March 22, 2002 at 22:33

1. Introduction. The ur-game for computers — Adventure — was originally written by Will Crowther in 1975 and greatly extended by Don Woods in 1976. I have taken Woods's original FORTRAN program for Adventure Version 1.0 and recast it in the CWEB idiom.

I remember being fascinated by this game when John McCarthy showed it to me in 1977. I started with no clues about the purpose of the game or what I should do; just the computer's comment that I was at the end of a forest road facing a small brick building. Little by little, the game revealed its secrets, just as its designers had cleverly plotted. What a thrill it was when I first got past the green snake! Clearly the game was potentially addictive, so I forced myself to stop playing — reasoning that it was great fun, sure, but traditional computer science research is great fun too, possibly even more so.

Now here I am, 21 years later, returning to the great Adventure after having indeed had many exciting adventures in Computer Science. I believe people who have played this game will be able to extend their fun by reading its once-secret program. Of course I urge everybody to *play the game first, at least ten times*, before reading on. But you cannot fully appreciate the astonishing brilliance of its design until you have seen all of the surprises that have been built in.

I believe this program is entirely faithful to the behavior of Adventure Version 1.0, except that I have slightly edited the computer messages (mostly so that they use both lowercase and uppercase letters). I have also omitted Woods's elaborate machinery for closing the cave during the hours of prime-time computing; I believe John McCarthy insisted on this, when he saw the productivity of his AI Lab falling off dramatically (although it is rumored that he had a special version of the program that allowed him to play whenever he wanted). And I have not adopted the encryption scheme by which Woods made it difficult for users to find any important clues in the binary program file or core image; such modifications would best be done by making a special version of CTANGLE. All of the spelunking constraints and interactive behavior have been retained, although the structure of this CWEB program is naturally quite different from the FORTRAN version I began with.

Many of the phrases in the following documentation have been lifted directly from comments in the FORTRAN code. Please regard me as merely a translator of the program, not as an author. I thank Don Woods for helping me check the validity of this translation.

By the way, if you don't like **goto** statements, don't read this. (And don't read any other programs that simulate multistate systems.)

— Don Knuth, September 1998

2. To run the program with, say, a UNIX shell, just type '`advent`' and follow instructions. (Many UNIX systems come with an almost identical program called '`adventure`' already built in; you might want to try it too, for comparison.)

```
#include <stdio.h>    /* basic input/output routines: fgets, printf */
#include <ctype.h>    /* isspace and tolower routines */
#include <string.h>    /* strncmp and strcpy to compare and copy strings */
#include <time.h>    /* current time, used as random number seed */
#include <stdlib.h>    /* exit */
  ⟨ Macros for subroutine prototypes 3 ⟩
  typedef enum {
    false, true
  } boolean;
  ⟨ Type definitions 5 ⟩
  ⟨ Global variables 7 ⟩
  ⟨ Subroutines 6 ⟩
  main( )
  {
    register int j, k;
    register char *p;
    ⟨ Additional local registers 22 ⟩;
    ⟨ Initialize all tables 200 ⟩;
    ⟨ Simulate an adventure, going to quit when finished 75 ⟩;
    ⟨ Deal with death and resurrection 188 ⟩;
  quit: ⟨ Print the score and say adieu 198 ⟩;
    exit(0);
  }
```

3. The subroutines of this program are declared first with a prototype, as in ANSI C, then with an old-style C function definition. The following preprocessor commands make this work correctly with both new-style and old-style compilers.

```
⟨ Macros for subroutine prototypes 3 ⟩ ≡
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ( )
#endif
```

This code is used in section 2.

4. The vocabulary. Throughout the remainder of this documentation, "you" are the user and "we" are the game author and the computer. We don't tell you what words to use, except indirectly; but we try to understand enough words of English so that you can play without undue frustration. The first part of the program specifies what we know about your language — about 300 words.

5. When you type a word, we first convert uppercase letters to lowercase; then we chop off all but the first five characters, if the word was longer than that, and we look for your word in a small hash table. Each hash table entry contains a string of length 5 or less, and two additional bytes for the word's type and meaning. Four types of words are distinguished: *motion_type*, *object_type*, *action_type*, and *message_type*.

```
⟨ Type definitions 5 ⟩ ≡
  typedef enum {
    no_type, motion_type, object_type, action_type, message_type
  } wordtype;
  typedef struct {
    char text[6];    /* string of length at most 5 */
    char word_type;    /* a wordtype */
    char meaning;
  } hash_entry;
```

See also sections 9, 11, 13, 18, and 19.

This code is used in section 2.

6. Here is the subroutine that puts words into our vocabulary, when the program is getting ready to run.

```
#define hash_prime 1009    /* the size of the hash table */
⟨ Subroutines 6 ⟩ ≡
  void new_word ARGS((char *, int));
  void new_word(w, m)
      char *w;    /* a string of length 5 or less */
      int m;    /* its meaning */
  {
    register int h, k;
    register char *p;
    for (h = 0, p = w; *p; p++) h = (*p + h + h) % hash_prime;
    while (hash_table[h].word_type) {
      h++; if (h ≡ hash_prime) h = 0;
    }
    for (k = 0, p = w; *p; p++, k++) hash_table[h].text[k] = *p;
    hash_table[h].word_type = current_type;
    hash_table[h].meaning = m;
  }
```

See also sections 8, 64, 65, 66, 71, 72, 154, 160, 194, and 197.

This code is used in section 2.

7.

```
⟨ Global variables 7 ⟩ ≡
  hash_entry hash_table[hash_prime];    /* the table of words we know */
  wordtype current_type;    /* the kind of word we are dealing with */
```

See also sections 15, 17, 20, 21, 63, 73, 74, 77, 81, 84, 87, 89, 96, 103, 137, 142, 155, 159, 165, 168, 171, 177, 185, 190, 193, 196, and 199.

This code is used in section 2.

8. While we're at it, let's write the program that will look up a word. It returns the location of the word in the hash table, or −1 if you've given a word like '`tickle`' or '`terse`' that is unknown.

```
#define streq(a,b) (strncmp(a,b,5) ≡ 0)    /* strings agree up to five letters */
⟨ Subroutines 6 ⟩ +≡
  int lookup ARGS((char *));

  int lookup(w)
      char *w;    /* a string that you typed */
  {
    register int h;
    register char *p;
    register char t;

    t = w[5];
    w[5] = '\0';    /* truncate the word */
    for (h = 0, p = w; *p; p++) h = (*p + h + h) % hash_prime;    /* compute starting address */
    w[5] = t;    /* restore original word */
    if (h < 0) return −1;    /* a negative character might screw us up */
    while (hash_table[h].word_type) {
      if (streq(w, hash_table[h].text)) return h;
      h++; if (h ≡ hash_prime) h = 0;
    }
    return −1;
  }
```

9. The *motion* words either specify a direction or a simple action or a place. Motion words take you from one location to another, when the motion is permitted. Here is a list of their possible meanings.

```
⟨ Type definitions 5 ⟩ +≡
  typedef enum {
    N, S, E, W, NE, SE, NW, SW, U, D, L, R, IN, OUT, FORWARD, BACK,
    OVER, ACROSS, UPSTREAM, DOWNSTREAM,
    ENTER, CRAWL, JUMP, CLIMB, LOOK, CROSS,
    ROAD, HILL, WOODS, VALLEY, HOUSE, GULLY, STREAM, DEPRESSION, ENTRANCE, CAVE,
    ROCK, SLAB, BED, PASSAGE, CAVERN, CANYON, AWKWARD, SECRET, BEDQUILT, RESERVOIR,
    GIANT, ORIENTAL, SHELL, BARREN, BROKEN, DEBRIS, VIEW, FORK,
    PIT, SLIT, CRACK, DOME, HOLE, WALL, HALL, ROOM, FLOOR,
    STAIRS, STEPS, COBBLES, SURFACE, DARK, LOW, OUTDOORS,
    Y2, XYZZY, PLUGH, PLOVER, OFFICE, NOWHERE
  } motion;
```

10. And here is how they enter our vocabulary.

If I were writing this program, I would allow the word `woods`, but Don apparently didn't want to.

