1. Introduction. Nob sent me another problem today, and as usual I couldn't put it down. The challenge is to count how many ways we can pack isosceles triangles into a regular decagon. There are two kinds of triangles, one with base angles 72° and the other with base angles 36°. We are given 25 of the former and 5 of the latter.

These triangles have nice properties, which makes the problem appealing. If we represent angles as multiples of 36°, the large triangles have angles 2, 2, 1 and sides 1, 1, ϕ^{-1}; the small triangles have angles 1, 1, 3 and sides ϕ^{-2}, ϕ^{-2}, ϕ^{-1}. The area of the 10-gon, which has unit sides, is $10\phi^2 A$, where A is the area of a large triangle. The small triangle has $\phi^{-3}A$. And it turns out that $25 + 5\phi^{-3} = 10\phi^2$, because $\phi^{-3} = \sqrt{5} - 2$ and $2\phi^2 = \sqrt{5} + 3$.

My backtrack program works by maintaining a residual polygon-to-be-filled, since I don't think I can use cells as I do with polyominoes or polyhexes. Each polygon is represented as a cyclic list of the form a_0, x_0, a_1, x_1, a_2, ..., a_n, x_n, a_0, where the a's are angles and the x's are lengths. When $a_k < 5$, angle a_k is a convex corner; at each stage we choose a convex corner and replace (a_k, x_k) by $(a_k - \theta_1, s_1, 10 - \theta_2, s_2, 5 - \theta_3, x_k - s_3)$, where $(\theta_1, s_1, \theta_2, s_2, \theta_3, s_3)$ is one of the six ways to place a triangle at that corner. Adjustments are then made so that all the a's and x's are positive.

Here are transformations that convert to positive, when possible: If $x_k = 0$ we replace $(a_{k-1}, 0, a_k)$ by $(a_{k-1} + a_k - 5)$. If $x_k < 0$ we replace (a_{k-1}, x_k, a_k) by $(a_{k-1} + 5, -x_k, a_k - 5)$. If $a_k = 0$ we replace $(a_{k-1}, x_{k-1}, 0, x_k)$ by $(a_{k-1} - 5, x_k - x_{k-1})$. When all x's are positive and no a's are zero, the polygon is erroneous if any a_k is negative or ≥ 10.

I thought about replacing $(x_{k-1}, 5, x_k)$ by $(x_{k-1} + x_k)$, but decided against it. Later, after watching the method in action, I decided to do that replacement after all.

The first draft of this program seemed to work fine on special cases, but when I ran it to completion on the full decagon problem it missed some of the solutions. (It found only 5463628, while I knew from symmetry considerations that the correct total would have the form $20x + 32$. The correct total is 5464292.) A serious bug in my original reasoning, explained below, had to be corrected, hence the program is now considerably more complicated than I thought it would be.

```
#define big 25 /* this many big triangles must be placed */
#define small 5 /* and this many small ones */
#define total_req (big + small)
#define eps (argc > 2) /* causes PostScript output, one file per solution */
#define debug (argc > 3) /* enables regular consistency checks */
#define verbose (argc > 4) /* causes extra printout */
#define sin SIN /* get around bug in clang */
#define cos COS /* get around bug in clang */
#include <stdio.h>
  ⟨ Preprocessor definitions ⟩
  ⟨ Type definitions 2 ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 3 ⟩
  main(argc, argv)
      int argc;
      char *argv[];
  {
    int i, j, k;
    register int l; /* level of backtracking */
    int vert = 0; /* number of vertices known */
    int count = 0, interval = 1, eps_interval = 1, big_need = big, small_need = small;
    register node *p, *q, *pp, *qq, *r, *rr;
    if (argc > 1) {
```

```
    sscanf(argv[1], "%d", &interval);
    if (eps) sscanf(argv[2], "%d", &eps_interval);
  }
  ⟨ Initialize the tables 7 ⟩;
  ⟨ Backtrack through all solutions 12 ⟩;
  printf("Altogether␣%d␣solutions.\n", count);
}
```

2. Polygons. Circular lists that represent polygons are doubly linked in a straightforward way. The only slightly tricky thing is that we represent lengths in the form $s\phi^{-1} + t\phi^{-2}$, where s and t are integers.

