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

1. Intro. This is an interactive program to do calculations associated with Dekking's generalized dragon curves and the associated calculus of tiles, as described in my notes on "diamonds and dragons."

When prompted, the user can do the following things:

- `p`⟨path⟩
  Set the current zigzag path to the sequence of directions specifed by ⟨path⟩. (Directions are the digits `0`, `1`, `2`, `3`, meaning "right," "up," "left," and "down," respectively; they must begin with `0` and alternate in parity.) The computer responds with the value of z, which is the point reached at the end of the path in the complex plane that starts at 0 and moves by i^k when taking direction k. For example, `p01012` yields $z = 1 + 2i$. At the beginning of computation the current path is simply `0`, and $z = 1$.
- ⟨folding sequence⟩
  Set the current zigzag path to the specified ⟨folding sequence⟩, which is a sequence of `D`'s and `U`'s. A folding sequence of length $s - 1$ corresponds to the path of length s that starts in direction 0 and then changes the direction by $+1 \pmod 4$ for each `D` and $-1 \pmod 4$ for each `U`. For example, the command `DUDD` is equivalent to the command `p01012`. (I apologize for the historical baggage of this notation, according to which the *down*-fold `D` corresponds to making the actual direction go *up*.)
- `*`⟨path⟩ or `*`⟨folding sequence⟩
  Multiply the current path by the specified path or folding sequence, using Dekking's folding product. For example, if the current path is `01012`, the command `*03` or `*U` will change it to `0101210303` and set $z \leftarrow 3 + i$.
- ⟨tile⟩`*`⟨tile⟩
  Compute the folding product of two tiles with respect to the current value of z. Here ⟨tile⟩ is a list of two integers separated by a comma. For example, `3,2*-2,3` will yield the result `-8,1` when $z = 1 + 2i$, because $(3 + 2i) * (-2 + 3i) = i(3 + 2i) + z(-2 + 2i) = -8 + i$.
- `a*`⟨tile⟩
  Compute the folding product $v * w$ of all tiles v in the polyomino of the current path with the specified tile w. In particular, if the specified tile is the unit tile `1,0`, the effect is simply to list all of the current polyomino tiles v.
- `c`⟨tile⟩ or `c`
  Show the congruence class and type of the specified tile. Or, if no tile is specified, show the congruence classes and types of all tiles in the current polynomino.
- `f`⟨tile⟩ or `F`⟨tile⟩
  "Factor" the given tile u to obtain v and w such that $u = v * w$ with respect to the current path, where v is a tile in the current polyomino. With `F` instead of `f`, proceed to factor w in the same way, until cycling. These commands are allowed only when the current path is plane-filling.
- `m`
  Output METAPOST commands to draw the current path.
- `v`⟨integer⟩
  Specify the level of verbosity, where `v0` gives the minimum amount of output and `v-1` gives the maximum.
- `q`
  Quit the program.
- `%`⟨comment⟩
  Do nothing, but politely think about whatever comment has been given.
- `i`⟨filename⟩
  Take commands from the specified file, then come back for more (unless the file included a "quit" command). The file may contain any command except another `i` command, because I don't want to bother maintaining a stack of included files.

Please realize that I had to write this program in an awful hurry, because of many other commitments.

