010 DATA YOU'VE,I'VE,YOU ARE,I AM,YOU'
RE,I'M,YOU,I
11020 DATA SHE HAS,SHE HAS,SHE IS,SHE IS
,SHE'S,SHE'S,SHE,SHE
11030 DATA THEY'VE,THEY'VE,THEY ARE,THEY
 ARE,THEY'RE,THEY'RE,THEY,THEY
11040 DATA HE HAS,HE HAS,HE IS,HE IS,HE'
S,HE'S,HE,HE,WE HAVE,WE HAVE
11050 DATA WE'VE,WE'VE,WE ARE,WE ARE,WE'
RE,WE'RE,WE,WE,I,YOU
12000 FOR N=0 TO SU%
12010 READ SU$(N,0),SU$(N,1)
12020 SU$(N,0)=SU$(N,0)+" ":SU$(N,1)=SU$
(N,1)+" "
12030 NEXT N
```

The next DATA line contains the main verbs which are READ into VB%$(n). The verb 'to be' is omitted as its variations are so complicated, and many of its versions are already accounted for in the 'subject' check.

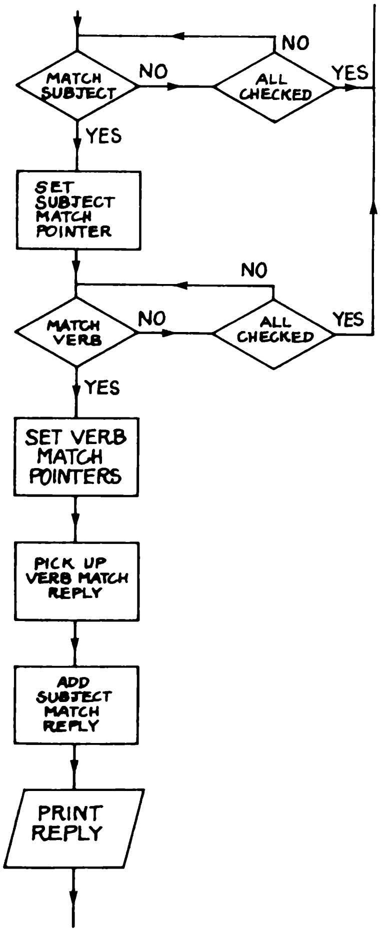
```
11060 DATA HATE,LOVE,KILL,DISLIKE,LIKE,F
EEL,KNOW
12040 FOR N=0 TO VB%
12050 READ VB$(N)
12060 NEXT N
```

The last set of DATA contains the replies which are put into RP$(n), before control RETURNs to the main part of the program. To make things simple to understand and check at this stage, all the replies contain the original verb, although of course they could say anything.

```
11070 DATA PROBABLY HATE YOU AS WELL,LOV
E YOU TOO,KILL YOU
11080 DATA DISLIKE LOTS OF THINGS,LIKE C
HIPS,FEEL POWERFUL,KNOW EVERYTHING
12070 FOR N=0 TO RP%
12080 READ RP$(N)
12090 NEXT N
13000 RETURN
```

Matching

The input string is now compared with the list of subjects in the first dimension of SU$(n,n) (see **Flowchart 4.3**). If there is no match then a new



Flowchart 4.3 Setting Match Pointers

input is requested, or else a subject match variable SM% is set to the element number at which a match was found. (Note that IS$ is always used in this program to indicate the position of an INSTR match.)

```
200 FOR M=0 TO SU%
210 ST%=1:TA$=SU$(M,0)):GOSUB 5000
220 IF IS%=0 THEN NEXT M:GOTO 100
230 SM%=M
```

The verb array is now compared with IN$. If no verb is found, then the input is rejected, or else the verb match variable VM% is set.

```
240 FOR M=0 TO VB%
250 TA$=VB$(M)):GOSUB 5000
260 IF IS%=0 THEN NEXT M:GOTO 100
270 VM%=M
```

Making reply

Now that the subject and verb have been identified, we can pick up the appropriate reply by using VM% as a pointer to the reply array RP$(n).

```
500 RL$=RP$(VM%)
```

In the simplest case we can just add the appropriate subject to the front of RL$ before we print it.

```
520 RL$=SU$(SM%,0)+RL$
550 PRINT RL$
560 GOTO 100
```

Now, for example, if you type in:

I HATE COMPUTERS

the program will reply with:

I PROBABLY HATE YOU AS WELL

and:

I KNOW A LOT

generates:

I KNOW EVERYTHING

Alternative subjects

If you prefer the machine to agree with you rather than trying to beat you at your own game, then just change the subject added to RL$ to the second element of the array (the 'opposite').

```
520 RL$=SU$(SM%,1)+RL$
```

now:

I KNOW A LOT

generates:

YOU KNOW EVERYTHING

For more variety, you can pick the subject at random from the first or second element, so that the reply is not predictable.

```
510 RS%=INT(RND(1)+0.5)
520 RL$=SU$(SM%,RS%)+RL$
```

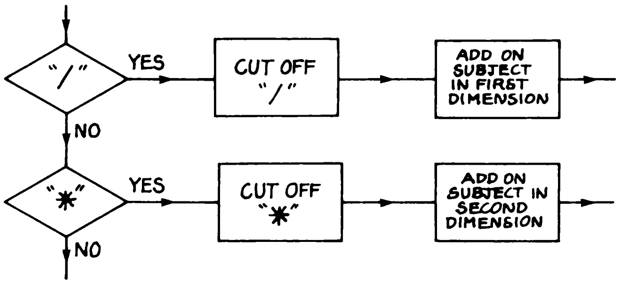
Putting the subject in context

It would be more sensible altogether if we chose the correct subject according to the context of the reply, but to do that we must have markers in the reply array. We will use a slash sign '/' to indicate that the word in the first dimension of the subject array is to be used, and an asterisk '*' to indicate that the word in the second dimension is to be used.

```
11070 DATA /PROBABLY HATE YOU AS WELL,/L
OVE YOU TOO,/KILL YOU
11080 DATA *DISLIKE LOTS OF THINGS,/LIKE
 CHIPS,*FEEL POWERFUL,*KNOW EVERYTHING
```

We can search the reply string RP$(VM%) pointed to by the verb marker VM% for a slash sign '/', provided that we rename this as IN$ before we enter the INSTR check. If a slash sign is found, then the contents of the first dimension of the subject array SU$(SM%,0) are added to the reply RL$, less the first character (the slash sign, see **Flowchart 4.4**).

```
500 RL$=RP$(VM%)
510 IN$=RL$:ST%=1:TA$="/":GOSUB 5000
520 IF IS%>0 THEN 800
800 RL$=SU$(SM%,0)+RIGHT$(RL$,LEN(RL$)-1
)
810 GOTO 530
```


"/"
YES
NO
CUT OFF "/"
ADD ON SUBJECT IN FIRST DIMENSION
"*"
YES
NO
CUT OFF "*"
ADD ON SUBJECT IN SECOND DIMENSION

Flowchart 4.4 Putting the Subject in Context

On the other hand if no slash sign is found in the reply then a second search is made for an asterisk '*'. If this is found, then the second dimension of SU$(n,n) is used in the same way.

```
530 ST$=1:TA$="*":GOSUB 5000
540 IF IS%>0 THEN 820
820 RL$=SU$(SM%,1)+RIGHT$(RL$,LEN(RL$)-1
)
830 GOTO 550
```

Now:

I LOVE ME

will give:

I LOVE YOU TOO

but:

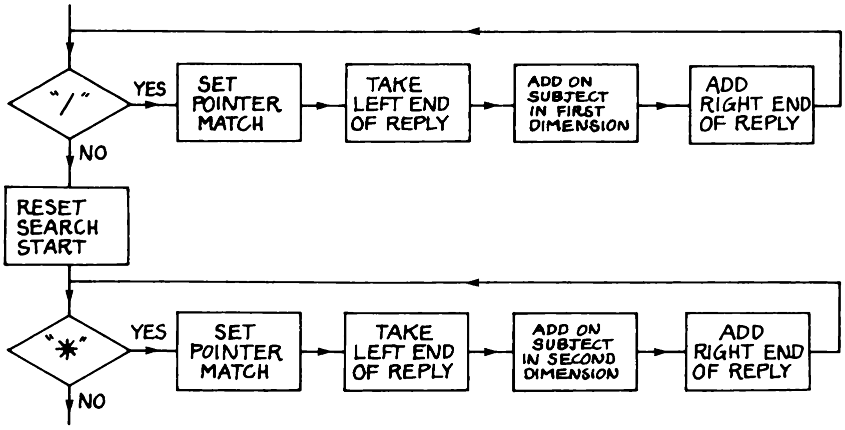
I FEEL POWERFUL

produces:

YOU FEEL POWERFUL

Inserting into sentences

To make things simple, we have always started our reply sentences with the subject, but in real life this is not always the case. Now that we have markers in the replies to indicate what type of subject is to be added, we can also use them to indicate where in the reply to insert this word or phrase. First we



Flowchart 4.5 Inserting into a Sentence

will amend the DATA so that the word to be inserted is never at the start, to make the insertion process obvious.

```
11070 DATA DO YOU REALISE THAT /PROBABLY
 HATE YOU AS WELL,WELL /LOVE YOU TOO
11080 DATA IF /DON'T KILL YOU FIRST,SO W
HAT *DISLIKE LOTS OF THINGS
11090 DATA DO /LIKE CHIPS,WHY DO *FEEL P
OWERFUL,HOW DO *KNOW EVERYTHING
```

We actually already have a record of where to insert the word as IS% tells us where in the reply the slash or asterisk was found. All we need to do is to take the part of the reply before the marker, LEFT$(RL$,IS%–1), add the correct version of SU$(SM%,n), and then the rest of the reply RIGHT$(RL$,LEN(RL$)–IS%)

```
800 RL$=LEFT$(RL$,IS%-1)+SU$(SM%,0)+RIGH
T$(RL$,LEN(RL$)-IS%)
```

```
820 RL$=LEFT$(RL$,IS%-1)+SU$(SM%,1)+RIGH
T$(RL$,LEN(RL$)-IS%)
```

Now:

I WILL KILL HIM

produces:

IF I DON'T KILL YOU FIRST

and:

I DISLIKE COMPUTERS

gives:

SO WHAT YOU DISLIKE LOTS OF THINGS

Although we are now inserting the subject into the reply sentence more naturally, we are only dealing with one subject per sentence. Another minor modification will allow us to insert any number of subjects into a sentence. All we need to do is to keep repeating the search for markers until no more are found. A start variable ST% is defined as 1 in line 500, and then a search is made for the first type of marker. When a match is found, ST% is reset to one more than the match position. When RL$ has been modified by line 800 we now need to jump back to 510 to look for more markers. If no match is found for the first marker then ST% is reset to 1. The second type of marker is then checked for in the same way.

```
500 RL$=RP$(VM%):ST%=1
510 IN%=RL$:TA$="/":GOSUB 5000
520 IF IS%>0 THEN ST%=IS%+1:GOTO 800
525 ST%=1
530 IN%=RL$:TA$="*":GOSUB 5000
540 IF IS%>0 THEN ST%=IS%+1:GOTO 820
810 GOTO 510
830 GOTO 530

11070 DATA DO YOU REALISE THAT /PROBABL
Y HATE YOU AS WELL,WELL /LOVE YOU TOO
11080 DATA IF /DON'T KILL YOU FIRST
11085 DATA SO WHAT /DISLIKE LOTS OF THI
NGS ESPECIALLY *
11090 DATA DO /LIKE CHIPS,WHY DO *FEEL
POWERFUL,*THINK *KNOW EVERYTHING
```

Now:

I KNOW EVERYTHING

produces:

YOU THINK YOU KNOW EVERYTHING

and:

I DISLIKE COMPUTERS

gives:

SO WHAT I DISLIKE LOTS OF THINGS ESPECIALLY YOU

OBJECTions on the SUBJECT

Everything is starting to look rosy until you try something like:

I HATE YOU

which replies:

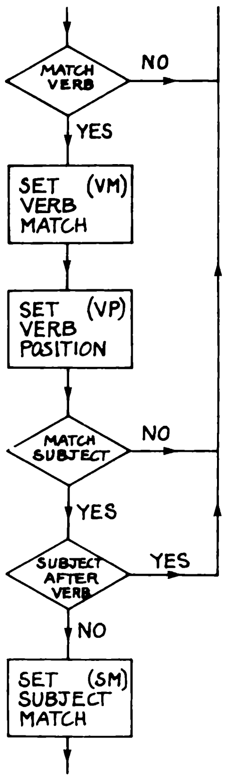
DO YOU REALISE THAT YOU PROBABLY HATE YOU AS WELL

The problem here is that we are jumping out of the search routine as soon as the first match is found, and that although we are checking for the subject 'I' we are finding the object 'YOU' first. As 'YOU' comes before 'I' in the subject array this is found first, in spite of the fact that it comes later in the sentence.

As we cannot practically mimic all the intricacies of the human brain, we will have to make the assumption that the subject always comes before the verb, and the object after it. In the program so far we have been checking for the subject before we checked for the verb, and we will have to reverse that order. The verb position in the input is the value of IS% when a verb is found, so we will save that as a verb position VP% pointer.

```
200 FOR M=0 TO VB%
210 ST%=1:TA$=VB$(M):GOSUB 5000
220 IF IS%=0 THEN NEXT M:GOTO 100
230 VM%=M:VP%=IS%
```

Now when a match with the subject array is found, we can compare that position IS% with the stored verb pointer VP%, and reject the match if the subject is positioned after the verb (see **Flowchart 4.6**).



Flowchart 4.6 Rejecting Object Matches

```
240 FOR M=0 TO SU%
250 ST%=1:TA$=SU$(M):GOSUB 5000
260 IF IS%=0 THEN NEXT M:GOTO 100
270 IF IS%>VP% THEN NEXT M:GOTO 100
280 SM%=M
```

(If you are too lazy to retype those lines you can add a couple of jumps to rearrange the order instead.)

```
140 GOTO 240
231 GOTO 500
271 GOTO 200
270 VM%=M:VP%=IS%
225 IF IS%>VP% THEN NEXT M:GOTO 100
```

A change of tense

If we change to the past tense of the verb, we may or may not find this. With the first five verbs the situation is straightforward: to change to the past tense we just add 'D' to the end of the present tense. Both forms are therefore accepted.

HATE	HATED
LOVE	LOVED
KILL	KILLED
DISLIKE	DISLIKED
LIKE	LIKED

However, with the last two verbs the word changes completely, so there can be no simple match. Although we might get away with checking for 'KN', as this is a rare combination, it would not be practical for us to use such a common group as 'FE' as a keyword.

FEEL	FELT
KNOW	KNEW

It is easier if we treat all verbs in the same way and, if there are no constraints on memory, then we can simply put all the possible versions into the verb array in pairs.

```
10000 SU%=26:VB%=13:RP%=6
11060 DATA HATE,HATED,LOVE,LOVED,KILL,KI
LLLED,DISLIKE,DISLIKED
11065 DATA LIKE,LIKED,FEEL,FELT,KNOW,KNE
W
```

Unless we want to have different replies for the different tenses, we will now have to divide the verb variable VM% by two, to point to the correct reply for both forms.

```
230 VM=M/2:VP%=IS%
```

CHAPTER 5
Expert systems

A human expert is someone who knows a great deal about a particular subject and who can give you sensible advice ('expert opinion') on it. Such expertise is only acquired after long training and a great deal of experience, so unfortunately real experts are few and far between. In addition they are often not on hand when a problem needs to be solved.

Scientists have therefore applied themselves to the problem of producing computer programs which mimic the functions of such human experts. Such programs have the advantage that they can be copied very easily to produce an infinite number of experts, and of course they do not need tea-breaks, sleep, pay-rises, etc, either! Of course, the computer must be totally logical and can still only follow pre-programmed instructions entered by the programmer. It is interesting to note that science fiction authors have envisaged problems when the ultimate 'experts' (such as HAL in '2001: A Space Odyssey' or Isaac Asimov's positronic robots) are faced with alternative courses which conflict with more than one of their prime directives and which produce not system crashes but 'pseudo-nervous breakdowns'.

Before we can start writing programs for 'expert systems', we must ask ourselves how a human expert works.

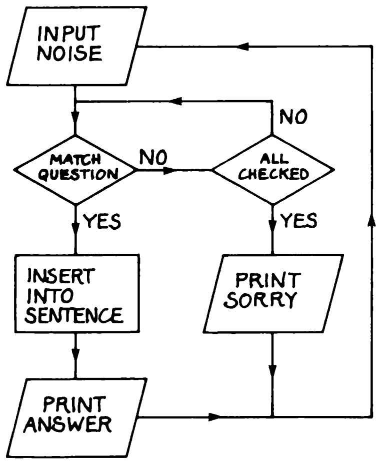
Let us first consider the simplest situation, where the expert's task is to find the answer to a known problem.

First of all he takes in information on the current task.

Secondly he compares this with information stored in his brain and looks for a match.

Finally he reports whether or not a match has been found.

What we need here is simply a database program which tries to match the input against stored information. (See **Flowchart 5.1**). A user-friendly system would accept natural language (see earlier), but to keep things simple here we will stick to a fixed input format. To start with, let's look at recognising animals by the sound they make. We set up two arrays: the question array QU$(n) contains the sounds which are known, and each



Flowchart 5.1 A Simple 'Expert'

element of the answer array AN$(n) contains the name of the relevant animal.

```
10 GOSUB 10000
10000 DIM QU$(4),AN$(4)
10010 DATA MIAOW,CAT,WUFF,DOG,MOO,COW,HO
OT,OWL,NEIGH,HORSE
10020 FOR N=0 TO 4:READ QU$(N),AN$(N):NE
XT N
10030 RETURN
```

Now we just need to ask for a sound and compare it with the contents of QU$(n).

