November 24, 2020 at 13:23

1. Intro. This quick-and-dirty program prepares TEX files for use with the weird fonts CELTICA and CELTICB. You can use it to print amazing pictures that look like stylized Celtic knots. Sometimes the resulting pictures look very elegant. (Plug: You can see the author's most elegant example, so far, in Chapter 46 of the book *Selected Papers on Fun and Games*, published by CSLI in 2010.)

The "knots" we print consist of one or more closed loops in which no three points are concurrent. Therefore we can draw them in such a way that the paths go alternately over-under-over-under, etc., whenever they cross. The loops consist of segments that cut a square grid either at corner points or at midpoints between adjacent corners. So we can think of the total picture as composed of square tiles, where each tile has eight

possible entry or exit points, numbered thus:

There are three kinds of tiles, each representable as a sequence of four characters:

- A blank tile, represented by four blanks.
- A tile with a single segment from i to j, where $i < j$, represented by ij and two blanks.
- A tile with segments from i to j and from i' to j', where $i < j$ and $i' < j'$ and $i < i'$, represented by $iji'j'$.

The segments from i to j are also subject to several additional restrictions:

- i and j cannot be adjacent on the periphery.
- i and j cannot both be even.

Thus, for example, if $i = 0$ the only possibilities for j are 3 and 5. But if $i = 1$ we can have $j = 3$, 4, 5, 6, or 7. It follows that 14 different one-segment tiles are legal, and there are 47 with two segments.

The input to this program, for an $m \times n$ picture, consists of m lines of $5n$ characters, where each line contains n tile specifications separated by blanks and followed by a period. For example, the simple 3×4 input

```
35   57   35   57  .
13   1537 1537 17  .
     13   17       .
```

yields the nice little picture ‘’.

The rules are illustrated more fully by the following 6×7 example, which was used in the author's initial tests:

```
                    35   57       .
     35   57   35   1537 1357 57  .
     1435 1657 15   1435 1637 17  .
35   1725 0514 16   1325 0537 57  .
13   1735 1725 03   17   13   17  .
     13   17                      .
```

From that input (on *stdin*), the output of this program (on *stdout*) is a TEX file that prints a “poodle”:

2. Whenever a tile contains a segment through some boundary point, the neighboring tiles must also contain a segment through that common point. For example, tile ‘`35`’ can be used only if its neighbor to the right uses its point `7`, and only if its neighbor below uses its point `1`. More significantly, if a tile uses a corner point, it has three neighbors that touch the same corner, and all three of them must use that corner. Paths therefore always make an ‘X’ crossing, at right angles, whenever they pass through a corner of the grid.

All regions of the final illustration are filled in with black, if they don't lie completely outside of all paths.

3. OK, let's get going. This program ought to be fun, once we get through the tedious details of preparing font tables and of reading/checking the input.

```
#define maxm 100 /* at most this many rows; mustn't exceed 4096 */
#define maxn 100 /* at most this many columns; mustn't exceed 4096 */
#define bufsize 5 * maxn + 2
#include <stdio.h>
#include <stdlib.h>
  char buf[bufsize], entry[8];
  int a[maxm][maxn]; /* the input */
  int b[maxm][maxn]; /* endpoints touched in each tile */
  int codetable[#7778]; /* mapping from tiles to font positions */
  char bw[maxm][maxn][8]; /* black/white coloring of regions */
  int inout[maxm][maxn][8]; /* inside/outside coloring of regions */
  ⟨ Declare the magic tables 11 ⟩;
  main()
  {
    register int i, j, k, ii, jj, kk, m, n, s, t;
    ⟨ Initialize codetable 4 ⟩;
    ⟨ Read the input into a, and check it for consistency 5 ⟩;
    ⟨ Do the black/white coloring 8 ⟩;
    ⟨ Do the inside/outside coloring 9 ⟩;
    ⟨ Produce the output 13 ⟩;
  }
```