2. Here we go.

```
#define maxm (1 ≪ 15)  /* length of longest path allowed */
#define maxd (1 ≪ 8)  /* anything ≥ √(2maxm) is safe here */
#define maxp 100  /* how much memory is allowed for cycle detection? */
#define bufsize 1024  /* maximum length of commands */
#define verbose_echo (1 ≪ 0)  /* should commands of included files be echoed? */
#define verbose_folds (1 ≪ 1)  /* should folds be printed when directions given? */
#define verbose_dirs (1 ≪ 2)  /* should directions be printed when folds given? */
#define metapost_name "/tmp/dragon-calc.mp"  /* file name for METAPOST output */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  int vbose;
  FILE *infile, *outfile;
  char buf[bufsize];
  char dir[maxm], fold[maxm];  /* directions and folds of current path */
  int s;  /* length of current path */
  typedef struct pair_struct {
    long x, y;
  } pair;
  pair e, u, v, w, z, uu, vv;
  pair ipower[4] = {{1, 0}, {0, 1}, {−1, 0}, {0, −1}};
  pair sqrt8i = {2, 2};
  pair poly[maxm];  /* polyomino of current path (i.e., its tiles) */
  int congclass[maxd][4 * maxm];  /* congruence class table */
  int fill[maxm];  /* mapping from classes to tiles of a plane-filling path */
  pair cyc[maxp];  /* elements to check for cycling in F commands */
  int cycptr;  /* number of relevant elements in cyc */
  int count;  /* this many paths have been output */
  ⟨Subroutines 23⟩;
  main()
  {
    register int c, d, j, k, neg;
    register char *p, *q;
    long qq;
    int including = 0;
    ⟨Reset the current path to the unit path 3⟩;
    while (1) {
      if (including) ⟨Read a new command from infile 6⟩
      else ⟨Prompt the user for a new command 5⟩;
      ⟨Do the command in buf 7⟩;
      while (*p ≡ '␣') p++;
      if (*p ≠ '\n') printf("Junk␣at␣end␣of␣command␣has␣been␣ignored:␣%s", p);
    }
  done: ⟨Make sure that outfile is closed 39⟩
  }
```

3. ⟨Reset the current path to the unit path 3⟩ ≡

```
  s = 1, z.x = 1, z.y = 0;
  ⟨Clear the current auxiliary tables 4⟩;
```

This code is used in sections 2 and 9.

4. We compute the *poly* table only when it's needed. After it has been computed, *poly*[0] will be {1, 0}. Similarly, we compute the *congclass* and *fill* tables only when necessary.

⟨ Clear the current auxiliary tables 4 ⟩ ≡

```
poly[0].x = 0, congclass[0][0] = −1;
fill[0] = −1;
```

This code is used in sections 3 and 9.

5. ⟨ Prompt the user for a new command 5 ⟩ ≡

```
{
  printf(">␣"); fflush(stdout);
  fgets(buf, bufsize, stdin);
}
```

This code is used in section 2.

6. ⟨ Read a new command from *infile* 6 ⟩ ≡

```
{
  if (¬fgets(buf, bufsize, infile)) {
    including = 0;
    continue;
  }
}
```

This code is used in section 2.

7. ⟨ Do the command in *buf* 7 ⟩ ≡

```
for (p = buf; *p ≡ '␣'; p++) ;
if (*p ≡ '\n') {
  if (¬including) printf("Please␣type␣a␣command,␣or␣say␣q␣to␣quit.\n");
  continue;
}
if (including ∧ (vbose & verbose_echo)) printf("%s", buf);
switch (*p) {
case 'q': goto done;
case 'i':
  if (¬including) {
    for (p = buf + 1; *p ≡ '␣'; p++) ;
    for (q = p + 1; *q ≠ '\n'; q++) ;
    *q = '\0';
    if (infile = fopen(p, "r")) including = 1;
    else printf("Sorry␣---␣I␣couldn't␣open␣file␣`%s'␣for␣reading!\n", p);
  } else printf("Sorry;␣you␣can't␣include␣one␣file␣in␣another.\n");
case '%': continue;
case 'v': p++;
  ⟨ Scan an integer to k 8 ⟩;
  vbose = k; break;
  ⟨ Cases for nontrivial commands 9 ⟩;
}
```

This code is used in section 2.

8. ⟨ Scan an integer to k 8 ⟩ ≡

```
{
  while (*p ≡ '␣') p++;
  if (*p ≡ '-') neg = 1, p++;
  else neg = 0;
  for (k = 0; *p ≥ '0' ∧ *p ≤ '9'; p++) k = 10 * k + *p − '0';
  if (neg) k = −k;
}
```

This code is used in sections 7, 20, 25, 30, and 34.