⟨ Build the vocabulary 10 ⟩ ≡

```
current_type = motion_type;
new_word("north",N); new_word("n",N);
new_word("south",S); new_word("s",S);
new_word("east",E); new_word("e",E);
new_word("west",W); new_word("w",W);
new_word("ne",NE);
new_word("se",SE);
new_word("nw",NW);
new_word("sw",SW);
new_word("upwar",U); new_word("up",U); new_word("u",U); new_word("above",U);
new_word("ascen",U);
new_word("downw",D); new_word("down",D); new_word("d",D); new_word("desce",D);
new_word("left",L);
new_word("right",R);
new_word("inwar",IN); new_word("insid",IN); new_word("in",IN);
new_word("out",OUT); new_word("outsi",OUT);
new_word("exit",OUT);
new_word("leave",OUT);
new_word("forwa",FORWARD); new_word("conti",FORWARD); new_word("onwar",FORWARD);
new_word("back",BACK); new_word("retur",BACK); new_word("retre",BACK);
new_word("over",OVER);
new_word("acros",ACROSS);
new_word("upstr",UPSTREAM);
new_word("downs",DOWNSTREAM);
new_word("enter",ENTER);
new_word("crawl",CRAWL);
new_word("jump",JUMP);
new_word("climb",CLIMB);
new_word("look",LOOK); new_word("exami",LOOK); new_word("touch",LOOK);
new_word("descr",LOOK);
new_word("cross",CROSS);
new_word("road",ROAD);
new_word("hill",HILL);
new_word("forest",WOODS);
new_word("valle",VALLEY);
new_word("build",HOUSE); new_word("house",HOUSE);
new_word("gully",GULLY);
new_word("strea",STREAM);
new_word("depre",DEPRESSION);
new_word("entra",ENTRANCE);
new_word("cave",CAVE);
new_word("rock",ROCK);
new_word("slab",SLAB); new_word("slabr",SLAB);
new_word("bed",BED);
new_word("passa",PASSAGE); new_word("tunne",PASSAGE);
new_word("caver",CAVERN);
new_word("canyo",CANYON);
new_word("awkwa",AWKWARD);
new_word("secre",SECRET);
```

```
new_word("bedqu",BEDQUILT);
new_word("reser",RESERVOIR);
new_word("giant",GIANT);
new_word("orien",ORIENTAL);
new_word("shell",SHELL);
new_word("barre",BARREN);
new_word("broke",BROKEN);
new_word("debri",DEBRIS);
new_word("view",VIEW);
new_word("fork",FORK);
new_word("pit",PIT);
new_word("slit",SLIT);
new_word("crack",CRACK);
new_word("dome",DOME);
new_word("hole",HOLE);
new_word("wall",WALL);
new_word("hall",HALL);
new_word("room",ROOM);
new_word("floor",FLOOR);
new_word("stair",STAIRS);
new_word("steps",STEPS);
new_word("cobbl",COBBLES);
new_word("surfa",SURFACE);
new_word("dark",DARK);
new_word("low",LOW);
new_word("outdo",OUTDOORS);
new_word("y2",Y2);
new_word("xyzzy",XYZZY);
new_word("plugh",PLUGH);
new_word("plove",PLOVER);
new_word("main",OFFICE); new_word("offic",OFFICE);
new_word("null",NOWHERE); new_word("nowhe",NOWHERE);
```

See also sections 12, 14, and 16.

This code is used in section 200.

11. The *object* words refer to things like a lamp, a bird, batteries, etc.; objects have properties that will be described later. Here is a list of the basic objects. Objects `GOLD` and higher are the "treasures." Extremely large objects, which appear in more than one location, are listed more than once using '`_`'.

```
#define min_treasure GOLD
#define is_treasure(t) (t ≥ min_treasure)
#define max_obj CHAIN
```

⟨ Type definitions 5 ⟩ +≡

```
typedef enum {
  NOTHING, KEYS, LAMP, GRATE, GRATE_, CAGE, ROD, ROD2, TREADS, TREADS_,
  BIRD, DOOR, PILLOW, SNAKE, CRYSTAL, CRYSTAL_, TABLET, CLAM, OYSTER,
  MAG, DWARF, KNIFE, FOOD, BOTTLE, WATER, OIL,
  MIRROR, MIRROR_, PLANT, PLANT2, PLANT2_, STALACTITE, SHADOW, SHADOW_,
  AXE, ART, PIRATE, DRAGON, DRAGON_, BRIDGE, BRIDGE_, TROLL, TROLL_, TROLL2, TROLL2_,
  BEAR, MESSAGE, GEYSER, PONY, BATTERIES, MOSS,
  GOLD, DIAMONDS, SILVER, JEWELS, COINS, CHEST, EGGS, TRIDENT, VASE,
  EMERALD, PYRAMID, PEARL, RUG, RUG_, SPICES, CHAIN
} object;
```

12. Most of the objects correspond to words in our vocabulary.

⟨ Build the vocabulary 10 ⟩ +≡

```
current_type = object_type;
new_word("key",KEYS); new_word("keys",KEYS);
new_word("lamp",LAMP); new_word("lante",LAMP); new_word("headl",LAMP);
new_word("grate",GRATE);
new_word("cage",CAGE);
new_word("rod",ROD);
new_word("bird",BIRD);
new_word("door",DOOR);
new_word("pillo",PILLOW);
new_word("snake",SNAKE);
new_word("fissu",CRYSTAL);
new_word("table",TABLET);
new_word("clam",CLAM);
new_word("oyste",OYSTER);
new_word("magaz",MAG); new_word("issue",MAG); new_word("spelu",MAG);
new_word("\"spel",MAG);
new_word("dwarf",DWARF); new_word("dwarv",DWARF);
new_word("knife",KNIFE); new_word("knive",KNIFE);
new_word("food",FOOD); new_word("ratio",FOOD);
new_word("bottl",BOTTLE); new_word("jar",BOTTLE);
new_word("water",WATER); new_word("h2o",WATER);
new_word("oil",OIL);
new_word("mirro",MIRROR);
new_word("plant",PLANT); new_word("beans",PLANT);
new_word("stala",STALACTITE);
new_word("shado",SHADOW); new_word("figur",SHADOW);
new_word("axe",AXE);
new_word("drawi",ART);
new_word("pirat",PIRATE);
new_word("drago",DRAGON);
new_word("chasm",BRIDGE);
new_word("troll",TROLL);
new_word("bear",BEAR);
new_word("messa",MESSAGE);
new_word("volca",GEYSER); new_word("geyse",GEYSER);
new_word("vendi",PONY); new_word("machi",PONY);
new_word("batte",BATTERIES);
new_word("moss",MOSS); new_word("carpe",MOSS);
new_word("gold",GOLD); new_word("nugge",GOLD);
new_word("diamo",DIAMONDS);
new_word("silve",SILVER); new_word("bars",SILVER);
new_word("jewel",JEWELS);
new_word("coins",COINS);
new_word("chest",CHEST); new_word("box",CHEST); new_word("treas",CHEST);
new_word("eggs",EGGS); new_word("egg",EGGS); new_word("nest",EGGS);
new_word("tride",TRIDENT);
new_word("ming",VASE); new_word("vase",VASE); new_word("shard",VASE);
new_word("potte",VASE);
new_word("emera",EMERALD);
new_word("plati",PYRAMID); new_word("pyram",PYRAMID);
```

```
new_word("pearl",PEARL);
new_word("persi",RUG); new_word("rug",RUG);
new_word("spice",SPICES);
new_word("chain",CHAIN);
```