```
20 PRINT"WHAT NOISE DOES IT MAKE";
30 INPUT IN$
40 FOR N=0 TO 4:IF IN$=QU$(N) THEN 100
50 NEXT N
60 PRINT"SORRY I DON'T KNOW THAT ONE"
70 GOTO 20
100 PRINT "AN ANIMAL THAT ";QU$(N);"S IS
 A ";AN$(N)
110 GOTO 20
```

Perhaps we should say at this point that our computer expert may well be better at this task than the human kind, as it cannot make subjective judgements, become bored, or accidentally forget to check all of the information in its memory. On the other hand it is not very literate as it reports 'A OWL', etc. (We will leave you to tidy that up by adding a routine which checks whether the first letter of the answer array match is a vowel.)

Branching out

The example above is very simple as only one question is asked, and there is only one possible answer. In reality we need to be able to deal with more difficult problems, where the answer cannot be found without asking a whole series of questions. For example, what should an expert do if he put the key in the ignition switch of his car and turned it, but nothing happened?

There could be a number of reasons for this:

```
FLAT BATTERY
BAD CONNECTIONS
SWITCH BROKEN
STARTER JAMMED
STARTER BROKEN
SOLENOID BROKEN
```

To find the cause, he should follow a logical path and make a number of checks. The first thing to do is to check whether it is only the starter motor which is not working:

```
IS IGNITION LIGHT ON? (Y/N)
```

If the answer to this is 'N' then there is no power at the switch, so the cause must be one of the first three possibilities listed above. We can narrow things down more by finding out if the lights work:

```
DO LIGHTS WORK CORRECTLY? (Y/N)
```

If the answer is yes, then the battery cannot be flat and it must be connected to the light switch correctly. So presumably the switch is broken and a suggestion can be made that you replace it.

```
REPLACE IGNITION SWITCH
```

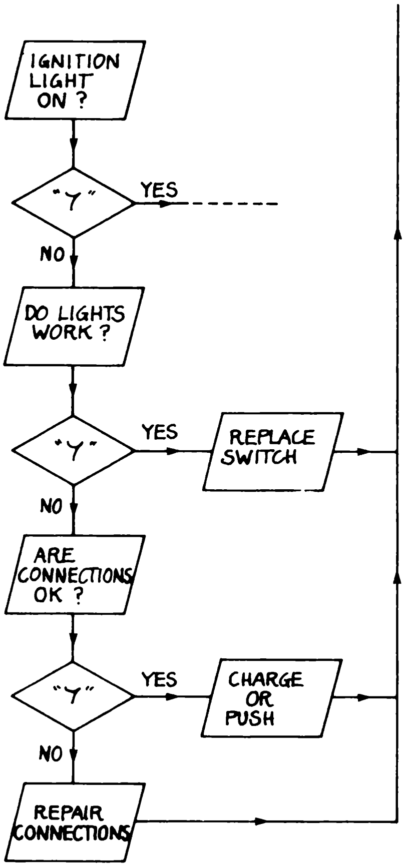
If the lights do not work, then the connections should be checked.

```
ARE BATTERY CONNECTIONS OK? (Y/N)
```

If the answer is yes, then the battery is flat so you must charge it (or push!)

CHARGE BATTERY OR PUSH CAR

In the same way, a sequence of checks could be made to deal with a situation where there is a power but the starter mechanism itself does not work (the last three possibilities).



Flowchart 5.2 A Branching 'Expert'

The simplest way to program this branching structure is by a series of IF-THEN tests (see **Flowchart 5.2**).

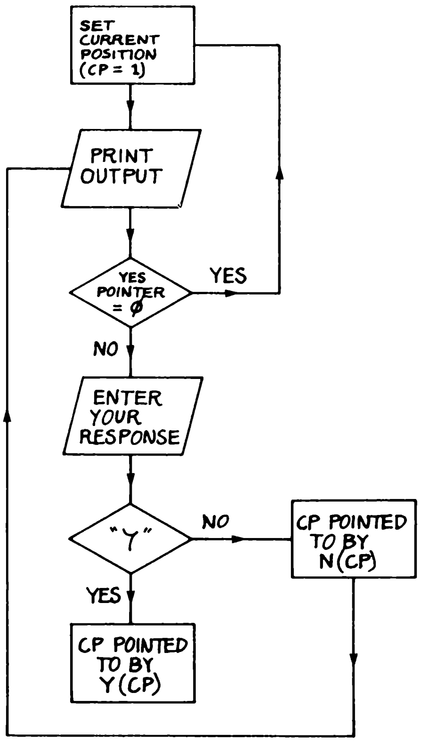
```
10 PRINT"FAULT DIAGNOSIS"
20 PRINT
30 PRINT"IS IGNITION LIGHT ON (Y/N)"
40 INPUT IN$
50 IF IN$="Y" THEN 180
60 PRINT"DO LIGHTS WORK CORRECTLY (Y/N)"
```

```
70 INPUT IN$
80 IF IN$="Y" THEN 110
90 PRINT"REPLACE IGNITION SWITCH"
100 RUN
110 PRINT"ARE BATTERY CONNECTIONS OK (Y/
N)"
120 INPUT IN$
130 IF IN$="Y" THEN 160
140 PRINT"REPAIR CONNECTIONS"
150 RUN
160 PRINT"CHARGE BATTERY OR PUSH CAR!"
170 RUN
180 ----- etc ------
```

This sort of program is relatively easy to write, but as usual is inefficient as it becomes longer and more complicated.

Pointing the way

A more efficient way to deal with the situation is to put the text into arrays



Flowchart 5.3 Pointing to the Next Output

and have pointers which direct you to the next question or reply, according to whether you answer yes or no to the current question (see **Flowchart 5.3**).

The format for entering the DATA for each branch point is, then:

(TEXT),(Pointer for 'YES'),(Pointer for'NO')

The first question was:

IS IGNITION LIGHT ON? (Y/N) ... 1

If the answer was 'N' then you need to ask the second question:

DO LIGHTS WORK CORRECTLY? (Y/N) ... 2

Otherwise you need to continue with the other part of the diagnosis (which we have not included but which would be point 7).

We need to set up three arrays: OP$(n) contains the output (text), Y(n) the pointer for 'yes', and N(n) the pointer for 'no'. To make the program easy to modify, a variable NP is used for the number of points. The DATA is read in groups of three into each element in these arrays. Where the DATA point is a possible end of the program, this is indicated by the Y(n) and N(n) pointers being set at zero.

```
10 GOSUB 10000
10000 NP=7
10010 DIM OP$(NP),Y(NP),N(NP)
11000 DATA "IS IGNITION LIGHT ON",7,2
11010 DATA "DO LIGHTS WORK CORRECTLY",3,
4
11020 DATA "REPLACE SWITCH",0,0
11030 DATA "ARE BATTERY CONNECTIONS OK",
5,6
11040 DATA "CHARGE BATTERY OR PUSH CAR",
0,0
11050 DATA "REPAIR CONNECTIONS",0,0
11060 DATA "-rest of program-",0,0
12000 FOR N=1 TO NP
12010 READ OP$(N),Y(N),N(N)
12020 NEXT N
13000 RETURN
```

The actual running routine is very simple. A pointer CP is used to indicate the current position in the array: to begin with this is set to 1 and the first text printed. If this is an end point Y(CP)=0 (hardly likely just yet!), then CP is reset to 1 so that the sequence starts again. On the other hand, if a real pointer is present then an INPUT is requested. If the input is 'Y', then CP is set to the value contained in the appropriate element of the Y(n) array, otherwise it is set to the value contained in the N(n) array.

```
20 CP=1
30 PRINT OP$(CP)
40 IF Y(CP)=0 THEN 20
50 INPUT IN$
60 IF IN$="Y" THEN CP=Y(CP):GOTO 30
70 CP=N(CP)
80 GOTO 30
```

A parallel approach

An alternative to the sequential branching method described above is the parallel approach which always asks all the possible questions before it reaches its conclusion. This method usually takes longer than following an efficient tree structure, but it is more likely to produce the correct answer as no points of comparison are omitted.

Let us consider how we might distinguish between various forms of transport.

We will consider eight features and mark 1 or 0 for the presence or absence of these in each of our five modes of transport (see **Table 5.1**). If you look closely you will notice that the pattern of results varies for each of the different possibilities so it must be possible to distinguish between them by these features.

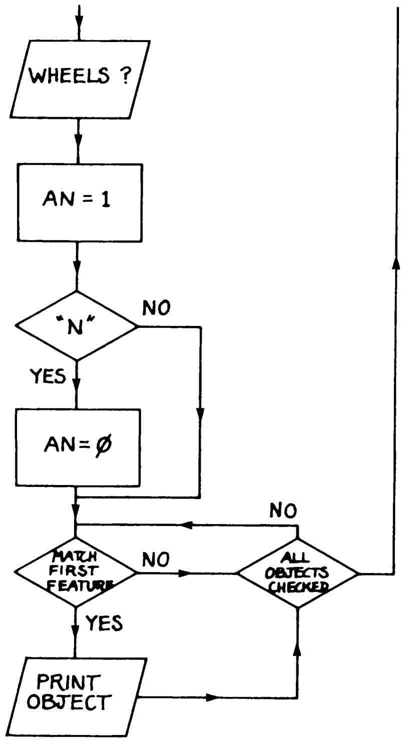
Table 5.1: Presence or Absence of Features

	bicycle	car	train	plane	horse
wheels	1	1	1	1	0
wings	0	0	0	1	0
engine	0	1	1	1	0
tyres	1	1	0	1	0
rails	0	0	1	0	0
windows	0	1	1	1	0
chain	1	0	0	0	0
steering	1	1	0	1	1

We will enter these values as DATA and then READ them into a two-dimensional array FE(n,n) which will hold a copy of this pattern, together with a string array containing the names of the objects OB$(n).

```
10 GOSUB 10000
10000 DIM OB$(5),FE(5,8)
11000 DATA BICYCLE,1,0,0,1,0,0,1,1
11010 DATA CAR,1,0,1,1,0,1,0,1
11020 DATA TRAIN,1,0,1,0,1,1,0,0
11030 DATA PLANE,1,1,1,1,0,1,0,1
11040 DATA HORSE,0,0,0,0,0,0,0,1
12000 FOR N=1 TO 5
12010 READ OB$(N)
12020 FOR M=1 TO 8
12030 READ FE(N,M)
12040 NEXT M,N
13000 RETURN
```

We can now ask whether the first feature is present or not and use the reply to print out which modes of transport match at this particular point (see **Flowchart 5.4**).



Flowchart 5.4 A Parallel Approach

```
100 PRINT"DOES IT HAVE WHEELS"
500 INPUT IN$
510 AN=1:IF IN$="N" THEN AN=0
520 FOR N=1 TO 5
530 IF FE(N,1)=AN THEN PRINT OB$(N)
540 NEXT N
```

In this case, answering ‘Y’ will produce a print-out of:

BICYLE
CAR
TRAIN
PLANE

and answering ‘N’ will produce a print-out of only:

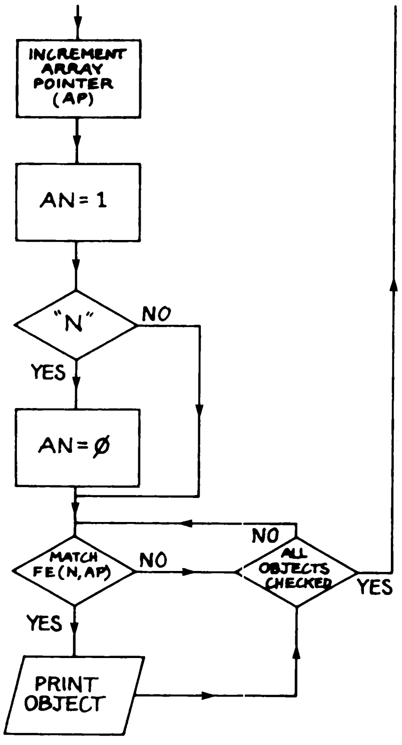
HORSE

This clearly demonstrates a possible disadvantage of the parallel method as, although we have just shown that only a horse does not have wheels, the program insists that we still ask all the other questions before it commits itself. This is not really as silly as it seems at first, as if you answer ‘Y’ to the next question (‘does it have wings’) you will see that the computer quite logically refuses to believe in flying horses.

If we put the actual comparison part as a subroutine we can use it to check for all eight features in turn. We would need to make slight modifications, adding an array pointer AP which is incremented to check the next element of the feature array FE(N,AP) in each cycle (see **Flowchart 5.5**).

```
100 PRINT"DOES IT HAVE WHEELS"
110 GOSUB 500
120 PRINT"DOES IT HAVE WINGS"
130 GOSUB 500
140 PRINT"DOES IT HAVE AN ENGINE"
150 GOSUB 500
160 PRINT"DOES IT HAVE TYRES"
170 GOSUB 500
180 PRINT"DOES IT NEED RAILS"
190 GOSUB 500
200 PRINT"DOES IT HAVE WINDOWS"
210 GOSUB 500
220 PRINT"DOES IT HAVE A CHAIN"
230 GOSUB 500
```

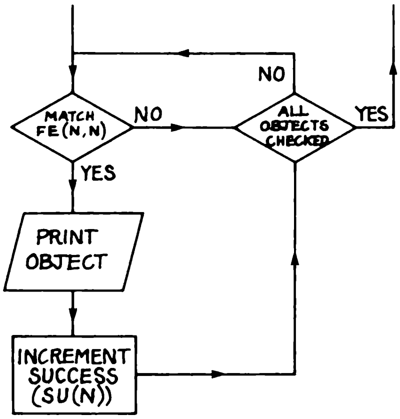
```
240 PRINT"IS IT STEERABLE"
250 GOSUB 500
400 PRINT
410 RUN
510 AP=AP+1:AN=1:IF IN$="N" THEN AN=0
530 IF FE(N,AP)=AN THEN PRINT OB$(N)
560 RETURN
```



Flowchart 5.5 Checking the Features in Turn

Top of the pops

The previous routine will print out a list of matches for each individual question as it proceeds, but it will not actually tell us which set of DATA is an overall match for the answers to all the questions. We can produce a SCORE which shows how well the answers match the DATA by having a success array element SU(n) for each object, which is only incremented when a match is found FE(N,AP)=AN (see **Flowchart 5.6**).



Flowchart 5.6 Measuring Success

```
260 PRINT
270 PRINT"SCORE"
280 PRINT
300 FOR N=1 TO 5
310 PRINT OB$(N),SU(N)
320 NEXT N
530 IF FE(N,AP)=AN THEN PRINT OB$(N):SU(
N)=SU(N)+1
10010 DIM SU(5)
```

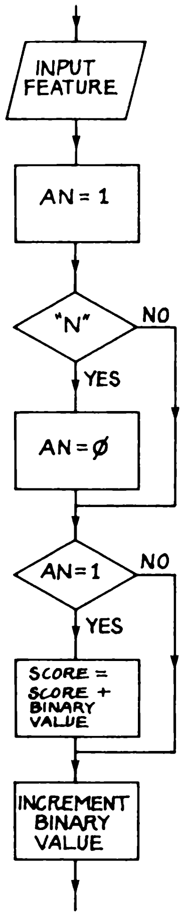
If a complete match is found then SU(n) will be equal to 8. Where one or more points were incorrect the score will be lowered. Scoring in this way is particularly useful where the correct answers to the questions are more a matter of opinion than fact (eg is a horse really steerable?), as the highest score actually obtained probably points to the correct answer anyway. (Notice that in this case each correct answer has equal weighting.)

Better in bits

You may have noticed that we just happened to use eight features for comparison and it may have occurred to you that this choice was not entirely accidental as there are eight bits in a byte. If we consider each feature as representing a binary digit (see **Table 5.2**), rather than an absolute value, then each object can be described by a single decimal number which is the sum of the binary digits, instead of by eight separate values. We will convert to decimal with the least significant bit at the top so that, starting from the top at 'wheels', each feature is equivalent to 1, 2, 4, 8, 16, 32, 64, 128 in decimal notation.

Table 5.2: Binary Weighted Features

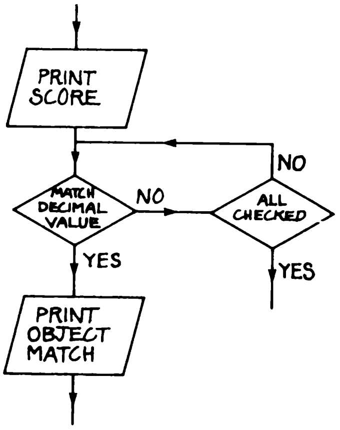
	bicycle	car	train	plane	horse
wheels	1	1	1	1	0
wings	0	0	0	2	0
engine	0	4	4	4	0
tyres	8	8	0	8	0
rails	0	0	16	0	0
windows	0	32	32	32	0
chain	64	0	0	0	0
steering	128	128	0	128	128
sum total	201	173	53	175	128



Flowchart 5.7 Producing a Binary Score

It is not too difficult to convert our 'score' of 1 to 8 into the appropriate binary value, as long as we remember that the decimal value of the binary digit BV must double each time we move down, and that we must only add the current binary value to the score if the answer was 'yes' (AN=1, see **Flowchart 5.7**).

If you consider for a moment, you will realise that we only need to keep track of the total number produced, SU, by adding the binary values of the 'yes' answers. There is no need to loop through and check each part of the array contents each time, or even to have a two-dimensional array at all! The only DATA we need to enter are the overall decimal values for each object, DV(n), and when all the questions have been asked we can check these against the decimal value obtained by the binary conversion of the 'yes/no' answers, SU (see **Flowchart 5.8**). The simplest thing for you to do



Flowchart 5.8 Matching the Decimal Value

now is to delete everything after line 260 and start entering from scratch again!

```
270 PRINT,"SCORE";SU
280 PRINT
300 FOR N=1 TO 5
310 IF DV(N)=SU THEN PRINT,"OB$(N):GOTO
400
320 NEXT N
330 PRINT,"OBJECT NOT FOUND"
400 PRINT
410 RUN
500 INPUT IN$
```

```
510 AN=1:IF IN$="N" THEN AN=0
520 IF AN=1 THEN SU=SU+BV
530 BV=BV+BV
540 RETURN
10000 DIM OB$(5),DV(5)
10010 BV=1
11000 DATA BICYCLE,201
11010 DATA CAR,173
11020 DATA TRAIN,53
11030 DATA PLANE,175
11040 DATA HORSE,129
12000 FOR N=1 TO 5
12010 READ OB$(N),DV(N)
12020 NEXT N
13000 RETURN
```

This approach obviously saves a lot of memory and time, as each array element takes up several bytes and must be located before it can be compared, so it is particularly useful where you are dealing with large amounts of information. On the other hand, it does mean that you have to calculate the decimal equivalents of all of the bit patterns before you can use them, and it also gives you no clues when a complete match is not found. (Note that you cannot simply take the nearest decimal value here, as the decimal equivalent value of each correct answer depends on its position.) Of course you could do the calculations the hard way, but if you enter the bit pattern as a string, I$, then it is quite easy to convert it to the equivalent decimal value DV by comparing each single character slice MID$(I$,N,1) with '1' and then adding on the value of the appropriate binary digit BD if a match is found.

