(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Intro. Given the specification of a SuperSlitherlink puzzle in *stdin*, this program outputs MCC data for the problem of finding all solutions. (It's based on SLITHERLINK-DLX, which handles ordinary Slitherlink.)

SuperSlitherlink extends the former rules by allowing several cells to participate in the same clue. Each superclue is identified by a letter, and the relevant cells are marked with that letter. An edge that lies between two cells with the same letter is called "internal" and it cannot participate in the solution. An edge that lies between a cell with a letter and a cell that either has a numeric clue or a blank clue or lies off the board is a "boundary edge" for that letter. An edge that lies between cells with different letters is a boundary edge for both of those letters. The clue for a letter is the number of boundary edges that should appear in the solution.

For reasons of speed and simplicity, this program does *not* solve SuperSlitherlink puzzles in full generality: In weird cases it can generate options that contain two identical items. It does, however, always work when the superclue groups are subrectangles, and in many other cases. (The similar but more complex program SUPERSLITHERLINK-GENERAL-DLX can be used to handle arbitrarily weird superclues.)

No attempt is made to enforce the "single loop" condition; solutions found by DLX3 will consist of disjoint loops. But DLX3-LOOP will weed out any disconnected solutions. In fact, it will nip most of them in the bud.

The specification begins with m lines of n characters each; those characters should be either '`0`' or '`1`' or '`2`' or '`3`' or '`4`' or '`.`' or a letter. (Here '`.`' means "no clue.") Those opening lines should be followed by lines of the form `!`⟨letter⟩`=`⟨clue⟩, one for each letter in the opening lines. For example, here's the specification for a simple SuperSlitherlink puzzle that Johan de Ruiter designed for Pi Day 2025:

```
aaa....
aaabb..
aaabb..
.......
..ccddd
..ccddd
....ddd
!a=3
!b=1
!c=4
!d=1
```

(See `https://cs.stanford.edu/~knuth/news25.html`.)

2. Here now is the general outline.

```
#define maxn 30    /* m and n must be at most this */
#define bufsize 80
#define panic(message)
        { fprintf(stderr, "%s:␣%s", message, buf); exit(−1); }
#include <stdio.h>
#include <stdlib.h>
  char buf[bufsize];
  int board[maxn][maxn];    /* the given clues */
  char super[128];    /* did this letter appear? */
  int clue[128];    /* numeric clues for letters that have appeared */
  char code[] = {#c, #a, #5, #3, #9, #6, #0};
  char edge[2 * maxn + 1][2 * maxn + 1];    /* is this a legal edge? */
  void main()
  {
    register int d, i, j, k, m, n;
    ⟨Read the input into board 3⟩;
    ⟨Determine the legal edges 5⟩;
    ⟨Print the item-name line 6⟩;
    for (i = 0; i ≤ m; i++)
      for (j = 0; j ≤ n; j++) ⟨Print the options for tile (i, j) 8⟩;
  }
```

3. ⟨Read the input into *board* 3⟩ ≡

```
  printf("|␣slitherlink-dlx:\n");
  for (i = 0; i < maxn; ) {
    if (¬fgets(buf, bufsize, stdin)) break;
    printf("|␣%s", buf);
    if (buf[0] ≡ '!') ⟨Record the clue for a letter and continue 4⟩;
    for (j = 0; j < maxn ∧ buf[j] ≠ '\n'; j++) {
      k = buf[j];
      if (k ≡ '|' ∨ k ≡ ':' ∨ k < 0) panic("Illegal␣character");
      if (k ≠ '.' ∧ (k < '0' ∨ k > '4')) super[k] = 1, clue[k] = −1;    /* we treat k as a letter */
      board[i][j] = k;
    }
    if (i++ ≡ 0) n = j;
    else if (n ≠ j) panic("row␣has␣wrong␣number␣of␣clues");
  }
  m = i;
  if (m < 2 ∨ n < 2) panic("the␣board␣dimensions␣must␣be␣2␣or␣more");
  for (k = 0; k < 128; k++)
    if (super[k] ∧ clue[k] < 0) fprintf(stderr, "No␣`!%c=...'!\n", k);
  fprintf(stderr, "OK,␣I've␣read␣a␣%dx%d␣array␣of␣clues.\n", m, n);
```

This code is used in section 2.

4. ⟨ Record the clue for a letter and **continue** 4 ⟩ ≡

```
{
  if ($buf[2] \neq$ '=') panic("no␣=␣sign");
  $k = buf[1]$;
  if ($\neg super[k]$) fprintf(stderr, "letter␣`%c'␣never␣occurred!\n", $k$);
  else {
    for ($d = 0, j = 3$; $buf[j] \geq$ '0' $\land$ $buf[j] \leq$ '9'; $j$++) $d = 10 * d + buf[j] -$ '0';
    $clue[k] = d$;
  }
  continue;
}
```

This code is used in section 3.

