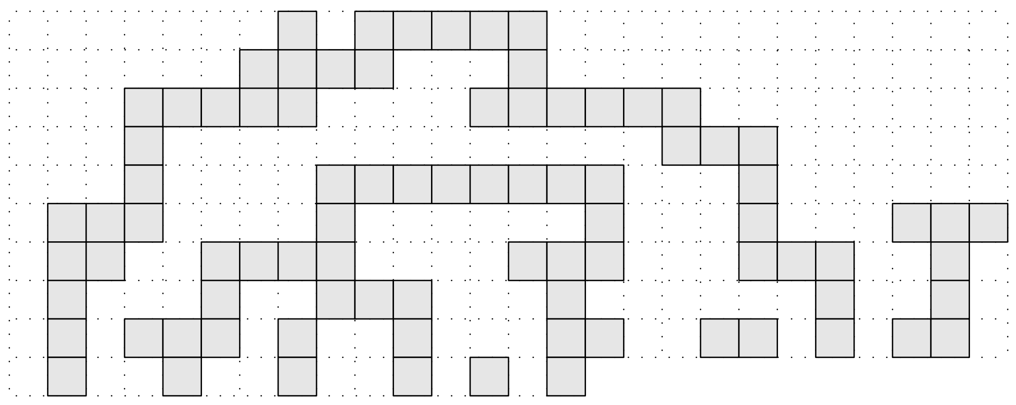
(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Introduction. The purpose of this program is to enumerate fixed polyominoes of up to 55 cells (although I won't really be able to get that far until Moore's law carries on for a few more years). The method is essentially that of Iwan Jensen [`arXiv:cond-mat/0007239`, to appear in *Journal of Statistical Physics*, spring 2001], who discovered that the important techniques of Andrew Conway [*Journal of Physics* **A28** (1995), 335–349] can be vastly improved in the special case of polyomino enumeration.

The basic idea is quite simple: We will count the number of fixed polyominoes that span a rectangle that is h cells high and w cells wide, where h and w are as small as possible; then we will add the totals for all relevant h and w. We can assume that $h \ge w$. For each h and w that we need, we enumerate the spanning polyominoes by considering one cell at a time of an $h \times w$ array, working from left to right and top to bottom, deciding whether or not that cell is occupied, and combining results for all boundary configurations that are equivalent as far as future decisions are concerned. For example, we might have a polyomino that starts out like this:



(This partial polyomino obviously has more than 54 cells already, but large examples will help clarify the concepts that are needed in the program below.)

Most of the details of the upper part of this pattern have no effect on whether the yet-undetermined cells will form a suitable polyomino or not. All we really need to know in order to answer that question is whether the bottom cells of each column-so-far are occupied or unoccupied, and which of the occupied ones are already connected to each other. We also need to know whether the left column and right column are still blank.

In this case the 26 columns have occupancy pattern `01001001001010110011010110` at the bottom, and the occupied cells belong to six connected components, namely

```
01000000000000000000010000
00001000001000110000000000
00000001000000000000000000
00000000000010000000000000
00000000000000000011000000
00000000000000000000000110
```

Fortunately the fact that polyominoes lie in a plane forces the components to be nested within each other; we can't have "crossings" like `1000100` with `0010001`. Therefore we can encode the component and occupancy information conveniently as

```
0(00(00100-010-)00()0)0()0
```

using a five-character alphabet:

`0`	means the cell is unoccupied;
`1`	means the cell is a single-cell component;
`(`	means the cell is the leftmost of a multi-cell component;
`)`	means the cell is the rightmost of a multi-cell component;
`-`	means the cell is in the midst of a multi-cell component.

Furthermore we can treat the cases where the entire leftmost column is nonblank by considering that the left edge of the array belongs to the leftmost cell component; and the rightmost column can be treated similarly. With these conventions, we encode the boundary condition at the lower fringe of the partially filled array above by the 26-character string

$$\texttt{0(00(00100-010-}{}_{\wedge}\texttt{)00()0)0(-0}\,, \tag{$*$}$$

using ‘${}_\wedge$’ to show where the last partial row ends.

A string like $(*)$ represents a so-called *configuration.* If no ‘${}_\wedge$’ appears, the partial row is assumed to be a complete row. The number of rows above a given occupancy pattern is implicitly part of the configuration but not explicitly shown in the notation, because this program never has to deal simultaneously with configurations that have different numbers of rows above the current partial row.

2. A bit of theory may help the reader internalize these rules: It turns out that the set of all connectivity/occupancy configuration codes at the end of a row has the interesting unambiguous context-free grammar

$$\begin{aligned}
S &\to L\,\texttt{0}J_0R \mid Z\texttt{-}I\texttt{-}Z \mid Z\texttt{-}Z\\
Z &\to \epsilon \mid \texttt{0}Z\\
L &\to \epsilon \mid Z\,\texttt{)} \mid Z\texttt{-}I\texttt{)}\\
R &\to \epsilon \mid \texttt{(}Z \mid \texttt{(}I\texttt{-}Z\\
I_0 &\to \epsilon \mid \texttt{0}J_0\\
J_0 &\to I_0 \mid A\,\texttt{0}J_0\\
I &\to \epsilon \mid \texttt{-}I \mid \texttt{0}J\\
J &\to I \mid A\,\texttt{0}J\\
A &\to \texttt{1} \mid \texttt{(}I\texttt{)}
\end{aligned}$$

[Translation: I is any sequence of `0`'s and `-`'s and A's with each A preceded and followed by `0`; I_0 is similar, but with no `-` at top level.] The number s_n of strings of length n in S has the generating function

$$\sum s_n z^n = \frac{1-4z^2-4z^3+z^4-(1+z)(1-z^2)\sqrt{1+z}\sqrt{1-3z}}{2z^3(1-z)} = 2z+6z^2+16z^3+\cdots;$$

hence s_n is asymptotically proportional to $3^n/n^{3/2}$.

3. Any polyomino with the configuration ($*$) in the midst of row 10 must have occupied at least $28+18+1+1+2+4=54$ cells so far, and 16 more will be needed to connect it up and make it touch the left edge of the array. Moreover, the array has only 10 rows so far, but it has 26 columns, and we will only be interested in arrays for which $h \ge w$. Therefore at least 15 additional cells must be occupied if we are going to complete a polyomino in an $h \times 26$ array.

In other words, we would need to consider the boundary configuration ($*$) only if we were enumerating polyominoes of at least $54+16+15=85$ cells. Such considerations greatly limit the number of configurations that can arise; Jensen's experiments show that the total number of cases we must deal with grows approximately as c^n where c is slightly larger than 1.4. This is exponentially large, but it is considerably smaller than the $3^{n/2}$ configurations that arise in Conway's original method; and it's vastly smaller than the total number of polyominoes, which turns out to be approximately 4.06^n times $0.3/n$.

4. This program doesn't actually do the whole job of enumeration; it only outputs a set of instructions that tell how to do it. Another program reads those instructions and completes the computation.

```
#include <stdio.h>
  ⟨ Type definitions 12 ⟩
  ⟨ Global variables 7 ⟩
  ⟨ Subroutines 5 ⟩
  main(int argc, char *argv[])
  {
    ⟨ Local variables 67 ⟩;
    ⟨ Scan the command line 6 ⟩;
    ⟨ Initialize 15 ⟩;
    ⟨ Output instructions for the postprocessor 78 ⟩;
    ⟨ Print statistics about this run 72 ⟩;
    ⟨ Empty the buffer and close the output file 16 ⟩;
    exit(0);
  }
```

5. ⟨ Subroutines 5 ⟩ ≡

```
  void panic(char *mess)
  {
    fprintf(stderr, "%s!\n", mess);
    exit(−1);
  }
```

See also sections 9, 10, 11, 13, 14, 17, 30, 31, 32, 35, 55, 56, 59, 63, 64, 69, 73, 75, and 76.

This code is used in section 4.

6. The user specifies the maximum size of polyominoes to be counted, n, and the desired width, w, on the command line. All $h \times w$ rectangles spanned by polyominoes of n cells or less will be counted, for $w \le h \le n + 1 - w$. (No solutions are possible for $h > n + 1 - w$.)

The present version of this program restricts w to be at most 23, for simplicity. But the packing and unpacking routines below could be adapted via `CWEB` change files in order to deal with values of w as large as 27, when we're pushing the envelope.

The command line should also specify the amount of memory allocated for configurations in this program and for individual counters in the output. Statistics will be printed for guidance in the choice of those parameters.

The base name of the output file should be given as the final command-line argument. This name will actually be extended by `.0`, `.1`, ..., as explained below, because there might be an enormous amount of output.

```
#define wmax 23   /* for quinary/octal packing into two tetrabytes */
#define nmax (wmax + wmax + 126)
#define bad(k, v) sscanf(argv[k], "%d", &v) ≠ 1
```

⟨ Scan the command line 6 ⟩ ≡

```
  if (argc ≠ 6 ∨ bad(1, n) ∨ bad(2, w) ∨ bad(3, conf_size) ∨ bad(4, slave_size)) {
    fprintf(stderr, "Usage:␣%s␣n␣w␣confsize␣slavesize␣outfilename\n", argv[0]);
    exit(−2);
  }
  if (w > wmax) panic("Sorry,␣that␣w␣is␣too␣big␣for␣this␣implementation");
  if (w < 2) panic("No,␣w␣must␣be␣at␣least␣2");
  if (n < w + w − 1) panic("There␣are␣no␣solutions␣for␣such␣a␣small␣n");
  if (n > w + w + 126) panic("Eh?␣That␣n␣is␣incredible");
  base_name = argv[5];
```

This code is used in section 4.

7. ⟨ Global variables 7 ⟩ ≡

```
  int n;   /* we will count polyominoes of n or fewer cells */
  int w;   /* provided that they span a rectangle of width w and height ≥ w */
  int conf_size;   /* the number of config structures in our memory */
  int slave_size;   /* the number of counter positions in the slave program memory */
```

See also sections 8, 18, 27, 34, 39, 57, 65, 68, 71, 85, 98, 108, and 110.

This code is used in section 4.

8. Output. Let's get the basics of output out of the way first, so that we know where we're heading. The postprocessing program POLYSLAVE will interpret instructions according to a compact binary format, with either one or four bytes per instruction.

Several gigabytes might well be generated, and my Linux system is not real happy with files of length greater than $2^{31} - 1 = 2147483647$. Therefore this program breaks the output up into a sequence of files called `foo.0`, `foo.1`, ..., each at most one large gigabyte in size. (That's one GGbyte = 2^{30} bytes.)

Some unfortunate hardware failures led me to add a *checksum* feature.

```
#define filelength_threshold (1 ≪ 30)   /* maximum file size in bytes */
#define buf_size (1 ≪ 16)   /* buffer size, should be a divisor of filelength_threshold */
⟨Global variables 7⟩ +≡
  FILE *out_file;   /* the output file */
  union {
    unsigned char buf[buf_size + 4];   /* place for binary output */
    int foo;   /* force out.buf to be aligned somewhat sensibly */
  } out;
  unsigned char *buf_ptr;   /* our current place in the buffer */
  int bytes_out;   /* the number of bytes so far in the current output file */
  unsigned int checksum;   /* a way to help identify bad I/O */
  FILE *ck_file;   /* the checksum file */
  int file_extension;   /* the number of GGbytes output */
  char *base_name, filename[100];
```

9. ⟨Subroutines 5⟩ +≡

```
  void open_it()
  {
    sprintf(filename, "%.90s.%d", base_name, file_extension);
    out_file = fopen(filename, "wb");
    if (¬out_file) {
      fprintf(stderr, "I␣can't␣open␣file␣%s", filename);
      panic("␣for␣output");
    }
    bytes_out = checksum = 0;
  }
```

10. ⟨Subroutines 5⟩ +≡

```
  void close_it()
  {
    if (fwrite(&checksum, sizeof(unsigned int), 1, ck_file) ≠ 1)
      panic("I␣couldn't␣write␣the␣check␣sum");
    if (fclose(out_file) ≠ 0) panic("I␣couldn't␣close␣the␣output␣file");
    printf("[%d␣bytes␣written␣on␣file␣%s,␣checksum␣%u.]\n", bytes_out, filename, checksum);
  }
```

11. ⟨ Subroutines 5 ⟩ +≡

```
void write_it(int bytes)
{
  register int k;
  register unsigned int s;
  if (bytes_out ≥ filelength_threshold) {
    if (bytes_out ≠ filelength_threshold) panic("Improper␣buffer␣size");
    close_it();
    file_extension++;
    open_it();
  }
  if (fwrite(&out.buf, sizeof(unsigned char), bytes, out_file) ≠ bytes) panic("Bad␣write");
  bytes_out += bytes;
  for (k = s = 0; k < bytes; k++) s = (s ≪ 1) + out.buf[k];
  checksum += s;
}
```

12. A four-byte instruction has the binary form $(0xaaaaaa)_2$, $(bbbbbbbb)_2$, $(cccccccc)_2$, $(dddddddd)_2$, where $(aaaaaabbbbbbbbccccccccdddddddd)_2$ is a 30-bit address specified in big-endian fashion. If $x = 0$ it means, "This is the new source address s." If $x = 1$ it means, "This is the new target address t."