4. (Begin tedium.) Fonts CELTICA and CELTICB have a peculiar encoding scheme, not terribly systematic, governed by a mapping from tile specs (in hexadecimal) to character positions (in octal).

```
⟨ Initialize codetable 4 ⟩ ≡
  for (i = 1; i < #7778; i++) codetable[i] = −1;
  codetable[#1537] = 0; codetable[0] = °40;
  codetable[#0300] = °44, codetable[#0500] = °46, codetable[#1300] = °50, codetable[#1400] = °52,
      codetable[#1500] = °54, codetable[#1600] = °56, codetable[#1700] = °60, codetable[#2500] = °62,
      codetable[#2700] = °64, codetable[#3500] = °66, codetable[#3600] = °70, codetable[#3700] = °72,
      codetable[#4700] = °74, codetable[#5700] = °76, codetable[#1357] = °100, codetable[#1735] = °104,
      codetable[#0513] = °110, codetable[#2735] = °114, codetable[#1457] = °120, codetable[#1736] = °124,
      codetable[#0357] = °130, codetable[#1725] = °134, codetable[#1347] = °140, codetable[#1635] = °144,
      codetable[#0514] = °150, codetable[#2736] = °154, codetable[#0347] = °160, codetable[#1625] = °164,
      codetable[#0314] = °170, codetable[#2536] = °174, codetable[#0547] = °200, codetable[#1627] = °204,
      codetable[#0315] = °210, codetable[#2537] = °214, codetable[#1547] = °220, codetable[#1637] = °224,
      codetable[#0537] = °230, codetable[#1527] = °234, codetable[#1437] = °240, codetable[#1536] = °244,
      codetable[#0316] = °250, codetable[#0325] = °254, codetable[#2547] = °260, codetable[#1647] = °264,
      codetable[#0527] = °270, codetable[#1425] = °274, codetable[#1436] = °300, codetable[#0536] = °304,
      codetable[#0317] = °310, codetable[#1325] = °314, codetable[#3547] = °320, codetable[#1657] = °324,
      codetable[#0517] = °330, codetable[#1327] = °334, codetable[#1435] = °340, codetable[#3657] = °344,
      codetable[#0516] = °350, codetable[#0327] = °354, codetable[#1425] = °360, codetable[#3647] = °364;
```

This code is used in section 3.

5. (More tedium. I do try to check carefully for errors, because the task of preparing the input is even more tedious than the task of writing this code.)

The rows are numbered from 0 to $m-1$, and the columns from 0 to $n-1$.

```
⟨ Read the input into a, and check it for consistency 5 ⟩ ≡
  for (m = 0; ; m++) {
    if (¬fgets(buf, bufsize, stdin)) break;
    for (j = 0; ; j++) {
      k = 5 * j;
      if (j ≡ n ∧ m > 0) {
        fprintf(stderr, "Missing ‘.’ at the end of row %d!\n", m);
        exit(−1);
      }
      ⟨ Parse the entry for row m and column j 6 ⟩;
      if (buf[k + 4] ≡ '.') {
        if (m ≡ 0) n = j + 1;
        else if (n ≠ j + 1) {
          fprintf(stderr, "Premature ‘.’ in row %d!\n", m);
          exit(−2);
        }
        break;
      } else if (buf[k + 4] ≠ ' ') {
        fprintf(stderr, "Tile spec in row %d, col %d not followed by blank!\n", m, j);
        exit(−5);
      }
    }
    continue;
  badentry: fprintf(stderr, "Bad entry (%s) in row %d and column %d!\n", entry, m, j);
    exit(−3);
  }
  if (m ≡ 0) {
    fprintf(stderr, "There was no input!\n");
    exit(−4);
  }
  fprintf(stderr, "OK, I've successfully read %d rows and %d columns.\n", m, n);
  ⟨ Check for consistency 7 ⟩;
```

This code is used in section 3.

6. ⟨ Parse the entry for row m and column j 6 ⟩ ≡

```
for (jj = 0; jj < 4; jj++) entry[jj] = buf[5 * j + jj];
if (entry[0] ≡ '␣') {
  if (entry[1] ≠ '␣' ∨ entry[2] ≠ '␣' ∨ entry[3] ≠ '␣') goto badentry;
  else a[m][j] = 0;
} else {
  if (entry[0] < '0' ∨ entry[0] > '7') goto badentry;
  if (entry[1] < '0' ∨ entry[1] > '7') goto badentry;
  a[m][j] = ((entry[0] − '0') ≪ 12) + ((entry[1] − '0') ≪ 8);
  b[m][j] = (1 ≪ (entry[0] − '0')) + (1 ≪ (entry[1] − '0'));
  if (entry[2] ≡ '␣') {
    if (entry[3] ≠ '␣') goto badentry;
  } else {
    if (entry[2] < '0' ∨ entry[2] > '7') goto badentry;
    if (entry[3] < '0' ∨ entry[3] > '7') goto badentry;
    a[m][j] += ((entry[2] − '0') ≪ 4) + (entry[3] − '0');
    b[m][j] += (1 ≪ (entry[2] − '0')) + (1 ≪ (entry[3] − '0'));
  }
}
if (codetable[a[m][j]] < 0) {
  fprintf(stderr, "Sorry,␣%s␣isn't␣a␣legal␣tile␣(row␣%d,␣col␣%d)!\n", entry, m, j);
  exit(−4);
}
```

This code is used in section 5.