5. The primary items are "tiles," "cells," and "clues." Tiles control the loop path; the name of tile (i, j) is $(2i, 2j)$. Cells represent numeric clues; the name of cell (i, j) is $(2i+1, 2j+1)$. A cell item is present only if a numeric clue has been given for that cell of the board.

The secondary items are "edges" of the path. Their names are the midpoints of the tiles they connect.

An edge is illegal if it is internal to the clue of some letter. A boundary edge for a zero clue is also illegal.

⟨ Determine the legal edges 5 ⟩ ≡

```
for ($i = 0$; $i \leq m + m$; $i$++)
  for ($j = 0$; $j \leq n + n$; $j$++)
    if ($(i + j)$ & 1) $edge[i][j] = 1$;
for ($i = 0$; $i < m$; $i$++)
  for ($j = 0$; $j < n$; $j$++) {
    $k = board[i][j]$;
    if ($k \equiv$ '0')
      $edge[i+i][j+j+1] = edge[i+i+1][j+j] = edge[i+i+1][j+j+2] = edge[i+i+2][j+j+1] = 0$;
    else if ($super[k]$) {
      if ($j < n - 1 \land board[i][j + 1] \equiv k$) $edge[i + i + 1][j + j + 2] = 0$;
      if ($i < m - 1 \land board[i + 1][j] \equiv k$) $edge[i + i + 2][j + j + 1] = 0$;
      if ($clue[k] \equiv 0$)
        $edge[i+i][j+j+1] = edge[i+i+1][j+j] = edge[i+i+1][j+j+2] = edge[i+i+2][j+j+1] = 0$;
    }
  }
for ($k = 0$; $k < 128$; $k$++)
  if ($super[k] \land clue[k] \equiv 0$) $super[k] = 0$;
```

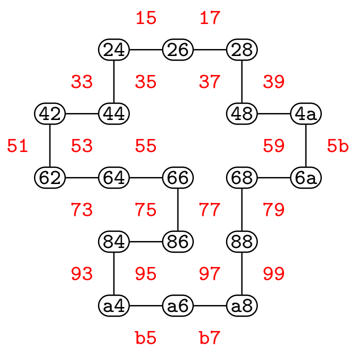
This code is used in section 2.

6. A two-coordinate "name" (x,y) actually appears as two encoded digits (in order to match the conventions of DLX3-LOOP).

```
#define encode(x) ((x) < 10 ? (x) + '0' : (x) < 36 ? (x) − 10 + 'a' : (x) < 62 ? (x) − 36 + 'A' : '?')
⟨Print the item-name line 6⟩ ≡
  for (i = 0; i ≤ m; i++)
    for (j = 0; j ≤ n; j++) printf("%c%c␣", encode(i + i), encode(j + j));
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      if (board[i][j] ≥ '1' ∧ board[i][j] ≤ '4')
        printf("%d|%c%c␣", board[i][j] − '0', encode(i + i + 1), encode(j + j + 1));
  for (k = 0; k < 128; k++)
    if (super[k]) printf("%d|%c␣", clue[k], k);
  printf("|");
  for (i = 0; i ≤ m + m; i++)
    for (j = 0; j ≤ n + n; j++)
      if (edge[i][j]) printf("␣%c%c", encode(i), encode(j));
  printf("\n");
```

This code is used in section 2.

7. This program relies on a beautiful property of loop paths in a grid, which can perhaps be best understood by looking at a concrete example. Consider the loop 24 — 26 — 28 — 48 — 4a — 6a — 68 — 88 — a8 — a6 — a4 — 84 — 86 — 66 — 64 — 62 — 42 — 44 — 24, which passes through 18 tiles. (Coordinates are specified here in hexadecimal notation.) It has, of course, 18 edges, namely 25, 27, 38, 49, 5a, 69, 78, 98, a7, a5, 94, 85, 76, 65, 63, 52, 43, 34. Here's a picture, showing also the cells that are adjacent to the edges:



We assign two cells to each pair of adjacent edges in the loop, using a "counterclockwise rule":

25, 27 ↦ 35, 17	78, 98 ↦ 99, 77	76, 65 ↦ 75, 77
27, 38 ↦ 37, 39	98, a7 ↦ 97, b7	65, 63 ↦ 73, 55
38, 49 ↦ 39, 37	a7, a5 ↦ b5, 97	63, 52 ↦ 53, 51
49, 5a ↦ 59, 5b	a5, 94 ↦ 95, 93	52, 43 ↦ 53, 33
5a, 69 ↦ 59, 79	94, 85 ↦ 95, 75	43, 34 ↦ 33, 53
69, 78 ↦ 79, 59	85, 76 ↦ 75, 95	34, 25 ↦ 35, 15

(When the edges take a 90° turn, they frame the first cell, and the second cell is obtained by rotating counterclockwise. A similar rule applies when the edges are parallel.)