Each angle contains a vertex number. The first few vertices come from the initial input. Each level of backtracking adds two more, some of which will be identified with earlier vertices.

```
⟨ Type definitions 2 ⟩ ≡
  typedef struct node_struct {
    struct node_struct *next, *prev;
    int s;      /* angle, or first component of length */
    int t;      /* vertex number, or second component of length */
    int dir;    /* direction to the next vertex (used only in angle nodes */
  } node;
```

This code is used in section 1.

3. The nodes are allocated with a normal sort of available space list.

```
⟨ Subroutines 3 ⟩ ≡
  node *get_avail( )
  {
    register node *p;
    if (avail) {
      p = avail;
      avail = p→next;
    }
    else if (next_node ≡ bad_node) {
      printf("ALLOCATING...\n");      /* temporary */
      p = (node *) calloc(1000, sizeof(node));
      if (p ≡ Λ) {
        printf("Out␣of␣memory!\n");
        exit(−1);
      }
      next_node = p + 1;
      bad_node = p + 1000;
    }
    else p = next_node++;
    return p;
  }
```

See also sections 35 and 38.

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
  node *avail;      /* a node that was recycled */
  node *next_node;  /* the next node not yet used */
  node *bad_node;   /* end of currently allocated block of nodes */
```

See also sections 5, 6, 8, 11, 13, 33, and 36.

This code is used in section 1.

5. Here are the six ways to place triangles—three ways each.

```
⟨ Global variables 4 ⟩ +≡
  int triang[6][9] = {{2, 1, 1, 1, 1, 1, 2, 1, 0}, {1, 1, 1, 2, 1, 0, 2, 1, 1}, {2, 1, 0, 2, 1, 1, 1, 1, 1},
      {1, 0, 1, 3, 0, 1, 1, 1, 0}, {3, 0, 1, 1, 1, 0, 1, 0, 1}, {1, 1, 0, 1, 0, 1, 3, 0, 1}};
```

6. A complication mentioned later will make it necessary to work with more than one polygon in certain cases. So in general we assume that there is a stack of polygons to be filled, pointed to by $poly[0]$, ..., $poly[top]$; we will currently be working on $poly[top]$.

```
⟨ Global variables 4 ⟩ +≡
  node *poly[total_req];      /* polygons to be filled */
  int top;      /* index to the topmost one */
```

7. The *dir* fields of the polygon will always satisfy the invariant relation $p\rightarrow dir = p\rightarrow prev\rightarrow prev\rightarrow dir + 5 - p\rightarrow s$, modulo 10, when p is an angle node. Moreover, the sum of $5 - s$ over all angle nodes of a polygon will equal 10.

```
#define init_pts 10
⟨ Initialize the tables 7 ⟩ ≡
  p = get_avail( );
  poly[0] = p;
  for (j = 0; j < init_pts; j++) {
    q = get_avail( );
    p→s = 4;
    p→t = vert++;
    p→dir = j;
    p→next = q;
    q→prev = p;
    p = (j < init_pts − 1 ? get_avail( ) : poly[0]);
    q→s = 1;
    q→t = 1;
    q→next = p;
    p→prev = q;
  }
```

See also sections 9, 10, and 37.

This code is used in section 1.

8. Coordinates. The method I sketched in the introduction sounded good to me at first, but it has a fatal flaw. The problem occurs when we try to branch at a convex corner that has already been covered by another part of the polygon. (Consider, for example, the case where the polygon consists of two nonadjacent triangles, separated by a crooked path traced in both directions so that it contributes nothing to the total area.) To avoid this bug, it is necessary to know more than the sequence of lengths and angles; we need to be able to tell when two vertices are identical as points in the plane.

Floating-point arithmetic could be used for this purpose, with care, but I prefer to use exact integer arithmetic. We can regard each vertex as a point in the complex plane, represented in the form $\sum_{k=0}^{9} x_k\zeta^k$, where the x's are integers and $\zeta = e^{\pi i/5}$ is a 10th root of unity. This is possible because the location of each point is the location of a previous point plus a number of the form $(s+t\phi^{-1})\zeta^k$, and because $\phi^{-1} = \zeta^2 - \zeta^3$. (We scale all dimensions up by ϕ for convenience.) Such a representation is highly redundant, because ζ satisfies the equation $\zeta^4 - \zeta^3 + \zeta^2 - \zeta + 1 = 0$; but it is unique if $x_4 = \cdots = x_9 = 0$, because that equation is irreducible over the rationals. (See *Seminumerical Algorithms*, exercise 4.6.2-32.)

The absolute values of x_0, x_1, x_2, and x_3 will be small in any covering, because they are obtained by adding small numbers of the form $(s + t\phi^{-1})\zeta^k$ for at most 30 values of (s, t, k). So we will represent each point internally as a single 32-bit number,

$$(x_3 + 128)\cdot 2^{24} + (x_2 + 128)\cdot 2^{16} + (x_1 + 128)\cdot 2^8 + x_0 + 128\,.$$

To compute the coordinates of each point it suffices to have short tables for the amounts to add to the representation when we want to add ζ^k or $\phi^{-1}\zeta^k$.

#define *pack*(a,b,c,d) $(a \ll 24) + (b \ll 16) + (c \ll 8) + d$

⟨ Global variables 4 ⟩ +≡