9. ⟨ Cases for nontrivial commands 9 ⟩ ≡

```
case 'p': for (s = 0, z.x = z.y = 0, p++; *p ≥ '0' ∧ *p ≤ '3'; s++, p++) {
    if (s ≡ 0 ∧ *p ≠ '0') {
      printf("A␣path␣must␣start␣in␣direction␣0!\n");
      goto bad_path;
    } else if ((*p ⊕ s) & #1) {
      printf("Direction␣%c␣in␣this␣path␣has␣bad␣parity!\n", *p);
    bad_path: ⟨ Reset the current path to the unit path 3 ⟩; break;
    }
    ⟨ Set dir[s] and update z 11 ⟩;
  }
  if (s > maxm) {
  too_long: printf("Sorry,␣I␣can't␣deal␣with␣paths␣longer␣than␣%d;␣recompile␣me!\n", maxm);
    goto bad_path;
  }
  ⟨ Convert the directions to folds 13 ⟩;
finish_dirs: ⟨ Print the current folds 10 ⟩;
print_path_params: printf("␣s=%d,␣z=", s);
  ⟨ Print the complex number z 12 ⟩;
  printf("\n");
  ⟨ Clear the current auxiliary tables 4 ⟩;
  break;
```

See also sections 14, 17, 20, 25, 30, 34, and 37.

This code is used in section 7.

10. ⟨ Print the current folds 10 ⟩ ≡

```
  if (vbose & verbose_folds) printf("␣%s,", fold);
```

This code is used in sections 9 and 18.

11. ⟨ Set $dir[s]$ and update z 11 ⟩ ≡

```
  if (s < maxm) dir[s] = *p − '0';
  switch (*p) {
  case '0': z.x++; break;
  case '1': z.y++; break;
  case '2': z.x−−; break;
  case '3': z.y−−; break;
  }
```

This code is used in section 9.

12. ⟨ Print the complex number z 12 ⟩ ≡

```
  if (z.x) printf("%ld", z.x);
  else if (¬z.y) printf("0");
  if (z.y) {
    if (z.y ≡ 1) printf("+i");
    else if (z.y > 0) printf("+%ldi", z.y);
    else if (z.y ≡ −1) printf("-i");
    else printf("-%ldi", −z.y);
  }
```

This code is used in section 9.

13. ⟨ Convert the directions to folds 13 ⟩ ≡

```
  for (j = k = 0; j < s − 1; j++) fold[j] = ((dir[j + 1] − dir[j]) & #2 ? 'U' : 'D');
  fold[j] = '\0';
```

This code is used in section 9.

14. ⟨ Cases for nontrivial commands 9 ⟩ +≡

```
case 'D': case 'U': for (s = 0; *p ≡ 'D' ∨ *p ≡ 'U'; s++, p++)
    if (s < maxm) fold[s] = *p;
  if (++s > maxm) goto too_long;
finish_folds: ⟨ Convert the folds to directions 16 ⟩;
  ⟨ Print the current directions 15 ⟩;
  goto print_path_params;
```

15. ⟨ Print the current directions 15 ⟩ ≡

```
  if (vbose & verbose_dirs) {
    printf("␣");
    for (k = 0; k < s; k++) printf("%d", dir[k]);
  }
```

This code is used in sections 14 and 19.

16. ⟨ Convert the folds to directions 16 ⟩ ≡

```
  for (j = k = 0, z.x = z.y = 0; k < s; k++) {
    dir[k] = j;
    switch (j) {
    case 0: z.x++; break;
    case 1: z.y++; break;
    case 2: z.x−−; break;
    case 3: z.y−−; break;
    }
    j = (j + (fold[k] ≡ 'D' ? 1 : −1)) & #3;
  }
```

This code is used in sections 14 and 19.