7. **#define** $eqbit(k,kk)$ $(((i \gg k) \oplus (ii \gg kk)) \,\&\, 1)$

```
⟨ Check for consistency 7 ⟩ ≡
  t = 0;
  for (j = 0; j ≤ m; j++)
    for (jj = 0; jj < n; jj++) {
      i = (j > 0 ? b[j − 1][jj] : 0);
      ii = (j < m ? b[j][jj] : 0);
      if (eqbit(4, 2) + eqbit(5, 1) + eqbit(6, 0)) {
        fprintf(stderr, "Inconsistent␣tiles␣%04x/%04x␣(row␣%d,␣col␣%d)!\n", j > 0 ? a[j − 1][jj] : 0,
            a[j][jj], j, jj);
        t++;
      }
    }
  for (jj = 0; jj ≤ n; jj++)
    for (j = 0; j < m; j++) {
      i = (jj > 0 ? b[j][jj − 1] : 0);
      ii = (jj < n ? b[j][jj] : 0);
      if (eqbit(2, 0) + eqbit(3, 7) + eqbit(4, 6)) {
        fprintf(stderr, "Inconsistent␣tiles␣%04x,%04x␣(row␣%d,␣col␣%d)!\n",
            jj > 0 ? a[j][jj − 1] : 0, a[j][jj], j, jj);
        t++;
      }
    }
  if (t) {
    fprintf(stderr, "Sorry,␣I␣can't␣go␣on␣(errs=%d).\n", t);
    exit(−69);
  }
```

This code is used in section 5.

8. OK, now the fun begins: We've got decent input, and we want to figure out how to typeset it.

The given loops partition the plane into regions, and the key idea is to assign "colors" to each region of each tile. We use two different bicolorings: (1) Regions are either black or white, where the color changes at each boundary edge between regions. (2) Regions are either inside or outside of the total picture. These two colorings are related by the fact that outside regions are always white.

A blank tile has only one region. A tile ij has two. And a tile $iji'j'$ has either three or four, depending on whether ij and $i'j'$ intersect. We unify these cases by considering eight subregions along the boundary, namely $0\,..\,1$, $1\,..\,2$, $\ldots$, $7\,..\,0$, some of which are known to be identical.

The black/white coloring is easily done in one pass. (This code is in fact overkill.)

```
⟨ Do the black/white coloring 8 ⟩ ≡
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      bw[i][j][0] = (i ≡ 0 ? 0 : bw[i − 1][j][5]);
      for (k = 1; k < 8; k++) bw[i][j][k] = (b[i][j] & (1 ≪ k) ? 1 : 0) ⊕ bw[i][j][k − 1];
    }
```

This code is used in section 3.

9. The inside/outside coloring is trickier, because connectivity to the outside can twist around, and because three-region tiles behave differently from four-region tiles.

The following algorithm is essentially a depth-first search to find all subregions that are connected to the upper left corner. A stack is maintained within the data structure. At the end of the process, $inout[i][j][k]$ is nonzero if and only if that subregion is outside.

```
#define pop(i,j,k)  i = s >> 16, j = (s >> 4) & #fff, k = s & #7, s = inout[i][j][k]
#define push(ii,jj,kk)
        { if (inout[ii][jj][(kk) & #7] == 0)
            inout[ii][jj][(kk) & #7] = s, s = ((ii) << 16) + ((jj) << 4) + ((kk) & #7); }
⟨ Do the inside/outside coloring 9 ⟩ ≡
  inout[0][0][0] = -1;
  for (s = 0; s >= 0; ) {
    pop(i,j,k);
    ⟨ Push all unseen neighbors of subregion [i][j][k] onto the stack 10 ⟩;
  }
```

This code is used in section 3.