```
unsigned int x[init_pts + 2 * total_req];      /* the coordinates */
unsigned int delta_s[10] = {pack(0,0,0,1), pack(0,0,1,0), pack(0,1,0,0), pack(1,0,0,0), pack(1,-1,1,
    -1),
    pack(0,0,0,-1), pack(0,0,-1,0), pack(0,-1,0,0), pack(-1,0,0,0), pack(-1,1,-1,1)};
unsigned int delta_t[10] = {pack(-1,1,0,0), pack(0,1,-1,1), pack(1,-1,1,0), pack(0,0,1,-1), pack(0,
    1,-1,0),
    pack(1,-1,0,0), pack(0,-1,1,-1), pack(-1,1,-1,0), pack(0,0,-1,1), pack(0,-1,1,0)};
```

9. ⟨ Initialize the tables 7 ⟩ +≡

```
x[0] = pack(128,128,128,128);
for (j = 1, p = poly[0]; j < init_pts; j++, p = p→next→next)
  x[j] = x[j-1] + p→next→s * delta_s[p→dir] + p→next→t * delta_t[p→dir];
```

10. We will occasionally need to decide whether a number of the form $s + t\phi^{-1}$ is positive, negative, or zero. There is an interesting recursive way to make this test: The answer is obvious unless $st < 0$; and in the latter case $s + t\phi^{-1}$ has the same sign as $s\phi + t = s + t + s\phi^{-1}$, so we can replace (s, t) by the pair $(s + t, s)$.

But for our purposes it is sufficient simply to test the sign of $13s + 8t$, since s and t will not get large enough to make this trick fail.

⟨ Initialize the tables 7 ⟩ +≡

```
for (j = 0; j < 6; j++) {
  thresh1[j] = 13 * triang[j][1] + 8 * triang[j][2];
  thresh3[j] = 13 * triang[j][7] + 8 * triang[j][8];
}
```

11. ⟨ Global variables 4 ⟩ +≡

```
int thresh1[6];    /* encoded version of the first length */
int thresh3[6];    /* encoded version of the third length */
```

12. Backtracking.

Two heuristics allow a quick decision: A triangle position is impossible if the existing angle is too small, or if the existing side is too small and between two convex angles (i.e., between angles that each have $s < 5$).