13. The *action* words tell us to do something that is usually nontrivial.

⟨ Type definitions 5 ⟩ +≡

```
typedef enum {
  ABSTAIN, TAKE, DROP, OPEN, CLOSE, ON, OFF, WAVE, CALM, GO, RELAX,
  POUR, EAT, DRINK, RUB, TOSS, WAKE, FEED, FILL, BREAK, BLAST, KILL,
  SAY, READ, FEEFIE, BRIEF, FIND, INVENTORY, SCORE, QUIT
} action;
```

14. Many of the action words have several synonyms. If an action does not meet special conditions, we will issue a default message.

#define *ok* *default_msg*[RELAX]

⟨ Build the vocabulary 10 ⟩ +≡

```
current_type = action_type;
new_word("take",TAKE); new_word("carry",TAKE); new_word("keep",TAKE);
new_word("catch",TAKE); new_word("captu",TAKE); new_word("steal",TAKE);
new_word("get",TAKE); new_word("tote",TAKE);
default_msg[TAKE] = "You␣are␣already␣carrying␣it!";
new_word("drop",DROP); new_word("relea",DROP); new_word("free",DROP);
new_word("disca",DROP); new_word("dump",DROP);
default_msg[DROP] = "You␣aren't␣carrying␣it!";
new_word("open",OPEN); new_word("unloc",OPEN);
default_msg[OPEN] = "I␣don't␣know␣how␣to␣lock␣or␣unlock␣such␣a␣thing.";
new_word("close",CLOSE); new_word("lock",CLOSE);
default_msg[CLOSE] = default_msg[OPEN];
new_word("light",ON); new_word("on",ON);
default_msg[ON] = "You␣have␣no␣source␣of␣light.";
new_word("extin",OFF); new_word("off",OFF);
default_msg[OFF] = default_msg[ON];
new_word("wave",WAVE); new_word("shake",WAVE); new_word("swing",WAVE);
default_msg[WAVE] = "Nothing␣happens.";
new_word("calm",CALM); new_word("placa",CALM); new_word("tame",CALM);
default_msg[CALM] = "I'm␣game.␣␣Would␣you␣care␣to␣explain␣how?";
new_word("walk",GO); new_word("run",GO); new_word("trave",GO); new_word("go",GO);
new_word("proce",GO); new_word("explo",GO); new_word("goto",GO); new_word("follo",GO);
new_word("turn",GO);
default_msg[GO] = "Where?";
new_word("nothi",RELAX);
default_msg[RELAX] = "OK.";
new_word("pour",POUR);
default_msg[POUR] = default_msg[DROP];
new_word("eat",EAT); new_word("devou",EAT);
default_msg[EAT] = "Don't␣be␣ridiculous!";
new_word("drink",DRINK);
default_msg[DRINK] =
    "You␣have␣taken␣a␣drink␣from␣the␣stream.␣␣The␣water␣tastes␣strongly␣of\n\
    minerals,␣but␣is␣not␣unpleasant.␣␣It␣is␣extremely␣cold.";
new_word("rub",RUB);
default_msg[RUB] = "Rubbing␣the␣electric␣lamp␣is␣not␣particularly␣rewarding.␣Anyway,\n\
    nothing␣exciting␣happens.";
new_word("throw",TOSS); new_word("toss",TOSS);
default_msg[TOSS] = "Peculiar.␣␣Nothing␣unexpected␣happens.";
new_word("wake",WAKE); new_word("distu",WAKE);
default_msg[WAKE] = default_msg[EAT];
new_word("feed",FEED);
default_msg[FEED] = "There␣is␣nothing␣here␣to␣eat.";
new_word("fill",FILL);
default_msg[FILL] = "You␣can't␣fill␣that.";
new_word("break",BREAK); new_word("smash",BREAK); new_word("shatt",BREAK);
default_msg[BREAK] = "It␣is␣beyond␣your␣power␣to␣do␣that.";
```

```
new_word("blast",BLAST); new_word("deton",BLAST); new_word("ignit",BLAST);
new_word("blowu",BLAST);
default_msg[BLAST] = "Blasting␣requires␣dynamite.";
new_word("attac",KILL); new_word("kill",KILL); new_word("fight",KILL);
new_word("hit",KILL); new_word("strik",KILL); new_word("slay",KILL);
default_msg[KILL] = default_msg[EAT];
new_word("say",SAY); new_word("chant",SAY); new_word("sing",SAY); new_word("utter",SAY);
new_word("mumbl",SAY);
new_word("read",READ); new_word("perus",READ);
default_msg[READ] = "I'm␣afraid␣I␣don't␣understand.";
new_word("fee",FEEFIE); new_word("fie",FEEFIE); new_word("foe",FEEFIE);
new_word("foo",FEEFIE); new_word("fum",FEEFIE);
default_msg[FEEFIE] = "I␣don't␣know␣how.";
new_word("brief",BRIEF);
default_msg[BRIEF] = "On␣what?";
new_word("find",FIND); new_word("where",FIND);
default_msg[FIND] = "I␣can␣only␣tell␣you␣what␣you␣see␣as␣you␣move␣about␣and␣manipulate\n\
    things.␣␣I␣cannot␣tell␣you␣where␣remote␣things␣are.";
new_word("inven",INVENTORY);
default_msg[INVENTORY] = default_msg[FIND];
new_word("score",SCORE);
default_msg[SCORE] = "Eh?";
new_word("quit",QUIT);
default_msg[QUIT] = default_msg[SCORE];
```

15. ⟨ Global variables 7 ⟩ +≡

```
char *default_msg[30];    /* messages for untoward actions, if nonzero */
```

16. Finally, our vocabulary is rounded out by words like `help`, which trigger the printing of fixed messages.

#define *new_mess*(*x*) *message*[*k*++] = *x*
#define *mess_wd*(*w*) *new_word*(*w*, *k*)