10. The neighbors of a subregion within a tile are either in an adjacent tile or in the same tile.

```
⟨ Push all unseen neighbors of subregion [i][j][k] onto the stack 10 ⟩ ≡
  switch (k) {
  case 0: case 1: if (i > 0) push(i-1,j,5-k); break;
  case 2: case 3: if (j < n-1) push(i,j+1,9-k); break;
  case 4: case 5: if (i < m-1) push(i+1,j,5-k); break;
  case 6: case 7: if (j > 0) push(i,j-1,9-k); break;
  }
  if ((b[i][j] & (1 << k)) == 0) push(i,j,k+7);      /* move counterclockwise */
  kk = (k+1) & #7;
  if ((b[i][j] & (1 << kk)) == 0) push(i,j,kk);      /* move clockwise */
  ⟨ Check neighbors in three-region tiles 12 ⟩;
```

This code is used in section 9.

11. A tile that contains two nonintersecting segments consists of a middle region and two others. The middle region needs to be identified so that we can "jump" from one of its edges to the other.

Three-region tiles are characterized by having $codetable[a[i][j]]$ between $\circ 100$ and $\circ 164$, inclusive. When that happens, we can pack the necessary logic into a magic four-byte table, which contains the four endpoints $\{i, j, i', j'\}$ in the correct clockwise order for processing.

```
⟨ Declare the magic tables 11 ⟩ ≡
  char magic[56] = {
      1,3,5,7,
      7,1,3,5,
      1,3,5,0,
      7,2,3,5,
      1,4,5,7,
      7,1,3,6,
      0,3,5,7,
      7,1,2,5,
      1,3,4,7,
      6,1,3,5,
      1,4,5,0,
      7,2,3,6,
      0,3,4,7,
      6,1,2,5};
```

See also section 22.

This code is used in section 3.

12.

```
⟨ Check neighbors in three-region tiles 12 ⟩ ≡
  kk = codetable[a[i][j]] - °100;
  if (kk < °70 ∧ kk ≥ 0) {
    if (k ≡ magic[kk + 1]) push(i, j, magic[kk] + 7);
    if (k ≡ ((magic[kk] + 7) & #7)) push(i, j, magic[kk + 1]);
    if (k ≡ magic[kk + 3]) push(i, j, magic[kk + 2] + 7);
    if (k ≡ ((magic[kk + 2] + 7) & #7)) push(i, j, magic[kk + 3]);
  }
```

This code is used in section 10.

13. And at last, when every subregion has been colored in both of the bicolorings, we come to the *denouement*: Typesetting can proceed.

Suppose $codetable[a[i][j]]$ is k. Then the tile in row i, column j is typeset in font CELTICA or CELTICB, depending on whether its subregion 0 is white or black, respectively; that is, depending on whether $bw[i][j][0]$ is 0 or 1, respectively.

A blank tile, when $k = \circ 40$, is typeset only if it's inside. (Character $\circ 40$ is ‘’.)

A two-region tile is typeset from k if both regions are inside, from $k + 1$ if one region is outside. (For example, character $\circ 44$ is ‘’; character $\circ 45$ is ‘’ in CELTICA and ‘’ in CELTICB.)

A three-region or four-region tile is typeset from k, $k+1$, $k+2$, or $k+3$, depending on the inside/outside configurations; details are worked out below.

```
⟨ Produce the output 13 ⟩ ≡
  ⟨ Publish the preamble 14 ⟩;
  for (i = 0; i < m; i++) ⟨ Publish row i 16 ⟩;
  ⟨ Publish the postamble 15 ⟩;
```

This code is used in section 3.

14. ⟨ Publish the preamble 14 ⟩ ≡

```
printf("%% begin output of CELTIC-PATHS\n");
printf("\\font\\celtica=celtica13 \\font\\celticb=celticb13\n\n");
printf("\\begingroup\\celtica\n");
printf("\\catcode`\\|=\\active \\def|#1|{\\hskip#1em}\n");
printf("\\catcode`\\-=\\active \\def-#1#2#3{\\celtica\\char'#1#2#3}\n");
printf("\\catcode`\\+=\\active \\def+#1#2#3{\\celticb\\char'#1#2#3}\n");
printf("\\offinterlineskip\\baselineskip=1em\n");
printf("\\let\\par=\\cr \\obeylines \\halign{#\\hfil\n");
```

This code is used in section 13.

