This specifies the input function interface more clearly, and shows where the printing is done. It also implies that we need a procedure that tests whether there is a path, which is obviously the hard part. In fact, since this is a big job, it might be desirable to write READMAZE completely and test it before we go on with path-finding.

We will represent the maze by a bit array with one value for a wall and the other for a non-wall, just as in the original. Then READMAZE is the following:

```
READMAZE: PROCEDURE RETURNS (BIT(1));

   ON ENDFILE(SYSIN)
      GOTO EOF;
   GET LIST (M, N);
   IF M < 2 | M > 50 | N < 2 | N > 50 THEN DO;
      PUT SKIP LIST (M, N, 'BAD DIMENSIONS');
      RETURN(NO);
   END;
   GET EDIT (((MAZE(I,J) DO J = 1 TO N) DO I = 1 TO M))
      (COLUMN(1), (N)B(1));
   PUT PAGE EDIT (((MAZE(I,J) DO J = 1 TO N) DO I = 1 TO M))
      (COLUMN(1), (N)B(1));
   RETURN(YES);
EOF:
   RETURN(NO);
END READMAZE;
```

Now we can write an abbreviated main routine that calls only READMAZE, and test it before any more code is added to confuse the logic.

With READMAZE out of the way, we can continue with the procedure FINDPATH, which searches a maze for a path. Basically, FINDPATH must probe at the maze from each edge. If it ever finds a path, it returns YES; otherwise it returns NO.

```
FINDPATH()
   IF (path from left side)
      return(YES)
   ELSE IF (path from right side)
      return(YES)
   ELSE IF (path from top)
      return(YES)
   ELSE IF (path from bottom)
      return(YES)
   ELSE
      return(NO)
```

Suppose we put the details of how to look for a path from a particular edge into a separate procedure called TRY, where we can use arguments to indicate what edge and direction is of interest in a particular call.

TRY looks at the cell under consideration. If this is a wall, then there can be no path, and TRY can return NO immediately. If the cell is not a wall, TRY can search from the cell in each direction in turn.