A one-byte instruction has the binary form $(1ooopppp)_2$, with a 3-bit opcode $(ooo)_2$ and a 4-bit parameter $(pppp)_2$. If the parameter is zero, the following byte is regarded as an 8-bit parameter $(pppppppp)_2$, and it should not be zero. (In that case the "one-byte instruction" actually occupies two bytes.)

In the instruction definitions below, p stands for the parameter, s stands for the current source address, and t stands for the current target address. The slave processor operates on a large array called *count*.

Opcode 0 (*sync*) means, "We have just finished row p." A report is given to the user.
Opcode 1 (*clear*) means, "Set $count[t + j] = 0$ for $0 \le j < p$."
Opcode 2 (*copy*) means, "Set $count[t + j] = count[s + j]$ for $0 \le j < p$."
Opcode 3 (*add*) means, "Set $count[t + j] \mathrel{+}= count[s + j]$ for $0 \le j < p$."
Opcode 4 (*inc_src*) means, "Set $s \mathrel{+}= p$."
Opcode 5 (*dec_src*) means, "Set $s \mathrel{-}= p$."
Opcode 6 (*inc_trg*) means, "Set $t \mathrel{+}= p$."
Opcode 7 (*dec_trg*) means, "Set $t \mathrel{-}= p$."

```
#define targ_bit #40000000    /* specifies t in a four-byte instruction */
```

⟨ Type definitions 12 ⟩ ≡

```
typedef enum {
  sync, clear, copy, add, inc_src, dec_src, inc_trg, dec_trg
} opcode;
```

See also sections 26, 28, 29, 33, 54, 60, and 61.

This code is used in section 4.

13. **#define** *end_of_buffer* &*out.buf*[*buf_size*]

⟨ Subroutines 5 ⟩ +≡

```
void put_inst(unsigned char o, unsigned char p)
{
  register unsigned char *b = buf_ptr;
  *b++ = #80 + (o << 4) + (p < 16 ? p : 0);
  if (p >= 16) *b++ = p;
  if (b >= end_of_buffer) {
    write_it(buf_size);
    out.buf[0] = out.buf[buf_size];
    b -= buf_size;
  }
  buf_ptr = b;
}
```

14. ⟨ Subroutines 5 ⟩ +≡

```
void put_four(register unsigned int x)
{
  register unsigned char *b = buf_ptr;
  *b = x >> 24;
  *(b + 1) = (x >> 16) & #ff;
  *(b + 2) = (x >> 8) & #ff;
  *(b + 3) = x & #ff;
  b += 4;
  if (b >= end_of_buffer) {
    write_it(buf_size);
    out.buf[0] = out.buf[buf_size];
    out.buf[1] = out.buf[buf_size + 1];
    out.buf[2] = out.buf[buf_size + 2];
    b -= buf_size;
  }
  buf_ptr = b;
}
```

15. The first six bytes of the instruction file are, however, special. Byte 0 is the number n of cells in the largest polyominoes being enumerated. When a *sync* is interpreted, POLYSLAVE outputs the current values of $count[j]$ for $1 \le j \le n$.

Byte 1 is the number of the final row. If this number is r, POLYSLAVE will terminate after interpreting the instruction *sync* r.

Bytes 2–5 specify the (big-endian) number of elements in the *count* array.

Initially $s = t = 0$, $count[0] = 1$, and $count[j]$ is assumed to be zero for $1 \le j \le n$.

```
⟨ Initialize 15 ⟩ ≡
  sprintf(filename, "%.90s.ck", base_name);
  ck_file = fopen(filename, "wb");
  if (¬ck_file) panic("I␣can't␣open␣the␣checksum␣file");
  open_it();
  out.buf[0] = n;
  out.buf[1] = n + 2 − w;
  buf_ptr = &out.buf[2];
  put_four(slave_size);
```

See also sections 58, 62, 66, and 77.

This code is used in section 4.

16. Here's what we'll do when it's all over.

```
⟨ Empty the buffer and close the output file 16 ⟩ ≡
  if (buf_ptr ≠ &out.buf[0]) write_it(buf_ptr − &out.buf[0]);
  close_it();
```

This code is used in sections 4 and 106.

17. Most of the output is generated by the *basic_inst* routine.

```
⟨ Subroutines 5 ⟩ +≡
  void basic_inst(int op, int src_addr, int trg_addr, unsigned char count)
  {
    register int del;
    if (verbose > 1) {
      if (op ≡ clear) printf("{clear␣%d␣->%d}\n", count, trg_addr);
      else printf("{%s␣%d␣%d->%d}\n", sym[op], count, src_addr, trg_addr);
    }
    del = src_addr − cur_src;
    if (del > 0 ∧ del < 256) put_inst(inc_src, del);
    else if (del < 0 ∧ del > −256) put_inst(dec_src, −del);
    else if (del) put_four(src_addr);
    cur_src = src_addr;
    del = trg_addr − cur_trg;
    if (del > 0 ∧ del < 256) put_inst(inc_trg, del);
    else if (del < 0 ∧ del > −256) put_inst(dec_trg, −del);
    else if (del) put_four(trg_addr + targ_bit);
    cur_trg = trg_addr;
    put_inst(op, count);
  }
```

18.
```
⟨ Global variables 7 ⟩ +≡
  char *sym[4] = {"sync", "clear", "copy", "add"};
  int cur_src, cur_trg;   /* current source and target addresses in the slave */
  int verbose = 0;   /* set nonzero when debugging */
```

19. Connectivity. The hardest task that confronts us is to figure out how to determine the cutoff threshold: Given a configuration like $\texttt{0(00(00100-010-}\underset{\wedge}{\texttt{)}}\texttt{00()0)0(-0}$, what is the minimum number of additional cells that are needed to connect it up and to make it stretch out to at least a given number of further rows? We claimed above, without proof, that this particular configuration needs at least 16 more cells before it will be connected and touch the left boundary. Now we want to prove that claim, and solve the general problem as well.

Some cases of this problem are easy. For example, let's consider first the case when we are at the beginning or end of a complete row. Then it is clear that a configuration like `00-)0(0--)00(--0` needs at least $3+4$ more cells to become occupied. [Well, if this *isn't* clear, please stop now and think about it until it *is*. Remember that it stands for a pattern with three connected components of occupied cells; the left component is connected to the left edge, the right component is connected to the right edge, and the middle component is standing alone.]

Suppose we have a pattern like $\texttt{0}^{g_0}\alpha_1\texttt{0}^{g_1}\alpha_2\ldots\texttt{0}^{g_{k-1}}\alpha_k\texttt{0}^{g_k}$, where each α_j is a separate component beginning with `(` and ending with `)`. For example, a typical α might be `()` or `(-)` or `(-00--00-0-0)`, etc. Again the problem we face is easily solved: We need to occupy g_0+1 cells in order to connect α_1 to the left edge, g_j+2 cells to connect α_j to α_{j+1} for $1\le j<k$, and g_k+1 cells to connect α_k to the right edge. The same formula holds if any α_j is simply `1`, denoting a singleton component, except that we can subtract 1 for every such α_j. For example, the cost of connecting up `0100(0-)00010(0)0` is $2+4+5+3+2-2$.

If α_1 is already connected to the left edge, we save g_0+1, but we cannot take the bonus if α_1 is `1`; a similar consideration applies at the right.

20. The situation gets more interesting when components are nested. Suppose, for example, that α_1, α_2, $\ldots$, α_{k-1} are distinct, but α_k is part of the same component as α_1. Then we still must pay g_0+1 to connect α_1 at the left and g_k+1 to connect $\alpha_k\equiv\alpha_1$ at the right; but in this new case we are allowed to keep α_j disconnected from α_{j+1} for any single choice of j we like, in the range $1\le j<k$. That will save g_j+2 from the formula stated above, except that it will cost one or two bonus points if α_j and/or α_{j+1} had length 1. For example, to connect the configuration `0(00010(0)00-0)00`, which has the form $\texttt{0}^1\alpha_1\texttt{0}^3\alpha_2\texttt{0}^1\alpha_3\texttt{0}^2\alpha_4\texttt{0}^2$ with $\alpha_1\equiv\alpha_4$ and potential bonuses at α_1 and α_2, we have three options. Disconnecting α_1 from α_2 costs $2+0+3+4+3-0$; disconnecting α_2 from α_3 costs $2+5+0+4+3-1$; disconnecting α_3 from α_4 costs $2+5+3+0+3-2$. The third alternative is best, even though it doesn't disconnect the largest gap $\texttt{0}^3$, because it retains the 2 bonus points.

21. Now look at the configuration `-00(010-010)00()00()00-`, which is connected to left and right edges and which also contains the subcomponent `(010-010)`. The best way to handle the subcomponent is to occupy 5 cells below it, spanning the middle region `10-01`. But then we need $4+4+4$ additional cells to connect up the whole diagram. If instead we use 6 cells within the subcomponent, spanning `(01` at the left and `10)` at the right, we need only $3+3+4$ additional cells to finish. Thus the environment can affect the optimal behavior within a subcomponent.

These examples give us one way to think about the minimum connection cost for the general pattern $\texttt{0}^{g_0}\alpha_1\texttt{0}^{g_1}\alpha_2\ldots\texttt{0}^{g_{k-1}}\alpha_k\texttt{0}^{g_k}$, when α_i is already connected to α_j for certain pairs (i,j), namely to start by charging $(g_0+1)+(g_1+2)+\cdots+(g_{k-1}+2)+(g_k+1)$ and then to deduct some of the terms for gaps g_j that are legitimately left unconnected: The term (g_0+1) can be deducted if α_1 is connected to the left edge, and (g_k+1) can be deducted if α_k is connected to the right edge. If $\alpha_i\equiv\alpha_j$ for some $j>i$ and if the components α_{i+1}, $\ldots$, α_j are mutually disconnected, then we are allowed to deduct any one of the terms (g_i+2), $\ldots$, $(g_{j-1}+2)$, after which we can treat $\alpha_i\ldots\alpha_j$ as a *single* component with respect to further deductions. Finally after choosing a subset of terms to deduct, we get a bonus for each α_j of length 1 such that neither g_{j-1} nor g_j were left disconnected. (Length 1 means that the code for α_j is a single character, either `1` or `(` or `)` or `-`.)

22. A recursive strategy can be used to solve the minimum connectivity problem in linear time, but we must design it carefully because of the examples considered earlier. The key idea will be to associate four costs c_{ij} with each subcomponent, where $0 \le i, j \le 1$. Cost c_{ij} is the minimum number of future occupied cells needed to connect everything up within the component, with the further proviso that there is a cell below the leftmost cell if $i = 1$, and a cell below the rightmost cell if $j = 1$. The 2×2 matrix (c_{ij}) will then represent all we need to know about connecting this component at a higher level.

For example, the cost matrix for a single-character component is $\binom{0\,1}{1\,1}$, and the cost matrix for a multi-character component with no internal subcomponents is $\binom{0\,1}{1\,2}$. (The 0 in the upper left corner signifies that we don't need any cells to connect it up, since it's already connected. But we may have to occupy one or two cells as "hooks" at the left and/or the right if the environment wants them.)