17. ⟨ Cases for nontrivial commands 9 ⟩ +≡

```
case '*': p++;
  if (*p ≡ 'D' ∨ *p ≡ 'U') ⟨ Multiply by a folding sequence 18 ⟩
  else if (*p ≡ 'O') ⟨ Multiply by a direction sequence 19 ⟩
  else {
    printf("Improper␣multiplication!\n");
    break;
  }
```

18. ⟨ Multiply by a folding sequence 18 ⟩ ≡

```
{
  for (k = j = s - 1; *p ≡ 'D' ∨ *p ≡ 'U'; p++) {
    if (k + s ≥ maxm) goto too_long;
    fold[k++] = *p;
    if (j)
      for ( ; j; j--) fold[k++] = 'U' + 'D' - fold[j - 1];
    else
      for ( ; j < s - 1; j++) fold[k++] = fold[j];
  }
  fold[k] = '\0', s = k + 1;
  ⟨ Print the current folds 10 ⟩;
  goto finish_folds;
}
```

This code is used in section 17.

19. ⟨ Multiply by a direction sequence 19 ⟩ ≡

```
{
  for (k = j = s - 1, p++; *p ≥ '0' ∧ *p ≤ '3' ∧ ((*p ⊕ *(p - 1)) & #1); p++) {
    if (k + s ≥ maxm) goto too_long;
    fold[k++] = (*p - *(p - 1)) & #2 ? 'U' : 'D';
    if (j)
      for ( ; j; j--) fold[k++] = 'U' + 'D' - fold[j - 1];
    else
      for ( ; j < s - 1; j++) fold[k++] = fold[j];
  }
  fold[k] = '\0', s = k + 1;
  ⟨ Convert the folds to directions 16 ⟩;
  ⟨ Print the current directions 15 ⟩;
  goto finish_dirs;
}
```

This code is used in section 17.

20. #**define** *must_see*(*c*)
```
        while (*p ≡ '␣') p++; if (*p++ ≠ c) goto bad_command
#define check_tile(v)
        if (((v.x + v.y) & #1) ≡ 0) {
          printf("Bad␣tile␣(%ld,%ld)!\n", v.x, v.y); break; }
```

⟨ Cases for nontrivial commands 9 ⟩ +≡

```
default: ⟨ Scan an integer to k 8 ⟩;
  v.x = k;
  while (*p ≡ '␣') p++;
  if (*p++ ≠ ',') {
  bad_command: p−−;
    if (including ∧ ¬(vbose & verbose_echo))
      printf("Sorry,␣I␣don't␣understand␣the␣command␣%s", buf);
    else printf("Sorry,␣I␣don't␣understand␣that␣command!\n");
    break;
  }
  ⟨ Scan an integer to k 8 ⟩;
  v.y = k;
  check_tile(v);
  must_see('*');
  ⟨ Scan an integer to k 8 ⟩;
  w.x = k;
  must_see(',');
  ⟨ Scan an integer to k 8 ⟩;
  w.y = k;
  check_tile(w);
  ⟨ Compute u = v * w 21 ⟩;
  printf("␣%ld,%ld\n", u.x, u.y);
  break;
```

21. ⟨ Compute $u = v * w$ 21 ⟩ ≡

```
  ⟨ Set d to the type of w and e to the triply even neighbor 22 ⟩;
  u = sum(prod(ipower[(−d) & #3], v), prod(z, e));
```

This code is used in sections 20 and 25.

22. #**define** *typ*(*w*) (((*w.x* & #1) + ((*w.x* + *w.y*) & #2) + 3) & #3) /* yes it works! */

⟨ Set *d* to the type of *w* and *e* to the triply even neighbor 22 ⟩ ≡

```
  d = typ(w);
  e = sum(w, ipower[(2 − d) & #3]);
```

This code is used in section 21.