```
⟨ Backtrack through all solutions 12 ⟩ ≡
  l = 0;
newlev:
  if (l ≡ total_req) {
    if (top < 0) ⟨ Record a solution 30 ⟩;
    goto backup;
  }
  ht[l] = top;
  lb[l] = (big_need ≡ 0 ? 3 : 0);
  ub[l] = (small_need ≡ 0 ? 3 : 6);
  ⟨ Find corner to branch on 14 ⟩;
  way[l] = lb[l];
tryit: j = way[l];
  p = choice[l];
  if (p→s < triang[j][0]) goto nogood;      /* angle is too small */
  q = p→next;
  r = q→next;
  if (r→s < 5) {
    if (13 * q→s + 8 * q→t < thresh3[j]) goto nogood;      /* side after p is too small */
    if (13 * q→s + 8 * q→t ≡ thresh3[j] ∧ r→s < triang[j][6]) goto nogood;
  }
  if (p→s ≡ triang[j][0] ∧ p→prev→prev→s < 5) {
    if (13 * p→prev→s + 8 * p→prev→t < thresh1[j]) goto nogood;      /* side preceding p is too small */
    if (13 * p→prev→s + 8 * p→prev→t ≡ thresh1[j] ∧ p→prev→prev→s < triang[j][3]) goto nogood;
  }
  ⟨ Install triangle j at position choice[l] 15 ⟩;
  if (debug) ⟨ Examine the current choice and its ramifications 27 ⟩;
  if (way[l] < 3) big_need−−; else small_need−−;
  l++;
  vert += 2;
  goto newlev;
nogood:
  if (++way[l] < ub[l]) goto tryit;
backup:
  if (l ≡ 0) goto done;
  l−−;
  vert −= 2;
  if (way[l] < 3) big_need++; else small_need++;
  ⟨ Undo the changes made in level l 28 ⟩;
  goto nogood;
done:
```

This code is used in section 1.

13. ⟨ Global variables 4 ⟩ +≡

```
node *bhoice[total_req];      /* convex corner where branching occurs */
int way[total_req];      /* which way we tried to place a triangle */
node *choice[total_req];      /* where we tried to place it */
node *save[total_req];      /* polygons to restore when backtracing */
int lb[total_req], ub[total_req];      /* bounds on way */
int ht[total_req];      /* size of stack when the choice was made */
```

14. After experimenting with simpler rules here, I decided it was best to choose a corner that results in the fewest possibilities with respect to the two heuristics just mentioned.

We also must restrict the branch point to a convex corner. Otherwise we might miss important solutions.

⟨ Find corner to branch on 14 ⟩ ≡

```
for (p = poly[top], rr = p→prev→prev, i = 10000000; ; rr = p, p = r) {
  q = p→next;
  r = q→next;
  if (p→s < 5) {
    for (j = lb[l], k = 0; j < ub[l]; j++)
      if (p→s ≥ triang[j][0] ∧ (r→s > 5 ∨ (13 * (q→s) + 8 * (q→t)) ≥ thresh3[j]) ∧ (p→s > triang[j][0] ∨ rr→s >
          5 ∨ 13 * p→prev→s + 8 * p→prev→t ≥ thresh1[j])) k++;
    if (k < i) i = k, pp = p;
  }
  if (r ≡ poly[top]) break;
}
choice[l] = pp;
```

This code is used in section 12.

15. ⟨ Install triangle j at position $choice[l]$ 15 ⟩ ≡

⟨ Copy the current polygon and save the old version 16 ⟩;
⟨ Create new vertices pp, qq, and the line r between them 17 ⟩;
⟨ Insert qq at the choice point; split into two polygons if necessary 18 ⟩;
⟨ Insert pp at the choice point; split into two polygons if necessary 22 ⟩;

This code is used in section 12.

16. My first draft program avoided full copying by copying only the nodes that changed. It was rather elegant, but alas—it implemented a bad algorithm. The correct algorithm manipulates the lists in more complex ways, hence partial copying is no longer feasible; it would be too complicated.

```
⟨ Copy the current polygon and save the old version 16 ⟩ ≡
  save[l] = poly[top];
  rr = get_avail( );
  for (pp = rr, p = choice[l]; ; p = p→next) {
    pp→s = p→s;
    pp→t = p→t;
    pp→dir = p→dir;
    qq = get_avail( );
    pp→next = qq;
    qq→prev = pp;
    p = p→next;
    qq→s = p→s;
    qq→t = p→t;
    if (p→next ≡ choice[l]) break;
    pp = get_avail( );
    qq→next = pp;
    pp→prev = qq;
  }
  qq→next = rr;
  rr→prev = qq;      /* poly[top] has not been updated */
```

This code is used in section 15.

17.