The theory that underlies the algorithm below is best understood in terms of *min-plus matrix multiplication* $C = AB$, where $c_{ij} = \min_k(a_{ik} + b_{kj})$. (I should probably use a special symbol to denote this multiplication, like $A \overset{\wedge}{+} B$ instead of AB; see, for example, exercise 1.3.1′–32 in *The Art of Computer Programming*, Fascicle 1. But that would clutter up a huge number of the formulas below. And I have no use for ordinary matrix multiplication in the present program. Therefore min-plus multiplication will be assumed to need no special marking in the following discussion.)

Let X_g be the matrix $\binom{\infty\ \ \infty}{\infty\ \ g}$. Then if $\alpha_1, \ldots, \alpha_k$ are distinct subcomponents with cost matrices $A_1, \ldots, A_k$, respectively, the cost of connecting up $\mathtt{0}^{g_0}\alpha_1\mathtt{0}^{g_1}\alpha_2\ldots\mathtt{0}^{g_{k-1}}\alpha_k\mathtt{0}^{g_k}$ and touching the left and right edges is the upper left corner entry of

$$OX_{g_0}A_1X_{g_1}\ldots X_{g_{k-1}}A_kX_{g_k}O, \qquad \text{where } O = \binom{0\ \ 0}{0\ \ 0},$$

because matrix X_g essentially means "occupy g cells below and require the cells to the left and right of these cells to be occupied as well." Notice that this rule yields $(g_0+1)+(g_1+2)+\cdots+(g_{k-1}+2)+(g_k+1)$ in the special case when each matrix A_j is $\binom{0\,1}{1\,2}$, and there's a bonus of 1 whenever we replace A_j by $\binom{0\,1}{1\,1}$. Notice also that $X_gX_h = X_{g+h}$; thus we can essentially think of each $\mathtt{0}$ in the configuration as a multiplication by X_1.

23. If α and β are subcomponents that are already connected to each other, having cost matrices A and B respectively, the cost matrix for $\alpha\,\mathtt{0}^g\beta$ as a single component is ANB, where $N = \binom{0\ \ \infty}{\infty\ \ \infty}$ is the matrix meaning "don't occupy anything below the $\mathtt{0}^g$ and don't insist that the cells to the left and right of that gap must be occupied." For example, this rule gives $\binom{0\,1}{1\,1}N\binom{0\,1}{1\,1} = \binom{0\,1}{1\,2}$ as it should.

And in general, suppose $\alpha_0, \alpha_1, \ldots, \alpha_k$ are subcomponents that are distinct except that α_0 is already connected to α_k. Then the cost matrix for $\alpha_0\mathtt{0}^{g_1}\alpha_1\ldots\alpha_{k-1}\mathtt{0}^{g_k}\alpha_k$ as a single subcomponent can be expressed in terms of the individual cost matrices $A_0, A_1, \ldots, A_k$ by using the formula

$$\min_{1\le j\le k}\bigl(A_0X_{g_1}\ldots X_{g_{j-1}}A_{j-1}NA_jX_{g_{j+1}}\ldots X_{g_k}A_k\bigr), \qquad (**)$$

because we are allowed to leave any one of the gaps $\mathtt{0}^{g_j}$ disconnected. Rule $(**)$ is the basic principle that makes a recursive algorithm work.

24. Our discussion so far has assumed for simplicity that we're at the end of a row. But we also need to consider the case that a gap $\mathtt{0}^g$ might be $\mathtt{0}^a\underset{\wedge}{\mathtt{0}}{}^b$ in the vicinity of the place where a partial row is being filled. Fortunately the same theory applies with only a slight variation: Instead of $X_g = X_{a+b}$, we use X_{a+b+1} if $a > 0$, or we use $Y_b = \binom{\infty\ \ b}{\infty\ \ \infty}$ if $a = 0$; either a or b or both might be zero. Matrix Y_b means, "Occupy the cells below $\mathtt{0}^b$ and require the next cell on the right (but not necessarily the left) to be occupied as well." The extra 1 in X_{a+b+1} is needed when $a > 0$ because one of the newly occupied cells must pass through the current row r in order to reach row $r + 1$.

25. After calculating the minimum cost of connection, we might also need to add the cost of extension to span a $w \times w$ square. Thus, if row r is partially filled and row $r-1$ is completely filled, and if $r < w$, we must add $w - r - 1$ cells if the minimum connection can be achieved with at least one cell in row $r+1$; otherwise we must add $w - r$ cells.

Consider, for example, the configuration `-0‸010)00`. The best way to connect this up is to occupy the five cells in row r that occur below the `10)00`. But then we need $w - r$ more for extension. On the other hand, in a configuration like `-0‸010000)00`, there are two essentially different ways to do the connection with eight more cells, and we must choose the one that uses cells of row $r+1$. (A configuration like `-0‸01000)00` can be connected in seven cells without using row $r+1$, or in eight with the use of that row, so it's a tossup.)

The configurations `-01‸00-` and `-01‸00)0(` also present interesting tradeoffs that our algorithm must handle properly.

The solution adopted here is to add ϵ to the cell costs after the '‸', so that newly occupied cells in row r are slightly more expensive than cells in row $r+1$. This makes the latter cells more attractive, in cases when they need to be.

26. OK, we've got a reasonably clean theory; now it's time to put it into practice. The first step is to define the encoding of our alphabet `0`, `1`, `(`, `)`, and `-`, to which we'll add an "edge of line" character. The following encoding is designed to make it easy to test whether a character is either `-` or `)`:

```
#define mid_or_rt(x)   (((x) & 2) ≡ 2)
⟨ Type definitions 12 ⟩ +≡
  typedef enum {
    zero, one, rt, mid, lft, eol
  } code;      /* 0, 1, ), -, (, or edge delimiter in a configuration string */
```

27. ⟨ Global variables 7 ⟩ +≡

```
  char decode[5] = {'0', '1', ')', '-', '('};
  code reflect[5] = {zero, one, lft, mid, rt};
```

28. Then in the cost matrices and in our answer describing the minimum connection cost of a given configuration, we will represent numbers $a + b\epsilon$ as short integers, knowing that a and b will never exceed 27.

```
#define unity   (1 ≪ 8)
#define epsilon   1
#define uunity   (unity + epsilon)
#define int_part(x)   ((x) ≫ 8)
#define eps_part(x)   ((x) & #ff)
⟨ Type definitions 12 ⟩ +≡
  typedef unsigned short cost;      /* a + bε represented as (a ≪ 8) + b */
  typedef struct {
    cost c[2][2];
  } cost_matrix;
```

29. We will need to distinguish between the cost matrices $X_g = \binom{\infty\ \ \infty}{\infty\ \ g}$ and $Y_g = \binom{\infty\ \ g}{\infty\ \ \infty}$, as well as a third case that arises when a configuration is already connected to the left edge.

```
⟨ Type definitions 12 ⟩ +≡
  typedef enum {
    ytyp, xtyp, otyp
  } gap_type;
```

30. A few easy subroutines do the basic operations on cost matrices that we will need.

```
⟨ Subroutines 5 ⟩ +≡
  cost_matrix a_n_b(cost_matrix a, cost_matrix b)    /* computes ANB */
  {
    cost_matrix c;
    c.c[0][0] = a.c[0][0] + b.c[0][0];
    c.c[0][1] = a.c[0][0] + b.c[0][1];
    c.c[1][0] = a.c[1][0] + b.c[0][0];
    c.c[1][1] = a.c[1][0] + b.c[0][1];
    return c;
  }
```

31.

```
⟨ Subroutines 5 ⟩ +≡
  cost_matrix a_x_b(cost_matrix a, cost_matrix b, gap_type typ, cost g)
        /* computes AX_g B, AY_g B, or B, depending on typ */
  {
    cost_matrix c;
    if (typ ≡ otyp) return b;
    c.c[0][0] = a.c[0][typ] + g + b.c[1][0];
    c.c[0][1] = a.c[0][typ] + g + b.c[1][1];
    c.c[1][0] = a.c[1][typ] + g + b.c[1][0];
    c.c[1][1] = a.c[1][typ] + g + b.c[1][1];
    return c;
  }
```

32.

```
⟨ Subroutines 5 ⟩ +≡
  void min_mat(cost_matrix *a, cost_matrix b)    /* sets A = min(A, B) */
  {
    if ((*a).c[0][0] > b.c[0][0]) (*a).c[0][0] = b.c[0][0];
    if ((*a).c[0][1] > b.c[0][1]) (*a).c[0][1] = b.c[0][1];
    if ((*a).c[1][0] > b.c[1][0]) (*a).c[1][0] = b.c[1][0];
    if ((*a).c[1][1] > b.c[1][1]) (*a).c[1][1] = b.c[1][1];
  }
```

33. The algorithm we use is inherently recursive, but we implement it iteratively using a stack because it involves only simple algebraic operations. Each stack entry typically represents a string of `0`'s and a subcomponent that has not been completely scanned as yet; the string of `0`'s is represented by a cost matrix X_g or Y_g, and the incomplete subcomponent is represented by a partial evaluation of formula ($**$).

At stack level 0, however, there is no incomplete subcomponent and only the *closed_cost* field is relevant. This field corresponds to the cost matrix of all components scanned so far.

```
⟨ Type definitions 12 ⟩ +≡
  typedef struct {
    cost gap;    /* zeros to cover in the previous gap */
    gap_type gap_typ;    /* type of previous gap (xtyp or ytyp or otyp) */
    cost_matrix closed_cost;    /* cost matrix with no gaps */
    cost_matrix open_cost;    /* cost matrix with optional gap */
  } stack_entry;
```

34. The given configuration string will appear in $c[1]$ through $c[w]$; also $c[0]$ and $c[w+1]$ will be set to *eol*.

```
⟨ Global variables 7 ⟩ +≡
  code c[64];      /* codes of the current configuration string */
  stack_entry stk[64];      /* partially evaluated costs */
```

35. This program operates in two phases that are almost identical: Before *row_end* has been sensed, the cost of a connection cell is *unity*, but afterwards it is *unity* + *epsilon*. In order to streamline the code I'm using a little trick explained in Example 7 of my paper "Structured programming with **goto** statements": I make two almost identical copies of the code, one for the actions to be taken before $k \equiv$ *row_end* and one for the actions to be taken subsequently. The first copy jumps to the second as soon as the condition $k \equiv$ *row_end* is sensed. This avoids all kinds of conditional coding and makes "variables" into "constants," although it does have the somewhat disconcerting feature of jumping from one loop into the body of another.

The program could be made faster if I would look at high speed for special cases like subcomponents of the form `(0--00-)`, since that subcomponent is equivalent to `()` with respect to the connectivity measure we are computing. But I intentionally avoided tricky optimizations in order to keep this program simpler and easier to verify.

On the other hand, I do reduce `0`-less forms like `(---)` to `()`, and `(-`$_\wedge$`--)` to `(`$_\wedge$`)`, etc. Such optimizations aren't strictly necessary but I found them irresistible, because they arise so frequently.

Variable *row_end* in the following routine points to the character just following the '$_\wedge$'.

```
⟨ Subroutines 5 ⟩ +≡
  cost connectivity(register int row_end)
  {
    register int k;      /* our place in the string */
    register int s;      /* the number of open items on the stack */
    int g;      /* the current gap size */
    gap_type typ;      /* its type */
    int open;      /* set nonzero if previous token was ( or - */
    ⟨ Get ready to compute connectivity 36 ⟩;
  scan_zeros0: ⟨ Scan for zeros in Phase 0 37 ⟩;
  scan_tokens0: ⟨ Scan a nonzero token cluster in Phase 0 38 ⟩;
  scan_zeros1: ⟨ Scan for zeros in Phase 1 48 ⟩;
  scan_tokens1: ⟨ Scan a nonzero token cluster in Phase 1 49 ⟩;
    ⟨ Finish the connectivity bound calculation and return the answer 52 ⟩;
  }
```

36. In practice *row_end* will not be zero. But I decided to make the algorithm general enough to work correctly also in that case, because only one more line of code was needed.

```
⟨ Get ready to compute connectivity 36 ⟩ ≡
  s = open = 0;
  stk[0].closed_cost = zero_cost;
  k = 1;
  if (mid_or_rt(c[1])) {
    g = 0, typ = xtyp;      /* xtyp and ytyp are equivalent at the left edge */
    if (row_end ≡ 1) goto scan_tokens1;
    else goto scan_tokens0;
  }
```

This code is used in section 35.

