1. Intro. This program produces a DLX file that corresponds to the problem of packing a given set of polycubes into a given box.

Cells of the box have three coordinates xyz, in the range $0 \le x, y, z < 62$, specified by means of the extended hexadecimal "digits" `0`, `1`, ..., `9`, `a`, `b`, ..., `z`, `A`, `B`, ..., `Z`.

As in DLX format, any line of *stdin* that begins with '`|`' is considered to be a comment.

The first noncomment line specifies the cells of the box. It's a list of triples xyz, where each coordinate is either a single digit or a set of digits enclosed in square brackets. For example, '`[01]a[9b]`' specifies four cells, `0a9`, `0ab`, `1a9`, `1ab`. Brackets may also contain a range of items, with UNIX-like conventions; for instance, '`[0-1][a-a][9-b]`' specifies six cells, `0a9`, `0aa`, `0ab`, `1a9`, `1aa`, `1ab`, and '`[1-3][1-4][1-5]`' specifies a $3 \times 4 \times 5$ cuboid.

Individual cells may be specified more than once, but they appear just once in the box. For example,

```
[123]22    2[123]2    22[123]
```

specifies seven cells, namely `222` and its six neighbors. The cells of a box needn't be connected.

The other noncomment lines consist of a piece name followed by typical cells of that piece. These typical cells are specified in the same way as the cells of a box.

The typical cells lead to up to 24 "base placements" for a given piece, corresponding to general rotations in three-dimensional space. The piece can then be placed by choosing one of its base placements and shifting it by an arbitrary amount, provided that all such cells fit in the box. The base placements themselves need not fit in the box.

Each piece name should be distinguishable from the coordinates of the cells in the box. (For example, a piece should not be named `000` unless cell `000` isn't in the box.)

A piece that is supposed to occur more than once can be preceded by its multiplicity and an asterisk; for example, one can give its name as '`4*Z`'. (This feature will produce a file that can be handled only by DLX solvers that allow multiplicity.)

Several lines may refer to the same piece. In such cases the placements from each line are combined.

```
#define bufsize 1024 /* input lines shouldn't be longer than this */
#define maxpieces 100 /* at most this many pieces */
#define maxnodes 10000 /* at most this many elements of lists */
#define maxbases 1000 /* at most this many base placements */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  char buf[bufsize];
  ⟨ Type definitions 8 ⟩;
  ⟨ Global variables 7 ⟩;
  ⟨ Subroutines 2 ⟩;
  main()
  {
    register int i, j, k, p, q, r, t, x, y, z, dx, dy, dz, xyz0;
    register long long xa, ya, za;
    ⟨ Read the box spec 16 ⟩;
    ⟨ Read the piece specs 22 ⟩;
    ⟨ Output the DLX column-name line 31 ⟩;
    ⟨ Output the DLX rows 32 ⟩;
    ⟨ Bid farewell 33 ⟩;
  }
```

2. Low-level operations. I'd like to begin by building up some primitive subroutines that will help to parse the input and to publish the output.

For example, I know that I'll need basic routines for the input and output of radix-62 digits.

```
⟨ Subroutines 2 ⟩ ≡
  int decode(char c)
  {
    if (c ≤ '9') {
      if (c ≥ '0') return c − '0';
    } else if (c ≥ 'a') {
      if (c ≤ 'z') return c + 10 − 'a';
    } else if (c ≥ 'A' ∧ c ≤ 'Z') return c + 36 − 'A';
    if (c ≠ '\n') return −1;
    fprintf(stderr, "Incomplete␣input␣line:␣%s", buf);
    exit(−888);
  }
  char encode(int x)
  {
    if (x < 0) return '-';
    if (x < 10) return '0' + x;
    if (x < 36) return 'a' − 10 + x;
    if (x < 62) return 'A' − 36 + x;
    return '?';
  }
```

See also sections 3, 11, 12, 13, 14, and 27.

This code is used in section 1.