```
20000 BD=1:INPUT I$
20010 FOR N=1 TO 8
20020 IF MID$(I$,N,1)="1" THEN DV=DV+BD
20030 BD=BD+BD
20040 NEXT N
20050 PRINT DV
20060 RUN
```

CHAPTER 6

Making Your Expert System Learn for Itself

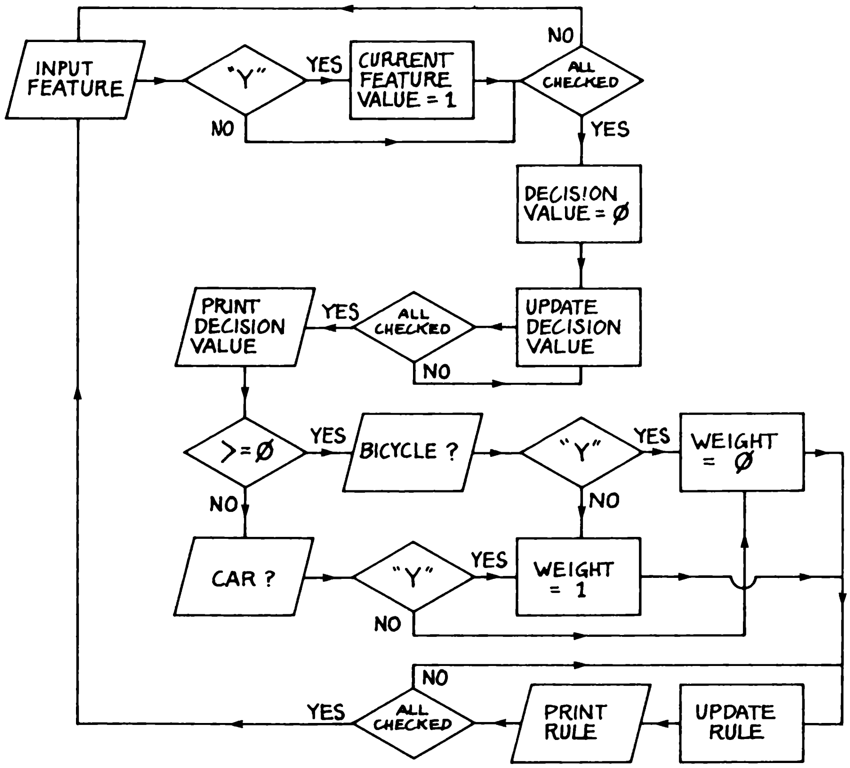
Although the 'expert' systems described so far will function all right, they all require you to give them the correct rules on which to base their decisions in advance, which can be very tedious.

However, it is possible to construct an expert program which can learn from its mistakes and work out the decision rules for itself, provided that you can tell it when (although not where) it goes wrong. This is obviously an advantage if you are not altogether sure of the correct rules yourself anyway! In this case we start out with a series of features which should enable us to distinguish between the different objects, but without any pre-defined yes/no pattern of these features ('decision rule') to guide us. Instead we use the program itself to calculate what the pattern should be.

We will work with our familiar transport example and begin by setting up some variables. FE% is the number of features to be considered (8), FE$(n) is an array containing the names of these features, FV(n) is an array which will hold the values which you give to each feature as input at any particular point (0 or 1), and RU(n) is an array which will hold the current overall values of the decision rule on each feature.

```
10 GOSUB 10000
10000 FE%=8
10010 DIM FE$(FE%),FV(FE%),RU(FE%)
10020 FOR N=1 TO FE%
10030 READ FE$(N)
10040 NEXT N
11000 DATA WHEELS,WINGS,ENGINE,TYRES,RAI
LS,WINDOWS,CHAIN,STEERING
12000 RETURN
```

Each feature is considered in turn (see **Flowchart 6.1**). First the current feature value FV(n) for this cycle is set to zero, and then a 'yes/no' input IN$ is requested from the user on each point. If IN$ is 'Y' the feature value element FV(N) is set to 1; otherwise it remains set at zero. This will produce a pattern which describes the object as '0' and '1' in array FV(n).



Flowchart 6.1 Learning to Distinguish Between Two Objects

```
60 FOR N=1 TO FE%
70 FV(N)=0
80 PRINT FE$(N);" ";
90 GET IN$:IF IN$="" THEN 90
100 PRINT IN$,
110 IF IN$="Y" THEN FV(N)=1
120 NEXT N
```

Before you start a decision variable DE% is set to zero. This is recalculated as the sum of the *current* value of DE%, plus each of the feature values FV(N) entered, multiplied by the current decision rule values RU(N).

```
125 DE%=0
130 FOR N=1 TO FE%
150 DE%=DE%+FV(N)*RU(N)
160 NEXT N
170 PRINT "DE%= ";DE%
```

Which is which?

To start with we will consider the simplest situation where there are only two possibilities — a bicycle or a car. Initially we make the distinction between these quite arbitrarily by saying that if the final value of DE% is equal to or greater than 0 then it is a bicycle, whereas if DE% is less than 0 then it is a car. It does not really matter that this is not actually true as the system will soon correct itself. When the program has made a decision on the basis of the value of DE% it requests confirmation (or otherwise) of the result.

```
180 IF DE%>=0 THEN PRINT"IS IT A BICYCLE
 ";:INPUT IN$:GOTO 200
190 IF DE%<0 THEN PRINT"IS IT A CAR ";:I
NPUT IN$:GOTO 220
```

Three possible courses of action may be taken according to whether or not the computer's decision was correct. If it was correct then effectively no action is taken (a weighting variable WT% is set to zero), and the program loops back for another try. If DE% was >=0 but the computer was wrong, then the weighting variable WT% is set to minus one, whilst if DE% was <0 but the computer was wrong then WT% is set to plus one.

```
200 IF IN$="Y" THEN WT%=0:GOTO 240
210 WT%=-1:GOTO 240
220 IF IN$="Y" THEN WT%=0:GOTO 240
230 WT%=1
```

The effect of the weighting variable is to modify the values in the rule array RU(N), pulling them down when they are too high, and pulling them up when they are too low.

```
240 FOR N=1 TO FE%
250 RU(N)=RU(N)+FV(N)*WT%
260 PRINT RU(N),
270 NEXT N
280 PRINT
290 GOTO 60
```

The way the system operates is best seen by a demonstration. Type RUN and then follow this sequence of entries. (Note that the punctuation has been designed to give a screen format which clearly indicates the relationship between your input values and the decision rule values.)

First of all enter these values:

WHEELS Y	WINGS N	ENGINE N	TYRES Y
RAILS N	WINDOWS N	CHAIN Y	STEERING Y

The program will return with a decision value DE% of zero, as this is the initial value and no modifications have yet taken place:

```
DE%=0
```

As DE% is 0, the system assumes that this is a bicycle and asks for confirmation, to which the answer is of course 'yes'.

```
IS IT A BICYCLE ? Y
```

The print-out of the contents of the rule array RU(n) shows that these have not changed from zero as the correct answer was, by pure chance, obtained:

```
 0          0          0          0
 0          0          0          0
```

Now try entering this sequence, which describes a car:

```
WHEELS Y    WINGS N      ENGINE Y    TYRES Y
RAILS N     WINDOWS Y    CHAIN N     STEERING Y
```

DE% is still zero, so the wrong conclusion is reached and the wrong question is asked, to which the answer must be 'no':

```
DE%=0
IS IT A BICYCLE ? N
```

Now, as a mistake was made, the decision rule is modified by subtracting one from each value in the rule array where a 'yes' answer was given. The contents of the rule array thus become:

```
-1          0          -1         -1
0           -1         0          -1
```

If you once more enter the values which describe a car, the program will come up with the correct answer:

```
WHEELS Y    WINGS N      ENGINE Y    TYRES Y
RAILS N     WINDOWS Y    CHAIN N     STEERING Y
```

```
DE%=-5
IS IT A CAR ? Y
```

```
-1          0          -1         -1
0           -1         0          -1
```

Before you feel too pleased with yourself, try giving it the values for a bicycle again, which it will get wrong!

```
WHEELS Y    WINGS N      ENGINE N    TYRES Y
RAILS N     WINDOWS N    CHAIN Y     STEERING Y

DE%=-3
IS IT A CAR ? N

 0          0            -1          0
 0          -1           1           0
```

However the positive features which are common to the bicycle and the car are now automatically increased by one, so that if you repeat this last sequence it will now produce the correct conclusion:

```
WHEELS Y    WINGS N      ENGINE N    TYRES Y
RAILS N     WINDOWS N    CHAIN Y     STEERING Y

DE%=1
IS IT A BICYCLE ? Y

 0          0            -1          0
 0          -1           1           0
```

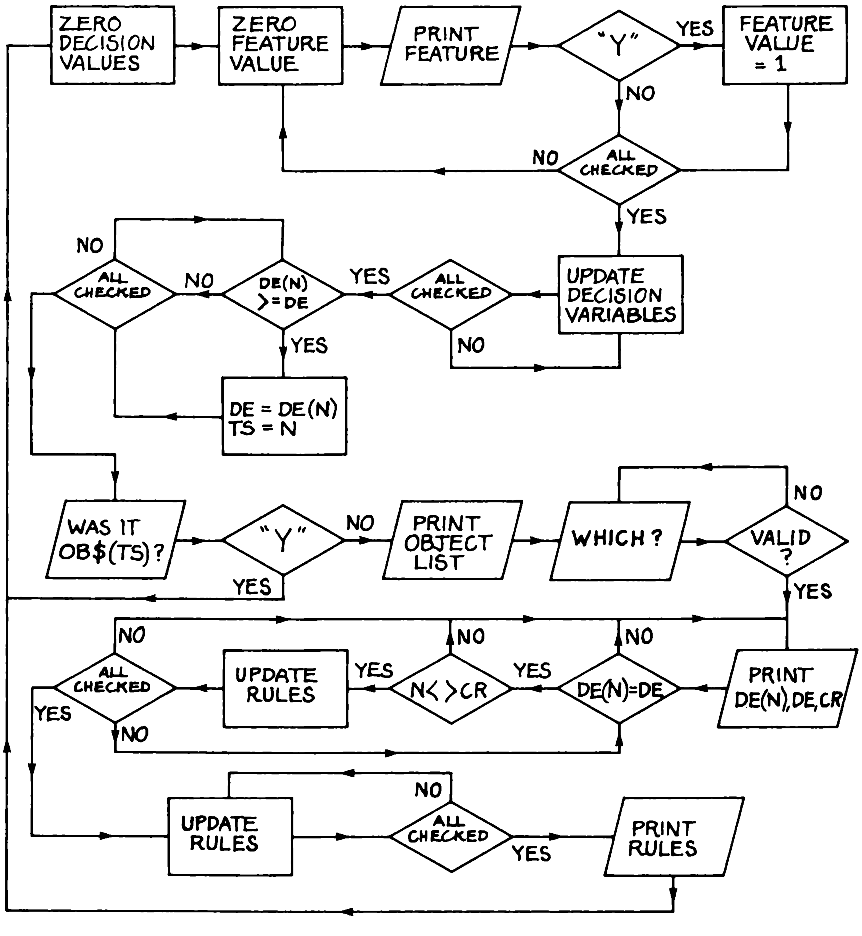
The situation has now stabilised and the program will always recognise both car and bicycle correctly every time you enter the features which describe them:

```
WHEELS Y    WINGS N      ENGINE Y    TYRES Y
RAILS N     WINDOWS Y    CHAIN N     STEERING Y

DE%=-2
IS IT A CAR ? Y

0           0            -1          0
0           -1           1           0
```

Notice that the final value of DE% for a bicycle is 1, and for a car −2. If you look at the rule array values, you will see that these correspond in both number and position to the unique features which distinguish these objects (CHAIN for bicycle, and ENGINE and WINDOWS for car).

A wider spectrum

Although you have now managed to teach your computer something, it is not exactly earth-shattering for it to be able to distinguish between only two objects. Let's expand the system to deal with a wider spectrum of possibilities (see **Flowchart 6.2**). To start with we need to define the



Flowchart 6.2 Learning the Rules for a Wider Spectrum of Possibilities

number of objects we wish to be able to recognise OB%, name them as DATA which we READ into a new array OB$(OB%), change our decision rule array into a two-dimensional form, RU(FE%,OB%), which can hold rules for each of the objects separately, and set up a decision array DE(n) to hold decision values for each object.

```
10 GOSUB 10000
10000 FE%=8:OB%=5
10010 DIM FE$(FE%),FV(FE%),RU(FE%,OB%),O
B$(OB%),DE(OB%)
10020 FOR N=1 TO FE%
10030 READ FE$(N)
10040 NEXT N
10050 FOR N=1 TO OB%
10060 READ OB$(N)
10070 NEXT N
11000 DATA WHEELS,WINGS,ENGINE,TYRES,RAI
LS,WINDOWS,CHAIN,STEERING
11010 DATA BICYCLE,CAR,TRAIN,PLANE,HORSE
12000 RETURN
```

Rather than just having a single decision variable DE%, we need here to determine a decision value for each object each time. In each cycle we must first set DE% to zero, and then zero every element in the decision array DE(n) so that we start with a clean slate for every object.

```
20 DE%=0
30 FOR N=1 TO OB%
40 DE(N)=0
50 NEXT N
```

The values for each feature are then entered in exactly the same way as before.

```
60 FOR N=1 TO FE%
70 FV(N)=0
80 PRINT FE$(N);" ";
90 GET IN$:IF IN$="" THEN 90
100 PRINT IN$
110 IF IN$="Y" THEN FV(N)=1
120 NEXT N
```

Each element of the decision array DE(n) is now updated according to the status of the entered values FV(n) and the contents of the appropriate rule array element RU(n,m).

```
130 FOR N=1 TO FE%
140 FOR M=1 TO OB%
150 DE(M)=DE(M)+FV(N)*RU(N,M)
160 NEXT M,N
```

We now need to look to see if any of the decision values for any of the objects DE(n) are greater than or equal to the overall decision value DE%. If this is true, we set a 'top score' TS% variable equal to the number of the object producing the best match.

```
170 FOR N=1 TO OB%
180 IF DE(N)>=DE% THEN DE%=DE(N):TS%=N
190 NEXT N
```

The best guess of the system is that this is the correct answer, so once again it asks for confirmation, and simply returns for a new input without making any changes if the answer was correct.

```
200 PRINT "WAS IT ";OB$(TS%);" ";
210 GET IN$:IF IN$="" THEN 210
215 PRINT IN$
220 IF IN$="Y" THEN 20
```

If this was not the correct answer, the names and numbers of all the objects are printed out and you are asked for the number of the correct answer CR%. (The limitations on CR% prevent you crashing the program by entering an illegal value.)

```
230 FOR N=1 TO OB%
240 PRINT N,OB$(N)
250 NEXT N
260 PRINT "WHICH WAS IT";
270 GET CR%:IF CR%<1 OR CR%>5 THEN 270
275 PRINT CR%
```

A check is now made to see if the decision value for each object DE(n) is greater than or equal to the overall decision value DE% *and* whether the object being considered is *not* the correct answer. If *both* of these are true then the rules are updated again by subtracting the correct feature values FV(n) to bias in favour of the correct answer.

```
280 FOR N=1 TO OB%
290 PRINT DE(N),DE%,CR%
300 IF DE(N)>=DE% AND N<>CR% THEN FOR M=
1 TO FE%:RU(M,N)=RU(M,N)-FV(M):NEXT M
310 NEXT N
```

Now the correct feature values FV(n) are added to the rule array for the correct object, to bias in the opposite direction.

```
320 FOR M=1 TO FE%
330 RU(M,CR%)=RU(M,CR%)+FV(M)
340 NEXT M
```

Finally the status of the rule arrays are printed out so that you can see what is happening.

```
350 FOR M=1 TO OB%
360 FOR N=1 TO FE%
370 PRINT RU(N,M);
380 NEXT N
390 PRINT
400 NEXT M
410 GOTO 20
```

Once again a demonstration is the best way to understand what is happening so enter the following sequence:

WHEELS Y	WINGS N	ENGINE N	TYRES Y
RAILS N	WINDOWS N	CHAIN Y	STEERING Y

The program will come back with the erroneous conclusion that it was a horse, so you must tell it that this was wrong, when it will ask you for the correct answer (bicycle = 1):

WAS IT HORSE N

1	BICYCLE
2	CAR
3	TRAIN
4	PLANE
5	HORSE

WHICH WAS IT 1

The statuses of the various decision and rule arrays are now printed out for your information (note that the labels shown here are not included on the screen).

(DE(N))	(DE%)	(CR%)
0	0	1
0	0	1
0	0	1
0	0	1
0	0	1

```
 1   0   0    1   0   0    1    1        (bicycle)
-1   0   0   -1   0   0   -1   -1            (car)
-1   0   0   -1   0   0   -1   -1          (train)
-1   0   0   -1   0   0   -1   -1          (plane)
-1   0   0   -1   0   0   -1   -1          (horse)

 A   B   C    D   E   F    G    H
```

(A=wheels B=wings C=engine D=tyres
E=rails F=windows G=Chain H=Steering)

If you look closely you will see that the features which have caused alterations in the rule arrays are wheels, tyres, chain and steering — all features which we defined as part of a bicycle and not found in a horse. In addition, you will see that the values for these features in the bicycle rule array are now all plus one, whilst the values for these features for all the other objects are now all minus one.

Now give it the features of a car, which it will think a bicycle, and then correct it. Notice that the rule arrays for bicycle and car are now amended to take into account the new information.

```
WHEELS Y      WINGS N        ENGINE Y      TYRES Y
RAILS N       WINDOWS Y      CHAIN N       STEERING Y

WAS IT BICYCLE N

1     BICYCLE
2     CAR
3     TRAIN
4     PLANE
5     HORSE

WHICH WAS IT 2
 3            3            2
-3            3            2
-3            3            2
-3            3            2
-3            3            2

 0   0   -1    0   0   -1    1    0  (bicycle)
 0   0    1    0   0    1   -1    0  (car)
-1   0    0   -1   0    0   -1   -1  (train)
-1   0    0   -1   0    0   -1   -1  (plane)
-1   0    0   -1   0    0   -1   -1  (horse)
```

```
A    B      C      D    E      F      G      H

(A=wheels      B=wings        C=engine       D=tyres
E=rails        F=windows      G=chain        H=steering)
```

Next give it a plane, which it decides is a car, and correct it again.