37. ⟨Scan for zeros in Phase 0 37⟩ ≡

```
  if ($k \equiv row\_end$) {
    $g = 0, typ = ytyp$;
    goto $scan\_zeros1x$;
  }
  if ($c[k]$) $panic$("Syntax␣error,␣0␣expected");
  $typ = xtyp, g = unity, k$++;
  while ($c[k] \equiv 0$) {
    if ($k \equiv row\_end$) {
      $g$ += $unity + uunity, k$++;    /* correction for straddling rows */
      goto $scan\_zeros1x$;
    }
    $g$ += $unity, k$++;
  }
  if ($k \equiv row\_end$) {
    $g$ += $unity$;
    goto $scan\_tokens1$;
  }
```

This code is used in section 35.

38. ⟨Scan a nonzero token cluster in Phase 0 38⟩ ≡

```
  $cm = base\_cost0$;
  $k$++; switch ($c[k-1]$) {
case $lft$: if ($\neg mid\_or\_rt(c[k])$) goto $scan\_open0$;
  ⟨Compress -*) following ( into a single token in Phase 0 40⟩
  if ($c[k-1] \neq rt$) goto $scan\_open0$;
case $one$: ⟨Append $cm$ to the current partial component 42⟩;
  $open = 0$; goto $scan\_zeros0$;
case $mid$: if ($\neg(mid\_or\_rt(c[k]))$) goto $scan\_mid0$;
  ⟨Compress -*) following - into a single token in Phase 0 41⟩
  if ($c[k-1] \neq rt$) goto $scan\_mid0$;
case $rt$: if ($\neg s$) {
    if ($stk[0].closed\_cost.c[1][1]$) $panic$("Unmatched␣)");
    $stk[0].closed\_cost = cm$;    /* already connected to left edge */
  } else {
    ⟨Append $cm$ to $stk[s].open\_cost$ 43⟩;
    ⟨Combine the top two items on the stack 44⟩;
  }
  $open = 0$; goto $scan\_zeros0$;
case $eol$: goto $scan\_eol0$;
default: $panic$("Illegal␣code"); }
$scan\_open0$: ⟨Finish processing $lft$ 46⟩; goto $check\_eol0$;
$scan\_mid0$: ⟨Finish processing $mid$ 47⟩;
$check\_eol0$: $open = 1$;
  if ($c[k] \neq eol$) goto $scan\_zeros0$;
  if ($k \equiv row\_end$) goto $scan\_eol1$;
$scan\_eol0$: $panic$("Row␣end␣missed");
```

This code is used in section 35.

39. ⟨ Global variables 7 ⟩ +≡

```
gap_type typ;    /* the current type of gap g */
cost_matrix cm;    /* the current cost matrix */
cost_matrix acm;    /* another cost matrix */
const cost_matrix zero_cost = {0, 0, 0, 0};    /* $\binom{0\,0}{0\,0}$ */
const cost_matrix base_cost0 = {0, unity, unity, unity};    /* $\binom{0\,1}{1\,1}$ */
const cost_matrix base_cost1 = {0, uunity, uunity, uunity};    /* $\begin{pmatrix} 0 & 1+\epsilon \\ 1+\epsilon & 1+\epsilon \end{pmatrix}$ */
```

40. ⟨ Compress -*) following (into a single token in Phase 0 40 ⟩ ≡

```
{
  do {
    if (k ≡ row_end) goto scan_tokens1a;
    k++;
    if (c[k − 1] ≡ rt) break;
  } while (mid_or_rt(c[k]));
  cm.c[1][1] = unity + unity;    /* now cm is $\binom{0\,1}{1\,2}$ */
}
```

This code is used in section 38.

41. ⟨ Compress -*) following - into a single token in Phase 0 41 ⟩ ≡

```
{
  do {
    if (k ≡ row_end) goto scan_tokens1b;
    k++;
    if (c[k − 1] ≡ rt) break;
  } while (mid_or_rt(c[k]));
  cm.c[1][1] = unity + unity;    /* now cm is $\binom{0\,1}{1\,2}$ */
}
```

This code is used in section 38.

42. ⟨ Append *cm* to the current partial component 42 ⟩ ≡

```
if (s) ⟨ Append cm to stk[s].open_cost 43 ⟩;
stk[s].closed_cost = a_x_b(stk[s].closed_cost, cm, typ, g);
```

This code is used in sections 38 and 49.

43. ⟨ Append *cm* to *stk*[*s*].*open_cost* 43 ⟩ ≡

```
{
  acm = a_n_b(stk[s].closed_cost, cm);
  stk[s].open_cost = a_x_b(stk[s].open_cost, cm, typ, g);
  min_mat(&stk[s].open_cost, acm);
}
```

This code is used in sections 38, 42, 47, and 49.

44. ⟨ Combine the top two items on the stack 44 ⟩ ≡

```
s−−;
if (s) ⟨ Append stk[s + 1].open_cost to stk[s].open_cost 45 ⟩;
stk[s].closed_cost = a_x_b(stk[s].closed_cost, stk[s + 1].open_cost, stk[s + 1].gap_typ, stk[s + 1].gap);
```

This code is used in sections 38, 49, and 52.

45. ⟨ Append $stk[s+1].open\_cost$ to $stk[s].open\_cost$ 45 ⟩ ≡

```
{
    $acm = a\_n\_b(stk[s].closed\_cost, stk[s+1].open\_cost)$;
    $stk[s].open\_cost = a\_x\_b(stk[s].open\_cost, stk[s+1].open\_cost, stk[s+1].gap\_typ, stk[s+1].gap)$;
    $min\_mat(\&stk[s].open\_cost, acm)$;
}
```

This code is used in section 44.

46. ⟨ Finish processing *lft* 46 ⟩ ≡

```
$stk[{+}{+}s].gap\_typ = typ$;
$stk[s].gap = g$;
$stk[s].closed\_cost = stk[s].open\_cost = cm$;
```

This code is used in sections 38 and 49.

47. ⟨ Finish processing *mid* 47 ⟩ ≡

```
if $(\neg s)$ {
    if $(stk[0].closed\_cost.c[1][1])$ $panic$("Unmatched␣-");
    $s = 1, stk[1].gap\_typ = otyp, stk[1].closed\_cost = stk[1].open\_cost = cm$;
} else {
    ⟨ Append $cm$ to $stk[s].open\_cost$ 43 ⟩;
    $stk[s].closed\_cost = stk[s].open\_cost$;
}
```

This code is used in sections 38 and 49.

48. ⟨ Scan for zeros in Phase 1 48 ⟩ ≡

```
  if $(c[k])$ $panic$("Syntax␣error,␣0␣expected");
  $typ = xtyp, g = uunity, k{+}{+}$;
$scan\_zeros1x$: while $(c[k] \equiv 0)$ {
    $g \mathrel{+}= uunity, k{+}{+}$;
  }
```

This code is used in section 35.

49. ⟨ Scan a nonzero token cluster in Phase 1 49 ⟩ ≡

```
  cm = base_cost1;
  k++; switch (c[k − 1]) {
case lft: if (¬mid_or_rt(c[k])) goto scan_open1;
  ⟨ Compress -*) following ( into a single token in Phase 1 50 ⟩
  if (c[k − 1] ≠ rt) goto scan_open1;
case one: ⟨ Append cm to the current partial component 42 ⟩;
  open = 0; goto scan_zeros1;
case mid: if (¬(mid_or_rt(c[k]))) goto scan_mid1;
  ⟨ Compress -*) following - into a single token in Phase 1 51 ⟩
  if (c[k − 1] ≠ rt) goto scan_mid1;
case rt: if (¬s) {
      if (stk[0].closed_cost.c[1][1]) panic("Unmatched␣)");
      stk[0].closed_cost = cm;      /* already connected to left edge */
  } else {
      ⟨ Append cm to stk[s].open_cost 43 ⟩;
      ⟨ Combine the top two items on the stack 44 ⟩;
  }
  open = 0; goto scan_zeros1;
case eol: goto scan_eol1;
default: panic("Illegal␣code"); }
scan_open1: ⟨ Finish processing lft 46 ⟩; goto check_eol1;
scan_mid1: ⟨ Finish processing mid 47 ⟩;
check_eol1: open = 1;
  if (c[k] ≠ eol) goto scan_zeros1;
scan_eol1:      /* fall through to return the answer */
```

This code is used in section 35.

50. ⟨ Compress -*) following (into a single token in Phase 1 50 ⟩ ≡

```
  {
  scan_tokens1a: do {
        k++;
        if (c[k − 1] ≡ rt) break;
     } while (mid_or_rt(c[k]));
     cm.c[0][1] = uunity;
     cm.c[1][1] += uunity;      /* now cm is $\binom{0\ \ 1+\epsilon}{1\ \ 2+\epsilon}$ or $\binom{0\ \ \ 1+\epsilon}{1+\epsilon\ \ 2+2\epsilon}$ */
  }
```

This code is used in section 49.

51. ⟨ Compress -*) following - into a single token in Phase 1 51 ⟩ ≡

```
  {
  scan_tokens1b: do {
        k++;
        if (c[k − 1] ≡ rt) break;
     } while (mid_or_rt(c[k]));
     cm.c[0][1] = uunity;
     cm.c[1][1] += uunity;      /* now cm is $\binom{0\ \ 1+\epsilon}{1\ \ 2+\epsilon}$ or $\binom{0\ \ \ 1+\epsilon}{1+\epsilon\ \ 2+2\epsilon}$ */
  }
```

This code is used in section 49.

52. The cost of reaching the right edge is g if g has the form $b + b\epsilon$, but it is $a + b$ if g has the exceptional form $a + 1 + b + b\epsilon$ that arises when filling an unfilled row.

```
⟨Finish the connectivity bound calculation and return the answer 52⟩ ≡
  if (open) {
    ⟨Combine the top two items on the stack 44⟩;
    if (s) panic("Missing␣)");
    return stk[0].closed_cost.c[0][0];
  } else {      /* we need to reach the right edge */
    if (s) panic("Missing␣)");
    if (int_part(g) ≡ eps_part(g)) return stk[0].closed_cost.c[0][typ] + g;
    return stk[0].closed_cost.c[0][1] + ((int_part(g) − 1) ≪ 8);
  }
```

This code is used in section 35.

53. Data structures. If we want to count n-ominoes efficiently for large n, our most precious resource turns out to be the random-access memory that is available. I think at least 20 bytes of memory are needed per active configuration, and that limits us to about 50 million active configurations per gigabyte of memory. Under such circumstances I don't mind recomputing results several times in order to save space. For example, many configurations will occur on several different rows, and this program recomputes their connectivity cost each time they appear.

The main loop of our computation will consist of taking a viable configuration α and looking at its two successors α_0 and α_1. The value of *row_end* in α_0 and α_1 (the position of '$_\wedge$') will be one greater than its value in α; α_0 will leave the new cell empty, but α_1 will occupy it.

We gain time when α_0 and/or α_1 have been seen before; a hash table helps us determine whether or not they are déjà vu. Newly seen configurations are subjected to the *connectivity* bound, and accepted into the computation only if they are found to be viable.

Each configuration α has an associated *generating function* $g(\alpha)$. For example, the generating function $5z^8 + 2z^9$ would mean that there are 5 ways to reach α with 8 cells occupied and 2 ways to reach it with 9 cells occupied (so far). We will add $g(\alpha)$ to $g(\alpha_0)$ if α_0 is viable, and $zg(\alpha)$ to $g(\alpha_1)$ if α_1 is viable. After that we are able to forget α and $g(\alpha)$, reclaiming precious memory space.