3. I'll also want to decode the specification of a given set of digits, starting at position p in *buf*. Subroutine *pdecode* sets the global variable *acc* to a 64-bit number that represents the digit or digits mentioned there. Then it returns the next buffer position, so that I can continue scanning.

```
⟨ Subroutines 2 ⟩ +≡
  int pdecode(register int p)
  {
    register int x;
    if (buf[p] ≠ '[') {
      x = decode(buf[p]);
      if (x ≥ 0) {
        acc = 1LL ≪ x;
        return p + 1;
      }
      fprintf(stderr, "Illegal␣digit␣at␣position␣%d␣of␣%s", p, buf);
      exit(−2);
    } else ⟨ Decode a bracketed specification 4 ⟩;
  }
```

4. We want to catch illegal syntax such as '`[-5]`', '`[1-]`', '`[3-2]`', '`[1-2-3]`', '`[3--5]`', while allowing '`[7-z32-4A5-5]`', etc. (The latter is equivalent to '`[2-57-A]`'.)

Notice that the empty specification '`[]`' is legal, but useless.

```
⟨ Decode a bracketed specification 4 ⟩ ≡
  {
    register int t, y;
    for (acc = 0, t = x = -1, p++; buf[p] ≠ ']'; p++) {
      if (buf[p] ≡ '\n') {
        fprintf(stderr, "No␣closing␣bracket␣in␣%s", buf);
        exit(-4);
      }
      if (buf[p] ≡ '-') ⟨ Get ready for a range 5 ⟩
      else {
        x = decode(buf[p]);
        if (x < 0) {
          fprintf(stderr, "Illegal␣bracketed␣digit␣at␣position␣%d␣of␣%s", p, buf);
          exit(-3);
        }
        if (t < 0) acc |= 1LL ≪ x;
        else ⟨ Complete the range from t to x 6 ⟩;
      }
    }
    return p + 1;
  }
```

This code is used in section 3.

5.

```
⟨ Get ready for a range 5 ⟩ ≡
  {
    if (x < 0 ∨ buf[p + 1] ≡ ']') {
      fprintf(stderr, "Illegal␣range␣at␣position␣%d␣of␣%s", p, buf);
      exit(-5);
    }
    t = x, x = -1;
  }
```

This code is used in section 4.

6.

```
⟨ Complete the range from t to x 6 ⟩ ≡
  {
    if (x < t) {
      fprintf(stderr, "Decreasing␣range␣at␣position␣%d␣of␣%s", p, buf);
      exit(-6);
    }
    acc |= (1LL ≪ (x + 1)) - (1LL ≪ t);
    t = x = -1;
  }
```

This code is used in section 4.

7.

```
⟨ Global variables 7 ⟩ ≡
  long long acc;      /* accumulated bits representing coordinate numbers */
  long long accx, accy, accz;      /* the bits for each dimension of a partial spec */
```

See also sections 10, 20, and 21.

This code is used in section 1.

8. Data structures. The given box is remembered as a sorted list of cells xyz, represented as a linked list of packed integers $(x \ll 16) + (y \ll 8) + z$. The base placements of each piece are also remembered in the same way.

All of the relevant information appears in a structure of type **box**.

```
⟨ Type definitions 8 ⟩ ≡
  typedef struct {
    int list;      /* link to the first of the packed triples xyz */
    int size;      /* the number of items in that list */
    int xmin, xmax, ymin, ymax, zmin, zmax;      /* extreme coordinates */
    int pieceno;      /* the piece, if any, for which this is a base placement */
  } box;
```

See also section 9.

This code is used in section 1.

9. Elements of the linked lists appear in structures of type **node**.

All of the lists will be rather short. So I make no effort to devise methods that are asymptotically efficient as things get infinitely large. My main goal is to have a program that's simple and correct. (And I hope that it will also be easy and fun to read, when I need to refer to it or modify it.)

```
⟨ Type definitions 8 ⟩ +≡
  typedef struct {
    int xyz;      /* data stored in this node */
    int link;      /* the next node of the list, if any */
  } node;
```

10. All of the nodes appear in the array *elt*. I allocate it statically, because it doesn't need to be very big.

```
⟨ Global variables 7 ⟩ +≡
  node elt[maxnodes];      /* the nodes */
  int curnode;      /* the last node that has been allocated so far */
  int avail;      /* the stack of recycled nodes */
```

11. Subroutine *getavail* allocates a new node when needed.

```
⟨ Subroutines 2 ⟩ +≡
  int getavail(void)
  {
    register int p = avail;
    if (p) {
      avail = elt[avail].link;
      return p;
    }
    p = ++curnode;
    if (p < maxnodes) return p;
    fprintf(stderr, "Overflow! Recompile me by making maxnodes bigger than %d.\n", maxnodes);
    exit(-666);
  }
```