23. Complex addition, subtraction, and multiplication are easy.

⟨ Subroutines 23 ⟩ ≡

```
pair sum(pair a, pair b)
{
  pair res;
  res.x = a.x + b.x;
  res.y = a.y + b.y;
  return res;
}
pair diff(pair a, pair b)
{
  pair res;
  res.x = a.x − b.x;
  res.y = a.y − b.y;
  return res;
}
pair prod(pair a, pair b)
{
  pair res;
  res.x = a.x ∗ b.x − a.y ∗ b.y;
  res.y = a.x ∗ b.y + a.y ∗ b.x;
  return res;
}
```

See also section 24.

This code is used in section 2.

24. We also need complex division, but only when it is known to be exact.

```
#define norm(z) (z.x ∗ z.x + z.y ∗ z.y)
```

⟨ Subroutines 23 ⟩ +≡

```
pair quot(pair a, pair b)
{
  pair res;
  long n = norm(b);
  res.x = (a.x ∗ b.x + a.y ∗ b.y)/n;
  res.y = (−a.x ∗ b.y + a.y ∗ b.x)/n;
  return res;
}
```

25. ⟨ Cases for nontrivial commands 9 ⟩ +≡

```
case 'a': ⟨ Make sure poly is uptodate 26 ⟩;
  p++;
  must_see('*');
  ⟨ Scan an integer to k 8 ⟩;
  w.x = k;
  must_see(',');
  ⟨ Scan an integer to k 8 ⟩;
  w.y = k;
  check_tile(w);
  for (k = 0; k < s; k++) {
    v = poly[k];
    ⟨ Compute u = v * w 21 ⟩;
    printf("␣%ld,%ld", u.x, u.y);
  }
  printf("\n");
  break;
```

26. ⟨ Make sure poly is uptodate 26 ⟩ ≡

```
  if (¬poly[0].x) {
    for (k = 0, u.x = u.y = 0; k < s; k++) {
      v = u;
      switch (dir[k]) {
      case 0: u.x++; break;
      case 1: u.y++; break;
      case 2: u.x--; break;
      case 3: u.y--; break;
      }
      poly[k] = sum(u, v);
    }
  }
```

This code is used in sections 25, 32, and 33.

27. Congruence classes. Finally we get to the most interesting part of the program, which determines whether tiles are congruent.

Let $Z = (2 + 2i)z = A + Bi$, and let $D = \gcd(A, B)$. The first task, when we want to find the congruence class of a given tile w, is to reduce w modulo Z. To do this, we set up the *congclass* table as follows: We essentially find p and q such that $pA + qB = D$. Then we let $U = (A - Bi)Z/D = (A^2 + B^2)/D$ and $V = (pi + q)Z = (qA - pB) + Di$. By subtracting an appropriate multiple of V from w, we reduce its imaginary part, mod D. Then we can reduce the real part, mod u. If the result is $w' = x + yi$, the class of w is stored in $congclass[y \gg 1][x]$. It's OK to shift y right in this formula (saving a factor of 2 in space) because $x + y$ is always odd.

```
#define classof(w) congclass[w.y ≫ 1][w.x]

⟨ Make sure congclass is uptodate 27 ⟩ ≡
  if (congclass[0][0] < 0) {
    ⟨ Compute U and V 28 ⟩;
    for (j = 0; j < vv.y ≫ 1; j++)
      for (k = 0; k < uu.x; k++) congclass[j][k] = −1;
    for (c = j = 0; j < vv.y ≫ 1; j++)
      for (k = 0; k < uu.x; k++)
        if (congclass[j][k] < 0) {
          congclass[j][k] = c;
          v.x = k, v.y = 2 ∗ j + 1 − (k & #1);
          for (d = 1; d < 4; d++) {
            w = prod(v, ipower[d]);
            ⟨ Reduce w mod Z 29 ⟩;
            classof(w) = c;
          }
          c++;
        }
  }
```

This code is used in sections 30 and 33.