```
WHEELS Y       WINGS Y        ENGINE Y       TYRES Y
RAILS N        WINDOWS Y      CHAIN N        STEERING Y

WAS IT CAR N

1    BICYCLE
2    CAR
3    TRAIN
4    PLANE
5    HORSE

WHICH WAS IT 4
```

And now a train, which it still gets wrong!

```
WHEELS Y       WINGS N        ENGINE Y       TYRES N
RAILS N        WINDOWS Y      CHAIN N        STEERING N

WAS IT PLANE N

1    BICYCLE
2    CAR
3    TRAIN
4    PLANE
5    HORSE

WHICH WAS IT 3
```

And finally a horse, which comes out as a plane!

```
WHEELS N       WINGS N        ENGINE N       TYRES N
RAILS N        WINDOWS N      CHAIN N        STEERING Y

WAS IT PLANE N

1    BICYCLE
2    CAR
```

```
3    TRAIN
4    PLANE
5    HORSE

WHICH WAS IT 5
```

If you continue to feed your expert information, eventually it will get the right answer every time. How long this will take depends upon the extent of the differences between the features of the objects, and on the order in which the objects are presented to the expert. Be warned that it can take a long time before it becomes infallible. Here is one sequence which eventually came out right every time.

plane (train)	car (plane)	bicycle (YES)
car (YES)	plane (car)	plane (YES)
horse (YES)	plane (bicycle)	car (plane)
plane (car)	plane (car)	car (plane)
car (YES)	plane (car)	plane (YES)
car (YES)	plane (YES)	horse (YES)
bicycle (YES)	train (car)	train (YES)
bicycle (YES)	car (plane)	car (YES)
plane (car)	plane (YES)	car (plane)
car (YES)	plane (YES)	car (YES)
bicycle (car)	car (YES)	plane (YES)
train (YES)	horse (YES)	bicycle (YES)

To see the final state of the rule array when the ultimate state is reached, you can stop the program and then type GOTO 350 as a direct command. As the final scale of values ranges from +6 to −2, you should not be surprised that it took a long time to get there.

1	0	−1	1	0	−2	3	0	(bicycle)
−1	4	1	0	−1	1	−2	0	(car)
0	−1	1	−2	2	1	−1	−2	(train)
−2	6	0	0	−1	0	−2	−2	(plane)
−1	0	0	−1	0	0	−1	0	(horse)
A	B	C	D	E	F	G	H	

(A=wheels	B=wings	C=engine	D=tyres
E=rails	F=windows	G=chain	H=steering)

Of course, in a real application of such an expert system you could feed it a mass of collected information and conclusions on a subject area and then leave it alone to digest this and to come up with the rules in its own good time. As these rules are stored in arrays you could easily write a routine to save these for re-use later.

CHAPTER 7
Fuzzy Matching

Computers are totally logical but our own memory banks are much more unreliable, which can lead to problems when you are trying to recover information on a particular subject. For example, English is a very variable language and there are frequently alternative spellings of the same (or very similar) surnames, which can cause difficulties.

One way around this problem is to try to match the sound of the word, rather than the actual letters in it, by means of 'Soundex Coding', which was originally developed to assist processing of the 1890 census in the USA. This method of coding ensures that similar-sounding words have almost the same code sequence.

The rules for coding a word are as follows:

1) Always retain the first letter of the word as the first character of the code.

From the second letter onward:

2) Ignore vowels (a, e, i, o, u).

3) Ignore the letters w, y, q and h.

4) Ignore punctuation marks.

5) Code the other letters with the values 1–6 as follows:

Letters	Code
bfpv	1
cgjksxz	2
dt	3
l	4
mn	5
r	6

6) Where adjacent letters have the same code only the first one is retained.

7) If length of code is greater than four characters then take first four only.

8) If length of code is less than four characters then pad out to four characters with zeros.

To make this clear here are some examples of Soundex Coded names:

BRAIN - B650

(B is retained, R is 6, A and I are dropped, N is 5 and a zero is added to pad out the code.)

CUNNINGHAM - C552

(C is retained, U is dropped, both Ns are represented by the single code 5, I is dropped, the third N is represented by 5, G is 2, H and A are dropped, and M is coded as 5 — but the resulting code (C5525) is truncated to four characters.)

GORE - G600

(G is retained, O is dropped, R is 6, E is dropped and zeros are added to pad the code.)

IRELAND - I645

(I is retained, R is 6, E is dropped, L is 4, A is dropped, N is 5 and D is 3— but the resulting code (I6453) is truncated to four characters.)

SCOT - S230

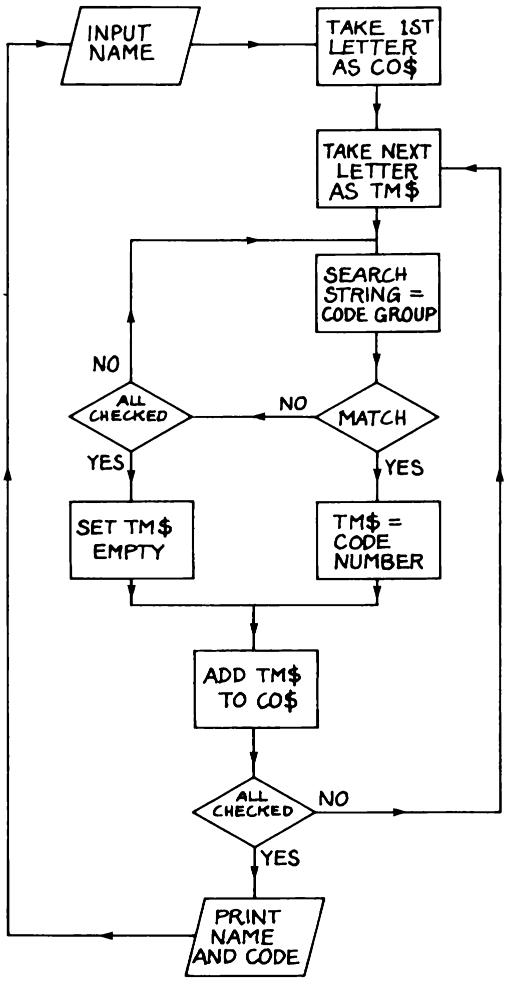
(S is retained, C is 2, O is dropped, T is 3 and zero is added to pad the code.)
If your name is full of vowels and other rejected letters, then you will find that your code is somewhat abbreviated!

HEYHOE - H000

(H is retained, all the other letters are rejected (!), and the code is filled up with zeros.)

Coding routine

To save all that brainwork, let's develop a program which allows you to input a word in English and output it in Soundex Code (see **Flowchart 7.1**) The first thing to do is to jump to a set-up routine which reads each group of the retained letters into one element of a Soundex Code string array SC$(n). (Note that the letters are arranged so that they are in the array element corresponding to their code value.)



Flowchart 7.1 Producing a Soundex Code

```
10 GOSUB 10000
10000 DIM SC$(6)
11000 DATA BFPV,CGJKSXZ,DT,L,MN,R
12000 FOR N=1 TO 6
12010 READ SC$(N)
12020 NEXT N
13000 RETURN
```

We can now input the word to be converted, IN$, and, to begin with, make the coded version of this, CO$, the first letter of that word (following the first rule above).

```
100 INPUT IN$
110 CO$=LEFT$(IN$,1)
```

We now need to check the other letters of the word, 2 TO LEN(IN$), in turn after first making a temporary string TM$ equal to the current letter to be translated.

```
120 FOR N=2 TO LEN(IN$)
130 TM$=MID$(IN$,N,1)
```

As conversion to the code numbers will be required at various points in the final problem, we will set up this process as a subroutine at line 1000.

```
140 GOSUB 1000
```

We have to check TM$ against each individual letter in each group of letters SC$(n) to find a match. To check each letter group, we have to go round six times, making a search string SE$ the current Soundex Code group, and jumping to an INSTR routine which checks each letter in the group against TM$ in turn.

```
1000 FOR P=1 TO 6
1010 SE$=SC$(P)
1020 GOSUB 5000
```

The INSTR routine is similar to the one used in previous chapters.

When the INSTR check has been made, we have to determine whether a match has been found to any of the Soundex groups, and if so, to which group. If no match is found then SP% will be set to zero. If a match *is* found then SP% will be set to M which will point to the value of the code group matched.

```
5000 FOR M=1 TO LEN(SE$)
5010 IF MID$(SE$,M,1)=TM$ THEN SP%=M:RET
URN
5020 NEXT M
5030 SP%=0
5040 RETURN
```

If a match is found, SP%>0, then we convert the numeric value of the loop scanning the code groups P to a string TM$ which replaces our original temporary string. (The STR$ command converts a number into a string, but we also need to use RIGHT$ as STR$ automatically adds a space on to the front of the number string.)

```
1030 IF SP%>0 THEN TM$=RIGHT$(STR$(P),1)
:RETURN
```

If no match is found in that group, we have to check the next group.

```
1040 NEXT P
```

If no match is found at all, then TM$ must contain one of the characters to be ignored. So we reset TM$ empty [$=""] and RETURN.

```
1050 TM$=""
1060 RETURN
```

We can now make the coded string CO$ equal to the original coded string plus the newly converted character TM$.

```
170 CO$=CO$+TM$
```

Now we loop back to deal with the next character in IN$.

```
180 NEXT N
```

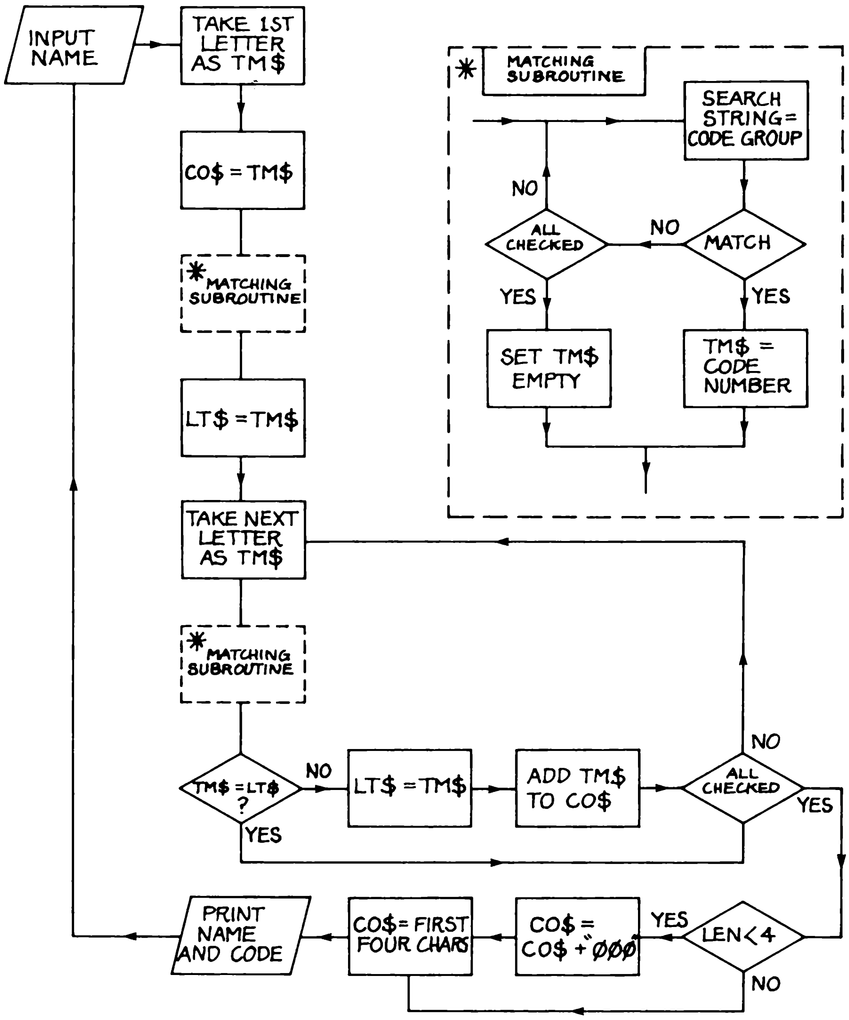
When the end of IN$ is reached, we print out the input IN$ and the entire coded string CO$ before going back to 100 for another input.

```
210 PRINT:PRINT "NAME","CODE":PRINT IN$,
CO$
320 GOTO 100
```

If you input the name STEVEN you will now get the code S315 which is correct. However, if you try BRAIN or CUNNINGHAM you will get the codes B65 and C55525 respectively. The code for BRAIN is too short and needs padding out with zeros, and the code for CUNNINGHAM is too long and the same codes are repeated one after another for the letter N.

Dealing with the details

To solve the problem of the repetition of the same code for adjacent letters, we need to keep a record of the last temporary string LT$. We need to make LT$ the code of the first character in IN$ to start with, so that the initial letter is not repeated. As we go through the FOR–NEXT loop, we must compare LT$ with TM$, and if they are the same we must not add TM$ to CO$. Otherwise we need to make LT$ the latest TM$.



Flowchart 7.2 Dealing with the Details

```
110 TM$=LEFT$(IN$,1):CO$=TM$:GOSUB 1000:
LT$=TM$
150 IF TM$=LT$ THEN GOTO 180
160 LT$=TM$
```

Now we can sort out the problem of the code being too short. First of all we check the length of the string LEN(CO$)<4. If it is too short, we add three zeros on to the end and then use LEFT$ to cut the string back down to the correct size (four characters).

```
190 IF LEN(CO$)<4 THEN CO$=CO$+"000":CO$
=LEFT$(CO$,4)
```

Finally, if the string is too long then we cut it down to size with LEFT$(CO$,4) again.

```
200 IF LEN(CO$)>4 THEN CO$=LEFT$(CO$,4)
```

Matchmaking

Now that we have a reliable method of producing the Soundex Codes, let's give it something to work on. The first task is to read a list of names out of DATA statements into a name string array NA$(n). Our demonstration list only consists of eighteen names — if you want more, a quick flick through your local telephone directory should soon solve that problem! Note that the number of words is also stored as NW%.

```
10010 NW%=17:DIM NA$(NW%)
11010 DATA ABRAHAM,ABRAHAMS,ABRAMS,ADAM,
ADAMS,ADDAMS,ADAMSON,ALAN,ALLAN,ALLEN
11020 DATA ANTHANY,ANTHONY,ANTONY,ANTROB
US,APPERLEY,APPLEBEE,APPLEBY,APPLEFORD
12030 FOR N=0 TO 17
12040 READ NA$(N)
12050 NEXT N
```

The whole idea of matching with Soundex Codes relies on the fact that you use the Soundex Code to make the match before printing the possible words. We therefore have to find the codes for each of the names from the DATA and put these coded into an equivalent string array NC$(n). The routine to find the Soundex Code is virtually identical to the one used to find the code of an input, as described above.

```
10020 DIM NC$(NW%)
12060 PRINT:PRINT "NAME","CODE":PRINT
12070 FOR Q=0 TO NW%
12090 PRINT NA$(Q),
12090 TM$=LEFT$(NA$(Q),1):CO$=TM$:GOSUB
1000:LT$=TM$
12100 FOR N=2 TO LEN(NA$(Q))
12110 TM$=MID$(NA$(Q),N,1)
12120 GOSUB 1000
12130 IF TM$=LT$ THEN NEXT N:GOTO 12170
12140 LT$=TM$
```

```
12150 CO$=CO$+TM$
12160 NEXT N
12170 IF LEN(CO$)<4 THEN CO$=CO$+"000":C
O$=LEFT$(CO$,4)
12180 IF LEN(CO$)>4 THEN CO$=LEFT$(CO$,4)
12190 PRINT CO$
12200 NC$(Q)=CO$
12210 NEXT Q
```

If you RUN this now, you will see all the codes for the DATA produced before the input request.

NAME	CODE
ABRAHAM	A165
ABRAHAMS	A165
ABRAMS	A165
ADAM	A350
ADAMS	A352
ADDAMS	A352
ADAMSON	A352
ALAN	A450
ALLAN	A450
ALLEN	A450
ANTHANY	A535
ANTHONY	A535
ANTONY	A535
ANTROBUS	A536
APPERLEY	A164
APPLEBEE	A141
APPLEBY	A141
APPLEFORD	A141

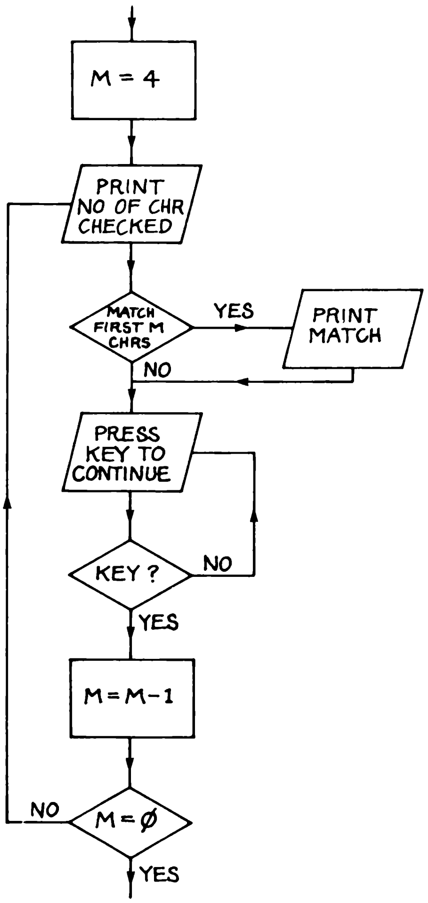
The only thing we need to do now is to find which codes of these names match the code of your input and then to print out these names with a FOR-NEXT loop.

```
240 PRINT
250 FOR N=0 TO NN%
260 IF CO$=NC$(N) THEN PRINT NA$(N),NC$(
N)
270 NEXT N
```

This will only print words with exactly matching Soundex Codes. For example, if you try entering the name APPLEBE you will get the following response:

```
? APPLEBE

NAME          CODE
APPLEBE       A141

APPLEBEE      A141
APPLEBY       A141
APPLEFORD     A141
```

Although APPLEBE (one E at the end!) is not present in the DATA, we have found APPLEBEE and APPLEBY, as well as APPLEFORD (where the interesting sound at the end has been chopped off).



Flowchart 7.3 Partial Matching

Partial matching

Notice that on the other hand APPERLEY has been rejected, even though it sounds quite similar at first. It would therefore be useful if we could also print out partial matches.

This can easily be done by adding an extra FOR–NEXT loop, which compares a decreasing section of the input LEFT$(CO$,M) with decreasing lengths of the stored codes LEFT$(NC$(N),M) (see **Flowchart 7.3**).