12. Conversely, *putavail* recycles a list of nodes that are no longer needed.

```
⟨ Subroutines 2 ⟩ +≡
  void putavail(int p)
  {
    register int q;
    if (p) {
      for (q = p; elt[q].link; q = elt[q].link) ;
      elt[q].link = avail;
      avail = p;
    }
  }
```

13. The *insert* routine puts new (x, y, z) data into the list of *newbox*, unless (x, y, z) is already present.

```
⟨ Subroutines 2 ⟩ +≡
  void insert(int x, int y, int z)
  {
    register int p, q, r, xyz;
    xyz = (x ≪ 16) + (y ≪ 8) + z;
    for (q = 0, p = newbox.list; p; q = p, p = elt[p].link) {
      if (elt[p].xyz ≡ xyz) return;      /* nothing to be done */
      if (elt[p].xyz > xyz) break;       /* we've found the insertion point */
    }
    r = getavail( );
    elt[r].xyz = xyz, elt[r].link = p;
    if (q) elt[q].link = r;
    else newbox.list = r;
    newbox.size++;
    if (x < newbox.xmin) newbox.xmin = x;
    if (y < newbox.ymin) newbox.ymin = y;
    if (z < newbox.zmin) newbox.zmin = z;
    if (x > newbox.xmax) newbox.xmax = x;
    if (y > newbox.ymax) newbox.ymax = y;
    if (z > newbox.zmax) newbox.zmax = z;
  }
```

14. Although this program is pretty simple, I do want to watch it in operation before I consider it to be reasonably well debugged. So here's a subroutine that's useful only for diagnostic purposes.

```
⟨ Subroutines 2 ⟩ +≡
  void printbox(box *b)
  {
    register int p, x, y, z;
    fprintf(stderr, "Piece␣%d,␣size␣%d,␣%d..%d␣%d..%d␣%d..%d:\n", b→pieceno, b→size, b→xmin, b→xmax,
        b→ymin, b→ymax, b→zmin, b→zmax);
    for (p = b→list; p; p = elt[p].link) {
      x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
      fprintf(stderr, "␣%c%c%c", encode(x), encode(y), encode(z));
    }
    fprintf(stderr, "\n");
  }
```

15. Inputting the given box. Now we're ready to look at the xyz specifications of the box to be filled. As we read them, we remember the cells in the box called *newbox*. Then, for later convenience, we also record them in a three-dimensional array called *occupied*.

16. ⟨ Read the box spec 16 ⟩ ≡

```
while (1) {
  if (¬fgets(buf, bufsize, stdin)) {
    fprintf(stderr, "Input␣file␣ended␣before␣the␣box␣specification!\n");
    exit(−9);
  }
  if (buf[strlen(buf) − 1] ≠ '\n') {
    fprintf(stderr, "Overflow!␣Recompile␣me␣by␣making␣bufsize␣bigger␣than␣%d.\n", bufsize);
    exit(−667);
  }
  printf("|␣%s", buf);      /* all input lines are echoed as DLX comments */
  if (buf[0] ≠ '|') break;
}
p = 0;
⟨ Put the specified cells into newbox, starting at buf[p] 17 ⟩;
givenbox = newbox;
⟨ Set up the occupied table 19 ⟩;
```

This code is used in section 1.

17. This spec-reading code will also be useful later when I'm inputting the typical cells of a piece.

⟨ Put the specified cells into *newbox*, starting at *buf*[*p*] 17 ⟩ ≡

```
newbox.list = newbox.size = 0;
newbox.xmin = newbox.ymin = newbox.zmin = 62;
newbox.xmax = newbox.ymax = newbox.zmax = −1;
for ( ; buf[p] ≠ '\n'; p++) {
  if (buf[p] ≠ '␣') ⟨ Scan an xyz spec 18 ⟩;
}
```

This code is used in sections 16 and 23.