⟨ Build the vocabulary 10 ⟩ +≡

```
current_type = message_type;
k = 0;
mess_wd("abra");
mess_wd("abrac");
mess_wd("opens");
mess_wd("sesam");
mess_wd("shaza");
mess_wd("hocus");
mess_wd("pocus");
new_mess("Good␣try,␣but␣that␣is␣an␣old␣worn-out␣magic␣word.");
mess_wd("help");
mess_wd("?");
new_mess("I␣know␣of␣places,␣actions,␣and␣things.␣Most␣of␣my␣vocabulary\n\
    describes␣places␣and␣is␣used␣to␣move␣you␣there.␣␣To␣move,␣try␣words\n\
    like␣forest,␣building,␣downstream,␣enter,␣east,␣west,␣north,␣south,\n\
    up,␣or␣down.␣␣I␣know␣about␣a␣few␣special␣objects,␣like␣a␣black␣rod\n\
    hidden␣in␣the␣cave.␣␣These␣objects␣can␣be␣manipulated␣using␣some␣of\n\
    the␣action␣words␣that␣I␣know.␣␣Usually␣you␣will␣need␣to␣give␣both␣the\n\
    object␣and␣action␣words␣(in␣either␣order),␣but␣sometimes␣I␣can␣infer\n\
    the␣object␣from␣the␣verb␣alone.␣␣Some␣objects␣also␣imply␣verbs;␣in\n\
    particular,␣\"inventory\"␣implies␣\"take␣inventory\",␣which␣causes␣me␣to\n\
    give␣you␣a␣list␣of␣what␣you're␣carrying.␣␣The␣objects␣have␣side\n\
    effects;␣for␣instance,␣the␣rod␣scares␣the␣bird.␣␣Usually␣people␣having\n\
    trouble␣moving␣just␣need␣to␣try␣a␣few␣more␣words.␣Usually␣people\n\
    trying␣unsuccessfully␣to␣manipulate␣an␣object␣are␣attempting␣something\n\
    beyond␣their␣(or␣my!)␣capabilities␣and␣should␣try␣a␣completely\n\
    different␣tack.␣␣To␣speed␣the␣game␣you␣can␣sometimes␣move␣long\n\
    distances␣with␣a␣single␣word.␣␣For␣example,␣\"building\"␣usually␣gets\n\
    you␣to␣the␣building␣from␣anywhere␣above␣ground␣except␣when␣lost␣in␣the\n\
    forest.␣␣Also,␣note␣that␣cave␣passages␣turn␣a␣lot,␣and␣that␣leaving␣a\n\
    room␣to␣the␣north␣does␣not␣guarantee␣entering␣the␣next␣from␣the␣south.\nGood␣luck!");
mess_wd("tree");
mess_wd("trees");
new_mess("The␣trees␣of␣the␣forest␣are␣large␣hardwood␣oak␣and␣maple,␣with␣an\n\
    occasional␣grove␣of␣pine␣or␣spruce.␣␣There␣is␣quite␣a␣bit␣of␣under-\n\
    growth,␣largely␣birch␣and␣ash␣saplings␣plus␣nondescript␣bushes␣of\n\
    various␣sorts.␣␣This␣time␣of␣year␣visibility␣is␣quite␣restricted␣by\n\
    all␣the␣leaves,␣but␣travel␣is␣quite␣easy␣if␣you␣detour␣around␣the\n\
    spruce␣and␣berry␣bushes.");
mess_wd("dig");
mess_wd("excav");
new_mess("Digging␣without␣a␣shovel␣is␣quite␣impractical.␣␣Even␣with␣a␣shovel\n\
    progress␣is␣unlikely.");
mess_wd("lost");
new_mess("I'm␣as␣confused␣as␣you␣are.");
new_mess("There␣is␣a␣loud␣explosion␣and␣you␣are␣suddenly␣splashed␣across␣the\n\
    walls␣of␣the␣room.");
```

```
new_mess("There␣is␣a␣loud␣explosion␣and␣a␣twenty-foot␣hole␣appears␣in␣the␣far\n\
    wall,␣burying␣the␣snakes␣in␣the␣rubble.␣␣A␣river␣of␣molten␣lava␣pours\n\
    in␣through␣the␣hole,␣destroying␣everything␣in␣its␣path,␣including␣you!");
mess_wd("mist");
new_mess("Mist␣is␣a␣white␣vapor,␣usually␣water,␣seen␣from␣time␣to␣time␣in\n\
    caverns.␣␣It␣can␣be␣found␣anywhere␣but␣is␣frequently␣a␣sign␣of␣a␣deep\n\
    pit␣leading␣down␣to␣water.");
mess_wd("fuck");
new_mess("Watch␣it!");
new_mess("There␣is␣a␣loud␣explosion,␣and␣a␣twenty-foot␣hole␣appears␣in␣the␣far\n\
    wall,␣burying␣the␣dwarves␣in␣the␣rubble.␣␣You␣march␣through␣the␣hole\n\
    and␣find␣yourself␣in␣the␣main␣office,␣where␣a␣cheering␣band␣of\n\
    friendly␣elves␣carry␣the␣conquering␣adventurer␣off␣into␣the␣sunset.");
mess_wd("stop");
new_mess("I␣don't␣know␣the␣word␣\"stop\".␣␣Use␣\"quit\"␣if␣you␣want␣to␣give␣up.");
mess_wd("info");
mess_wd("infor");
new_mess("If␣you␣want␣to␣end␣your␣adventure␣early,␣say␣\"quit\".␣␣To␣get␣full\n\
    credit␣for␣a␣treasure,␣you␣must␣have␣left␣it␣safely␣in␣the␣building,\n\
    though␣you␣get␣partial␣credit␣just␣for␣locating␣it.␣␣You␣lose␣points\n\
    for␣getting␣killed,␣or␣for␣quitting,␣though␣the␣former␣costs␣you␣more.\n\
    There␣are␣also␣points␣based␣on␣how␣much␣(if␣any)␣of␣the␣cave␣you've\n\
    managed␣to␣explore;␣in␣particular,␣there␣is␣a␣large␣bonus␣just␣for\n\
    getting␣in␣(to␣distinguish␣the␣beginners␣from␣the␣rest␣of␣the␣pack),\n\
    and␣there␣are␣other␣ways␣to␣determine␣whether␣you've␣been␣through␣some\n\
    of␣the␣more␣harrowing␣sections.␣␣If␣you␣think␣you've␣found␣all␣the\n\
    treasures,␣just␣keep␣exploring␣for␣a␣while.␣␣If␣nothing␣interesting\n\
    happens,␣you␣haven't␣found␣them␣all␣yet.␣␣If␣something␣interesting\n\
    DOES␣happen,␣it␣means␣you're␣getting␣a␣bonus␣and␣have␣an␣opportunity\n\
    to␣garner␣many␣more␣points␣in␣the␣master's␣section.\n\
    I␣may␣occasionally␣offer␣hints␣if␣you␣seem␣to␣be␣having␣trouble.\n\
    If␣I␣do,␣I'll␣warn␣you␣in␣advance␣how␣much␣it␣will␣affect␣your␣score\n\
    to␣accept␣the␣hints.␣␣Finally,␣to␣save␣paper,␣you␣may␣specify␣\"brief\",\n\
    which␣tells␣me␣never␣to␣repeat␣the␣full␣description␣of␣a␣place\n\
    unless␣you␣explicitly␣ask␣me␣to.");
mess_wd("swim");
new_mess("I␣don't␣know␣how.");
```

17. ⟨ Global variables 7 ⟩ +≡

```
char *message[13];    /* messages tied to certain vocabulary words */
```

18. Cave data. You might be in any of more than 100 places as you wander about in Colossal Cave. Let's enumerate them now, so that we can build the data structures that define the travel restrictions.

