flowchart with a correct solution, but in the process of going from flowchart to code, the increment of COUNT got moved to the wrong place. (The discrepancy between flowchart and code illustrates one of the problems of program documentation, a topic which we will study in Chapter 8.)

Rewriting in pseudo-code with only one counter leads to a neater solution:

```
read EOF test
sum = 0
i = 0
WHILE (get new value ¬= EOF)
   i = i + 1
   x(i) = new value
   sum = sum + new value
IF (i > 0)
   avg = sum / i
ELSE
   avg = 0
```

After we hand-test the pseudo-code at its boundaries, it can be translated into Fortran with some confidence in its correctness.

```
      REAL X(200)
      READ(5,100) TEST
100   FORMAT(F10.0)
      SUM = 0.0
      I = 0
 10   READ(5,100) VAL
      IF (VAL .EQ. TEST) GOTO 20
         I = I + 1
         X(I) = VAL
         SUM = SUM + VAL
         GOTO 10
 20   IF (I .GT. 0) AVG = SUM / FLOAT(I)
      IF (I .LE. 0) AVG = 0.0
      ...
```

As a small but satisfying sign of improvement, the array size needed to process 200 items is now 200 instead of 201, because the end of file signal is no longer stored in X.

Test programs at their boundary values.

Another way to head off potential disasters is to "program defensively." Anticipate that in spite of good intentions and careful checking, things will sometimes go awry, and take some steps to catch errors before they propagate too far. For example, here is a fragment of a checker-playing program — it counts the number of reds and blacks on the board. Reds are represented by +1, blacks by −1, and unoccupied squares by 0. There are no other legal values.

```
IF BOARD(I,J)=1 THEN REDS = REDS + 1 ;
IF BOARD(I,J)=-1 THEN BLACKS = BLACKS + 1 ;
```

Suppose we are debugging the code. In the best of all worlds, there would never be