18. I could make this faster by using bitwise trickery. But what the heck.

```
⟨ Scan an xyz spec 18 ⟩ ≡
  {
    p = pdecode(p), accx = acc;
    p = pdecode(p), accy = acc;
    p = pdecode(p), accz = acc;
    if (buf[p] ≠ '␣') {
      if (buf[p] ≡ '\n') p−−;      /* we'll reread the newline character */
      else {
        fprintf(stderr, "Missing␣space␣at␣position␣%d␣of␣%s", p, buf);
        exit(−11);
      }
    }
    for (x = 0, xa = accx; xa; x++, xa ≫= 1)
      if (xa & 1) {
        for (y = 0, ya = accy; ya; y++, ya ≫= 1)
          if (ya & 1) {
            for (z = 0, za = accz; za; z++, za ≫= 1)
              if (za & 1) insert(x, y, z);
          }
      }
  }
```

This code is used in section 17.

19.

```
⟨ Set up the occupied table 19 ⟩ ≡
  for (p = givenbox.list; p; p = elt[p].link) {
    x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
    occupied[x][y][z] = 1;
  }
```

This code is used in section 16.

20.

```
⟨ Global variables 7 ⟩ +≡
  box newbox;      /* the current specifications are placed here */
  char occupied[64][64][64];      /* does the box occupy a given cell? */
  box givenbox;
```

21. Inputting the given pieces. After I've seen the box, the remaining noncomment lines of the input file are similar to the box line, except that they begin with a piece name.

This name can be any string of one to eight nonspace characters allowed by DLX format, followed by a space. It should also not be the same as a position of the box.

I keep a table of the distinct piece names that appear, and their multiplicities.

And of course I also compute and store all of the base placements that correspond to the typical cells that are specified.

```
⟨ Global variables 7 ⟩ +≡
  char names[maxpieces][8];      /* the piece names seen so far */
  int piececount;      /* how many of them are there? */
  int mult[maxpieces];      /* what is the multiplicity? */
  box base[maxbases];      /* the base placements seen so far */
  int basecount;      /* how many of them are there? */
```

22. ⟨ Read the piece specs 22 ⟩ ≡

```
  while (1) {
    if (¬fgets(buf, bufsize, stdin)) break;
    if (buf[strlen(buf) − 1] ≠ '\n') {
      fprintf(stderr, "Overflow! Recompile me by making bufsize bigger than %d.\n", bufsize);
      exit(−777);
    }
    printf("| %s", buf);      /* all input lines are echoed as DLX comments */
    if (buf[0] ≡ '|') continue;
    ⟨ Read a piece spec 23 ⟩;
  }
```

This code is used in section 1.

23. ⟨ Read a piece spec 23 ⟩ ≡

```
  ⟨ Read the piece name, and find it in the names table at position k 24 ⟩;
  newbox.pieceno = k;      /* now buf[p] is the space following the name */
  ⟨ Put the specified cells into newbox, starting at buf[p] 17 ⟩;
  ⟨ Normalize the cells of newbox 26 ⟩;
  base[basecount] = newbox;
  ⟨ Create the other base placements equivalent to newbox 28 ⟩;
```

This code is used in section 22.

24. ⟨ Read the piece name, and find it in the *names* table at position k 24 ⟩ ≡

```
if (buf[1] ≠ '*') i = 1, p = q = 0;
else {
  i = decode(buf[0]), p = q = 2;      /* prepare for multiplicity i */
  if (i < 0) {
    fprintf(stderr, "Unknown multiplicity: %s", buf);
    exit(−4);
  }
}
for ( ; buf[p] ≠ '\n'; p++) {
  if (buf[p] ≡ ' ') break;
  if (buf[p] ≡ '|' ∨ buf[p] ≡ ':' ∨ buf[p] ≡ '*') {
    fprintf(stderr, "Illegal character in piece name: %s", buf);
    exit(−8);
  }
}
if (buf[p] ≡ '\n') {
  fprintf(stderr, "(Empty %s is being ignored)\n", p ≡ 0 ? "line" : "piece");
  continue;
}
⟨ Store the name in names[piececount] and check its validity 25 ⟩;
for (k = 0; ; k++)
  if (strncmp(names[k], names[piececount], 8) ≡ 0) break;
if (k ≡ piececount) {      /* it's a new name */
  if (++piececount > maxpieces) {
    fprintf(stderr, "Overflow! Recompile me by making maxpieces bigger than %d.\n", maxpieces);
    exit(−668);
  }
}
if (¬mult[k]) mult[k] = i;
else if (mult[k] ≠ i) {
  fprintf(stderr, "Inconsistent multiplicities for piece %.8s, %d vs %d!\n", names[k], mult[k],
      i);
  exit(−6);
}
```

This code is used in section 23.