A special negative value called *inhand* is the location code for objects that you are carrying. But you yourself are always situated in a place that has a nonnegative location code.

Nonnegative places ≤ *outside* are outside the cave, while places ≥ *inside* are inside. The upper part of the cave, places < *emist*, is the easiest part to explore. (We will see later that dwarves do not venture this close to the surface; they stay ≥ *emist*.)

Places between *inside* and *dead2*, inclusive, form the main cave; the next places, up to and including *barr*, form the hidden cave on the other side of the troll bridge; then *neend* and *swend* are a private cave.

The remaining places, ≥ *crack* are dummy locations, not really part of the maze. As soon as you arrive at a dummy location, the program immediately sends you somewhere else. In fact, the last three dummy locations aren't really even locations; they invoke special code. This device is a convenient way to provide a variety of features without making the program logic any more cluttered than it already is.

```
#define min_in_cave inside
#define min_lower_loc emist
#define min_forced_loc crack
#define max_loc didit
#define max_spec troll
⟨ Type definitions 5 ⟩ +≡
  typedef enum {
    inhand = −1, limbo,
    road, hill, house, valley, forest, woods, slit, outside,
    inside, cobbles, debris, awk, bird, spit,
    emist, nugget, efiss, wfiss, wmist,
    like1, like2, like3, like4, like5, like6, like7, like8, like9, like10, like11, like12, like13, like14,
    brink, elong, wlong,
    diff0, diff1, diff2, diff3, diff4, diff5, diff6, diff7, diff8, diff9, diff10,
    pony, cross, hmk, west, south, ns, y2, jumble, windoe,
    dirty, clean, wet, dusty, complex,
    shell, arch, ragged, sac, ante, witt,
    bedquilt, cheese, soft,
    e2pit, w2pit, epit, wpit,
    narrow, giant, block, immense,
    falls, steep, abovep, sjunc, tite, low, crawl, window,
    oriental, misty, alcove, proom, droom,
    slab, abover, mirror, res,
    scan1, scan2, scan3, secret,
    wide, tight, tall, boulders,
    scorr, swside,
    dead0, dead1, dead2, dead3, dead4, dead5, dead6, dead7, dead8, dead9, dead10, dead11,
    neside, corr, fork, warm, view, chamber, lime, fbarr, barr,
    neend, swend,
    crack, neck, lose, cant, climb, check, snaked, thru, duck, sewer, upnout, didit,
    ppass, pdrop, troll
  } location;
```

19. Speaking of program logic, the complex cave dynamics are essentially kept in a table. The table tells us what to do when you ask for a particular motion in a particular location. Each entry of the table is called an instruction; and each instruction has three parts: a motion, a condition, and a destination.

The motion part of an instruction is one of the motion verbs enumerated earlier. The condition part c is a small integer, interpreted as follows:

- if $c = 0$, the condition is always true;
- if $0 < c < 100$, the condition is true with probability $c/100$;
- if $c = 100$, the condition is always true, except for dwarves;
- if $100 < c <= 200$, you must have object $c \bmod 100$;
- if $200 < c <= 300$, object $c \bmod 100$ must be in the current place;
- if $300 < c <= 400$, $prop[c \bmod 100]$ must not be 0;
- if $400 < c <= 500$, $prop[c \bmod 100]$ must not be 1;
- if $500 < c <= 600$, $prop[c \bmod 100]$ must not be 2; etc.

(We will discuss properties of objects and the *prop* array later.) The destination d is either a location or a number greater than *max_loc*; in the latter case, if $d \le max\_spec$ we perform a special routine, otherwise we print $remarks[d - max\_spec]$ and stay in the current place.

If the motion matches what you said but the condition is not satisfied, we move on to the next instruction that has a different destination and/or condition from this one. The next instruction might itself be conditional in the same way. (Numerous examples appear below.)

⟨ Type definitions 5 ⟩ +≡
```
typedef struct {
    motion mot;     /* a motion you might have requested */
    int cond;     /* if you did, this condition must also hold */
    location dest;     /* and if so, this is where you'll go next */
} instruction;
```

20. Suppose you're at location l. Then $start[l]$ is the first relevant instruction, and $start[l+1]-1$ is the last. Also $long\_desc[l]$ is a string that fully describes l; $short\_desc[l]$ is an optional abbreviated description; and $visits[l]$ tells how many times you have been here. Special properties of this location, such as whether a lantern is necessary or a hint might be advisable, are encoded in the bits of $flags[l]$.

```
#define lighted 1     /* bit for a location that isn't dark */
#define oil 2     /* bit for presence of oil */
#define liquid 4     /* bit for presence of a liquid (oil or water) */
#define cave_hint 8     /* bit for hint about trying to get in the cave */
#define bird_hint 16     /* bit for hint about catching the bird */
#define snake_hint 32     /* bit for hint about dealing with the snake */
#define twist_hint 64     /* bit for hint about being lost in a maze */
#define dark_hint 128     /* bit for hint about the dark room */
#define witt_hint 256     /* bit for hint about Witt's End */
#define travel_size 740     /* at most this many instructions */
#define rem_size 15     /* at most this many remarks */
```

⟨ Global variables 7 ⟩ +≡
```
instruction travels[travel_size];     /* the table of instructions */
instruction *start[max_loc + 2];     /* references to starting instruction */
char *long_desc[max_loc + 1];     /* long-winded descriptions of locations */
char *short_desc[max_loc + 1];     /* short-winded descriptions, or 0 */
int flags[max_loc + 1];     /* bitmaps for special properties */
char *remarks[rem_size];     /* comments made when staying put */
int rem_count;     /* we've made this many comments */
int visits[max_loc + 1];     /* how often have you been here? */
```

21. Cave connections. Now we are ready to build the fundamental table of location and transition data, by filling in the arrays just declared. We will fill them in strict order of their **location** codes.

It is convenient to define several macros and constants.

```
#define make_loc(x, l, s, f)
        { long_desc[x] = l; short_desc[x] = s; flags[x] = f; start[x] = q; }
#define make_inst(m, c, d)
        { q->mot = m; q->cond = c; q->dest = d; q++; }
#define ditto(m)
        { q->mot = m; q->cond = (q - 1)->cond; q->dest = (q - 1)->dest; q++; }
#define holds(o) 100 + o    /* do instruction only if carrying object o */
#define sees(o) 200 + o    /* do instruction only if object o is present */
#define not(o, k) 300 + o + 100 * k    /* do instruction only if prop[o] ≠ k */
#define remark(m) remarks[++rem_count] = m
#define sayit max_spec + rem_count
```

⟨ Global variables 7 ⟩ +≡

```
  char all_alike[] = "You␣are␣in␣a␣maze␣of␣twisty␣little␣passages,␣all␣alike.";
  char dead_end[] = "Dead␣end.";
  int slit_rmk, grate_rmk, bridge_rmk, loop_rmk;    /* messages used more than once */
```

22. ⟨ Additional local registers 22 ⟩ ≡

```
  register instruction *q, *qq;
```

See also sections 68 and 144.