```
230 FOR M=4 TO 1 STEP -1
240 PRINT:PRINT M;"CHARACTERS MATCH":PRI
NT
260 IF LEFT$(CO$,M)=LEFT$(NC$(N),M) THEN
 PRINT NA$(N),NC$(N)
280 PRINT:PRINT "PRESS KEY TO CONTINUE"
290 GET IN$:IF IN$="" THEN 290
300 PRINT:PRINT
310 NEXT M
```

If you now try APPLEBE you can see the whole range of possibilities.

```
? APPLEBE

NAME          CODE
APPLEBE       A141

4 CHARACTERS MATCH
APPLEBEE      A141
APPLEBY       A141
APPLEFORD     A141
PRESS KEY TO CONTINUE

3 CHARACTERS MATCH
APPLEBEE      A141
APPLEBY       A141
APPLEFORD     A141
PRESS KEY TO CONTINUE

2 CHARACTERS MATCH
ABRAHAM       A165
ABRAHAMS      A165
ABRAMS        A165
APPERLEY      A164
```

```
APPLEBEE        A141
APPLEBY         A141
APPLEFORD       A141
PRESS KEY TO CONTINUE

1 CHARACTERS MATCH
ABRAHAM         A165
ABRAHAMS        A165
ABRAMS          A165
ADAM            A350
ADAMS           A352
ADDAMS          A352
ADAMSON         A352
ALAN            A450
ALLAN           A450
ALLEN           A450
ANTHANY         A535
ANTHONY         A535
ANTONY          A535
ANTROBUS        A536
APPERLEY        A164
APPLEBEE        A141
APPLEBY         A141
APPLEFORD       A141
PRESS KEY TO CONTINUE
```

CHAPTER 8
Recognising Shapes

We normally recognise objects using our senses of sight, sound, taste and feel, whereas of course our basic computer can only obtain information through the keyboard. Whilst it is possible to produce sensors which can be interfaced with your machine to give it another view of the outside world, constructing these requires a reasonable amount of electronic and mechanical knowledge and skill. We will make do instead with a simulation of the action of a light sensor to illustrate how shapes can be recognised.

Let us think for a start about three simple shapes — a vertical line, a square, and a right-angled triangle.

We can recognise these shapes by looking at the pattern they make on an imaginary grid and deciding whether or not there is a point set at each X and Y coordinate.

In the case of a line only the first X coordinate is used, but all of the Y coordinates. A square is a little more complicated, as all the X coordinates on Y rows 1 and 8 are set, and from Y rows 2 to 7 only the first and last X points are set. Finally, a triangle is even more complicated, as the slope is produced by incrementing the X axis each time

Table 8.1 Decimal Values of Shapes Described in Binary Form

Y row	line	square	triangle
1	1	255	1
2	1	129	3
3	1	129	5
4	1	129	9
5	1	129	17
6	1	129	33
7	1	129	65
8	1	255	255

One obvious way to describe these particular figures would be to represent each point by a single bit and to produce a decimal value for each row, in the same way as we did before when we were looking at expert systems (see **Table 8.1**). In fact this type of approach is used to produce the characters which you see on your screen display, the formats for which are stored in memory in just this form. For example **Figure 8.1** shows how the letter 'A' is built up.

There are now machines available (Optical Character Readers) which can reverse this process. They actually 'read' a printed page by scanning the paper in a grid pattern and measuring whether or not light is reflected at particular coordinates.

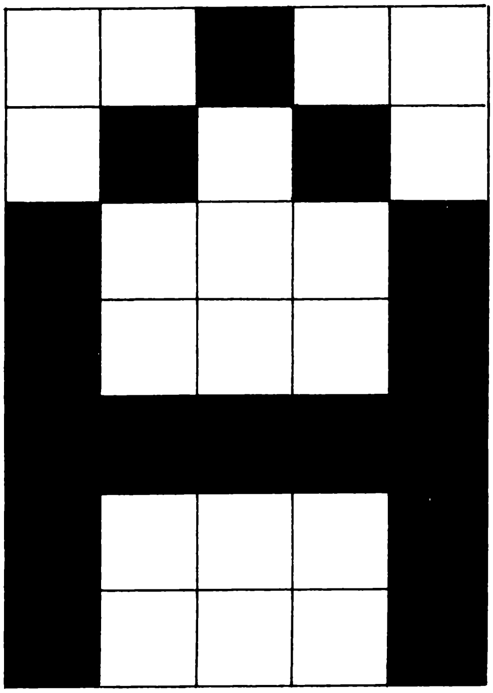


Figure 8.1 Forming the Letter 'A'

What they actually take in will be a pattern of 'yes' and 'no' for each coordinate, and of course this must then be decoded and compared with the patterns for known shapes. The most obvious way to make this comparison would be to consider every point in turn as a binary digit and then to convert each row back to a decimal value which could be compared with a table of known values. However this has the disadvantage that we must actually check every individual point on the grid (64 points).

A branching short cut

A quicker approach relies on the fact that each character can actually be detected by looking at only a much smaller number of critical features of the pattern. For example, **Figure 8.2** gives a decision tree which will find all

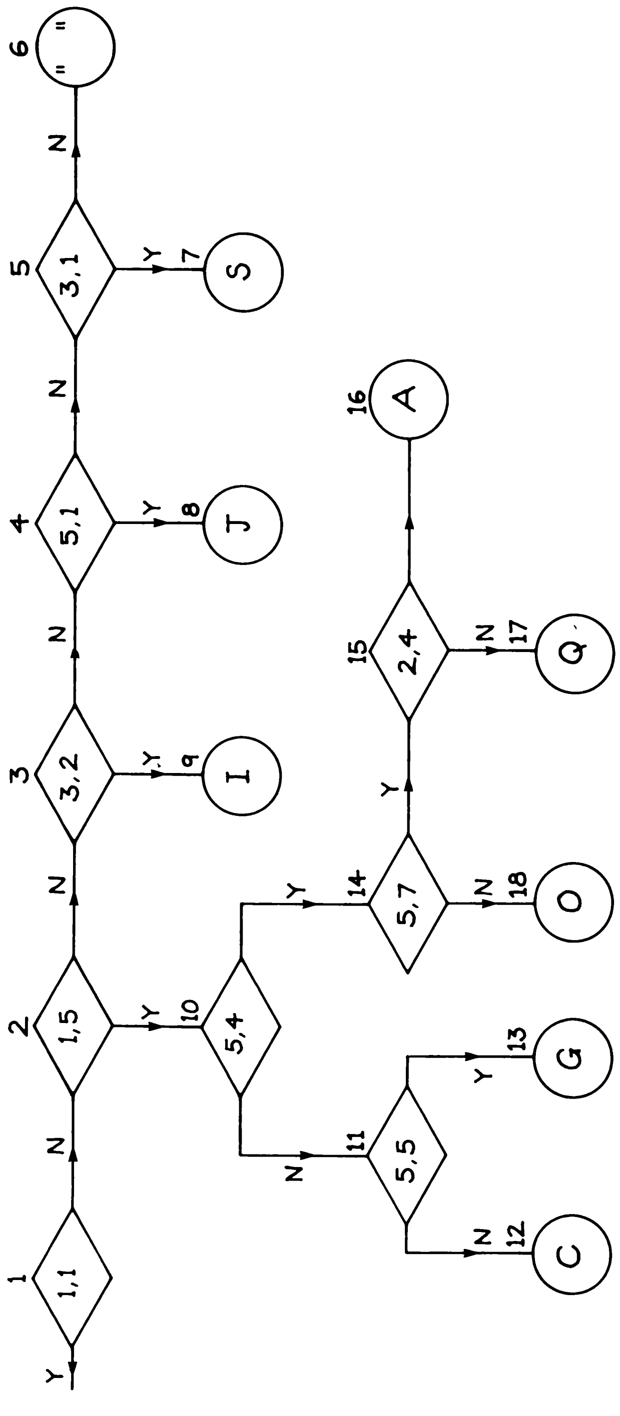


Figure 8.2a Decision Tree for Alphabet

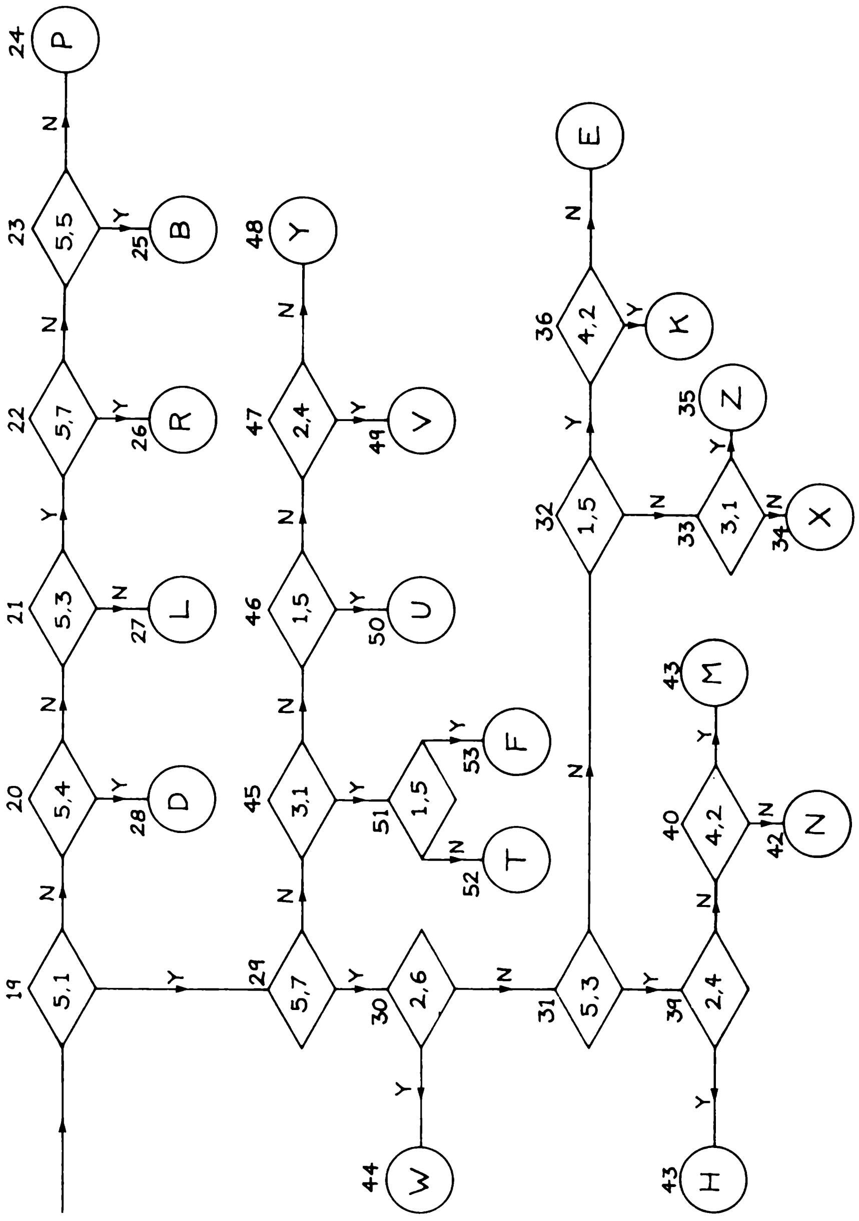

19
5,1
20
5,4
28
D
21
5,3
27
L
22
5,7
26
R
23
5,5
25
B
24
P
29
5,7
45
3,1
51
1,5
52
T
53
F
46
1,5
50
U
47
2,4
49
V
48
Y
30
2,6
44
W
31
5,3
32
1,5
33
3,1
34
X
35
Z
36
4,2
K
E
39
2,4
43
H
40
4,2
42
N
43
M
Y
N

Figure 8.2b

the capital letters of the alphabet using only 12 points (see **Figure 8.3**), and it is not even necessary to check all 12 in any particular case. If you follow each of the routes, you will see that the maximum number of steps to be

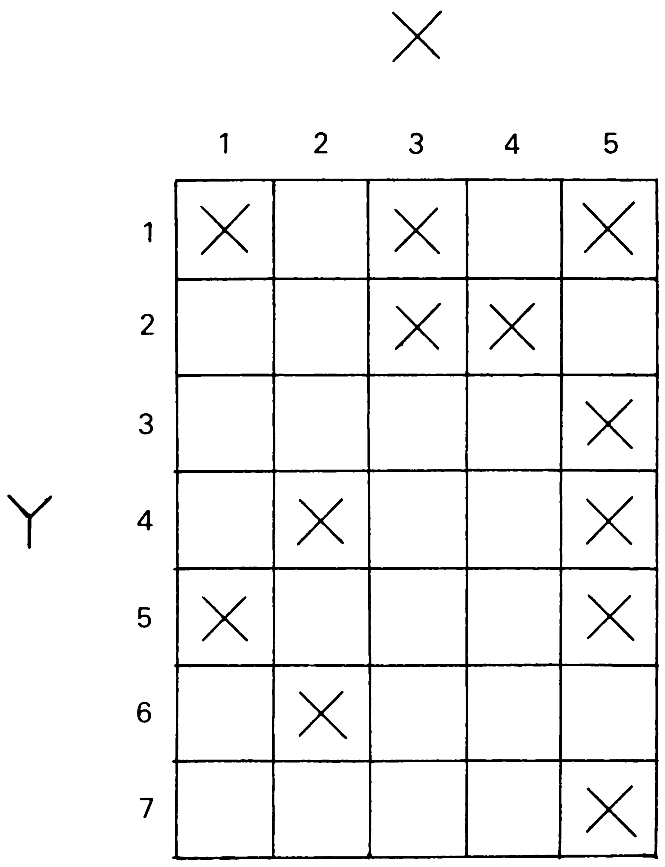


Figure 8.3 Points Used in Decision Tree

followed is seven, and that most letters are found in less than five steps (**Table 8.2**). This must obviously be quicker than comparing all 64 points!

Table 8.2 Numbers of Steps Required for Recognition of Each Character

3 steps	I, D
4 steps	L, J, C, G, O, W
5 steps	S, A, Q, R, T, F, U, space
6 steps	P, V, Y, H
7 steps	B, M, N, E, K, X, Z

To demonstrate how this approach works, we will simulate the action of the scanning head by producing a grid on the screen, on which you can construct characters.

The text screen start address 1024 and colour RAM offset 54272 are defined as variables, TS and CO respectively, as they are used frequently. The screen is cleared and a dark area 6 × 8 blocks is set up in the top lefthand corner. A lighter-coloured 5 × 7 grid is then superimposed on this to mark the actual working area (of course there must be a margin around the edge so that characters do not merge).

```
10 GOSUB 10000
10000 TS=1024:CO=54272
12000 PRINT "[CLR]"
12010 FOR N=1 TO 10
12020 PRINT
12030 NEXT N
13000 FOR X=0 TO 6
13010 FOR Y=0 TO 8
13020 POKE TS+CO+X+(Y*40),11
13030 POKE TS+X+(Y*40),224
13040 NEXT Y,X
13050 FOR X=1 TO 5
13060 FOR Y=1 TO 7
13070 POKE TS+CO+X+(Y*40),1
13080 NEXT Y,X
13090 X=1:Y=1
13100 RETURN
```

A flashing cursor is now produced to show your position. CP is the current position on the text screen, TS + offset, the current colour of which is saved as CC by PEEKing the equivalent position in the colour RAM. A different colour CC + 4 is then POKEd into place and the original colour (CC) POKEd back, so that there is no lasting effect.

```
20 GET A$
30 CP=TS+X+(Y*40):CC=PEEK(CP+CO):POKE CP
+CO,CC+4:POKE CP+CO,CC
40 IF A$="" THEN 20
```

The X and Y coordinates are updated according to the movement of the cursor keys, and if the spacebar is pressed the colour of the current position is set to black (0). If you make a mistake, the left arrow erases the current position by resetting the colour to 1, or CLR jumps to the set-up routine and erases all the current grid. Pressing RETURN leads to the decoding routine, or else the program loops back to the keycheck.

```
50 IF A$="[RIGHT CURSOR]" THEN X=X+1
60 IF A$="[LEFT CURSOR]" THEN X=X-1
70 IF A$="[DOWN CURSOR]" THEN Y=Y+1
80 IF A$="[UP CURSOR]" THEN Y=Y-1
90 IF A$=" " THEN POKE TS+CO+X+(Y*40)
,0
100 IF A$="[<--]" THEN POKE TS+CO+X+(Y*40
),1
110 IF A$="[CLR]" THEN GOSUB 13000
120 IF ASC(A$)=13 THEN 2000
170 GOTO 20
```

Limits must be set to prevent the cursor wandering off the 5 × 7 grid area.

```
130 IF X<1 THEN X=1
140 IF X>5 THEN X=5
150 IF Y<1 THEN Y=1
160 IF Y>7 THEN Y=7
```

The decision tree is held in a series of linked arrays where NB is the number of branches, LE$(n) holds the names of the letters, C1(n) the X coordinate to be checked next, C2(n) the Y coordinate to be checked next, N(n) the next element to use if the answer is 'no', and Y(n) the next element to use if the answer is 'yes'.

```
11000 NB=53
11010 DIM LE$(NB),C1(NB),C2(NB),N(NB),Y(
NB)
11020 FOR N=1 TO NB
11030 READ LE$(N),C1(N),C2(N),N(N),Y(N)
11040 NEXT N
```

The best way to enter the DATA is probably as 53 separate lines (one for each branch point), as this makes it easy to enter and to edit out any mistakes.