25. ⟨ Store the name in *names*[*piececount*] and check its validity 25 ⟩ ≡

```
if (p ≡ q ∨ p > q + 8) {
  fprintf(stderr, "Piece␣name␣is␣nonexistent␣or␣too␣long:␣%s", buf);
  exit(−7);
}
for (j = q; j < p; j++) names[piececount][j − q] = buf[j];
if (p ≡ q + 3) {
  x = decode(names[piececount][0]);
  y = decode(names[piececount][1]);
  z = decode(names[piececount][2]);
  if (x ≥ 0 ∧ y ≥ 0 ∧ z ≥ 0 ∧ occupied[x][y][z]) {
    fprintf(stderr, "Piece␣name␣conflicts␣with␣board␣position:␣%s", buf);
    exit(−333);
  }
}
```

This code is used in section 24.

26. It's a good idea to "normalize" the typical cells of a piece, by making the *xmin*, *ymin*, *zmin* fields of *newbox* all zero.

⟨ Normalize the cells of *newbox* 26 ⟩ ≡

```
xyz0 = (newbox.xmin ≪ 16) + (newbox.ymin ≪ 8) + newbox.zmin;
if (xyz0) {
  for (p = newbox.list; p; p = elt[p].link) elt[p].xyz −= xyz0;
  newbox.xmax −= newbox.xmin, newbox.ymax −= newbox.ymin, newbox.zmax −= newbox.zmin;
  newbox.xmin = newbox.ymin = newbox.zmin = 0;
}
```

This code is used in section 23.

27. Transformations. Now we get to the interesting part of this program, as we try to find all of the base placements that are obtainable from a given set of typical cells.

The method is a simple application of breadth-first search: Starting at the newly created base, we make sure that every elementary transformation of every known placement is also known.

This procedure requires a simple subroutine to check whether or not two placements are equal. We can assume that both placements are normalized, and that both have the same size. Equality testing is easy because the lists have been sorted.

```
⟨ Subroutines 2 ⟩ +≡
  int equality(int b)
  {    /* return 1 if base[b] matches newbox */
    register int p, q;
    for (p = base[b].list, q = newbox.list; p; p = elt[p].link, q = elt[q].link)
      if (elt[p].xyz ≠ elt[q].xyz) return 0;
    return 1;
  }
```

28. Just two elementary transformations suffice to generate them all.

```
⟨ Create the other base placements equivalent to newbox 28 ⟩ ≡
  j = basecount, k = basecount + 1;    /* bases j thru k − 1 have been checked */
  while (j < k) {
    ⟨ Set newbox to base[j] transformed by xy rotation 29 ⟩;
    for (i = basecount; i < k; i++)
      if (equality(i)) break;
    if (i < k) putavail(newbox.list);    /* already known */
    else base[k++] = newbox;    /* we've found a new one */
    ⟨ Set newbox to base[j] transformed by xyz cycling 30 ⟩;
    for (i = basecount; i < k; i++)
      if (equality(i)) break;
    if (i < k) putavail(newbox.list);    /* already known */
    else base[k++] = newbox;    /* we've found a new one */
    j++;
  }
  basecount = k;
  if (basecount + 24 > maxbases) {
    fprintf(stderr, "Overflow! Recompile me by making maxbases bigger than %d.\n",
        basecount + 23);
    exit(−669);
  }
```

This code is used in section 23.

29. The first elementary transformation replaces (x, y, z) by $(y, -x, z)$. It corresponds to 90-degree rotation about a vertical axis.

```
⟨ Set newbox to base[j] transformed by xy rotation 29 ⟩ ≡
  newbox.size = newbox.list = 0;
  newbox.xmax = base[j].ymax, t = newbox.ymax = base[j].xmax, newbox.zmax = base[j].zmax;
  for (p = base[j].list; p; p = elt[p].link) {
    x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
    insert(y, t − x, z);
  }
```

This code is used in section 28.

30. The other elementary transformation replaces (x, y, z) by (y, z, x). It corresponds to 120-degree rotation about a major diagonal.

```
⟨ Set newbox to base[j] transformed by xyz cycling 30 ⟩ ≡
  newbox.size = newbox.list = 0;
  newbox.xmax = base[j].ymax, newbox.ymax = base[j].zmax, newbox.zmax = base[j].xmax;
  for (p = base[j].list; p; p = elt[p].link) {
    x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
    insert(y, z, x);
  }
```

This code is used in section 28.

