operation to control the loop. Since that is not a simple operation, we make a separate module to isolate that complexity from the code that produces the output report.

The result:

```
CTR = 0;
DO WHILE (GETCARD() = YES);
   IF ¬ANY(NUM_TBL = CARD.NUM) THEN
      PUT SKIP LIST ('BAD CARD', CARD.NUM, CARD.AMT);
   ELSE DO;
      CTR = CTR + 1;
      IF MOD(CTR, 46) = 1 THEN DO;   /* HEADER */
         WRITE FILE (PRTFLE) FROM (HDR);
         WRITE FILE (PRTFLE) FROM (COL_HDR);
         WRITE FILE (PRTFLE) FROM (LINE);
      END;
      IF CARD.AMT > 0 THEN DO;
         DETAIL.CREDIT = CARD.AMT;
         DETAIL.DEBIT = 0;
      END;
      ELSE DO;
         DETAIL.CREDIT = 0;
         DETAIL.DEBIT = CARD.AMT;
      END;
      WRITE FILE (PRTFLE) FROM (DETAIL);
   END;
END;
```

This takes care of producing the report, leaving the problem of input to a separate module:

```
GETCARD: PROCEDURE RETURNS (BIT(1));
   ON ENDFILE (CARDIN)
      GOTO EOF;
   READ FILE (CARDIN) INTO (CARD);
   RETURN (YES);
EOF:
   RETURN (NO);
END;
```

GETCARD merely reads a card each time it is called; it signals end of file when there is no data left.

Let the data structure the program.

Modularity becomes most important when a program starts getting large, so we will devote the rest of this chapter to a single example that is big enough to illustrate several principles of program structure. The following program simulates a mouse trying to find a path through a maze by the simple rule, "Turn right if you can, left if you must." The maze is a Boolean matrix, with ones representing possible paths and zeros the walls. A path consists of a connected series of horizontal and vertical strings of ones that enters the maze somewhere and exits somewhere else. A path may not run along the edge, although its ends may both be on one side. For