This code is used in section 2.

23. The *road* is where you start; its *long_desc* is now famous, having been quoted by Steven Levy in his book *Hackers*.

The instructions here say that if you want to go west, or up, or on the road, we take you to *hill*; if you want to go east, or in, or to the house, or if you say '`enter`', we take you to *house*; etc. Of course you won't know about all the motions available at this point until you have played the game for awhile.

⟨ Build the travel table 23 ⟩ ≡

```
  q = travels;
  make_loc(road,
  "You␣are␣standing␣at␣the␣end␣of␣a␣road␣before␣a␣small␣brick␣building.\n\
      Around␣you␣is␣a␣forest.␣␣A␣small␣stream␣flows␣out␣of␣the␣building␣and\n\
      down␣a␣gully.",
  "You're␣at␣end␣of␣road␣again.", lighted + liquid);
  make_inst(W, 0, hill); ditto(U); ditto(ROAD);
  make_inst(E, 0, house); ditto(IN); ditto(HOUSE); ditto(ENTER);
  make_inst(S, 0, valley); ditto(D); ditto(GULLY); ditto(STREAM); ditto(DOWNSTREAM);
  make_inst(N, 0, forest); ditto(WOODS);
  make_inst(DEPRESSION, 0, outside);
```

See also sections 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, and 62.

This code is used in section 200.

24. There's nothing up the hill, but a good explorer has to try anyway.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(hill,
"You␣have␣walked␣up␣a␣hill,␣still␣in␣the␣forest.␣␣The␣road␣slopes␣back\n\
    down␣the␣other␣side␣of␣the␣hill.␣␣There␣is␣a␣building␣in␣the␣distance.",
"You're␣at␣hill␣in␣road.", lighted);
make_inst(ROAD, 0, road); ditto(HOUSE); ditto(FORWARD); ditto(E); ditto(D);
make_inst(WOODS, 0, forest); ditto(N); ditto(S);
```

25. The house initially contains several objects: keys, food, a bottle, and a lantern. We'll put them in there later.

Two magic words are understood in this house, for spelunkers who have been there and done that.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(house,
"You␣are␣inside␣a␣building,␣a␣well␣house␣for␣a␣large␣spring.",
"You're␣inside␣building.", lighted + liquid);
make_inst(ENTER, 0, road); ditto(OUT); ditto(OUTDOORS); ditto(W);
make_inst(XYZZY, 0, debris);
make_inst(PLUGH, 0, y2);
make_inst(DOWNSTREAM, 0, sewer); ditto(STREAM);
```

26. A foolish consistency is the hobgoblin of little minds. (Emerson)

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(valley,
"You␣are␣in␣a␣valley␣in␣the␣forest␣beside␣a␣stream␣tumbling␣along␣a\nrocky␣bed.",
"You're␣in␣valley.", lighted + liquid);
make_inst(UPSTREAM, 0, road); ditto(HOUSE); ditto(N);
make_inst(WOODS, 0, forest); ditto(E); ditto(W); ditto(U);
make_inst(DOWNSTREAM, 0, slit); ditto(S); ditto(D);
make_inst(DEPRESSION, 0, outside);
```

27. The instructions here keep you in the *forest* with probability 50%, otherwise they take you to the *woods*. This gives the illusion that we maintain more state information about you than we really do.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(forest,
"You␣are␣in␣open␣forest,␣with␣a␣deep␣valley␣to␣one␣side.",
"You're␣in␣forest.", lighted);
make_inst(VALLEY, 0, valley); ditto(E); ditto(D);
make_inst(WOODS, 50, forest); ditto(FORWARD); ditto(N);
make_inst(WOODS, 0, woods);
make_inst(W, 0, forest); ditto(S);
make_loc(woods,
"You␣are␣in␣open␣forest␣near␣both␣a␣valley␣and␣a␣road.",
short_desc[forest], lighted);
make_inst(ROAD, 0, road); ditto(N);
make_inst(VALLEY, 0, valley); ditto(E); ditto(W); ditto(D);
make_inst(WOODS, 0, forest); ditto(S);
```

28. You're getting closer. (But the program has forgotten that DEPRESSION leads *outside*; it knew this when you were at the *road* or the *valley*.)

⟨ Build the travel table 23 ⟩ +≡
```
make_loc(slit,
"At␣your␣feet␣all␣the␣water␣of␣the␣stream␣splashes␣into␣a␣2-inch␣slit\n\
    in␣the␣rock.␣␣Downstream␣the␣streambed␣is␣bare␣rock.",
"You're␣at␣slit␣in␣streambed.", lighted + liquid);
make_inst(HOUSE, 0, road);
make_inst(UPSTREAM, 0, valley); ditto(N);
make_inst(WOODS, 0, forest); ditto(E); ditto(W);
make_inst(DOWNSTREAM, 0, outside); ditto(ROCK); ditto(BED); ditto(S);
remark("You␣don't␣fit␣through␣a␣two-inch␣slit!");
make_inst(SLIT, 0, sayit); ditto(STREAM); ditto(D);
slit_rmk = sayit;
```

29. We'll see later that the GRATE will change from state 0 to state 1 if you unlock it. So let's hope you have the KEYS.

⟨ Build the travel table 23 ⟩ +≡
```
make_loc(outside,
"You␣are␣in␣a␣20-foot␣depression␣floored␣with␣bare␣dirt.␣␣Set␣into␣the\n\
    dirt␣is␣a␣strong␣steel␣grate␣mounted␣in␣concrete.␣␣A␣dry␣streambed\n\
    leads␣into␣the␣depression.",
"You're␣outside␣grate.", lighted + cave_hint);
make_inst(WOODS, 0, forest); ditto(E); ditto(W); ditto(S);
make_inst(HOUSE, 0, road);
make_inst(UPSTREAM, 0, slit); ditto(GULLY); ditto(N);
make_inst(ENTER, not(GRATE, 0), inside); ditto(ENTER); ditto(IN); ditto(D);
remark("You␣can't␣go␣through␣a␣locked␣steel␣grate!");
grate_rmk = sayit;
make_inst(ENTER, 0, sayit);
```

30. If you've come this far, you're probably hooked, although your adventure has barely begun.

⟨ Build the travel table 23 ⟩ +≡
```
make_loc(inside,
"You␣are␣in␣a␣small␣chamber␣beneath␣a␣3x3␣steel␣grate␣to␣the␣surface.\n\
    A␣low␣crawl␣over␣cobbles␣leads␣inwards␣to␣the␣west.",
"You're␣below␣the␣grate.", lighted);
make_inst(OUT, not(GRATE, 0), outside); ditto(OUT); ditto(U);
make_inst(OUT, 0, grate_rmk);
make_inst(CRAWL, 0, cobbles); ditto(COBBLES); ditto(IN); ditto(W);
make_inst(PIT, 0, spit);
make_inst(DEBRIS, 0, debris);
```