```
⟨ Create new vertices pp, qq, and the line r between them 17 ⟩ ≡
  pp = get_avail( );
  pp→t = vert;
  qq = get_avail( );
  qq→t = vert + 1;
  r = get_avail( );
  r→s = triang[j][4];
  r→t = triang[j][5];
  pp→next = r;
  r→prev = pp;
  r→next = qq;
  qq→prev = r;
  k = (rr→dir + triang[j][0] + 100) % 10;      /* direction from the choice node to pp */
  x[vert] = x[rr→t] + triang[j][1] * delta_s[k] + triang[j][2] * delta_t[k];
  k = (k + triang[j][3] + 5) % 10;
  pp→dir = k;
  pp→s = 10 − triang[j][3];
  x[vert + 1] = x[vert] + triang[j][4] * delta_s[k] + triang[j][5] * delta_t[k];
```

This code is used in section 15.

18. We maintain the following conditions in the polygons: (1) All angles s are in the range $s \le 9$, $s \ne 0$, $s \ne 5$. (2) All vertices are at distinct points in the plane.

We don't bother to check that the new polygon doesn't intersect itself, except when the point of intersection is at a vertex. Self-intersecting polygons of other types will not lead to solutions, since they will doubly cover some points and will therefore be incompletely filled when we have used up our quota of triangles. If we checked for self-intersection, the search tree would be smaller, but I think the total search time would be longer, because of the extra time spent in checking.

Previous steps have created nodes rr, pp, qq for the new triangle. Node rr is the choice point in the current polygon; we have not yet linked pp and qq into that polygon, nor have we recorded anything about it in $poly[top]$. The triangle will be inserted in such a way that the line from rr to qq runs along the existing line from rr to its successor.

```
⟨ Insert qq at the choice point; split into two polygons if necessary 18 ⟩ ≡
  q = rr→next;
  p = q→next;      /* q is the line between rr and p */
  k = 13 * q→s + 8 * q→t;
  if (k ≡ thresh3[j]) ⟨ Connect pp directly to existing vertex p ≡ qq 19 ⟩
  else {
    if (k > thresh3[j]) {      /* the line from rr to p is longer than needed */
      q→s -= triang[j][7];
      q→t -= triang[j][8];
      qq→s = 5 - triang[j][6];
    }
    else {      /* the line from rr to p is shorter than from rr to qq */
      p→s -= 5;      /* we know this is > 0 */
      q→s = triang[j][7] - q→s;
      q→t = triang[j][8] - q→t;
      qq→s = 10 - triang[j][6];
    }
    qq→next = q;
    q→prev = qq;
    qq→dir = pp→dir + 5 - qq→s;
    for (p = p→next→next; p ≠ rr; p = p→next→next)
      if (x[p→t] ≡ x[vert + 1]) {      /* qq coincides with a previous point */
        ⟨ Split off a polygon at position qq ≡ p 20 ⟩;
        break;
      }
  }
  ⟨ Remove angle 0 or 5 at p, if present after the qq insertion stage 21 ⟩;
```

This code is used in section 15.

19. Node r is the line between pp and qq; node q is the line between rr and p. We've discovered that these lines are identical; so we discard q and qq. If the new angle at p is negative, backtracking will occur at level $l+1$, so we don't bother to check for that unlikely event.

```
⟨ Connect pp directly to existing vertex p ≡ qq 19 ⟩ ≡
  {
    r→next = p;
    p→prev = r;
    q→next = qq;
    qq→next = avail;
    avail = q;
    p→s -= triang[j][6];
  }
```

This code is used in section 18.

20. This part of the program is the price I had to pay to fix my original ill-understood algorithm. We separate out the subpolygon $(qq, \ldots, p-pred, qq)$ and connect pp to p and its successors rather than to qq.

The split-off polygon might not have a winding number of 1 (I mean, the sum of its exterior angles $5-s$ might not be 10). Then it might have no convex corners. But in such a case, the remaining polygon would have a negative angle, so we would never have to look at the split-off polygon (which is lower in the stack). This reasoning is somewhat subtle, and the case may never arise, but I do want to record it here because I think it is correct and because I don't want to imply that I ignored a potential problem.