This rule has the magic property that each cell occurs precisely as often as it is adjacent to a edge of the loop. For example, cell 15 occurs once, cell 35 occurs twice, and cell 75 occurs thrice.

The magic property can be proved by induction on the length of the loop, considering all ways that a cell can be surrounded by 1, 2, 3, or 4 edges of a loop.

8. Each tile '$2i$,$2j$' controls the destiny of up to four edges that touch its central point (i,j). It is filled with either zero or two edges, and each edge potentially contributes to one or two clue counts. If there are two edges, we include the names of the adjacent cells, whenever a clue has been given for such cells.

The adjacent cells will never correspond to the same primary item, if the edges specify a right-angle turn, or if the clue groups are subrectangles.

```
#define N(i,j)  (i > 0 ∧ edge[i + i − 1][j + j])
#define W(i,j)  (j > 0 ∧ edge[i + i][j + j − 1])
#define E(i,j)  (j < n ∧ edge[i + i][j + j + 1])
#define S(i,j)  (i < m ∧ edge[i + i + 1][j + j])
⟨Print the options for tile (i,j) 8⟩ ≡
  {
    for (k = 0; k < 7; k++) {
      if ((code[k] & 8) ∧ ¬N(i,j)) continue;    /* can't go north */
      if ((code[k] & 4) ∧ ¬W(i,j)) continue;    /* can't go west */
      if ((code[k] & 2) ∧ ¬E(i,j)) continue;    /* can't go east */
      if ((code[k] & 1) ∧ ¬S(i,j)) continue;    /* can't go south */
      printf("%c%c", encode(i + i), encode(j + j));
      if (N(i,j)) printf("␣%c%c:%d", encode(i + i − 1), encode(j + j), code[k] ≫ 3);
      if (W(i,j)) printf("␣%c%c:%d", encode(i + i), encode(j + j − 1), (code[k] ≫ 2) & 1);
      if (E(i,j)) printf("␣%c%c:%d", encode(i + i), encode(j + j + 1), (code[k] ≫ 1) & 1);
      if (S(i,j)) printf("␣%c%c:%d", encode(i + i + 1), encode(j + j), code[k] & 1);
      switch (k) {
      case 0: ⟨Show NW clue 9⟩; ⟨Show SW clue 12⟩; break;    /* NW */
      case 1: ⟨Show NE clue 10⟩; ⟨Show NW clue 9⟩; break;    /* NE */
      case 2: ⟨Show SW clue 12⟩; ⟨Show SE clue 11⟩; break;    /* SW */
      case 3: ⟨Show SE clue 11⟩; ⟨Show NE clue 10⟩; break;    /* SE */
      case 4: ⟨Show SE clue 11⟩; ⟨Show NW clue 9⟩; break;    /* NS */
      case 5: ⟨Show SW clue 12⟩; ⟨Show NE clue 10⟩; break;    /* EW */
      case 6: break;    /* untouched */
      }
      printf("\n");
    }
  }
```

This code is used in section 2.

9. Finally we need to identify the clues, if any, that are relevant in each of the four quadrants of tile (i,j).

```
⟨Show NW clue 9⟩ ≡
  if (i > 0 ∧ j > 0) {
    register int k = board[i − 1][j − 1];
    if (k ≠ '.') {
      if (k ≥ '1' ∧ k ≤ '4') printf("␣%c%c", encode(i + i − 1), encode(j + j − 1));
      else printf("␣%c", k);
    }
  }
```

This code is used in section 8.

10. ⟨ Show NE clue 10 ⟩ ≡

```
if ($i > 0 \land j < n$) {
  register int $k = board[i-1][j]$;
  if ($k \neq$ '.') {
    if ($k \geq$ '1' $\land k \leq$ '4') printf("␣%c%c", encode($i+i-1$), encode($j+j+1$));
    else printf("␣%c", $k$);
  }
}
```

This code is used in section 8.

11. ⟨ Show SE clue 11 ⟩ ≡

```
if ($i < m \land j < n$) {
  register int $k = board[i][j]$;
  if ($k \neq$ '.') {
    if ($k \geq$ '1' $\land k \leq$ '4') printf("␣%c%c", encode($i+i+1$), encode($j+j+1$));
    else printf("␣%c", $k$);
  }
}
```

This code is used in section 8.

12. ⟨ Show SW clue 12 ⟩ ≡

```
if ($i < m \land j > 0$) {
  register int $k = board[i][j-1]$;
  if ($k \neq$ '.') {
    if ($k \geq$ '1' $\land k \leq$ '4') printf("␣%c%c", encode($i+i+1$), encode($j+j-1$));
    else printf("␣%c", $k$);
  }
}
```

This code is used in section 8.

13. Index.

SUPERSLITHERLINK-DLX