Indeed, we don't actually have enough space to deal with the generating functions $g(\alpha)$. Therefore this program compiles and outputs a sequence of instructions that will be interpreted by another program, POLYSLAVE; that program will subsequently do the actual additions, without needing a hash table or other space-hungry data.

The algorithm proceeds row by row, starting in row $r = 1$, and continues until no viable configurations remain. In each row it makes w complete passes over the existing configurations, with *row_end* running from 0 to $w - 1$. Every such pass processes and discards all configurations that were generated on the previous pass, and we are free to process them in any convenient order. Therefore we adopt a "rolling" strategy that uses memory with near-maximum efficiency: Suppose one pass has produced M configurations in the first M slots $\alpha^{(1)}, \ldots, \alpha^{(M)}$ of a memory pool consisting of $N =$ *conf_size* total slots. We start by finding the successors $\alpha_0^{(M)}$ and $\alpha_1^{(M)}$ of $\alpha^{(M)}$, putting them into slots N and $N - 1$. Then slot M is free and we turn to $\alpha^{(M-1)}$, etc., thereby running out of memory only if N configurations are fully in use. On the following pass we reverse direction, filling slots from 1 upwards instead of from N downwards.

The same strategy makes it easy to allocate the memory space needed for generating functions in the POLYSLAVE program. Indeed, this rolling allocation scheme fills memory almost perfectly, even though each generating function occupies a variable number of bytes. (Sometimes holes do appear, if a generating function turns out to need more bytes than we thought it would. But the behavior in general is quite satisfactory.)

54. Eight bytes suffice to encode any configuration $c_0 \ldots c_{w-1}$ in a 5-letter alphabet, since $5^{27} < 2^{64} < 5^{28}$ and we are assuming that w is at most 27. We define **cstring** to a union type, so that the hash function can readily access its individual bytes, yet packing and unpacking can be done with radix 5 or 8. (The hash function will evaluate differently on a big-endian machine versus a little-endian machine, but that doesn't matter.)

```
⟨ Type definitions 12 ⟩ +≡
  typedef struct {
    unsigned int h, l;      /* high-order and low-order halves */
  } octa;      /* two tetrabytes make one octabyte */
  typedef union {
    octa o;
    unsigned char byte[8];
  } cstring;      /* packed version of a configuration string */
```

55. The *pack* subroutine produces a **cstring** from the codes in array c. Since we are assuming provisionally that w is at most 23, we can use octal notation to put 10 codes in one tetrabyte and quinary notation to put 13 in the other. But quinary notation could obviously be used in both tetrabytes, or we could use pure quinary on octabytes, in variants of this program designed for larger values of w.

```
⟨ Subroutines 5 ⟩ +≡
  cstring packit( )
  {
    register int j, k;
    cstring packed;
    k = w − 1, j = c[w];
    if (w ≤ 10) packed.o.h = 0;
    else {
      for ( ; k > 10; k−−) j = (j ≪ 2) + j + c[k];
      packed.o.h = j;
      k = 9, j = c[10];
    }
    for ( ; k > 0; k−−) j = (j ≪ 3) + c[k];
    packed.o.l = j;
    return packed;
  }
```

56. That which can be packed can be unpacked. This routine puts the results into two arrays, *sc* and c, because it is used only when unpacking a new source configuration α.

There's an all-binary way to divide by 5 that is faster than division on some machines (see *TAOCP* exercise 4.4–9). But the simple '/5' works best on my computer, winning also over floating point division.

Curiously, I also found that '$x*5$' is slower than '$(x << 2)+x$', but '$y-5*x$' is faster than '$y-((x \ll 2)+x)$'. Some quirk of pipelining probably underlies these phenomena. I am content to leave such mysteries unexplained for now, because the present speed is acceptable.

```
⟨ Subroutines 5 ⟩ +≡
  void unpackit(cstring s)
  {
    register int j, k, q;
    if (w > 10) {
      for (k = 1, j = s.o.l; k < 10; k++) {
        sc[k] = c[k] = j & 7;
        j ≫= 3;
      }
      sc[10] = c[10] = j;
      for (k = 11, j = s.o.h; k < w; k++) {
        q = j/5;
        sc[k] = c[k] = j − 5 ∗ q;
        j = q;
      }
    } else
      for (k = 1, j = s.o.l; k < w; k++) {
        sc[k] = c[k] = j & 7;
        j ≫= 3;
      }
    sc[k] = c[k] = j;
  }
```

57. ⟨ Global variables 7 ⟩ +≡

```
code sc[64];    /* codes of the current source configuration */
```

58. ⟨ Initialize 15 ⟩ +≡

```
c[0] = sc[0] = c[w + 1] = sc[w + 1] = eol;
```

59. ⟨ Subroutines 5 ⟩ +≡

```
void print_config(int row_end)
{
  register int k;
  for (k = 1; k ≤ w; k++) {
    if (row_end ≡ k) printf("^");
    if (c[k] < eol) printf("%c", decode[c[k]]);
    else printf("?");
  }
}
```

60. A configuration can be in three states: Normally it is *active*, with a generating function represented as a sequence of counters; first, however, it is *raw*, meaning that the space for counters has been allocated but not yet cleared to zero. An inactive target node is marked *deleted* when its memory space has been recycled and made available for reuse.

⟨ Type definitions 12 ⟩ +≡

```
typedef enum {
  active, raw, deleted
} status;
```

61. Here then are the 20 precious bytes that represent a configuration. The rolling strategy allows us to get by with only one link field.

⟨ Type definitions 12 ⟩ +≡

```
typedef struct conf_struct {
  cstring s;    /* the configuration name */
  unsigned int addr;    /* where the slave keeps the generating function */
  struct conf_struct *link;    /* the next item in a hash chain or hole list */
  char lo;    /* smallest exponent of z in the current generating function */
  char hi;    /* largest exponent of z in the current generating function */
  char lim;    /* largest viable exponent of z, if this is a target */
  status state;    /* active, raw, or deleted */
} config;
```

62. ⟨ Initialize 15 ⟩ +≡

```
conf = (config *) calloc(conf_size, sizeof(config));
if (¬conf) panic("I␣can't␣allocate␣the␣config␣table");
conf_end = conf + conf_size;
```

63. The main high-level routine, called *update*, is used to add terms *p→lo* through *hi* of the generating function for configuration p to the generating function for configuration q. The special case $q = \Lambda$ is used to update the counters for polyominoes that have been completed.

```
⟨ Subroutines 5 ⟩ +≡
  void update(config *p, config *q, char hi)
  {
    if (¬q) basic_inst(add, p→addr, p→lo, hi + 1 − p→lo);
    else if (q→state ≡ raw) {
      q→state = active;
      if (q→lo ≠ p→lo ∨ q→hi ≠ hi) basic_inst(clear, cur_src, q→addr, q→hi + 1 − q→lo);
      basic_inst(copy, p→addr, q→addr + p→lo − q→lo, hi + 1 − p→lo);
    } else basic_inst(add, p→addr, q→addr + p→lo − q→lo, hi + 1 − p→lo);
  }
```

64. "Universal hashing" (*TAOCP* exercise 6.4–72) is used to get a good hash function, because most of the key bits tend to be zero.

```
#define hash_width 20    /* lg of hash table size */
#define hash_mask ((1 ≪ hash_width) − 1)
⟨ Subroutines 5 ⟩ +≡
  int mangle(cstring s)
  {
    register unsigned int h, l;
    for (l = 1, h = hash_bits[0][s.byte[0]]; l < 8; l++) h += hash_bits[l][s.byte[l]];
    return h & hash_mask;
  }
```

65. ⟨ Global variables 7 ⟩ +≡

```
  unsigned int hash_bits[8][256];    /* random bits for universal hashing */
  config *hash_table[hash_mask + 1];    /* heads of the hash chains */
```

66. The random number generator used here doesn't have to be of sensational quality. We can keep *hash_bits*[*j*][0] = 0 without loss of universality.

```
⟨ Initialize 15 ⟩ +≡
  row_end = 314159265;    /* borrow a register temporarily (bad style, sorry) */
  for (j = 0; j < 8; j++)
    for (k = 1; k < 256; k++) {
      row_end = 69069 * row_end + 1;
      hash_bits[j][k] = row_end ≫ (32 − hash_width);
    }
```

67. ⟨ Local variables 67 ⟩ ≡

```
  register int j, k;    /* all-purpose indices */
  register int row_end;    /* size of the current partial row r */
```

See also section 99.

This code is used in section 4.

68. On odd-numbered passes, *src* runs down towards *conf*, while *trg* starts at $conf\_end - 1$ and proceeds downward. On even-numbered passes, *src* runs up towards $conf\_end - 1$ and *trg* starts up from *conf*. The variables *ssrc* and *strg* have a similar significance but they refer to addresses in the slave memory.

```
⟨ Global variables 7 ⟩ +≡
  config *conf;   /* first item in the pool of configuration nodes */
  config *conf_end;   /* last item (plus 1) in the pool of configuration nodes */
  config *src;   /* the current configuration α about to be recycled */
  config *trg;   /* the first unused configuration slot */
  int ssrc, strg;   /* allocation pointers for slave counts */
```

69. When a new configuration is created, we allocate space for its generating function in the slave module. Later on, we might discover that more space is needed because another generating function (with more terms) must be combined with it. At such times, we copy the data to another slot, leaving a hole in the configuration array and in the slave's array of counters. All holes of a given size are linked together, so that they can hopefully be plugged again soon.

Here is the basic subroutine that allocates space for a configuration with $s + 1$ terms in its generating function. The subroutine has two versions, one for passes in which allocation goes upward and the other for passes in which allocation goes downward. It maintains statistics so that we can judge how fragmented the memory has become at the most stressful times.

```
⟨ Subroutines 5 ⟩ +≡
  config *get_slot_up(register int s)
  {
    register config *p = slot[s];
    if (p) {
      slot[s] = p→link;
      holes−−, sholes −= s + 1;
    } else {
      p = trg++;
      ⟨ Allocate p→addr (upward) and check that memory hasn't overflowed 70 ⟩;
    }
    p→state = raw;
    return p;
  }
```

70. ⟨ Allocate *p*→*addr* (upward) and check that memory hasn't overflowed 70 ⟩ ≡

```
  {
    if (src − trg < min_space) {
      min_space = src − trg;
      if (min_space < 0) panic("Memory␣overflow");
      min_holes = holes, space_row = r, space_col = re;
    }
    p→addr = strg;
    strg += s + 1;
    if (ssrc − strg < min_sspace) {
      min_sspace = ssrc − strg;
      if (min_sspace < 0) panic("Slave␣memory␣overflow");
      min_sholes = sholes, slave_row = r, slave_col = re;
    }
  }
```

This code is used in section 69.

71. ⟨ Global variables 7 ⟩ +≡

```
int holes;    /* current number of holes in the target area */
int sholes;    /* current number of vacated counters in slave target area */
int min_space = 1000000000;    /* how close did src and trg get? */
int min_holes;    /* and how many holes were present at that time? */
int space_row, space_col;    /* and where were we then? */
int min_sspace = 1000000000;    /* how close did ssrc and strg get? */
int min_sholes;    /* and how many wasted counters were present then? */
int slave_row, slave_col;    /* and where were we then? */
int moves;    /* the number of times a hole was created */
int configs;    /* total configurations recorded so far, mod 10^9 */
int hconfigs;    /* billions of configurations so far */
int r;    /* number of the partially filled row */
int re;    /* non-register copy of row_end */
config *slot[nmax + 1];    /* heads of the available-slot chains */
```

72. ⟨ Print statistics about this run 72 ⟩ ≡

```
printf("Altogether␣");
if (hconfigs) printf("%d%09d", hconfigs, configs);
else printf("%d", configs);
printf("␣viable␣configurations␣examined;\n");
printf("␣%d␣slots␣needed␣(with␣%d␣holes)␣in␣position␣(%d,%d);\n", conf_size − min_space,
      min_holes, space_row, space_col);
printf("␣%d␣counters␣needed␣(with␣%d␣wasted)␣in␣position␣(%d,%d);\n", slave_size − min_sspace,
      min_sholes, slave_row, slave_col);
printf("␣%d␣moves.\n", moves);
```

This code is used in sections 4 and 105.