28. We essentially do Euclid's algorithm on the imaginary parts here. The roles of D and $(A^2 + B^2)/D$ in the formulas above are played by $vv.y$ and $uu.x$, respectively.

```
⟨ Compute U and V 28 ⟩ ≡
  uu = prod(z, sqrt8i), vv.x = −uu.y, vv.y = uu.x;
  if (uu.y < 0) uu.x = −uu.x, uu.y = −uu.y;
  if (vv.y < 0) vv.x = −vv.x, vv.y = −vv.y;
  while (uu.y) {
    while (vv.y ≥ uu.y) vv = diff(vv, uu);
    w = vv, vv = uu, uu = w;
  }
  if (uu.x < 0) uu.x = −uu.x;
```

This code is used in section 27.

29. ⟨ Reduce w mod Z 29 ⟩ ≡

```
{
  if (w.y < 0) {
    qq = (vv.y − 1 − w.y)/vv.y;
    w.x += qq ∗ vv.x, w.y += qq ∗ vv.y;
  } else {
    qq = w.y/vv.y;
    w.x −= qq ∗ vv.x, w.y −= qq ∗ vv.y;
  }
  if (w.x < 0) {
    qq = (uu.x − 1 − w.x)/uu.x;
    w.x += qq ∗ uu.x;
  } else {
    qq = w.x/uu.x;
    w.x −= qq ∗ uu.x;
  }
}
```

This code is used in sections 27, 31, 33, and 35.

30. ⟨ Cases for nontrivial commands 9 ⟩ +≡

```
case 'c': ⟨ Make sure congclass is uptodate 27 ⟩;
  p++;
  while (∗p ≡ '␣') p++;
  if (∗p ≡ '\n') ⟨ Show congruence classes for all of poly 32 ⟩
  else {
    ⟨ Scan an integer to k 8 ⟩;
    w.x = k;
    must_see(',');
    ⟨ Scan an integer to k 8 ⟩;
    w.y = k;
    ⟨ Show the congruence class and type of w 31 ⟩;
  }
  break;
```

31. ⟨ Show the congruence class and type of w 31 ⟩ ≡

```
  v = w;
  ⟨ Reduce w mod Z 29 ⟩;
  printf("␣%ld,%ld␣is␣%d_%d\n", v.x, v.y, classof(w), typ(v));
```

This code is used in sections 30 and 32.

32. ⟨ Show congruence classes for all of *poly* 32 ⟩ ≡

```
{
  ⟨ Make sure poly is uptodate 26 ⟩;
  for (k = 0; k < s; k++) {
    w = poly[k];
    ⟨ Show the congruence class and type of w 31 ⟩;
  }
}
```

This code is used in section 30.

33. A plane-filling path has the property that $s = |z|^2$ and all of its tiles are incongruent. In such cases we set *fill*[*j*] = *k* when *poly*[*k*] has class *j*.

```
⟨ Make sure fill is uptodate 33 ⟩ ≡
  if (fill[0] < 0 ∧ (norm(z) ≡ s)) {
    ⟨ Make sure poly is uptodate 26 ⟩;
    ⟨ Make sure congclass is uptodate 27 ⟩;
    for (j = 1; j < s; j++) fill[j] = −1;
    for (k = 0; k < s; k++) {
      w = poly[k];
      ⟨ Reduce w mod Z 29 ⟩;
      if (fill[classof(w)] ≥ 0) {
        fill[0] = −1;
        break;      /* abort, since it's not plane-filling */
      }
      fill[classof(w)] = k;
    }
  }
```

This code is used in section 34.