15. ⟨ Publish the postamble 15 ⟩ ≡

```
printf("}\\endgroup\n");
```

This code is used in section 13.

16. ⟨ Publish row i 16 ⟩ ≡

```
{
  s = 0;      /* s holds the number of accumulated blanks */
  for (j = 0; j < n; j++) ⟨ Typeset tile (i, j) 17 ⟩;
  printf("\n");
}
```

This code is used in section 13.

17. ⟨ Typeset tile (i, j) 17 ⟩ ≡

```
{
  kk = codetable[a[i][j]];
  ii = bw[i][j][0];
  if (kk ≥ °100) ⟨ Handle a tile with three or four regions 21 ⟩
  else if (kk ≡ °40) ⟨ Handle a blank tile 18 ⟩
  else if (kk ≡ 0) ⟨ Handle tile 1537 20 ⟩
  else ⟨ Handle a two-region tile 19 ⟩;
  if (s) {
    printf("|%d|", s);
    s = 0;
  }
  printf("%c%03o", ii ? '+' : '-', kk);
}
```

This code is used in section 16.

18. ⟨ Handle a blank tile 18 ⟩ ≡

```
{
  if (inout[i][j][0]) {      /* normal case, blank and outside */
    s++; continue;
  }
}
```

This code is used in section 17.

19. ⟨ Handle a two-region tile 19 ⟩ ≡

```
{
  for (k = 0; k < 8; k++)
    if (inout[i][j][k]) {
      kk++; break;
    }
}
```

This code is used in section 17.

20. The fonts treat `1537` as a special case in which all sixteen combinations of black/white backgrounds are permissible. Only four of them can actually occur in the output of this program, because adjacent regions cannot both be 'outside'.

⟨ Handle tile `1537` 20 ⟩ ≡

```
{
  kk = (inout[i][j][1] ? 1 : 0) + (inout[i][j][3] ? 8 : 0) + (inout[i][j][5] ? 4 : 0) + (inout[i][j][7] ? 2 : 0);
}
```

This code is used in section 17.

21. In the most complex case, we walk clockwise around the edge of the tile and note the pattern of four inside/outside regions that we see. Four patterns are possible (either 0000, 1000, 0010, or 1010 in CELTICA, and either 0000, 0001, 0100, or 0101 in CELTICB); they cause us to add 0, 1, 2, or 3, respectively to the character code *kk*.

⟨ Handle a tile with three or four regions 21 ⟩ ≡

```
{
  t = (inout[i][j][0] ? 8 : 0);
  for (k = 1, jj = 4; jj; k++)
    if (b[i][j] & (1 ≪ k)) {
      t += (inout[i][j][k] ? jj : 0);
      jj ≫= 1;
    }
  kk += offset[t];
}
```

This code is used in section 17.

22. ⟨ Declare the magic tables 11 ⟩ +≡

```
char offset[16] = {0, 1, 2, 0, 2, 3, 0, 0, 1, 0, 3, 0, 0, 0, 0, 0};
```

23. Index.

⟨ Check for consistency 7 ⟩ Used in section 5.
⟨ Check neighbors in three-region tiles 12 ⟩ Used in section 10.
⟨ Declare the magic tables 11, 22 ⟩ Used in section 3.
⟨ Do the black/white coloring 8 ⟩ Used in section 3.
⟨ Do the inside/outside coloring 9 ⟩ Used in section 3.
⟨ Handle a blank tile 18 ⟩ Used in section 17.
⟨ Handle a tile with three or four regions 21 ⟩ Used in section 17.
⟨ Handle a two-region tile 19 ⟩ Used in section 17.
⟨ Handle tile `1537` 20 ⟩ Used in section 17.
⟨ Initialize *codetable* 4 ⟩ Used in section 3.
⟨ Parse the entry for row m and column j 6 ⟩ Used in section 5.
⟨ Produce the output 13 ⟩ Used in section 3.
⟨ Publish row i 16 ⟩ Used in section 13.
⟨ Publish the postamble 15 ⟩ Used in section 13.
⟨ Publish the preamble 14 ⟩ Used in section 13.
⟨ Push all unseen neighbors of subregion $[i][j][k]$ onto the stack 10 ⟩ Used in section 9.
⟨ Read the input into a, and check it for consistency 5 ⟩ Used in section 3.
⟨ Typeset tile (i, j) 17 ⟩ Used in section 16.

CELTIC-PATHS