```
14010 DATA ,1,1,2,19
14020 DATA ,1,5,3,10
14030 DATA ,3,2,4,9
14040 DATA ,5,1,5,9
14050 DATA ,3,1,6,7
```

```
14060 DATA " ",,,,
14070 DATA "S",,,,
14080 DATA "J",,,,
14090 DATA "I",,,,
14100 DATA ,5,4,11,14
14110 DATA ,5,5,12,13
14120 DATA "C",,,,
14130 DATA "G",,,,
14140 DATA ,5,7,18,15
14150 DATA ,2,4,17,16
14160 DATA "A",,,,
14170 DATA "Q",,,,
14180 DATA "O",,,,
14190 DATA ,5,1,20,29
14200 DATA ,5,4,21,29
14210 DATA ,5,3,27,22
14220 DATA ,5,7,23,26
14230 DATA ,5,5,24,25
14240 DATA "F",,,,
14250 DATA "B",,,,
14260 DATA "R",,,,
14270 DATA "L",,,,
14280 DATA "D",,,,
14290 DATA ,5,7,45,30
14300 DATA ,2,6,31,44
14310 DATA ,5,3,32,39
14320 DATA ,1,5,33,36
14330 DATA ,3,1,34,35
14340 DATA "X",,,,
14350 DATA "Z",,,,
14360 DATA ,4,2,38,37
14370 DATA "K",,,,
14380 DATA "E",,,,
14390 DATA ,2,4,40,43
14400 DATA ,4,2,42,41
14410 DATA "M",,,,
14420 DATA "N",,,,
14430 DATA "H",,,,
14440 DATA "W",,,,
14450 DATA ,3,1,46,51
14460 DATA ,1,5,47,50
14470 DATA ,2,4,48,49
14480 DATA "Y",,,,
14490 DATA "V",,,,
14500 DATA "U",,,,
14510 DATA ,1,5,52,53
14520 DATA "T",,,,
14530 DATA "F",,,,
```

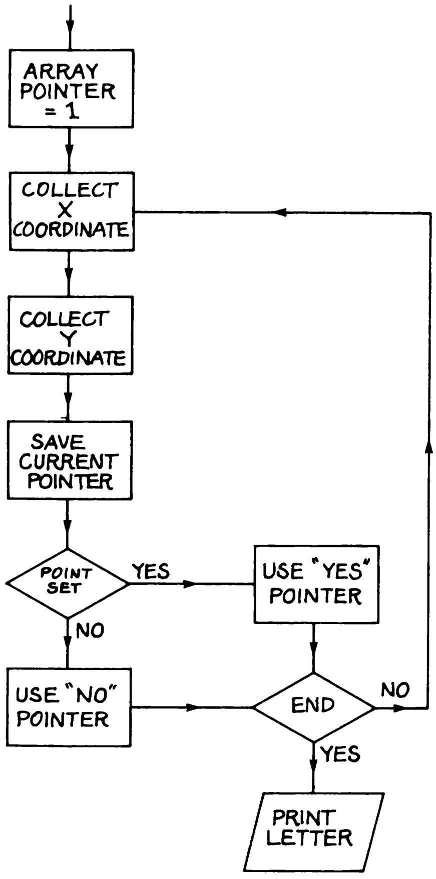
If you are more confident (or are trying to save space) then all the DATA can be condensed on to eight rather unreadable lines which are OK for those who are good at counting commas, but very difficult to edit.

```
14010 DATA ,1,1,2,19,,1,5,3,10,,3,2,4,9,
,5,1,5,8,,3,1,6,7," ",,,,,"S",,,,
14080 DATA "J",,,,,"I",,,,,,5,4,11,14,,5
,5,12,13,"C",,,,,"G",,,,,,5,7,18,15
14150 DATA ,2,4,17,16,"A",,,,,"Q",,,,,,"O
",,,,,,5,1,20,29,,5,4,21,28,,5,3,27,22
14220 DATA ,5,7,23,26,,5,5,24,25,"P",,,,
,"B",,,,,"R",,,,,"L",,,,,"D",,,,
14290 DATA ,5,7,45,30,,2,6,31,44,,5,3,32
,39,,1,5,33,36,,3,1,34;35,"X",,,,
14350 DATA "Z",,,,,,,4,2,38,37,"K",,,,,"E
",,,,,,2,4,40,43,,4,2,42,41,"M",,,,
14420 DATA "N",,,,,"H",,,,,"W",,,,,,3,1,
46,51,,1,5,47,50,,2,4,48,49,"Y",,,,
14490 DATA "V",,;,,"U",,,,,,1,5,52,53,"T
",,,,,"F",,,,
```

To check the design produced against the patterns available (see **Flowchart 8.1**), the array pointer AP is first set to 1 so that the search is started from the beginning. X and Y coordinates are read from the C1 (AP) and C2(AP) elements pointed to, and the last position LP pointer set equal to the current array pointer AP.

The point colour PC at these coordinates is determined by PEEK(TS+CO+X+(Y*40)) AND 15. If this is zero than the point has been set and the 'yes' pointer Y(AP) must be followed. If any other value is found then the 'no' pointer N(AP) is followed. In either case a check is made to see whether the element pointed to contains a zero (indicating the ultimate end of a branch), which shows that a character has been found. If so, the appropriate letter LE$(LP) is printed, and the display is held until a key is pressed, when a new cycle is initiated. As long as a higher value than zero is found then this must be another branch point and so the program loops back to 2010 and picks up the new values of C1(AP) and C2(AP).

To allow you to see which points have been checked, these are set to different colours as they are found. 'Yes' and 'no' branches can be distinguished as tested points which were not set, PC>0, and will now be light



Flowchart 8.1 Character Recognition

green, 3, whilst points which were set will be red (3+1). Any points which were set but not tested will remain black.

```
2000 AP=1
2010 X=C1(AP):Y=C2(AP):LP=AP
2020 PC=PEEK(TS+CO+X+(Y*40)) AND 15
2030 IF PC=0 THEN AP=Y(AP):GOTO 2050
2040 AP=N(AP)
2050 IF AP=0 THEN 2070
2060 POKE TS+CO+X+(Y*40),3+(PC=0):GOTO 2
010
2070 PRINT LE$(LP);
2080 GET A$:IF A$="" THEN 2080
2090 GOSUB 13000:GOTO 20
```

If you want to see which part of the tree was actually followed, then add these modifications which will print out the sequence. The grid is moved down the screen by adding an offset of 481 to SS and a blanking string BL$ defined which is used for partial screen clearance.

```
10000 SS=1024+481
10005 BL$="
      "

2005 PRINT"[HOME]";" AP":PRINT
2055 PRINT AP
2070 PRINT:PRINT "  ";LE$(LP):PRINT
2075 PRINT"PRESS A KEY TO CONTINUE"
2085 PRINT"[HOME]":FOR N=1 TO 10:PRINT BL$:NE
XT N
```

The disadvantage of this more rapid method, of only checking critical points, is that it will make a mistaken match if it encounters a shape that is not on the tree, whereas if all points are checked then no match will be found in such a case.

Early Optical Character Readers would only accept a single particular typeface, but the latest machines not only accept different styles of type, but actually learn the recognition rules for themselves by means of a built-in expert system. You teach these by showing them a few pages of text and then entering these same characters via the keyboard. However we feel that it will still be a long time before anyone can produce a machine that can read OUR handwriting!

CHAPTER 9
An Intelligent Teacher

Another place where artificial intelligence can be particularly useful is in teaching programs. It is all very well having a program which tests a student's knowledge at random, but this is not how a real human teacher works. As well as asking the questions, he keeps an eye on the progress of the students, increases the difficulty of the questions as experience increases, and tests them more rigorously on the types of problems with which they are having difficulties. For example, if a child takes a test involving addition, subtraction, multiplication, and division, but only gets the division-type questions wrong, then it follows that the child should be given more division questions in the future to provide more practice.

Let's have a look at how we can introduce these 'human' qualities into a teaching program.

Questions and answers

We need to create random numbers to be used in the first question, which we will make addition. Using INT(RND(1)*10) will give numbers between 0 and 9.

```
20 A%=INT(RND(1)*10)
30 B%=INT(RND(1)*10)
```

The computer adds these together and then goes on to an input and checking subroutine at 1000.

```
40 C%=A%+B%:GOSUB 1000
```

First, the routine must print the question and input your answer IP%.

```
1000 PRINT A%;"+";B%;"=";
1010 INPUT IP%
```

Your answer must then be checked. If the program answer C% is the same as your answer, then CORRECT is printed and the routine returns to line 40. Otherwise WRONG is printed followed by the correct answer.

```
1020 IF C%=IP% THEN PRINT "CORRECT":RETU
RN
1030 PRINT "WRONG, THE CORRECT ANSWER WA
S ";C%
1040 RETURN
```

The other three subjects (subtraction, multiplication, and division) can be easily dealt with in the same way if we replace the ' + ' sign in line 1000 by a sign string SG$, which we can set to the appropriate character at the time. As INT(RND(1)*10) is common to all the calculations, we might as well define this as a function RD.

```
15 DEF FNRD(X)=RND(1)*10
20 A%=FNRD(AD%)
30 B%=FNRD(AD%)
40 SG$="+":C%=A%+B%:GOSUB 1000
50 A%=FNRD(SU%)
60 B%=FNRD(SU%)
70 SG$="-":C%=A%-B%:GOSUB 1000
80 A%=FNRD(MU%)
90 B%=FNRD(MU%)
100 SG$="*":C%=A%*B%:GOSUB 1000
110 A%=FNRD(DI%)
120 B%=FNRD(DI%)
130 SG$="/":C%=A%/B%:GOSUB 1000
1000 PRINT A%;SG$;B%;"=";
```

Finally we jump back to line 20 to ask more questions.

```
140 GOTO 20
```

Dividing by zero!

As it stands, the program can crash if B% happens to be zero when a division is selected. This can be simply fixed by always adding one on to B% in this case:

```
120 B%=FNRD(DI%)+1
```

Deleting decimals

We are using integer variables to keep us to round numbers, but of course a division may still produce a fractional answer, which you cannot enter correctly: IP% will be rounded down, eg:

3/2 = 1.5

The program will accept 1, 1.5, 1.9 or any other number between 1 and 1.999 ... as correct.

To avoid producing decimals, A% needs to be a multiple of B%. To achieve this we calculate B% first and make A% equal to B% multiplied by a random number between 0 and 10.

```
110 B%=FNRD(DI%)+1
120 A%=INT(FNRD(DI%))*B%
```

Keeping a score

Now that we have the test itself working, we need to consider how to keep a score. The simplest thing is to increment a tries variable TR% each time the subroutine at 1000 is used, and to increment a score variable SC% each time a correct answer is obtained.

```
1010 INPUT IP%:TR%=TR%+1
1020 IF C%=IP% THEN:PRINT "CORRECT":SC%=
SC%+1:GOTO 1040
1040 PRINT "YOUR SCORE IS ";SC%;"/";TR%:
RETURN
```

If you prefer the score as a percentage then amend line 1040 as follows:

```
1040 PRINT "YOU HAVE HAD ";INT((SC%/TR%)
*100);"% CORRECT":RETURN
```

How many questions?

As it stands the program will ask one question of each type in sequence, ad infinitum. We can limit the number by defining the number of questions NQ% as a variable.

```
10 NQ%=32
```

Each time a question is asked, NQ% is decreased by 1, and when NQ%=0 the test ends (after eight questions of each type have been answered).

```
150 IF NQ%>0 THEN 20
160 END
1010 INPUT IP%:TR%=TR%+1:NQ%=NQ%-1
```

Shifting the emphasis

If we are going to bias the questions in favour of areas of difficulty, then we need to keep a record of performance in each individual area. We therefore need separate variables for each type of question (AD% for addition, SU% for subtraction, MU% for multiplication, and DI% for division). These variables are defined in terms of one eighth of the total number of questions to be asked NQ%.

```
10 NQ%=32:AD%=NQ%/8:SU%=AD%:MU%=AD%:DI%=
AD%
```

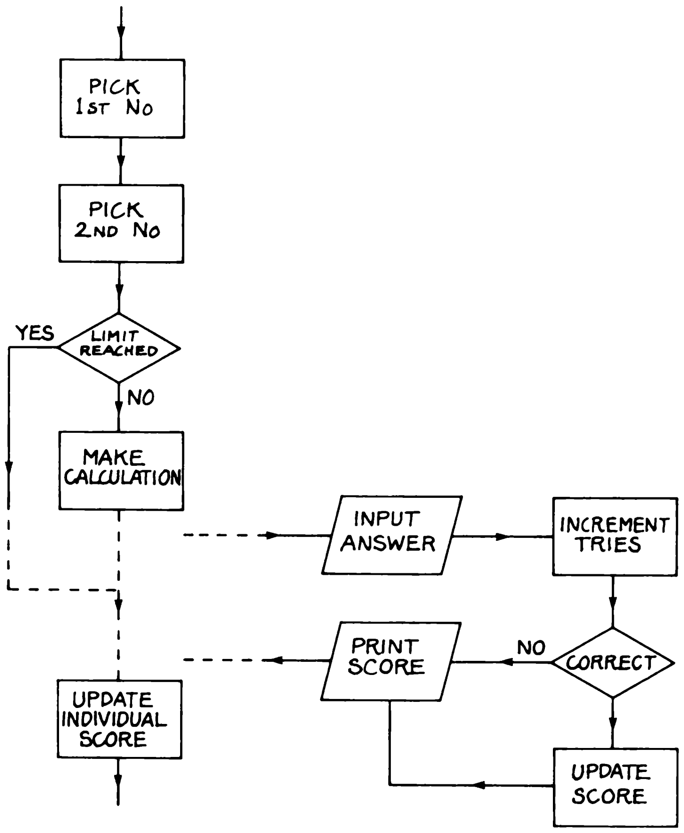
Now if the correct answer C% is the same as your answer IP% then an increment variable IN% is set to −1, CORRECT is printed, and the routine returns. Otherwise IN% is set to 1, and WRONG is printed followed by the correct answer.

```
1020 IF C%=IP% THEN IN%=-1:PRINT "CORREC
T":RETURN
1030 IN%=1:PRINT "WRONG, THE CORRECT ANS
WER WAS ";C%
1040 RETURN
```

IN% is added to the appropriate individual number of questions variable AD%, SU%, MU% or DI% on returning, producing an increase in this value if the answer was wrong, or a decrease if the answer was right.

```
40 SG$="+":C%=A%+B%:GOSUB 1000:AD%=AD%+I
N%
70 SG$="-":C%=A%-B%:GOSUB 1000:SU%=SU%+I
N%
100 SG$="*":C%=A%*B%:GOSUB 1000:MU%=MU%+
IN%
130 SG$="/":C%=A%/B%:GOSUB 1000:DI%=DI%+
IN%
```

Now we add a check to see whether all the questions of a particular type have not been correctly answered (eg AD%>0, see **Flowchart 9.1**) If all questions of one type have been correctly answered, then no more of this type will be asked as the line is jumped over. If the appropriate number of



Flowchart 9.1 Intelligent Teacher

each type has been answered correctly (AD%=0, SU%=0, MU%=0 DI%=0) then the program ends.

```
40 IF AD%>0 THEN SG$="+";C%=A%+B%:GOSUB
1000:AD%=AD%+IN%
70 IF SU%>0 THEN SG$="-":C%=A%-B%:GOSUB
 1000:SU%=SU%+IN%
100 IF MU%>0 THEN SG$="*":C%=A%*B%:GOSU
B 1000:MU%=MU%+IN%
130 IF DI%>0 THEN SG$="/":C%=A%/B%:GOSU
B 1000:DI%=DI%+IN%
140 IF AD%=0 AND SU%=0 AND MU%=0 AND DI%
=0 THEN 160
```

Notice that you are no longer asked questions about areas in which you have correctly answered four questions without making any errors. If you make a mistake then AD%, etc, will be increased and so you will have to answer more than four correctly before AD reaches zero.

Degrees of difficulty

How about making the questions easier or harder according to how well you are doing (ie the values of AD%, SU%, MU%, and DI%)? So far the current values for A% and B% have always been between 0 and 9 as they were produced by RND(1)*10, but we now need to bias the numbers produced for the questions towards higher values, if you are correct, and lower values, if you are incorrect. At the same time, we must ensure that you do not produce negative values if your performance is abysmal.

The 'worst case' will be if you get all the questions right in three of the groups, and all the questions wrong in the last group. In this case only four questions will be asked on the first three groups, leaving 32–(3*4)=20 questions to be asked on the last group. In addition we must remember that X (eg AD%) starts at a value of 4, so that the maximum value of X which could be obtained is 20+4=24.

We therefore set up a weighting variable WT%, which is calculated by subtracting three times the number of questions to be asked in each group (3*AD%) from the total number of questions NQ% and adding back on the number of questions in a group AD% at the start.

WT%=NQ%–(3*AD%)+AD%

This is more simply expressed as:

WT%=NQ%–(2*AD%)

```
10 NQ%=32:AD%=NQ%/8:SU%=AD%:MU%=AD%:DI%=
AD%:WT%=NQ%-(2*AD%)
```

We now replace the fixed value of ten by the difference betwen WT% and X.