```
⟨ Split off a polygon at position qq ≡ p 20 ⟩ ≡
  qq→prev = p→prev;
  p→prev→next = qq;
  k = qq→s + p→s − 10;
  qq→s = (p→prev→prev→dir + 105 − qq→dir) % 10;      /* qq→dir stays the same */
  p→prev = r;
  r→next = p;
  p→s = k − qq→s;      /* if negative, we'll discover a problem soon */
  k = qq→s;
  if (k ≡ 0 ∨ k ≡ 5) {      /* recall that q = qq→next */
    qq = qq→prev;
    if (k ≡ 5) qq→s += q→s, qq→t += q→t;
    else if (13 * (qq→s − q→s) + 8 * (qq→t − q→t) < 0)
      qq→s = q→s − qq→s, qq→t = q→t − qq→t, qq→prev→s -= 5, qq→prev→dir += 5;
    else qq→s -= q→s, qq→t -= q→t, q→next→s -= 5;
    qq→next = q→next;
    q→next→prev = qq;
    qq = q→next;
    q→next = avail;
    avail = q→prev;
  }
  poly[top++] = qq;
```

This code is used in section 18.

21. If the new angle at p is zero, there's a possibility that point pp is coincident with the successor of p. In that case we temporarily retain both points, with a line of length 0 between them.

⟨ Remove angle 0 or 5 at p, if present after the qq insertion stage 21 ⟩ ≡

```
if (p→s ≡ 0 ∨ p→s ≡ 5) {
  q = p→next;      /* at this point r = p→prev */
  if (p→s ≡ 5) r→s += q→s, r→t += q→t;
  else if (13 * (r→s − q→s) + 8 * (r→t − q→t) ≤ 0) r→s = q→s − r→s, r→t = q→t − r→t, pp→s −= 5, pp→dir += 5;
      /* pp = r→prev */
  else r→s −= q→s, r→t −= q→t, q→next→s −= 5;
  r→next = q→next;
  q→next→prev = r;
  q→next = avail;
  avail = p;
}
```

This code is used in section 18.

22. How do things stand now? We have a path from pp to rr, and r is the line out of pp; the length of r might be zero. Variables p, q, and qq are currently unused. The remaining task is to insert the line from rr to pp.

The happiest situation occurs when we find that the former angle at rr is just the angle of the new triangle, and the vertex preceding rr coincides with pp, and the length of r is zero. This means the current polygon has been completely filled, and we've made progress!

```
⟨ Insert pp at the choice point; split into two polygons if necessary 22 ⟩ ≡
  if (rr→s ≡ triang[j][0]) {
    q = rr→prev;
    p = q→prev;      /* q is the line between p and rr */
    k = 13 * q→s + 8 * q→t;
    if (k ≡ thresh1[j]) {
      if (p ≡ pp→next→next) {      /* hurray */
        rr→next = avail;
        avail = pp;
        top−−;
        goto insert_done;
      }
      ⟨ Connect existing vertex p ≡ pp directly to the path following pp 23 ⟩;
      goto insert_almost_done;
    }
    else ⟨ Connect vertex p to pp, removing node rr 24 ⟩;
  }
  else {
    rr→s −= triang[j][0];
    rr→dir += triang[j][0];
    q = get_avail( );
    q→prev = rr;
    rr→next = q;
    q→s = triang[j][1];
    q→t = triang[j][2];
    q→next = pp;
    pp→prev = q;
    p = rr;
  }
  for (p = p→prev→prev; p ≠ pp; p = p→prev→prev)
    if (x[p→t] ≡ x[vert]) {      /* pp coincides with a previous point */
      ⟨ Split off a polygon at position pp ≡ p 25 ⟩;
      break;
    }
insert_almost_done: ⟨ Remove angle 0 or 5 at p, if present after the pp insertion stage 26 ⟩;
  poly[top] = p;
insert_done:
```

This code is used in section 15.

23. At this point $q = rr\rightarrow prev$. We will recycle nodes q, rr, and pp.

⟨ Connect existing vertex $p \equiv pp$ directly to the path following pp 23 ⟩ ≡

```
p→next = pp→next;
pp→next→prev = p;
p→s -= 10 - pp→s;
p→dir = pp→dir;
pp→next = avail;
rr→next = pp;
avail = q;
```

This code is used in section 22.

24. ⟨ Connect vertex p to pp, removing node rr 24 ⟩ ≡