31. Go West, young man. (If you've got a lamp.)

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(cobbles,
"You␣are␣crawling␣over␣cobbles␣in␣a␣low␣passage.␣␣There␣is␣a␣dim␣light\n\
    at␣the␣east␣end␣of␣the␣passage.",
"You're␣in␣cobble␣crawl.", lighted);
make_inst(OUT, 0, inside); ditto(SURFACE); ditto(NOWHERE); ditto(E);
make_inst(IN, 0, debris); ditto(DARK); ditto(W); ditto(DEBRIS);
make_inst(PIT, 0, spit);
make_loc(debris,
"You␣are␣in␣a␣debris␣room␣filled␣with␣stuff␣washed␣in␣from␣the␣surface.\n\
    A␣low␣wide␣passage␣with␣cobbles␣becomes␣plugged␣with␣mud␣and␣debris\n\
    here,␣but␣an␣awkward␣canyon␣leads␣upward␣and␣west.␣␣A␣note␣on␣the␣wall\n\
    says␣\"MAGIC␣WORD␣XYZZY\".",
"You're␣in␣debris␣room.", 0);
make_inst(DEPRESSION, not(GRATE, 0), outside);
make_inst(ENTRANCE, 0, inside);
make_inst(CRAWL, 0, cobbles); ditto(COBBLES); ditto(PASSAGE); ditto(LOW); ditto(E);
make_inst(CANYON, 0, awk); ditto(IN); ditto(U); ditto(W);
make_inst(XYZZY, 0, house);
make_inst(PIT, 0, spit);
make_loc(awk,
"You␣are␣in␣an␣awkward␣sloping␣east/west␣canyon.", 0, 0);
make_inst(DEPRESSION, not(GRATE, 0), outside);
make_inst(ENTRANCE, 0, inside);
make_inst(D, 0, debris); ditto(E); ditto(DEBRIS);
make_inst(IN, 0, bird); ditto(U); ditto(W);
make_inst(PIT, 0, spit);
make_loc(bird,
"You␣are␣in␣a␣splendid␣chamber␣thirty␣feet␣high.␣␣The␣walls␣are␣frozen\n\
    rivers␣of␣orange␣stone.␣␣An␣awkward␣canyon␣and␣a␣good␣passage␣exit\n\
    from␣east␣and␣west␣sides␣of␣the␣chamber.",
"You're␣in␣bird␣chamber.", bird_hint);
make_inst(DEPRESSION, not(GRATE, 0), outside);
make_inst(ENTRANCE, 0, inside);
make_inst(DEBRIS, 0, debris);
make_inst(CANYON, 0, awk); ditto(E);
make_inst(PASSAGE, 0, spit); ditto(PIT); ditto(W);
make_loc(spit,
"At␣your␣feet␣is␣a␣small␣pit␣breathing␣traces␣of␣white␣mist.␣␣An␣east\n\
    passage␣ends␣here␣except␣for␣a␣small␣crack␣leading␣on.",
"You're␣at␣top␣of␣small␣pit.", 0);
make_inst(DEPRESSION, not(GRATE, 0), outside);
make_inst(ENTRANCE, 0, inside);
make_inst(DEBRIS, 0, debris);
make_inst(PASSAGE, 0, bird); ditto(E);
make_inst(D, holds(GOLD), neck); ditto(PIT); ditto(STEPS);
make_inst(D, 0, emist);     /* good thing you weren't loaded down with GOLD */
make_inst(CRACK, 0, crack); ditto(W);
```

32. Welcome to the main caverns and a deeper level of adventures.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(emist,
"You␣are␣at␣one␣end␣of␣a␣vast␣hall␣stretching␣forward␣out␣of␣sight␣to\n\
    the␣west.␣␣There␣are␣openings␣to␣either␣side.␣␣Nearby,␣a␣wide␣stone\n\
    staircase␣leads␣downward.␣␣The␣hall␣is␣filled␣with␣wisps␣of␣white␣mist\n\
    swaying␣to␣and␣fro␣almost␣as␣if␣alive.␣␣A␣cold␣wind␣blows␣up␣the\n\
    staircase.␣␣There␣is␣a␣passage␣at␣the␣top␣of␣a␣dome␣behind␣you.",
"You're␣in␣Hall␣of␣Mists.", 0);
make_inst(L, 0, nugget); ditto(S);
make_inst(FORWARD, 0, efiss); ditto(HALL); ditto(W);
make_inst(STAIRS, 0, hmk); ditto(D); ditto(N);
make_inst(U, holds(GOLD), cant); ditto(PIT); ditto(STEPS);
ditto(DOME); ditto(PASSAGE); ditto(E);
make_inst(U, 0, spit);
make_inst(Y2, 0, jumble);
```

33. To the left or south of the misty threshold, you might spot the first treasure.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(nugget,
"This␣is␣a␣low␣room␣with␣a␣crude␣note␣on␣the␣wall.␣␣The␣note␣says,\n\
    \"You␣won't␣get␣it␣up␣the␣steps\".",
"You're␣in␣nugget␣of␣gold␣room.", 0);
make_inst(HALL, 0, emist); ditto(OUT); ditto(N);
```

34. Unless you take a circuitous route to the other side of the Hall of Mists, via the Hall of the Mountain King, you should make the CRYSTAL bridge appear (by getting it into state 1).

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(efiss,
"You␣are␣on␣the␣east␣bank␣of␣a␣fissure␣slicing␣clear␣across␣the␣hall.\n\
    The␣mist␣is␣quite␣thick␣here,␣and␣the␣fissure␣is␣too␣wide␣to␣jump.",
"You're␣on␣east␣bank␣of␣fissure.", 0);
make_inst(HALL, 0, emist); ditto(E);
remark("I␣respectfully␣suggest␣you␣go␣across␣the␣bridge␣instead␣of␣jumping.");
bridge_rmk = sayit;
make_inst(JUMP, not(CRYSTAL, 0), sayit);
make_inst(FORWARD, not(CRYSTAL, 1), lose);
remark("There␣is␣no␣way␣across␣the␣fissure.");
make_inst(OVER, not(CRYSTAL, 1), sayit); ditto(ACROSS); ditto(W); ditto(CROSS);
make_inst(OVER, 0, wfiss);

make_loc(wfiss,
"You␣are␣on␣the␣west␣side␣of␣the␣fissure␣in␣the␣Hall␣of␣Mists.", 0, 0);
make_inst(JUMP, not(CRYSTAL, 0), bridge_rmk);
make_inst(FORWARD, not(CRYSTAL, 1), lose);
make_inst(OVER, not(CRYSTAL, 1), sayit); ditto(ACROSS); ditto(E); ditto(CROSS);
make_inst(OVER, 0, efiss);
make_inst(N, 0, thru);
make_inst(W, 0, wmist);
```