34. ⟨ Cases for nontrivial commands 9 ⟩ +≡

```
case 'f': case 'F': q = p++;
  ⟨ Make sure fill is uptodate 33 ⟩;
  if (fill[0] < 0) {
    printf("Sorry,␣the␣current␣path␣isn't␣plane-filling!\n");
    break;
  }
  ⟨ Scan an integer to k 8 ⟩;
  u.x = k;
  must_see(',');
  ⟨ Scan an integer to k 8 ⟩;
  u.y = k;
  check_tile(u);
  cyc[0] = u, cycptr = 1;
  while (1) {
    ⟨ Factor u 35 ⟩;
    if (*q ≡ 'f') break;
    ⟨ If we're in a cycle, break 36 ⟩;
    u = w;
  }
  break;
```

35. See my diamonds-and-dragons notes for the theory used here.

```
⟨ Factor u 35 ⟩ ≡
  w = u;
  ⟨ Reduce w mod Z 29 ⟩;
  v = poly[fill[classof(w)]];
  k = (typ(u) − typ(v)) & #3;
  e = quot(diff(u, prod(v, ipower[(−k) & #3])), z);
  w = sum(e, ipower[(−k) & #3]);
  printf("␣%ld,%ld␣=␣%ld,%ld␣*␣%ld,%ld\n", u.x, u.y, v.x, v.y, w.x, w.y);
```

This code is used in section 34.

36. The element in *cyc*[0] always has the smallest magnitude we've seen so far. If $|w| = 1$, we're done, because $1 * w = w$ in that case.

⟨ If we're in a cycle, **break** 36 ⟩ ≡

```
if (norm(w) ≡ 1) break;
if (norm(w) < norm(cyc[0])) cyc[0] = w, cycptr = 1;
else {
  for (k = 0, cyc[cycptr] = w; w.x ≠ cyc[k].x ∨ w.y ≠ cyc[k].y; k++) ;
  if (k < cycptr) break;
  cycptr++;
}
```

This code is used in section 34.

37. Graphic output. Finally, we have a rudimentary way to visualize general dragon curves, via METAPOST.

```
⟨ Cases for nontrivial commands 9 ⟩ +≡
case 'm': ⟨ Make sure that outfile is open 38 ⟩;
  count++, p++;
  fprintf(outfile, "\nbeginfig(%d)\n␣O", count);
  for (k = 0; k < s − 1; k++) {
    if (k % 32 ≡ 31) fprintf(outfile, "\n");
    fprintf(outfile, "␣%c", fold[k]);
  }
  fprintf(outfile, ";\nendfig;\n");
  break;
```

38. ⟨ Make sure that *outfile* is open 38 ⟩ ≡

```
  if (¬outfile) {
    outfile = fopen(metapost_name, "w");
    if (¬outfile) {
      fprintf(stderr, "Oops,␣I␣can't␣open␣%s␣for␣output!␣Have␣to␣quit...\n", metapost_name);
      exit(−99);
    }
    fprintf(outfile, "%%␣Output␣from␣DRAGON-CALC\n");
    fprintf(outfile, "numeric␣dd;␣pair␣rr,ww,zz;␣rr=(10bp,0);␣%%␣adjust␣rr␣if␣desired!\n");
    fprintf(outfile, "def␣D␣=␣dd:=dd+90;␣ww:=zz;␣zz:=ww+rr␣rotated␣dd;␣draw␣ww--zz;␣enddef;\n");
    fprintf(outfile, "def␣U␣=␣dd:=dd-90;␣ww:=zz;␣zz:=ww+rr␣rotated␣dd;␣draw␣ww--zz;␣enddef;\n");
    fprintf(outfile, "def␣O␣=␣zz:=origin;␣dd:=-90;␣D;␣enddef;\n");
  }
```

This code is used in section 37.

39. ⟨ Make sure that *outfile* is closed 39 ⟩ ≡

```
  if (outfile) {
    fprintf(outfile, "\nbye.\n");
    fclose(outfile);
    fprintf(stderr, "METAPOST␣output␣for␣%d␣paths␣written␣on␣%s.\n", count, metapost_name);
    outfile = Λ;
  }
```

This code is used in section 2.

40. Index.

DRAGON-CALC