```
{
  if (k > thresh1[j]) {      /* the line from p to rr is longer than needed */
    q→s -= triang[j][1];
    q→t -= triang[j][2];
    pp→s -= 5;
  }
  else {      /* it's shorter than from rr to pp */
    p→s -= 5;      /* we know this is > 0 */
    p→dir += 5;
    q→s = triang[j][1] - q→s;
    q→t = triang[j][2] - q→t;
  }
  q→next = pp;
  pp→prev = q;
  rr→next = avail;
  avail = rr;
}
```

This code is used in section 22.

25. ⟨ Split off a polygon at position $pp \equiv p$ 25 ⟩ ≡

```
qq = pp→next;
if (qq→next ≡ p) {      /* remove trivial length-0 leg */
  p→s += pp→s − 5;
  p→prev = pp→prev;
  pp→prev→next = p;
  qq→next = avail;
  avail = pp;
}
else {
  q = p→next;
  p→next = qq;
  qq→prev = p;
  pp→next = q;
  q→prev = pp;
  k = pp→dir;
  pp→dir = p→dir;
  p→dir = k;
  k = p→s + pp→s − 10;
  pp→s = (pp→prev→prev→dir + 105 − pp→dir) % 10;
  p→s = k − pp→s;      /* if negative, we'll catch it */
  k = pp→s;
  if (k ≡ 0 ∨ k ≡ 5) {
    pp = pp→prev;
    if (k ≡ 5) pp→s += q→s, pp→t += q→t;
    else if (13 ∗ (pp→s − q→s) + 8 ∗ (pp→t − q→t) < 0)
      pp→s = q→s − pp→s, pp→t = q→t − pp→t, pp→prev→s −= 5, pp→prev→dir += 5;
    else pp→s −= q→s, pp→t −= q→t, q→next→s −= 5;
    pp→next = q→next;
    q→next→prev = pp;
    pp = q→next;
    q→next = avail;
    avail = q→prev;
  }
  poly[top++] = pp;
}
```

This code is used in section 22.

26. I sure wish I had been able to figure out an elegant way to get rid of so many special cases. Sigh. This, at least, is the last.

The polygon we're left with consists entirely of old vertices, so they are distinct.

```
⟨ Remove angle 0 or 5 at p, if present after the pp insertion stage 26 ⟩ ≡
  if (p→s ≡ 0 ∨ p→s ≡ 5) {
    q = p→next;
    r = p→prev;
    if (p→s ≡ 5) r→s += q→s, r→t += q→t;
    else if (13 ∗ (r→s − q→s) + 8 ∗ (r→t − q→t) ≤ 0)
      r→s = q→s − r→s, r→t = q→t − r→t, r→prev→s −= 5, r→prev→dir += 5;
    else r→s −= q→s, r→t −= q→t, q→next→s −= 5;
    r→next = q→next;
    q→next→prev = r;
    q→next = avail;
    avail = p;
    p = r→next;
  }
```

This code is used in section 22.

27.

```
⟨ Examine the current choice and its ramifications 27 ⟩ ≡
  {
    int badsums = 0, negangle = 0;
    if (verbose) printf("Level␣%d:␣vertex␣%d␣with␣triangle␣%d\n", l, choice[l]→t, way[l]);
    for (j = ht[l]; j ≤ top; j++) {
      for (p = poly[j], k = p→prev→prev→dir, i = 0; ; ) {
        q = p→next;
        if (q→prev ≠ p) printf("␣badlink!");
        if (verbose) printf("␣%d(%d)", p→t, p→s);
        if (p→s ≡ 0 ∨ p→s ≡ 5) printf("␣badangle!");
        if (p→s < 0 ∧ j ≡ top) negangle = 1;
        if ((k + 105 − p→s − p→dir) % 10 ≠ 0) printf("baddir!");
        i += 5 − p→s;
        k = p→dir;
        p = q→next;
        if (p→prev ≠ q) printf("␣badlink!");
        if (verbose) printf("␣%d,%d", q→s, q→t);
        if (p ≡ poly[j]) break;
      }
      if (i ≠ 10) badsums++;
      if (verbose) printf("\n");
    }
    if (badsums ∧ ¬negangle) printf("␣badsum!\n");
  }
```

This code is used in section 12.

28. ⟨ Undo the changes made in level l 28 ⟩ ≡