73. ⟨ Subroutines 5 ⟩ +≡

```
config *get_slot_down(register int s)
{
  register config *p = slot[s];
  if (p) {
    slot[s] = p→link;
    holes−−, sholes −= s + 1;
  } else {
    p = trg−−;
    ⟨ Allocate p→addr (downward) and check that memory hasn't overflowed 74 ⟩;
  }
  p→state = raw;
  return p;
}
```

74. ⟨ Allocate *p→addr* (downward) and check that memory hasn't overflowed 74 ⟩ ≡

```
{
  if (trg − src < min_space) {
    min_space = trg − src;
    if (min_space < 0) panic("Memory␣overflow");
    min_holes = holes, space_row = r, space_col = re;
  }
  strg −= s + 1;
  if (strg − ssrc < min_sspace) {
    min_sspace = strg − ssrc;
    if (min_sspace < 0) panic("Slave␣memory␣overflow");
    min_sholes = sholes, slave_row = r, slave_col = re;
  }
  p→addr = strg + 1;
}
```

This code is used in section 73.

75. The *move_down* and *move_up* subroutines are invoked when an active target configuration *p* needs more space for its generating function. The global variable *hash* will have been set so that *hash_table*[*hash*] = *p*; we effectively move that configuration to another place in the sequential list of targets, and return a pointer to the new place. The former node *p* is now marked *deleted*, but its *addr* field remains valid (in case *get_slot* is able to reuse it).

⟨ Subroutines 5 ⟩ +≡

```
config *move_down(config *p, int lo, int hi)
{
  register config *q, *r;
  register int s = p→lo, t = p→hi;
  r = p→link;
  p→link = slot[t − s], slot[t − s] = p;
  p→state = deleted;
  holes++, sholes += t − s + 1;
  if (s > lo) s = lo;
  if (t < hi) t = hi;
  q = get_slot_down(t − s);
  q→lo = s, q→hi = t;
  q→s = p→s, q→lim = p→lim;
  hash_table[hash] = q, q→link = r;
  update(p, q, p→hi);
  moves++;
  return q;
}
```

76. ⟨ Subroutines 5 ⟩ +≡

```
config *move_up(config *p, int lo, int hi)
{
  register config *q, *r;
  register int s = p→lo, t = p→hi;
  r = p→link;
  p→link = slot[t − s], slot[t − s] = p;
  p→state = deleted;
  holes++, sholes += t − s + 1;
  if (s > lo) s = lo;
  if (t < hi) t = hi;
  q = get_slot_up(t − s);
  q→lo = s, q→hi = t;
  q→s = p→s, q→lim = p→lim;
  hash_table[hash] = q, q→link = r;
  update(p, q, p→hi);
  moves++;
  return q;
}
```

77. The main loop. Now that we have some infrastructure in place, we can map out the top levels of this program's main processing cycle.

We start with an all-*zero* configuration, at the very top of the width-w array of cells that we will conceptually traverse; this configuration serves as the great-...-great grandparent of all other configurations that will arise later. The slave module will begin by giving this configuration the trivial generating function '1' (namely z^0) in its counter cell number 0, meaning that there's just one way to reach the initial configuration, and that no cells are occupied so far.

There is no need to initialize *conf*[0].*lim* or *conf*[0].*link*, because those fields are used only when a configuration is a target. The other fields—namely *conf*[0].*s*, *conf*[0].*addr*, *conf*[0].*lo*, *conf*[0].*hi*, and *conf*[0].*state*— are initially zero by the conventions of C, and luckily those zeros happen to be just what we want.

```
⟨ Initialize 15 ⟩ +≡
  r = 0, row_end = w;
  trg = conf + 1;      /* pretend that the previous pass produced a single result */
  strg = 1;
```

78. Once again it seems best to write two nearly identical pieces of code, depending on whether the allocation is actually moving upward or downward. (We're supposed to be hoarding memory, but the space required for this program is small potatoes.)

```
⟨ Output instructions for the postprocessor 78 ⟩ ≡
  while (1) {
    ⟨ Get ready for a downward pass, or break when done 79 ⟩;
    ⟨ Pass downward over all configurations created on the previous pass 80 ⟩;
    ⟨ Get ready for an upward pass, or break when done 82 ⟩;
    ⟨ Pass upward over all configurations created on the previous pass 83 ⟩;
  }
```

This code is used in section 4.

79.
```
⟨ Get ready for a downward pass, or break when done 79 ⟩ ≡
  if (row_end < w) {
    row_end++;
    printf("Beginning␣column␣%d", row_end);
    ⟨ Print current stats and clear the hash/slot tables 81 ⟩;
  } else {
    if (r) {
      printf("Finished␣row␣%d", r);
      ⟨ Print current stats and clear the hash/slot tables 81 ⟩;
      if (r > w) put_inst(sync, r);
    }
    ⟨ Check if this run has gone on too long 105 ⟩;
    r++, row_end = 1;
  }
  if (trg ≡ conf) break;      /* the previous pass was sterile */
  src = trg − 1;      /* start the source pointer at the highest occupied node */
  ssrc = strg − 1;      /* and the highest occupied counter position */
  trg = conf_end − 1;      /* start the target pointer at the highest unoccupied node */
  strg = slave_size − 1;      /* and the highest unoccupied counter position */
  re = row_end;
```

This code is used in section 78.

80. ⟨ Pass downward over all configurations created on the previous pass 80 ⟩ ≡

```
while ($src \ge conf$) {
  if ($src\to state \equiv active$) {
    $unpackit(src\to s)$;    /* Put the source configuration $\alpha$ into $sc$ and $c$ */
    if ($verbose$) {
      $print\_config(row\_end)$; $printf$("\n");
    }
    ⟨ Change array $c$ for target $\alpha_0$ 84 ⟩;
    if ($viable$) ⟨ Process target configuration $c$ (downward) 97 ⟩;
    for ($k = 1$; $k \le w$; $k{+}{+}$) $c[k] = sc[k]$;
    ⟨ Change array $c$ for target $\alpha_1$ 90 ⟩;
    if ($viable$) ⟨ Process target configuration $c$ (downward) 97 ⟩;
  }
  $ssrc = src\to addr - 1$;
  $src{-}{-}$;    /* the old $src$ node is now outta here */
}
```

This code is used in section 78.

81. Timely progress reports let the user know that we are still chugging along. We are about to start an upward pass if and only if $src \equiv conf - 1$.

⟨ Print current stats and clear the hash/slot tables 81 ⟩ ≡

```
if ($src \equiv conf - 1$) $printf$("␣(%d,%d,", $conf\_end - 1 - trg$, $slave\_size - 1 - strg$);
else $printf$("␣(%d,%d,", $trg - conf$, $strg - n - 1$);
$printf$("%d,%d,%d,%d,%d)\n", $conf\_size - min\_space$, $min\_holes$, $slave\_size - min\_sspace$, $min\_sholes$,
    $bytes\_out$);
⟨ Print and clear the hash/slot tables 109 ⟩;
$fflush(stdout)$;
```

This code is used in sections 79 and 82.

82. Counter positions 1 through n in the slave memory are reserved for the final polyomino counts.

⟨ Get ready for an upward pass, or **break** when done 82 ⟩ ≡

```
if ($row\_end < w$) {
  $row\_end{+}{+}$;
  $printf$("Beginning␣column␣%d", $row\_end$);
  ⟨ Print current stats and clear the hash/slot tables 81 ⟩;
} else {
  if ($r$) {
    $printf$("Finished␣row␣%d", $r$);
    ⟨ Print current stats and clear the hash/slot tables 81 ⟩;
    if ($r > w$) $put\_inst(sync, r)$;
  }
  $r{+}{+}$, $row\_end = 1$;
}
if ($trg \equiv conf\_end - 1$) break;    /* the previous pass was sterile */
$src = trg + 1$;    /* start the source pointer at the lowest occupied node */
  /* and we'll soon set $ssrc$ to $src\to addr$, which equals $strg + 1$, the lowest occupied counter */
$trg = conf$;    /* start the target pointer at the lowest unoccupied node */
$strg = n + 1$;    /* and the lowest unoccupied counter position */
$re = row\_end$;
```

This code is used in section 78.

83. ⟨ Pass upward over all configurations created on the previous pass 83 ⟩ ≡

```
while (src < conf_end) {
  if (src→state ≡ active) {
    ssrc = src→addr;
    unpackit(src→s);     /* Put the source configuration α into sc and c */
    if (verbose) {
      print_config(row_end); printf("\n");
    }
    ⟨ Change array c for target α_0 84 ⟩;
    if (viable) ⟨ Process target configuration c (upward) 103 ⟩;
    for (k = 1; k ≤ w; k++) c[k] = sc[k];
    ⟨ Change array c for target α_1 90 ⟩;
    if (viable) ⟨ Process target configuration c (upward) 103 ⟩;
  }
  src++;     /* the old src node is now outta here */
}
```

This code is used in section 78.

84. Nitty-gritty. The basic logic of a so-called "transfer-matrix" approach is embedded in the following program steps, which change a configuration string when a new cell in row r is or is not to be occupied. Here, for example, we observe that when the previous configuration has '`1‸(`' at the end of a partial row, and if we occupy the new cell, the new configuration has '`(-‸`' instead. But if we don't occupy that cell, the new configuration has '`10‸`' and a further change must also be made because of the `(` that has disappeared.

Most of the cases that arise are completely straightforward. But each of the thirty combinations of two adjacent codes must of course be handled perfectly. The following chart summarizes what the program is supposed to do when the new cell is being left vacant.

	`0`	`1`	`(`	`-`	`)`
`0`	`00`	[a]	`00`[b]	`00`[c]	`00`[d]
`1`	`10`	[a]	`10`[b]	`10`	`10`[d]
`(`	`(0`	[e]	[e]	`(0`	`(0`[d]
`-`	`-0`	[a]	`-0`[b]	`-0`	`-0`[d]
`)`	`)0`	[a]	`-0`[b]	`)0`	`-0`[d]
left edge	`0`	[e]	[e]	`0`[c]	[a]

[a] Not viable
[b] Downgrade the successor of `(`
[c] A polyomino may have been completed
[d] Downgrade the predecessor of `)`
[e] Impossible case

Special handling is necessary when a `-` is eliminated, if it is preceded or followed by nothing but zeros; for example, `0(010-` must become `010(00`, and `-010()0-0` must become either `00)0()0(0` or `)010(-000`. These somewhat unusual cases are approached cautiously in the program below.

```
#define f(x, y)  ((x ≪ 3) + y)

⟨ Change array c for target α₀ 84 ⟩ ≡
  pair = f(sc[row_end − 1], sc[row_end]);
  c[row_end] = zero;
  viable = 1;
  switch (pair) {
  case f(zero, one): case f(one, one): case f(mid, one): case f(rt, one): case f(eol, rt): viable = 0;
      /* component would be isolated */
  case f(zero, zero): case f(one, zero): case f(lft, zero): case f(mid, zero): case f(rt, zero):
    case f(eol, zero): case f(lft, mid): case f(mid, mid): break;
  case f(zero, lft): case f(one, lft): case f(mid, lft): case f(rt, lft):
    ⟨ Downgrade the successor of the lft 86 ⟩; break;
  case f(zero, rt): case f(one, rt): case f(lft, rt): case f(mid, rt): case f(rt, rt):
    ⟨ Downgrade the predecessor of the rt 87 ⟩; break;
  case f(zero, mid): case f(eol, mid): ⟨ Cautiously delete a mid that may be leftmost 88 ⟩;
  case f(one, mid): case f(rt, mid): ⟨ Cautiously delete a mid that may be rightmost 89 ⟩;
    break;
  case f(lft, one): case f(lft, lft): case f(eol, one): case f(eol, lft):
    panic("Impossible␣configuration");
  default: panic("Impossible␣pair");
  }
```

This code is used in sections 80 and 83.

85. ⟨ Global variables 7 ⟩ +≡

```
  int pair;    /* the two codes surrounding the '‸' in sc */
  int viable;    /* might the target configuration lead to a relevant polyomino? */
```

86. In this step, we have just zeroed out a left parenthesis. If that `(` is followed by a `-`, we change the `-` to `(`; if it is followed by a `)`, we change the `)` to `1`.