31. Finishing up. In previous parts of this program, I've terminated abruptly when finding malformed input.

But when everything on *stdin* passes muster, I'm ready to publish all the information that has been gathered.

```
⟨ Output the DLX column-name line 31 ⟩ ≡
  printf("|␣this␣file␣was␣created␣by␣polycube-dlx␣from␣that␣data\n");
  for (p = givenbox.list; p; p = elt[p].link) {
    x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
    printf("␣%c%c%c", encode(x), encode(y), encode(z));
  }
  for (k = 0; k < piececount; k++) {
    if (mult[k] ≡ 1) printf("␣%.8s", names[k]);
    else printf("␣%c*%.8s", encode(mult[k]), names[k]);
  }
  printf("\n");
```

This code is used in section 1.

32.

```
⟨ Output the DLX rows 32 ⟩ ≡
  for (j = 0; j < basecount; j++) {
    for (dx = givenbox.xmin; dx ≤ givenbox.xmax − base[j].xmax; dx++)
      for (dy = givenbox.ymin; dy ≤ givenbox.ymax − base[j].ymax; dy++)
        for (dz = givenbox.zmin; dz ≤ givenbox.zmax − base[j].zmax; dz++) {
          for (p = base[j].list; p; p = elt[p].link) {
            x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
            if (¬occupied[x + dx][y + dy][z + dz]) break;
          }
          if (¬p) {      /* they're all in the box */
            printf("%.8s", names[base[j].pieceno]);
            for (p = base[j].list; p; p = elt[p].link) {
              x = elt[p].xyz ≫ 16, y = (elt[p].xyz ≫ 8) & #ff, z = elt[p].xyz & #ff;
              printf("␣%c%c%c", encode(x + dx), encode(y + dy), encode(z + dz));
            }
            printf("\n");
          }
        }
  }
```

This code is used in section 1.

33. Finally, when I've finished outputting the desired DLX file, it's time to say goodbye by summarizing what I did.

```
⟨ Bid farewell 33 ⟩ ≡
  fprintf(stderr, "Altogether␣%d␣cells,␣%d␣pieces,␣%d␣base␣placements,␣%d␣nodes.\n",
      givenbox.size, piececount, basecount, curnode + 1);
```

This code is used in section 1.

34. Index.

⟨ Bid farewell 33 ⟩ Used in section 1.
⟨ Complete the range from t to x 6 ⟩ Used in section 4.
⟨ Create the other base placements equivalent to *newbox* 28 ⟩ Used in section 23.
⟨ Decode a bracketed specification 4 ⟩ Used in section 3.
⟨ Get ready for a range 5 ⟩ Used in section 4.
⟨ Global variables 7, 10, 20, 21 ⟩ Used in section 1.
⟨ Normalize the cells of *newbox* 26 ⟩ Used in section 23.
⟨ Output the DLX column-name line 31 ⟩ Used in section 1.
⟨ Output the DLX rows 32 ⟩ Used in section 1.
⟨ Put the specified cells into *newbox*, starting at *buf*[p] 17 ⟩ Used in sections 16 and 23.
⟨ Read a piece spec 23 ⟩ Used in section 22.
⟨ Read the box spec 16 ⟩ Used in section 1.
⟨ Read the piece name, and find it in the *names* table at position k 24 ⟩ Used in section 23.
⟨ Read the piece specs 22 ⟩ Used in section 1.
⟨ Scan an *xyz* spec 18 ⟩ Used in section 17.
⟨ Set up the *occupied* table 19 ⟩ Used in section 16.
⟨ Set *newbox* to *base*[j] transformed by *xyz* cycling 30 ⟩ Used in section 28.
⟨ Set *newbox* to *base*[j] transformed by *xy* rotation 29 ⟩ Used in section 28.
⟨ Store the name in *names*[*piececount*] and check its validity 25 ⟩ Used in section 24.
⟨ Subroutines 2, 3, 11, 12, 13, 14, 27 ⟩ Used in section 1.
⟨ Type definitions 8, 9 ⟩ Used in section 1.

POLYCUBE-DLX