```
for (j = top; j ≥ ht[l]; j−−) {
  poly[j]→prev→next = avail;
  avail = poly[j];
}
top = ht[l];
poly[top] = save[l];
```

This code is used in section 12.

29. Solutions. The terminal gets only a minimum of information from which a tiling can be constructed.

30. ⟨ Record a solution 30 ⟩ ≡

```
{
  count++;
  if (count % interval ≡ 0) {
    printf("%d:", count);
    for (j = 0; j < l; j++) printf("␣%d-%d", choice[j]→t, way[j]);
    printf("\n");
  }
  if (eps ∧ count % eps_interval ≡ 0) ⟨ Output a PostScript version 31 ⟩;
}
```

This code is used in section 12.

31. Here's how we get encapsulated PostScript output for a solution.

⟨ Output a PostScript version 31 ⟩ ≡

```
{
  ⟨ Open eps_file for output, and define a triangle subroutine 32 ⟩;
  for (j = 0; j < l; j++) ⟨ Output the triangle for level j 34 ⟩;
  fclose(eps_file);
}
```

This code is used in section 30.

32. The PostScript '`t`' subroutine simply draws a triangle between three given points.

⟨ Open *eps_file* for output, and define a triangle subroutine 32 ⟩ ≡

```
sprintf(buffer, "%s.%d", argv[0], count);
eps_file = fopen(buffer, "w");
if (¬eps_file) {
  printf("Can't␣open␣file␣%s!\n", buffer);
  exit(−4);
}
fprintf(eps_file, "%%!\n%%%%BoundingBox:␣%d␣%d␣%d␣%d\n", bbxlo − 1, bbylo − 1, bbxhi + 1, bbyhi + 1);
fprintf(eps_file, "/t␣{␣moveto␣lineto␣lineto␣closepath␣stroke␣}␣bind␣def\n");
```

This code is used in section 31.

33. ⟨ Global variables 4 ⟩ +≡

```
char buffer[100];      /* output file name (e.g. 'decagon.1') */
FILE *eps_file;
int bbxlo, bbylo, bbxhi, bbyhi;      /* PostScript bounding box coordinates */
```

34. ⟨ Output the triangle for level *j* 34 ⟩ ≡

```
{
  print_coord(choice[j]→t);
  print_coord(init_pts + j + j);
  print_coord(init_pts + 1 + j + j);
  fprintf(eps_file, "␣t\n");
}
```

This code is used in section 31.

35. ⟨ Subroutines 3 ⟩ +≡

```
print_coord(j)
    int j;
{
  register float xx, yy;
  register int k;
  register unsigned b;
  for (xx = yy = 0.0, k = 0, b = x[j]; k < 4; k++, b >>= 8) {
    xx += ((int)(b & #ff) − 128) * cos[k];
    yy += ((int)(b & #ff) − 128) * sin[k];
  }
  fprintf(eps_file, "␣%d␣%d", (int) xx, (int) yy);
}
```

36.

```
#define cos36 80.9017 /* 100 times cos 36° */
#define cos72 30.9017 /* 100 times cos 72° */
#define sin36 58.7785 /* 100 times sin 36° */
#define sin72 95.1057 /* 100 times sin 72° */
```

⟨ Global variables 4 ⟩ +≡

```
float cos[] = {100.0, cos36, cos72, −cos72};
float sin[] = {0.0, sin36, sin72, sin72};
```

37. ⟨ Initialize the tables 7 ⟩ +≡

```
{
  float xx, yy;
  unsigned b;
  bbxlo = bbylo = 100000;
  bbxhi = bbyhi = −100000;
  for (j = 0; j < init_pts; j++) {
    for (xx = yy = 0.0, k = 0, b = x[j]; k < 4; k++, b >>= 8) {
      xx += ((int)(b & #ff) − 128) * cos[k];
      yy += ((int)(b & #ff) − 128) * sin[k];
    }
    if ((int) xx < bbxlo) bbxlo = (int) xx;
    if ((int) yy < bbylo) bbylo = (int) yy;
    if ((int) xx > bbxhi) bbxhi = (int) xx;
    if ((int) yy > bbyhi) bbyhi = (int) yy;
  }
}
```

38. Index.

⟨ Subroutines 3 ⟩ +≡

```
temp1( )
{
  printf("");
}
temp2( )
{
  printf("");
}
```

DECAGON