Here "followed by" really means "followed on the same level by," because nested subcomponents may intervene. We therefore need a level counter, j, as we scan to the right.

If the `(` had no successor because it simply marked a connection to the right edge, we shouldn't have deleted it; its component is now disconnected, so we set *viable* to zero.

```
⟨ Downgrade the successor of the lft 86 ⟩ ≡
  for (k = row_end + 1, j = 0; ; k++) {
    switch (c[k]) {
    case lft: j++;
    case zero: case one: continue;
    case mid: if (j) continue;
      c[k] = lft; break;
    case rt: if (j) { j--; continue; }
      c[k] = one; break;
    case eol: if (j) panic("Unexpected␣eol");
      viable = 0;
    }
    break;
  }
```

This code is used in section 84.

87. Contrariwise, an erased `)` is like an erased `(` but vice versa.

```
⟨ Downgrade the predecessor of the rt 87 ⟩ ≡
  for (k = row_end − 1, j = 0; ; k−−) {
    switch (c[k]) {
    case rt: j++;
    case zero: case one: continue;
    case mid: if (j) continue;
      c[k] = rt; break;
    case lft: if (j) { j−−; continue; }
      c[k] = one; break;
    case eol: if (j) panic("Unexpected␣eol");
      viable = 0;
    }
    break;
  }
```

This code is used in section 84.

88. ⟨Cautiously delete a *mid* that may be leftmost 88⟩ ≡

```
for (k = row_end − 1; c[k] ≡ zero; k−−) ;
if (c[k] ≡ eol) {     /* yes, the mid was leftmost */
  for (k = row_end + 1; c[k] ≡ zero; k++) ;
  switch (c[k]) {
  case mid: case rt: case eol: break;     /* no problem */
  default: if (c[k] ≡ one) c[k] = rt, j = 0;
    else c[k] = mid, j = 1;     /* c[k] was lft */
    for (k++; ; k++) {     /* we must downgrade the successor of the mid */
      switch (c[k]) {
      case lft: j++;
      case zero: case one: continue;
      case mid: if (j) continue;
        c[k] = lft; break;
      case rt: if (j) { j−−; continue; }
        c[k] = one; break;
      case eol: panic("This␣can't␣happen");
      }
      break;
    }
  }
}
```

This code is used in section 84.

89. ⟨Cautiously delete a *mid* that may be rightmost 89⟩ ≡

```
for (k = row_end + 1; c[k] ≡ zero; k++) ;
if (c[k] ≡ eol) {     /* yes, the mid was rightmost */
  for (k = row_end − 1; c[k] ≡ zero; k−−) ;
  switch (c[k]) {
  case mid: case lft: case eol: break;     /* no problem */
  default: if (c[k] ≡ one) c[k] = lft, j = 0;
    else c[k] = mid, j = 1;     /* c[k] was rt */
    for (k−−; ; k−−) {     /* we must downgrade the predecessor of the mid */
      switch (c[k]) {
      case rt: j++;
      case zero: case one: continue;
      case mid: if (j) continue;
        c[k] = rt; break;
      case lft: if (j) { j−−; continue; }
        c[k] = one; break;
      case eol: panic("This␣can't␣happen");
      }
      break;
    }
  }
}
```

This code is used in section 84.

90. A different kind of excitement awaits us when we consider occupying the new cell.

If the cases $f(lft, mid)$, $f(lft, rt)$, $f(mid, mid)$, and $f(mid, rt)$ are modified here to set $viable = 0$, the program will count *polyomino trees* instead of normal polyominoes. (In a polyomino tree there is exactly one way to get from one cell to another via rook moves.) These are the four cases in which already-connected cells are connected again.

	`0`	`1`	`(`	`-`	`)`
`0`	`01`[k]	`01`	`0(`	`0-`	`0)`
`1`	`()`	`()`	`(-`	`--`	`-)`
`(`	`(-`	[e]	[e]	`(-`[f]	`()`[f]
`-`	`--`	`--`	`--`[g]	`--`[f]	`-)`[f]
`)`	`-)`	`-)`	`--`	`--`[h]	`-)`[h]
left edge	`)`[i]	[e]	[e]	`-`	`)`

[e] Impossible case
[f] Not viable in polyomino trees
[g] Merge with mate of `(`
[h] Merge with mate of `)`
[i] Downgrade the next component if open
[j] Downgrade the previous component if open
[k] Or possibly `0)`[i] or `0(`[j]

The somewhat unusual case `01`[k] becomes `0)`[i] if no nonzero cells lie to the left but the left edge has already been occupied somewhere in the rows above. It becomes `0(`[j] if no nonzero cells lie to the right but the right edge has already been occupied somewhere in the rows above.

```
⟨ Change array c for target $\alpha_1$ 90 ⟩ ≡
  viable = 1;
  src→lo++, src→hi++;    /* implicitly multiply the generating function $g(\alpha)$ by $z$ */
  switch (pair) {
  case f(one, zero): case f(one, one): c[row_end − 1] = lft, c[row_end] = rt;
  case f(zero, one): case f(zero, lft): case f(zero, mid): case f(zero, rt): case f(lft, mid):
    case f(lft, rt): case f(mid, mid): case f(mid, rt): case f(eol, mid): case f(eol, rt): break;
  case f(one, lft): c[row_end − 1] = lft, c[row_end] = mid; break;
  case f(one, mid): case f(one, rt): c[row_end − 1] = mid; break;
  case f(lft, zero): case f(mid, zero): case f(mid, one): c[row_end] = mid; break;
  case f(mid, lft): c[row_end] = mid;
    ⟨ Merge with the mate of the former lft 91 ⟩; break;
  case f(rt, zero): case f(rt, one): c[row_end − 1] = mid, c[row_end] = rt; break;
  case f(rt, lft): c[row_end − 1] = c[row_end] = mid; break;
  case f(rt, mid): case f(rt, rt): c[row_end − 1] = mid;
    ⟨ Merge with the mate of the former rt 92 ⟩; break;
  case f(eol, zero): c[row_end] = rt;
    ⟨ Downgrade the next component if it is open 93 ⟩; break;
  case f(zero, zero): ⟨ Cautiously introduce a new one 94 ⟩; break;
  case f(lft, one): case f(lft, lft): case f(eol, one): case f(eol, lft):
    panic("Impossible␣configuration");
  default: panic("Impossible␣pair");
  }
  if (row_end ≡ w) ⟨ Make special corrections at the right edge 95 ⟩;
```

This code is used in sections 80 and 83.

91. ⟨ Merge with the mate of the former *lft* 91 ⟩ ≡

```
for (k = row_end + 1, j = 0; ; k++) {
  switch (c[k]) {
  case lft: j++;
  case zero: case one: case mid: continue;
  case rt: if (¬j) break;
    j−−; continue;
  case eol: panic("Unexpected␣eol");
  }
  c[k] = mid; break;
}
```

This code is used in section 90.

92. ⟨ Merge with the mate of the former *rt* 92 ⟩ ≡

```
for (k = row_end − 2, j = 0; ; k−−) {
  switch (c[k]) {
  case rt: j++;
  case zero: case one: case mid: continue;
  case lft: if (¬j) break;
    j−−; continue;
  case eol: panic("Unexpected␣eol");
  }
  c[k] = mid; break;
}
```

This code is used in section 90.

93. ⟨ Downgrade the next component if it is open 93 ⟩ ≡

```
for (k = 2; ; k++) {
  switch (c[k]) {
  case zero: continue;
  case mid: c[k] = lft;
  case one: case lft: case eol: break;
  case rt: c[k] = one;
  }
  break;
}
```

This code is used in section 90.

94. ⟨ Cautiously introduce a new *one* 94 ⟩ ≡

```
c[row_end] = one;
for (k = row_end − 2; c[k] ≡ zero; k−−) ;
if (¬k) {     /* we're introducing a new leftmost 1 */
  for (k = row_end + 1; ; k++) {
    switch (c[k]) {
    case zero: continue;
    case mid: c[k] = lft, c[row_end] = rt;
    case one: case lft: case eol: break;
    case rt: c[k] = one, c[row_end] = rt;
    }
    break;
  }
} else {
  for (j = row_end + 1; c[j] ≡ zero; j++) ;
  if (c[j] ≡ eol) {     /* we're introducing a new rightmost 1 */
    if (c[k] ≡ mid) c[k] = rt, c[row_end] = lft;
    else if (c[k] ≡ lft) c[k] = one, c[row_end] = lft;
  }
}
```

This code is used in section 90.

95. ⟨ Make special corrections at the right edge 95 ⟩ ≡

```
switch (c[row_end]) {
case rt: c[row_end] = mid;
case zero: case mid: case lft: break;
case one: c[row_end] = lft;
  ⟨ Downgrade the previous component if it is open 96 ⟩;
}
```

This code is used in section 90.

96. ⟨ Downgrade the previous component if it is open 96 ⟩ ≡

```
for (k = row_end − 1; ; k−−) {
  switch (c[k]) {
  case zero: continue;
  case mid: c[k] = rt;
  case one: case rt: case eol: break;
  case lft: c[k] = one;
  }
  break;
}
```

This code is used in section 95.

97. Nittier-and-grittier. The last nontrivial hurdle facing us is the problem of what to do after a target configuration has been constructed in $c[1]$ through $c[w]$. It's not a Big Problem, but it does require care, especially with respect to the generating function arithmetic.

An all-*zero* target configuration is preserved only in the first few passes, before we've reached the end of row 1.

```
⟨ Process target configuration c (downward) 97 ⟩ ≡
  {
    if (row_end ≡ w) ⟨ Canonize the configuration 100 ⟩;
    target = packit( );
    if (target.o.l ∨ target.o.h ∨ (r ≡ 1 ∧ row_end < w)) {
      ⟨ If target is already present, make p point to it 101 ⟩;
      if (¬p) ⟨ Get a downward slot for target if it is really viable 102 ⟩;
      if (p ∧ (src→lo ≤ (j = p→lim))) {
        if (src→hi < j) j = src→hi;
        if (j > p→hi ∨ src→lo < p→lo) p = move_down(p, src→lo, j);
        if (verbose) {
          printf("␣->␣"); print_config(row_end + 1); printf("\n");
        }
        update(src, p, j);
      }
    } else if (r > w) {
      if (verbose) printf("␣->␣0\n");
      update(src, Λ, src→hi);      /* polyominoes completed */
    }
  }
```

This code is used in section 80.

98. ⟨ Global variables 7 ⟩ +≡

```
  cstring target;      /* the packed name of the current target configuration */
  int hash;      /* its hash address */
```

99. ⟨ Local variables 67 ⟩ +≡

```
  register config *p;      /* current target of interest */
```

100. At the end of a row we change the configuration to its left-right reflection, if the reflection is lexicographically smaller. This reduction to a canonical form reduces the number of active configurations by a factor of nearly 2, at least for the next few passes. (The reduction can be justified by observing that we could have operated from right to left instead of from left to right, on each row that follows a left-heavy row.)

Notice that a code like `0(0` will be reflected to `0)0`.

```
⟨Canonize the configuration 100⟩ ≡
  {
    for ($j = 1, k = w$; $j \le k$; $j$++, $k$−−)
      if ($c[j] \ne reflect[c[k]]$) break;
    if ($c[j] > reflect[c[k]]$)
      for ( ; $j \le k$; $j$++, $k$−−) {
        register int $i = c[k]$;
        $c[k] = reflect[c[j]]$;
        $c[j] = reflect[i]$;
      }
  }
```

This code is used in sections 97 and 103.

101. We take care to move p to the top of its hash list, when present, and to set the global variable *hash* as required by *move_down* and *move_up*.

```
⟨If target is already present, make p point to it 101⟩ ≡
  $hash = mangle(target)$;
  $p = hash\_table[hash]$;
  if ($p \land \neg(p\to s.o.l \equiv target.o.l \land p\to s.o.h \equiv target.o.h)$) {
    register config $*q$;
    for ($q = p, p = p\to link$; $p$; $q = p, p = p\to link$)
      if ($p\to s.o.l \equiv target.o.l \land p\to s.o.h \equiv target.o.h$) break;
    if ($p$) {
      $q\to link = p\to link$;    /* remove p from its former place in the list */
      $p\to link = hash\_table[hash]$;    /* and insert it at the front */
      $hash\_table[hash] = p$;
    }
  }
```

This code is used in sections 97 and 103.

