variables at the start of the program, and refer back to these variables thereafter. This makes the program faster and neater, and it means that you can change these values without having to hunt through the program for every occurrence.

Even with the sort of formal approach that we have outlined here, it's difficult to eliminate bugs entirely, so it's important to adopt a disciplined method for finding and eradicating them. The commonest bugs are syntax errors, and you can usually correct them as soon as you encounter them. But this is not always the case. Consider:

```
10 PRINT"BIG BUGS HAVE LITTLE BUGS UPON"
20 PRINT"THEIR BACKS TO BITE THEM"
```

Such lines often cause an error message when executed if they're not keyed in as two separate lines. Line 10 contains 40 characters, so when you type it on a 40-column screen, the cursor finishes up at the start of the next screen line, which can cause you to forget to hit RETURN on line 10 before you start typing line 20. If so, then what look like two perfect lines in your program will actually be one line with a syntax error (the number 20) in the middle of it. One way of trapping these errors is to list suspect lines individually rather than as part of a piece of program.

Error messages, when they're not incomprehensible, can be misleading. Take for example:

```
25 DATA 10.2;34.56.9,0.008,15.6
30 FOR K=1 TO 5: READ N(K):NEXT K
```

This may fail to execute because of an alleged syntax error in line 30; whereas the error is actually in the data on line 25 (One of the zeros has been mis-keyed as the letter O).

Coding errors that don't result in syntax errors are the commonest bugs, and usually also the hardest to find. In this case, it is vital to be methodical. Begin by trying to find out roughly where the bug is in the program. This is reasonably easy with well-structured modular programs, and can be made easier by the TRACE utility, which causes the current program line number to be printed on the screen as it is executed. If your machine doesn't allow this, then you can create TRACE statements periodically throughout the program (PRINT "LINE 150" at the beginning of line 150, for example). Similarly, you can use the STOP command to halt program execution at significant places in the program so that you can examine the values of crucial variables. You can do this in direct mode using PRINT, or you can write a subroutine onto the end of your program:

```
11000 REM PRINT THE VARIABLES
11100 PRINT"SCORE,SIZE,FLAGS"
11200 PRINT SC;SZ;F1;F2
11300 PRINT"BOARD ARRAY"
11400 FOR K=1 TO 10:PRINT BD$(K):NEXT K
```

Consequently, when the program comes across a STOP command, you can type GOTO 11000, and have the current state of the variables displayed. You can even change them (by typing, say, SZ=17 and pressing RETURN), and then restart the program with the CONTinue command.

When you've found that the bug is lurking within certain lines, or in a particular variable, then you should be close to eliminating it, but tread carefully! Try one remedy at a time so that you can see what its exact effect on execution is. It's very easy to make several changes between runs, perhaps getting rid of one bug, but creating one or more new ones, and then forgetting exactly what it was you did!

Loops and branches, especially when they're nested, are particularly fertile ground for bugs, and require special care in both writing and de-bugging. Consider this piece of code:

```
460 IF SM< 0 AND SC<>-1 THEN IF SC>0 OR
    SM=SC-F9 THEN LT=500
470 FOR C1=1 TO LT:FOR C2=LT TO C1 STEP-1
480 SC=SM+SC*C2
490 NEXT C2:SM=0:NEXT C1
```

What does this all mean? Even if you know what it's meant to do, would you know if it were succeeding or failing? Putting statements inside a loop when they should be outside is a sure way to encourage bugs. And so is failing to cover all possible conditions when writing IF . . . THEN statements. A special case of this occurs when you write multiple statements after IF . . . THEN. For example:

```
655 IF A$="" THEN GOTO 980:A$=B$
660 PRINT A$
```

The statement A$=B$ will never be executed because either A$="", in which case control passes to line 980, or A$<>"", in which case the rest of line 655 is ignored.

Experience is the best teacher of de-bugging, but a step-by-step approach and a disciplined method are invaluable aids. Take your time, and — above all — DON'T PANIC!

Early Bug

To new programmers, bugs often seem to take on animate characteristics, such as hiding from the programmer and deliberately undermining all his efforts to find them. However, the first bug (at least the one from which the term is derived) really was animate. In trying to eliminate an error from a program she was developing on the Harvard Mrk II in 1945, Captain Grace Hopper discovered that a large moth had got caught up in the electromechanical working of the computer and was causing the fault. As a result of that incident, the term 'de-bugging' was coined

TONY LODGE