```
15 DEF FNRD(X)=RND(1)*(WT%-X)
```

To begin with, WT%=24 and X=4 so numbers between 0 and 19 will be selected. If a correct answer is given, then X will be reduced to 3 and numbers between 0 and 20 will be chosen. After four correct answers, X will not change (for this type of question) as it will have reached zero and the line will be skipped. The last values will therefore be between 0 and 22.

On the other hand if the first answer is incorrect then X will increase by 1 and the range of numbers produced reduced by 1 (0—18). In the 'worst case' X will be increased twenty times to 24 and (WT%–X) will fall to zero for both A% and B% (so you should be able to solve that particular problem!).

CHAPTER 10
Putting It All Together

In the previous chapters we have dealt, from first principles, with various aspects of Artificial Intelligence. In this final chapter we have linked together many of these individual ideas in a single complete program.

The original 'intelligent' program was the famous 'ELIZA', which was a pseudo-psychiatrist program written to send up a particular style of psychiatric therapy. We have resisted the temptation to follow this lead and have opted instead to produce a replacement for the average computer salesman. This program combines some ideas on the processing of natural language and on expert systems, to produce a result which should both understand your requests and make suggestions which take into account both your requirements and a number of hard commercial facts.

Enough words and values have already been included to make the program interesting, but you can easily customise it by adding your own ideas to the DATA. (We take no responsibility for the values included so far, which are for demonstration purposes only, or for the views on particular machines expressed by the program!) The program itself is quite complex but it follows the methods described earlier in the book and the functions of the various line variables and arrays are given in **Table 10.1**

Making conversation

The format of the program is that you are asked for your views on each of a number of possible features in turn (the exact wording of the question being selected at random from a selection of phrases). Note that the key word or phrase is inserted into the sentence where necessary, and that the correct conjugation is applied.

Your input is examined in detail for keywords, and a rule array updated according to your requests. (If you want actually to watch the rule array being updated then delete line 5490.) Many of the keywords are truncated so that one check can be made for a number of similar words, and a test is included to see if the matching string is at the start of a word.

The simplest answer is 'YES' or 'NO', which adds or subtracts 1 from the rule for that feature. If you mention the name of the feature (eg 'GRAPHICS') then a further 1 is added to the rule. In addition, using a

Table 10.1 Main Variables in 'Salesman'

SIMPLE VARIABLES

IS	INSTR start
I1$	target string
I2$	search string
IP	INSTR pointer
QP	no. of question sentences
Q	no. of questions
R	no. of rules
BB	bank balance
PH	phrase number
PH$	phrase words
M	match marker
OF	object flag
OM	object match
LD	like/dislike
FS	rest of sentence pointer
NP	negative pointer
S1	AND match pointer
S2	BUT match pointer
RU	rule update marker
OB	no. of objects
AJ	no. of adjectives
AV	no. of adverbs
LI	no. of likes
DL	no. of dislikes
NJ	no. of negative adjectives
NV	no. of negative adverbs
HM	no. of cheap/expensive
CO	no. of computers
FE	no. of features
CT	no. of cost ratings
CS	no. of cost suggestions
EX	no. of excuses
HI	no. of high price suggestions
LO	no. of low price suggestions
TC	total cost
TP	total profit

ARRAYS

OB$(OB)	objects
AJ$(AJ)	adjectives
NJ$(NJ)	negative addresses
AV$(AV)	adverbs
NV$(NV)	negative adverbs
LI$(LI)	likes
DL$(DL)	dislikes
Q$(Q)	question objects
QP$(QP)	question sentences
CR(Q)	cost rate
PR(Q)	profit rate
IC(Q)	total cost
IP(Q)	total profit
HM$(HM)	cheap/expensive
R(R)	rules
CO$(FE)	computer names
FE(CO,FE)	feature names
C(CT)	cost ratings
CS$(CS)	cost suggestions
EX$(EX)	excuses
HI$(HI)	high messages
LO$(LO)	low messages

'positive' adjective or adverb adds to the rule, whilst a 'negative' adjective or adverb subtracts from the rule. Separating the words into different classes allows you to make more than one change to the rule at the same time.

Thus:

YES	adds one
YES BASIC	adds two
YES BASIC NECESSARY	adds three
YES GOOD BASICNECESSARY	adds four

Whilst:

NO	subtracts one
NO MEMORY	subtracts two

Furthermore, verbs are grouped as 'likes' and 'dislikes', the last of which reverses the action of the rest of the words.
Thus:

I DETEST MACRODRIVES subtracts one

Both 'NO-' and 'N'T' are recognised, and most double negatives are interpreted correctly.
Thus:

I DON'T LIKE SOUND subtracts two

I DON'T DISLIKE SOUND adds one

If anything appears at the start of a sentence and is followed by a comma, it is usually cut off and effectively ignored.
Thus:

NO, I DON'T WANT GOOD SOUND subtracts three

The exception is when 'AND' or 'BUT' are included, when both parts of the sentence are acted on independently.
Thus if the question is:

DO YOU WANT GRAPHICS?

and the answer is:

NO, BUT I WANT GOOD SOUND

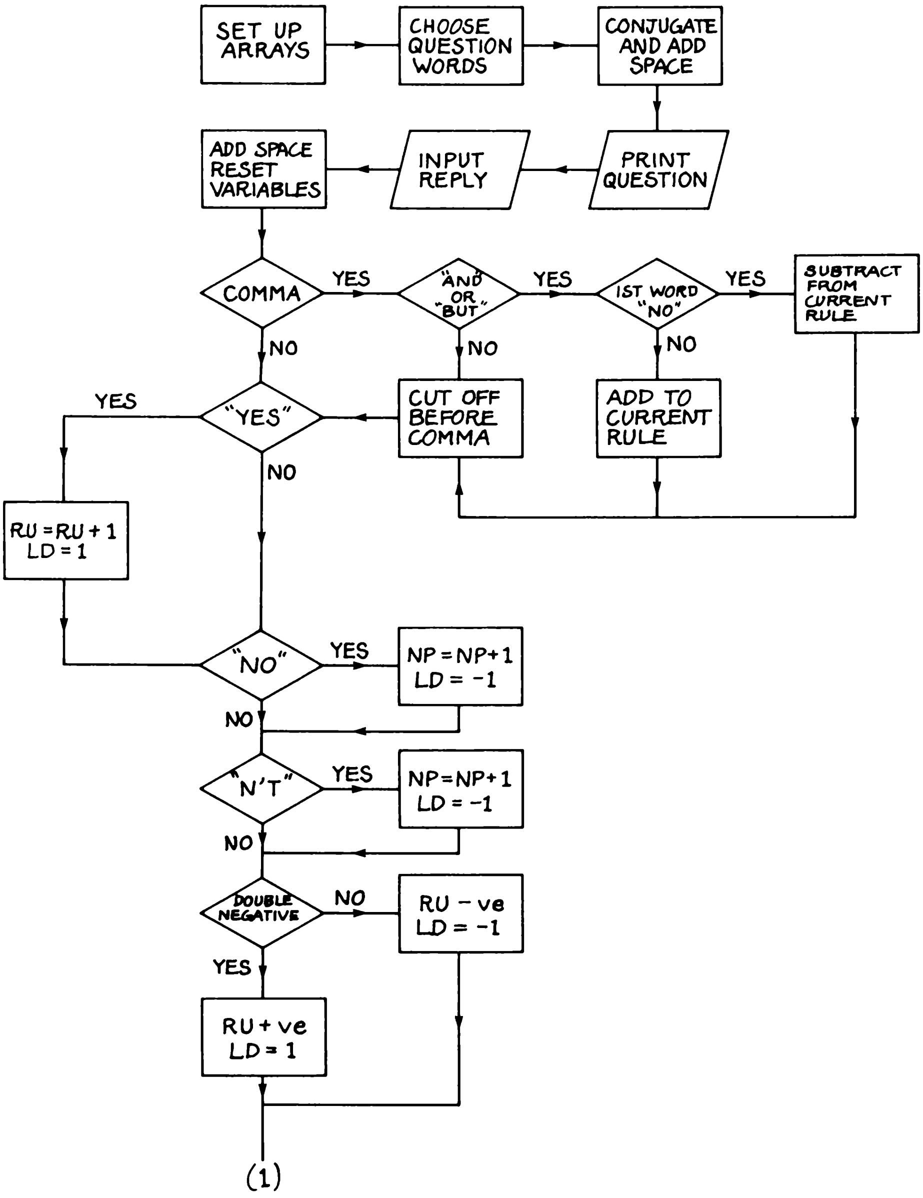
then one is subtracted from the graphics rule and two is added to the sound rule.
If the program does not find any keywords in the input, it politely asks you to try again:

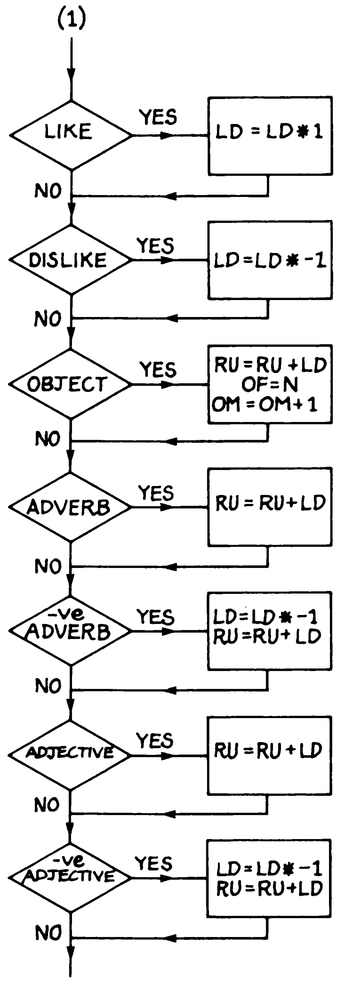
PARDON, EXCUSE ME BUT...

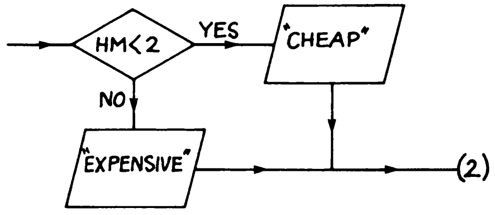
The program can only cope with one feature at a time, so if you try to ask for 'SOUND and GRAPHICS' at the same time, for example, you will get a request for a repeat of the question.

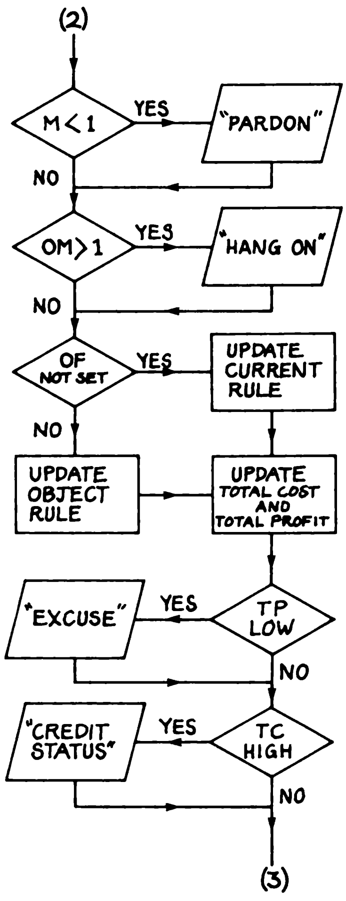
HANG ON — ONE THING AT A TIME

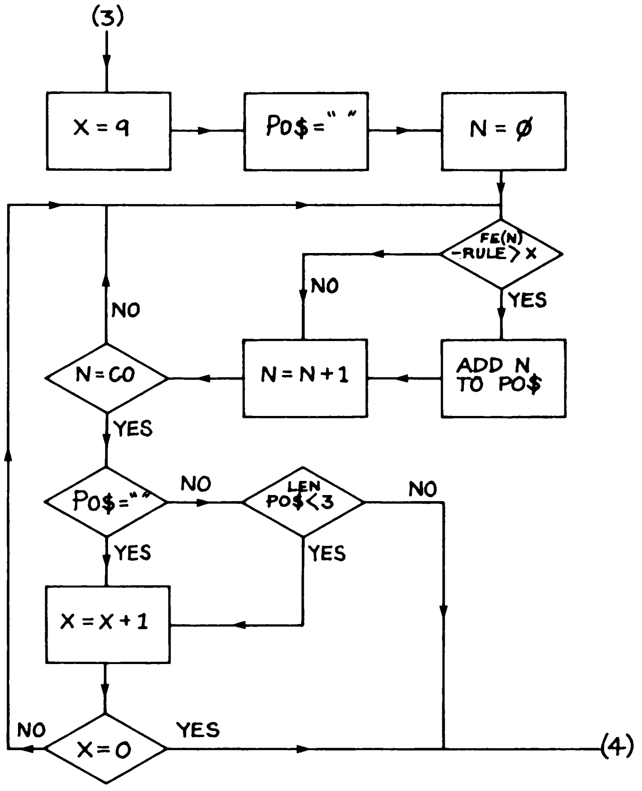
However, it is possible to make comments about single features that you are not being asked about at the time, and these entries will still update the rules (as in the 'BUT' example above).

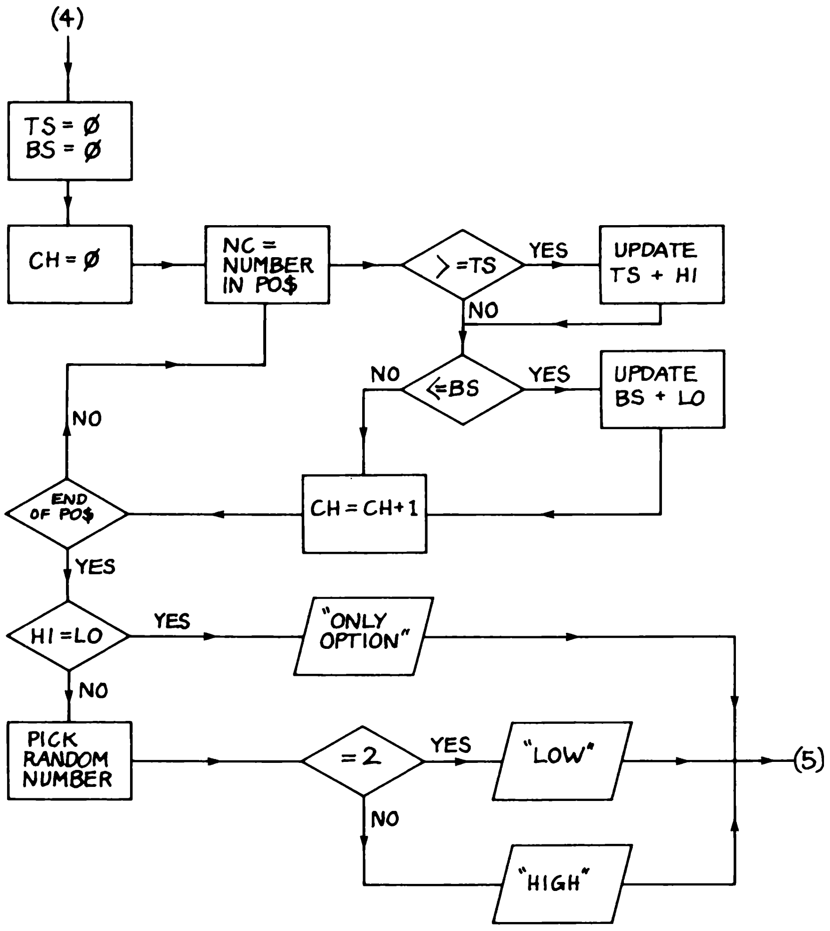

SET UP ARRAYS
CHOOSE QUESTION WORDS
CONJUGATE AND ADD SPACE
PRINT QUESTION
INPUT REPLY
ADD SPACE RESET VARIABLES
COMMA
YES
"AND" OR "BUT"
YES
1ST WORD "NO"
YES
SUBTRACT FROM CURRENT RULE
NO
NO
NO
ADD TO CURRENT RULE
CUT OFF BEFORE COMMA
YES
"YES"
NO
RU = RU + 1
LD = 1
"NO"
YES
NP = NP + 1
LD = -1
NO
"N'T"
YES
NP = NP + 1
LD = -1
NO
DOUBLE NEGATIVE
NO
RU - ve
LD = -1
YES
RU + ve
LD = 1
(1)


(1)
LIKE
YES
LD = LD*1
NO
DISLIKE
YES
LD=LD*-1
NO
OBJECT
YES
RU=RU+LD
OF=N
OM=OM+1
NO
ADVERB
YES
RU = RU+LD
NO
-ve ADVERB
YES
LD=LD*-1
RU=RU+LD
NO
ADJECTIVE
YES
RU=RU+LD
NO
-ve ADJECTIVE
YES
LD=LD*-1
RU=RU+LD
NO


HM<2
YES
"CHEAP"
NO
"EXPENSIVE"
(2)


(2)
M<1
YES
"PARDON"
NO
OM>1
YES
"HANG ON"
NO
OF NOT SET
YES
UPDATE CURRENT RULE
NO
UPDATE OBJECT RULE
UPDATE TOTAL COST AND TOTAL PROFIT
"EXCUSE"
YES
TP LOW
NO
"CREDIT STATUS"
YES
TC HIGH
NO
(3)


(3)
X = 9
PO$ = " "
N = Ø
FE(N) -RULE > X
NO
YES
ADD N TO PO$
N = N + 1
N = CO
NO
YES
PO$ = " "
NO
LEN PO$ < 3
NO
YES
YES
X = X + 1
X = 0
NO
YES
(4)


(4)
TS = Ø
BS = Ø
CH = Ø
NC = NUMBER IN PO$
> =TS
YES
UPDATE TS + HI
NO
<=BS
NO
YES
UPDATE BS + LO
NO
END OF PO$
CH = CH+1
YES
HI =LO
YES
"ONLY OPTION"
NO
PICK RANDOM NUMBER
=2
YES
"LOW"
(5)
NO
"HIGH"

Decisions

In addition to the rule array, there are two other arrays which are linked to this. The first is the 'cost array', which gives an indication of the cost of this particular option, and the second is the 'profit array' which indicates to the salesman how much effort it is worth putting into selling this feature. The values for these last two arrays are produced by multiplying the content of the corresponding rule array element by factors entered originally as DATA in lines 10100, etc, where the format is:

(phrase describing feature, cost, profit)

After each input, the salesman considers the consequences of your requests. First of all he looks to see if the sum total of the cost of all your requirements exceeds your bank balance. If so, he prints out one of a series of caustic comments on your credit-worthiness like:

THIS SPECIFICATION SEEMS TO BE EXCEEDING YOUR CREDIT LIMIT

He also looks at how much profit he is likely to make on the sale so far: if this drops too low, he will start to lose interest and come up with comments like:

I HAVE AN URGENT APPOINTMENT

or

WE CLOSE IN FIVE MINUTES

At the same time, he will be more helpful with regard to which of the available computers will fit your requirements, drawing up a short-list by comparing the rating given originally to this feature in the description of each computer with the value you put on it. The format for the descriptions is:

(name, value of feature 1, value of feature 2, value of feature 3, etc)

The highest rated machine will always be picked out first but, if possible, at least three machines (possibly with lower ratings) will be selected and the final choice is made from these. Either the highest or lowest cost computer (at random) will be selected for mention, for example:

IF YOU WANT A REAL ROLLS-ROYCE THEN JUST LOOK AT THE...

and

IF YOU ARE IN THE BUDGET MARKET THEN WHAT ABOUT THE...

If only one machine fits the bill, the program will come up with:

YOUR ONLY OPTION IS THE...

Salesman

```
10 GOSUB 9300
20 GOTO 200
100 FOR IS=FT TO LEN(I1$)
110 IF MID$(I1$,IS,LEN(I2$))=I2$ THEN IP
=IS:RETURN
120 NEXT IS
130 IP=0:RETURN
200 PH=RND((1)*(QP+1)):PH$=QP$(PH)
210 I1$=PH$:I2$="/":FT=1:GOSUB 100:SP=IP
220 IF SP=0 THEN 300
230 IF LEFT$(Q$(Q),1)="@" THEN PH$=LEFT$
(PH$,SP-1)+"ARE"+RIGHT$(PH$,LEN(PH$)-SP)
300 IF SP=0 THEN 400
310 IF LEFT$(Q$(Q),1)="&" THEN PH$=LEFT$
(PH$,SP-1)+"IS"+RIGHT$(PH$,LEN(PH$)-SP)
400 I1$=PH$:I2$="*":FT=1:GOSUB 100
410 IF SP=0 THEN 440
420 PH$=LEFT$(PH$,SP-1)+" "+RIGHT$(Q$(Q)
,LEN(Q$(Q))-1)+RIGHT$(PH$,LEN(PH$)-SP)
430 GOTO 500
440 PH$=PH$+" "+RIGHT$(Q$(Q),LEN(Q$(Q))-
1)
450 PRINT:PRINT
500 PRINT PH$;"?"
600 PRINT
700 IN$=" "
710 GET I$:PRINT "<[LEFT CURSOR]";
720 IF I$="" THEN 710
730 IF I$=CHR$(13) THEN 800
740 IN$=IN$+I$
750 PRINT I$;
760 GOTO 710
```