102. If the target is viable, p will be set to a fresh node with $p\to state = raw$. The reader can verify that *move_down* will not then be necessary.

```
⟨ Get a downward slot for target if it is really viable 102 ⟩ ≡
  {
    $j = connectivity(row\_end + 1)$;
    if $(r \ge w)$ $j = int\_part(j)$;
    else if $(int\_part(j) \equiv eps\_part(j))$ $j = int\_part(j) + (w - r)$;
    else $j = int\_part(j) + (w - 1 - r)$;      /* $j$ more cells are needed in a valid polyomino */
    if $(src\to lo + j \le n)$ {
      if $(\mathbin{++} configs \equiv 1000000000)$ $configs = 0, hconfigs\mathbin{++}$;
      $p = get\_slot\_down((src\to hi > n - j \mathbin{?} n - j : src\to hi) - src\to lo)$;
      $p\to link = hash\_table[hash], hash\_table[hash] = p$;
      $p\to s = target$;
      $p\to lo = src\to lo, p\to hi = src\to hi, p\to lim = n - j$;
      if $(p\to hi > p\to lim)$ $p\to hi = p\to lim$;
    }
  }
```

This code is used in section 97.

103.

```
⟨ Process target configuration c (upward) 103 ⟩ ≡
  {
    if $(row\_end \equiv w)$ ⟨ Canonize the configuration 100 ⟩;
    $target = packit()$;
    if $(target.o.l \lor target.o.h \lor (r \equiv 1 \land row\_end < w))$ {
      ⟨ If target is already present, make $p$ point to it 101 ⟩;
      if $(\neg p)$ ⟨ Get an upward slot for target if it is really viable 104 ⟩;
      if $(p \land (src\to lo \le (j = p\to lim)))$ {
        if $(src\to hi < j)$ $j = src\to hi$;
        if $(j > p\to hi \lor src\to lo < p\to lo)$ $p = move\_up(p, src\to lo, j)$;
        if $(verbose)$ {
          $printf$("␣->␣"); $print\_config(row\_end + 1)$; $printf$("\n");
        }
        $update(src, p, j)$;
      }
    } else if $(r > w)$ {
      if $(verbose)$ $printf$("␣->␣0\n");
      $update(src, \Lambda, src\to hi)$;      /* polyominoes completed */
    }
  }
```

This code is used in section 83.

104. ⟨ Get an upward slot for *target* if it is really viable 104 ⟩ ≡

```
{
  $j = connectivity(row\_end + 1)$;
  if ($r \ge w$) $j = int\_part(j)$;
  else if ($int\_part(j) \equiv eps\_part(j)$) $j = int\_part(j) + (w - r)$;
  else $j = int\_part(j) + (w - 1 - r)$;      /* $j$ more cells are needed in a valid polyomino */
  if ($src\to lo + j \le n$) {
    if ($\mathrel{+}+configs \equiv 1000000000$) $configs = 0, hconfigs\mathrel{+}+$;
    $p = get\_slot\_up((src\to hi > n - j\ ?\ n - j : src\to hi) - src\to lo)$;
    $p\to link = hash\_table[hash], hash\_table[hash] = p$;
    $p\to s = target$;
    $p\to lo = src\to lo, p\to hi = src\to hi, p\to lim = n - j$;
    if ($p\to hi > p\to lim$) $p\to hi = p\to lim$;
  }
}
```

This code is used in section 103.

105. Checkpointing. One of the goals of this program is to establish new world records. Thus, local resources are probably being stretched to their current limits, and several days of running time might well be involved.

It's prudent therefore to make the program stop at a suitable "checkpoint," firming up what has been accomplished so far; then we won't have to go back to square one when recovering from a disaster. We should also use POLYSLAVE to reduce the intermediate data at such times, thereby freeing up nearly all of the disk space we've been filling before we proceed to fill some more.

The code in this section is executed at a particularly convenient time: A new row is about to begin, and so is a new downward pass. It's as good a time as any to dump out the configuration-table-so-far in a form that can easily be used by a special version of this program to get going again when we're ready to resume. (See the change file `polynum-restart.ch` for details.)

```
#define gig_threshold 5
            /* try to avoid filling more than about twice this many gigabytes of disk space */

⟨ Check if this run has gone on too long 105 ⟩ ≡
  if (file_extension ≥ gig_threshold ∧ trg ≠ conf) {
    ⟨ Shut down the POLYSLAVE process 106 ⟩;
    sprintf(filename, "%.90s.dump", base_name);
    out_file = fopen(filename, "wb");
    if (¬out_file) panic("I␣can't␣open␣the␣dump␣file");
    ⟨ Dump all information needed to restart 107 ⟩;
    ⟨ Print statistics about this run 72 ⟩;
    printf("[%d␣bytes␣written␣on␣file␣%s.]\n", ftell(out_file), filename);
    exit(1);
  }
```

This code is used in section 79.

106. A special *sync* instruction with parameter 255 tells POLYSLAVE that it should invoke its own checkpointing activity.

```
⟨ Shut down the POLYSLAVE process 106 ⟩ ≡
  put_inst(sync, 255);
  ⟨ Empty the buffer and close the output file 16 ⟩;
  printf("Checkpoint␣stop:␣Please␣process␣that␣data␣with␣polyslave,\n");
  printf("then␣resume␣the␣computation␣with␣polynum-restart.\n");
```

This code is used in section 105.

107. Since we're at the beginning of a downward pass, the user will be able to restart this program with different values of *conf_size* and *slave_size* if desired.

```
⟨ Dump all information needed to restart 107 ⟩ ≡
  dump_data[0] = n;
  dump_data[1] = w;
  dump_data[2] = r;
  dump_data[3] = trg − conf;
  dump_data[4] = strg;
  if (fwrite(dump_data, sizeof(int), 5, out_file) ≠ 5) panic("Bad␣write␣at␣beginning␣of␣dump");
  if (fwrite(conf, sizeof(config), trg − conf, out_file) ≠ trg − conf)
    panic("Couldn't␣dump␣the␣configuration␣table");
```

This code is used in section 105.

108.
```
⟨ Global variables 7 ⟩ +≡
  int dump_data[5];     /* parameters needed to restart */
```

109. Computational experience. With a suitable change file it is not difficult to convert this program to a one-pass routine that does the evaluation directly, provided that n and w are reasonably small. For example, when $n = 30$ and $2 \le w \le 15$, all the computations were completed in 192 seconds (on 12 December 2000). The most difficult case, which took 68 seconds to complete, was for $w = 13$, when 100,488 slots (with 0 holes) and 218980 counters (with 4114 wasted) were needed.

The resulting number of n-ominoes for $n \le 30$ agreed perfectly with the answers obtained from a completely different algorithm, using my now-obsolete program POLYENUM. That program had taken more than 15 hours to count 30-ominoes, so Jensen's method ran almost 300 times faster.

Setting $n = 47$ led to much more of an adventure, of course, since all space and time requirements grow exponentially. Some runs lasted several days, and various glitches and hardware failures added to the excitement. Detailed statistics about the performance, including the histograms computed here, were helpful for planning and for diagnosing various problems.

```
#define hist_size 100
⟨ Print and clear the hash/slot tables 109 ⟩ ≡
  for (k = 0; k < hist_size; k++) hhist[k] = 0;
  for (k = 0; k ≤ nmax; k++) chist[k] = 0;
  jj = 0;
  for (k = 0; k ≤ hash_mask; k++) {
    for (p = hash_table[k], j = 0; p; p = p→link, j++) chist[p→hi − p→lo]++;
    if (j > jj) {
      if (j ≥ hist_size) j = hist_size − 1;
      jj = j;
    }
    hhist[j]++;
    hash_table[k] = Λ;
  }
  printf("Hash␣histogram:");
  for (j = 1; j ≤ jj; j++) printf("␣%d", hhist[j]);
  printf("\nCounters:");
  for (k = nmax; k ≥ 0; k−−)
    if (chist[k]) break;
  for (j = 0; j ≤ k; j++) printf("␣%d", chist[j]);
  for (k = nmax; k ≥ 0; k−−)
    if (slot[k]) break;
  if (k ≥ 0) {
    printf("\nHoles:");
    for (j = 0; j ≤ k; j++) {
      for (p = slot[j], jj = 0; p; p = p→link, jj++) ;
      printf("␣%d", jj);
      slot[j] = Λ;
    }
  }
  printf("\n");
  holes = sholes = 0;
```

This code is used in section 81.

110. ⟨ Global variables 7 ⟩ +≡

```
  int hhist[hist_size];    /* histogram of hash chain lengths */
  int chist[nmax + 1];    /* histogram of counter table lengths */
  int jj;    /* auxiliary variable for statistics calculations */
```

111. The greatest difficulty for $n = 47$ occurred when $w = 20$; indeed, more than 100 gigabytes of data were passed to POLYSLAVE in that case, and the computation lasted several days, so the checkpointing algorithm proved to be particularly helpful.

Here is a summary of the main statistics from those runs. The number of "configs" is the total of distinct configurations, summed over all passes. The number of "moves" is the number of times *move_up* or *move_down* was called to increase the space allocated to a generating function.

w	slots	counters	configs	moves	bytes	POLYNUM	POLYSLAVE
23	0.3M	0.3M	109M	3M	0.4G	14 min	5 min
22	6.2M	8.0M	2150M	129M	10.2G	267 min*	35 min*
21	28.6M	46.5M	9481M	1053M	58.8G	1911 min*	314 min*
20	40.2M	94.0M	12852M	2267M	103.6G	2960 min*	574 min*
19	31.5M	105.6M	9183M	2099M	86.8G	1803 min*	497 min*
18	19.4M	85.5M	5220M	1318M	52.9G	749 min*	324 min*
17	10.1M	58.3M	2514M	678M	26.7G	280 min*	172 min*
16	4.5M	34.2M	1091M	308M	11.9G	137 min*	104 min*
15	1.9M	18.2M	437M	127M	4.9G	51 min*	44 min*
14	0.7M	8.8M	167M	49M	2.0G	22 min	31 min
13	0.3M	4.1M	62M	19M	0.8G	8 min	12 min
12	98K	1.8M	23M	7M	0.3G	3 min	5 min
11	37K	789K	8.5M	2.6M	0.1G	84 sec	2 min
10	14K	334K	3.1M	0.9M	40M	45 sec	21 sec
9	5K	148K	1124K	340K	15M	30 sec	7 sec
8	2K	59K	402K	119K	5M	23 sec	2 sec
7	875	24K	144K	43K	1.9M	20 sec	1 sec
6	350	10K	50K	14K	618K	17 sec	0 sec
5	146	4K	17K	5K	206K	14 sec	0 sec
4	57	1484	5410	1460	59K	11 sec	0 sec
3	22	553	1658	430	17K	8 sec	0 sec
2	7	180	318	76	3K	6 sec	0 sec

* Done on computers with 1 gigabyte of memory, thanks to Andy Kacsmar of Stanford's database group.

(The case $w = 24$ was omitted because of the formula

$$8\binom{h+w-2}{w-1} - 3hw + 2h + 2w - 8,$$

which gives the total number of n-ominoes spanning an $h \times w$ rectangle when $n = h + w - 1$ and $h > 1$ and $w > 1$.)

112. I was glad to see that the allocation system used here for variable-length nodes, allowing "holes," worked quite well: More than 98 percent of the memory space was typically being put to good use when it was needed. In fact, only 8 holes were present when the maximum demand of 40,219,325 slots occurred in the runs for $n = 47$ (row 10 and column 11 when $w = 20$); only 105 counters were wasted when the maximum demand of 105,578,552 counters occurred (row 12 and column 10 when $w = 19$).

113. Joke: George Pólya was a polymath who worked on polyominoes.

114. Index.

POLYNUM