because the control logic is so curiously distributed. Let us analyze the code by sections.

The DO loops at RUNUD and RUNLR inspect the borders of the maze, looking for an entrance. TEST1 and TEST2 start the mouse going "in" from an opening by transferring control to the appropriate LOOKx test. These implement the right-turn rule described above, either to find a path continuation or to cause the mouse to back out of a dead end. SET stores each point along the path and tests whether the mouse has reached the border again. If the mouse comes out where it went in, this is not a path, so the program goes back to the appropriate RUNxx loop to continue searching the border where it left off. Otherwise it falls through to the SORT and MERGE loops, which determine the path to be printed out.

The RUNxx loops have elaborate control parameters so that they can be resumed. This is not an easy thing to do, so we are not surprised to find that it is done wrong. At TEST1, for instance, we see that NN remembers the index K2; but at RUNUD, NN is associated with the index K1. Sure enough, the lower limits for all four DO loops are stored incorrectly. The program cannot properly resume the border scan.

There are other problems with these DO loops. If the maze has no entry points, or if there is no path through it, control falls through to TEST1 and the program starts looking around outside the array MAZE. (Watch out for DO loops that "never" terminate normally.) By definition, it is not possible to go "in" from a corner, yet each is tried and could give an uninteresting answer — a path that only runs along the edge. In fact, any adjacent ones on the border will be reported as a path. Finally, the two inner loops (on K2) are handled differently even though they perform similar functions. This tips us off that one of them is incorrect. (As a matter of fact both are, but the details are not worth pursuing.)

The four LOOKx tests form one of those repeated patterns we have already encountered several times. Defining the appropriate data structures should permit us to summarize all four tests in one. Then perhaps we could avoid the dubious GOTO BRANCH, which is a PL/I equivalent of Fortran's assigned GOTO and equally obscure. Anything that disguises the flow of control should be avoided.

SORT and MERGE are correctly coded, but not well designed. It seems silly to save all the loops and dead ends encountered until the very end, when they can be readily eliminated along the way. Then it is not necessary to make a list of all the matching points so that loops can be skipped over on output. (POINT should be written as two arrays, by the way, and each should be much larger than 60 elements. POSITIONX and POSITIONY, on the other hand, need be only about two thirds of their current size of 2500.) SORT and MERGE are far more complicated than necessary.

Finally, the sequence

```
                GO TO ENDSORT;
        ENDSORT:END SORT;
```

is an open invitation to misunderstanding. The fact that a four line comment is needed to explain what is going on should be reason enough to rewrite the code. An even better reason is that the comment is wrong — *two* dummy statements are created to end *two* DO groups. How much easier and clearer it is to write