```
800 LD=1:OF=-1:FS=1:NP=0:RU=0:M=0:OM=0:S
1=0:S2=0
900 I1$=IN$:I2$=",":FT=1:GOSUB 100:CM=IP
910 IF CM=0 THEN 1600
1000 I1$=IN$:I2$="AND":FT=1:GOSUB 100:S1
=IP
1010 I2$=IN$:I2$="BUT":FT=1:GOSUB 100:S2
=IP
1200 IF S1+S2=0 THEN 1500
1300 IF LEFT$(IN$,3)<>" NO" THEN 1400
1310 R(Q)=R(Q)-1:IC(Q)=IC(Q)-CR(Q):IP(Q)
=IP(Q)-PR(Q):GOTO 1500
1400 R(Q)=R(Q)+1:IC(Q)=IC(Q)+CR(Q):IP(Q)
=IP(Q)+PR(Q)
1500 IN$=RIGHT$(IN$,LEN(IN$)-CM)
1600 I1$=IN$:I2$="YES":FT=FS:GOSUB 100:S
P=IP
1700 IF SP>0 THEN RU=RU+1:LD=1:M=1:FS=SP
+1:GOTO 1600
1800 I1$=IN$:I2$="NO":FT=FS:GOSUB 100:SP
=IP
1900 IF SP>0 THEN LD=-1:M=1:FS=SP+1:NP=N
P+1:GOTO 1800
2000 I1$=IN$:I2$="N'T":FT=FS:GOSUB 100:S
P=IP
2100 IF SP>0 THEN LD=-1:M=1:FS=SP+1:NP=N
P+1:GOTO 2000
2200 IF NP=0 THEN 2300
2210 IF INT(NP/2)=NP/2 THEN RU=RU+1:LD=1
:GOTO 2300
2250 RU=RU-1:LD=-1
2300 FOR N=0 TO LI
2400 I1$=IN$:I2$=LI$(N):FT=1:GOSUB 100:S
P=IP
2410 IF SP=0 THEN 2500
2420 IF MID$(IN$,SP-1,1)=" " THEN LD=LD*
1:M=1
2500 NEXT N
2600 FOR N=0 TO DL
2700 I1$=IN$:I2$=DL$(N):FT=1:GOSUB 100:S
P=IP
2710 IF SP>0 THEN IF MID$(IN$,SP-1,1)="
" THEN LD=LD*-1:M=1
2900 NEXT N
```

```
2900 FOR N=0 TO OB
3000 I1$=IN$:I2$=OB$(N):FT=1:GOSUB 100:S
P=IP
3010 IF SP>0 THEN IF MID$(IN$,SP-1,1)="
" THEN RU=RU+LD:OF=N:M=1:OM=OM+1
3100 NEXT N
3200 FOR N=0 TO AV
3300 I1$=IN$:I2$=AV$(N):FT=1:GOSUB 100:S
P=IP
3310 IF SP=0 THEN 3600
3400 IF MID$(IN$,SP-1,1)<>" " THEN 3600
3500 RU=RU+LD:M=1
3600 NEXT N
3700 FOR N=0 TO NV
3800 I1$=IN$:I2$=NV$(N):FT=1:GOSUB 100:S
P=IP
3810 IF SP=0 THEN 4100
3900 IF MID$(IN$,SP-1,1)<>" " THEN 4100
4000 LD=LD*-1:RU=RU+LD:M=1
4100 NEXT N
4200 FOR N=0 TO AJ
4300 I1$=IN$:I2$=AJ$(N):FT=1:GOSUB 100:S
P=IP
4310 IF SP=0 THEN 4600
4400 IF MID$(IN$,SP-1,1)<>" " THEN 4600
4500 RU=RU+LD:M=1
4600 NEXT N
4700 FOR N=0 TO NJ
4800 I1$=IN$:I2$=NJ$(N):FT=1:GOSUB 100:S
P=IP
4910 IF SP=0 THEN 5100
4900 IF MID$(IN$,SP-1,1)<>" "THEN 5100
5000 LD=LD*-1:RU=RU+LD:M=1
5100 NEXT N
5110 FOR N=0 TO HM
5120 I1$=IN$:I2$=HM$(N):FT=1:GOSUB 100:S
P=IP
5130 IF SP=0 THEN 5190
5140 IF MID$(IN$,SP-1)<>" " THEN 5190
5150 XX=N:IF XX<2 THEN PRINT"CHEAP AND N
ASTY":GOTO 5190
5160 IF XX>=2 THEN PRINT"RATHER EXPENSIV
E"
5170 NEXT N
```

```
5180 PRINT
5200 IF M<1 THEN PRINT "PARDON, PLEASE E
XCUSE ME BUT":GOTO 200
5300 IF OM>1 THEN PRINT "HANG ON - ONE T
HING AT A TIME":GOTO 200
5400 IF OF=-1 THEN 5440
5410 R(OF)=R(OF)+RU:IC(OF)=IC(OF)+(CR(OF
)*RU)
5420 IP(OF)=IP(OF)+(PR(OF)*RU)
5430 GOTO 5490
5440 R(Q)=R(Q)+RU:IC(Q)=IC(Q)+(CR(Q)*RU)
:IP(Q)=IP(Q)+(PR(Q)*RU)
5490 GOTO5900
5500 PRINT"[CLR]"
5600 FOR N=0 TO R:PRINT R(N);:NEXT N:PRI
NT
5700 FOR N=0 TO R:PRINT IC(N);:NEXT N:PR
INT
5800 FOR N=0 TO R:PRINT IP(N);:NEXT N:PR
INT
5900 FOR N=0 TO OB
6000 TC=TC+IC(N)
6100 TP=TP+IP(N)
6200 NEXT N
6300 IF TP<Q*5 THEN TX=RND(0)*EX:PRINT:P
RINT EX$(TX)
6400 IF TC>BB THEN PT=RND(0)*CS:PRINT:PR
INT CS$(PT)
6500 TC=0:TP=0
6700 FOR X=9 TO 0 STEP-1:PO$=""
6800 FOR N=0 TO CO
6900 IF FE(N,Q)-R(Q)>X THEN PO$=PO$+RIGH
T$(STR$(N),1):M=N
7000 NEXT N
7100 IF PO$="" THEN NEXT X:GOTO 7200
7110 IF LEN(PO$)<3 THEN NEXT X
7310 GOTO7900
7350 PRINT PO$
7400 IF PO$="" THEN 9200
7500 FOR N=1 TO LEN(PO$)
7600 PRINT CO$(VAL(MID$(PO$,N,1)))
7700 NEXT N
7800 PRINT
7900 TS=0:BS=10
```

```
8000 FOR CH=0 TO LEN(PO$)-1
8100 NC=VAL(MID$(PO$,CH+1,1))
8200 IF C(NC)>=TS THEN TS=C(NC):HI=NC
8300 IF C(NC)<=BS THEN BS=C(NC):LO=NC
8400 NEXT CH
8410 IF HI=LO THEN PRINT"YOUR ONLY OPTIO
N IS THE":PRINT CO$(HI):GOTO 9200
8500 HI$=CO$(HI):LO$=CO$(LO)
8600 SE=RND(1)+1
8700 SL=RND(1)*2
8800 IF SE=2 THEN 9100
8900 PRINT HI$(SL),,,,HI$
9000 GOTO 9200
9100 PRINT LO$(SL),,,,LO$
9200 Q=Q+1:IF Q<20 THEN 200 ELSE END
9300 QP=5:Q=19:R=0:OB=R:AJ=8:AV=5:LI=3:D
L=3:NJ=8:NV=2:HM=3:BB=100
9310 DIM OB$(OB),AJ$(AJ),NJ$(NJ),AV$(AV)
,NV$(NV),LI$(LI),DL$(DL),Q$(Q)
9320 DIM R(R),QP$(QP),CR(Q),PR(Q),IC(Q),
IP(Q),HM$(HM)
9400 DATA BASIC,GRAPHIC,SOUND,KEYBOARD,F
UNCTION,MEMORY,TAPE,MACRODRIVE,DISC
9410 DATA SOFTWARE,CARTRIDGE,JOYSTICK,AS
SEMBL,CENTRONIC,RS232,EXPAND
9420 DATA NETWORK,16-BIT,MULTITASK,SERVI
CE
9500 DATA GOOD,EXCEL,SUPER,MAGNIF,FIRST,
FAST,EFFIC,ESSENT,LOT
9600 DATA BAD,RUBBISH,POOR,SLOW,INEFFIC,
FEW,WORS,LEAST,LESS
9700 DATA REAL,VERY,OFTEN,FREQ,NECESS,TR
U
9800 DATA NEVER,UNNECESS,INFREQ
9900 DATA WANT,LIKE,NEED,REQUIRE
10000 DATA HATE,DISLIKE,LOATHE,DETEST
10100 DATA &GOOD BASIC,5,2,@GRAPHICS,7,2
,&SOUND,6,2,&A GOOD KEYBOARD,4,2
10110 DATA @FUNCTION KEYS,1,5,&A LARGE M
EMORY,3,6,&A TAPE INTERFACE,2,2
10120 DATA @MACRODRIVES,2,4,@DISCS,5,8,&
EXTENSIVE SOFTWARE,0,9
10130 DATA &A CARTRIDGE PORT,1,6
10200 DATA &A JOYSTICK PORT,1,7,&AN ASSE
```

```
MBLER,2,1,&A CENTRONICS PORT,2,5
10210 DATA &AN RS232 PORT,2,6,&EXPANDABI
LTY,2,9,&NETWORKING,3,4
10220 DATA &A 16-BIT CPU,1,7,&MULTITASKI
NG,5,5,&GOOD SERVICE,1,9
10300 DATA WOULD YOU LIKE,WHAT ABOUT,HOW
 ABOUT,DO YOU WANT,DO YOU REQUIRE
10310 DATA /* IMPORTANT
10320 DATA CHEAP,INEXPENSIVE
10330 DATA DEAR,EXPENSIVE
10400 FOR N=0 TO OB:READ OB$(N):NEXT N
10500 FOR N=0 TO AJ:READ AJ$(N):NEXT N
10600 FOR N=0 TO NJ:READ NJ$(N):NEXT N
10700 FOR N=0 TO AV:READ AV$(N):NEXT N
10800 FOR N=0 TO NV:READ NV$(N):NEXT N
10900 FOR N=0 TO LI:READ LI$(N):NEXT N
11000 FOR N=0 TO DL:READ DL$(N):NEXT N
11100 FOR N=0 TO Q:READ Q$(N),CR(N),PR(N
):NEXT N
11200 FOR N=0 TO QP:READ QP$(N):NEXT N
11210 FOR N=0 TO HM:READ HM$(N):NEXT N
11300 PRINT"[CLR]":Q=0
11400 PRINT "IT IS MY PLEASURE TO WELCOM
E YOU TO THE MULTIMEGA MICROSTORE"
11410 PRINT
11500 PRINT"WE ARE UNDOUBTEDLY THE ULTIM
ATE SOURCE  OF ALL COMPUTER PRODUCTS"
11505 PRINT
11510 PRINT"I SHALL HAVE PLEASURE IN HEL
PING          YOU SELECT YOUR NEW MACHINE"
11515 PRINT
11600 PRINT"SO THAT I CAN WORK OUT THE B
EST COMPUTER"
11610 PRINT"FOR YOUR PARTICULAR NEEDS PE
RHAPS"
11620 PRINT"YOU WOULD BE KIND ENOUGH TO
ANSWER A FEW QUESTIONS"
11650 PRINT
11700 PRINT:PRINT"ARE YOU READY";
11900 CO=9:FE=19:CT=9:DIM CO$(FE),FE(CO,
FE),DF(CO,FE),C(CT)
11900 DATA JCN PC,7,8,8,9,8,8,8,0,9,9,7,
7,0,7,6,8,8,9,9,9
12000 DATA KNACT SERIOUS,6,7,6,8,8,9,8,0
```

```
,8,8,0,0,0,7,6,8,8,9,9,7
12100 DATA CLEARSIN MT,9,9,9,7,7,8,8,9,9
,6,7,7,0,7,6,7,9,9,9,1
12200 DATA ACHRON ILLUSION,8,7,6,6,0,3,7
,0,5,5,0,0,6,0,0,4,1,0,0,2
12300 DATA BANANA IIE,3,5,2,5,0,4,6,0,3,
0,3,5,0,0,6,7,0,0,0,4
12400 DATA SI ELITE,9,8,9,7,7,9,8,0,7,2,
7,4,0,0,6,0,0,0,0,0
12500 DATA COLECTOVISION CABBAGE,5,5,5,5
,2,5,5,5,5,1,7,7,0,0,6,5,0,9,0,0
12600 DATA CANDY COLOURED COMPUTER,7,6,4
,2,0,2,7,0,4,9,8,7,0,0,6,3,0,0,0,6
12700 DATA COMANDEAR 64,2,8,9,7,7,6,5,0,
6,9,6,7,0,0,2,2,0,0,0,6
12800 DATA ATRIA-600GT,1,8,8,5,0,2,5,0,7
,7,7,7,0,0,6,6,0,0,0,5
12900 DATA 10,9,7,3,8,4,6,5,2,1
13000 FOR N=0 TO CO
13100 READ CO$(N)
13200 FOR M=0 TO FE
13300 READ FE(N,M)
13400 NEXT M,N
13500 FOR N=0 TO CT
13600 READ C(N)
13700 NEXT N
13800 GET A$:IF A$="" THEN 13800
13900 DATA I THINK YOU ARE GETTING OUT O
F YOUR      PRICE RANGE
13910 DATA THIS SPECIFICATION SEEMS TO B
E EXCEEDINGYOUR CREDIT LIMIT
13920 DATA I DON'T THINK THAT YOU CAN AF
FORD SUCH  LUXURIES
14000 DATA EXCUSE ME I CAN HEAR THE PHON
E RINGING,I HAVE AN URGENT APPOINTMENT
14010 DATA WE CLOSE IN FIVE MINUTES
14100 CS=2:EX=2:DIM CS$(CS):DIM EX$(EX)
14200 FOR N=0 TO CS:READ CS$(N):NEXT N
14300 FOR N=0 TO EX:READ EX$(N):NEXT N
14400 DATA IF YOU ARE IN THE BUDGET MARK
ET THEN     WHAT ABOUT THE
14410 DATA AN INEXPENSIVE CHOICE IS THE,
YOU GET GOOD VALUE FOR MONEY WITH THE
14500 DATA IF YOU WANT A FIRST-CLASS PRO
```

```
DUCT THEN  YOU MUST TRY THE
14510 DATA FOR STATE OF THE ART TECHNOLO
GY         YOU CAN'T BEAT THE
14520 DATA IF YOU WANT A ROLLS-ROYCE THE
N JUST LOOK AT THE
14600 HI=2:LO=2:DIM HI$(HI),LO$(LO)
14700 FOR N=0 TO LO:READ L0$(N):NEXT N
14800 FOR N=0 TO HI:READ HI$(N):NEXT N
14900 PRINT"[CLR]":RETURN
```

Commentary

Lines 100–130: Contain an INSTR routine.

Lines 200–440: Pick the words to be used in the next question, and select the correct conjugation.

Lines 500–800: Set up your INPUT and reset variables.

Lines 900–910: Check for a comma.

Lines 1000–1200: Check for 'AND' and 'BUT'. If neither of these is present the program jumps to line 1500.

Lines 1300–1310: Update the current rule negatively if 'AND' or 'BUT' are present and the first word is 'NO'.

Line 1400: Updates the current rule positively if 'AND' or 'BUT' are present and the first word is not 'NO'.

Line 1500: Deletes anything preceding a comma.

Line 1600-2100: Check for 'YES', 'NO' and 'N'T' and update the current rule accordingly.

Line 2200: Checks for a double negative.

Lines 2300–2500: Check for 'likes'.

Lines 2600–2800: Check for 'dislikes'.

Lines 2900–5100: Similarly check for objects, adjectives and adverbs.

Lines 5110–5190: Check matches for high and low cost key words.

Line 5200: Checks for no match and reports.

Line 5300: Checks for more than one object.

Lines 5400–5440: Update the current rule, or another rule, according to whether or not the object matches the current question.

Line 5490: Jumps over the print-out of the rules.

Lines 5500–5800: Print out the rules.

Lines 5900–6200: Update the total cost and total profit values.

Line 6300: Prints an excuse if the profit seems too low.

Line 6400: Prints a warning if the spending is too high.

Line 6500: Zeros the total cost and profit values.

Line 6700–7120: Search for computers which match your requirements.

Line 7310: Jumps over the print-out of matching machines.

Lines 7350 –7800: Print out the matches.

Lines 7900–8400: Pick the highest and lowest priced machines which match the specification.

Line 8140: Checks if only one machine was selected.

Lines 8500–9100: Print out the name of either the highest or lowest priced machine.

Line 9200: Updates the feature to be checked and returns for another input.

Lines 9300–11300: Enter the information on features, keywords, costs and profits.

Lines 11400–11700: Provide an introduction.

Lines 11800–13800: Enter the information on the names and virtues of particular machines.

Lines 13900–14300: Provide warnings and excuses.

Lines 14400–14900: Contain the words for high and low cost messages.

The rest is up to you

Artificial Intelligence is a fascinating subject, and we trust that we have given you enough information to get you started on your own experiments in this area. We have certainly enjoyed making our own explorations whilst putting this book together, but we have started to wonder how long it will be before someone designs an expert system program which writes books ...

Artificial Intelligence on the Commodore 64 shows you how to implement AI routines on your home micro and turn it into an intelligent machine which can hold a conversation with you, give you rational advice, learn from you (and teach you) and even write programs for you.

The book explains AI from first principles and assumes no previous knowledge of the subject. All the important aspects of AI are covered and are fully illustrated with example programs.

For many years science fiction books and films have contained 'intelligent' computers which appear to be at least the equal of man. Although some of the features described in these remain illusions, extensive research into AI has brought many of the ideas much nearer reality.

Keith and Steven Brain are a father and son team and have already published the best selling Dragon 32 Games Master and Advanced Sound and Graphics for the Dragon computer. They are both regular contributors to Popular Computing Weekly.

159.00

SUNSHINE

ISBN 0 946408 29 7

£6.95 net