35. What you see here isn't exactly what you get; N takes you east and S sucks you in to an amazing maze.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(wmist,
"You␣are␣at␣the␣west␣end␣of␣the␣Hall␣of␣Mists.␣␣A␣low␣wide␣crawl\n\
    continues␣west␣and␣another␣goes␣north.␣␣To␣the␣south␣is␣a␣little\n\
    passage␣6␣feet␣off␣the␣floor.",
"You're␣at␣west␣end␣of␣Hall␣of␣Mists.", 0);
make_inst(S, 0, like1); ditto(U); ditto(PASSAGE); ditto(CLIMB);
make_inst(E, 0, wfiss);
make_inst(N, 0, duck);
make_inst(W, 0, elong); ditto(CRAWL);
```

36. The twisty little passages of this maze are said to be all alike, but they respond differently to different motions. For example, you can go north, east, south, or west from *like1*, but you can't go north from *like2*. In that way you can psych out the whole maze of 14 similar locations. (And eventually you will want to know every place where treasure might be hidden.) The only exits are to *wmist* and *brink*.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(like1, all_alike, 0, twist_hint);
make_inst(U, 0, wmist);
make_inst(N, 0, like1);
make_inst(E, 0, like2);
make_inst(S, 0, like4);
make_inst(W, 0, like11);
make_loc(like2, all_alike, 0, twist_hint);
make_inst(W, 0, like1);
make_inst(S, 0, like3);
make_inst(E, 0, like4);
make_loc(like3, all_alike, 0, twist_hint);
make_inst(E, 0, like2);
make_inst(D, 0, dead5);
make_inst(S, 0, like6);
make_inst(N, 0, dead9);
make_loc(like4, all_alike, 0, twist_hint);
make_inst(W, 0, like1);
make_inst(N, 0, like2);
make_inst(E, 0, dead3);
make_inst(S, 0, dead4);
make_inst(U, 0, like14); ditto(D);
make_loc(like5, all_alike, 0, twist_hint);
make_inst(E, 0, like6);
make_inst(W, 0, like7);
make_loc(like6, all_alike, 0, twist_hint);
make_inst(E, 0, like3);
make_inst(W, 0, like5);
make_inst(D, 0, like7);
make_inst(S, 0, like8);
make_loc(like7, all_alike, 0, twist_hint);
make_inst(W, 0, like5);
make_inst(U, 0, like6);
make_inst(E, 0, like8);
make_inst(S, 0, like9);
make_loc(like8, all_alike, 0, twist_hint);
make_inst(W, 0, like6);
make_inst(E, 0, like7);
make_inst(S, 0, like8);
make_inst(U, 0, like9);
make_inst(N, 0, like10);
make_inst(D, 0, dead11);
make_loc(like9, all_alike, 0, twist_hint);
make_inst(W, 0, like7);
make_inst(N, 0, like8);
make_inst(S, 0, dead6);
```

```
make_loc(like10, all_alike, 0, twist_hint);
make_inst(W, 0, like8);
make_inst(N, 0, like10);
make_inst(D, 0, dead7);
make_inst(E, 0, brink);
make_loc(like11, all_alike, 0, twist_hint);
make_inst(N, 0, like1);
make_inst(W, 0, like11); ditto(S);
make_inst(E, 0, dead1);
make_loc(like12, all_alike, 0, twist_hint);
make_inst(S, 0, brink);
make_inst(E, 0, like13);
make_inst(W, 0, dead10);
make_loc(like13, all_alike, 0, twist_hint);
make_inst(N, 0, brink);
make_inst(W, 0, like12);
make_inst(NW, 0, dead2);      /* NW: a dirty trick! */
make_loc(like14, all_alike, 0, twist_hint);
make_inst(U, 0, like4); ditto(D);
```

37. ⟨ Build the travel table 23 ⟩ +≡

```
make_loc(brink,
"You␣are␣on␣the␣brink␣of␣a␣thirty-foot␣pit␣with␣a␣massive␣orange␣column\n\
    down␣one␣wall.␣␣You␣could␣climb␣down␣here␣but␣you␣could␣not␣get␣back\n\
    up.␣␣The␣maze␣continues␣at␣this␣level.",
"You're␣at␣brink␣of␣pit.", 0);
make_inst(D, 0, bird); ditto(CLIMB);
make_inst(W, 0, like10);
make_inst(S, 0, dead8);
make_inst(N, 0, like12);
make_inst(E, 0, like13);
```

38. Crawling west from *wmist* instead of south, you encounter this.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(elong,
"You␣are␣at␣the␣east␣end␣of␣a␣very␣long␣hall␣apparently␣without␣side\n\
    chambers.␣␣To␣the␣east␣a␣low␣wide␣crawl␣slants␣up.␣␣To␣the␣north␣a\n\
    round␣two-foot␣hole␣slants␣down.",
"You're␣at␣east␣end␣of␣long␣hall.", 0);
make_inst(E, 0, wmist); ditto(U); ditto(CRAWL);
make_inst(W, 0, wlong);
make_inst(N, 0, cross); ditto(D); ditto(HOLE);
make_loc(wlong,
"You␣are␣at␣the␣west␣end␣of␣a␣very␣long␣featureless␣hall.␣␣The␣hall\n\
    joins␣up␣with␣a␣narrow␣north/south␣passage.",
"You're␣at␣west␣end␣of␣long␣hall.", 0);
make_inst(E, 0, elong);
make_inst(N, 0, cross);
make_inst(S, 100, diff0);
```

39. Recall that the '100' on the last instruction above means, "Dwarves not permitted." It keeps them out of the following maze, which is based on an 11×11 latin square. (Each of the eleven locations leads to each of the others under the ten motions N, S, E, W, NE, SE, NW, SW, U, D — except that *diff0* goes down to the entrance location *wlong* instead of to *diff10*, and *diff10* goes south to the dead-end location *pony* instead of to *diff0*. Furthermore, each location is accessible from all ten possible directions.)

Incidentally, if you ever get into a "little twisting maze of passages," you're really lost.

⟨Build the travel table 23⟩ +≡

```
make_loc(diff0,
"You␣are␣in␣a␣maze␣of␣twisty␣little␣passages,␣all␣different.", 0, 0);
make_inst(S, 0, diff1); make_inst(SW, 0, diff2); make_inst(NE, 0, diff3); make_inst(SE, 0, diff4);
make_inst(U, 0, diff5); make_inst(NW, 0, diff6); make_inst(E, 0, diff7); make_inst(W, 0, diff8);
make_inst(N, 0, diff9); make_inst(D, 0, wlong);
make_loc(diff1,
"You␣are␣in␣a␣maze␣of␣twisting␣little␣passages,␣all␣different.", 0, 0);
make_inst(W, 0, diff0); make_inst(SE, 0, diff2); make_inst(NW, 0, diff3); make_inst(SW, 0, diff4);
make_inst(NE, 0, diff5); make_inst(U, 0, diff6); make_inst(D, 0, diff7); make_inst(N, 0, diff8);
make_inst(S, 0, diff9); make_inst(E, 0, diff10);
make_loc(diff2,
"You␣are␣in␣a␣little␣maze␣of␣twisty␣passages,␣all␣different.", 0, 0);
make_inst(NW, 0, diff0); make_inst(U, 0, diff1); make_inst(N, 0, diff3); make_inst(S, 0, diff4);
make_inst(W, 0, diff5); make_inst(SW, 0, diff6); make_inst(NE, 0, diff7); make_inst(E, 0, diff8);
make_inst(D, 0, diff9); make_inst(SE, 0, diff10);
make_loc(diff3,
"You␣are␣in␣a␣twisting␣maze␣of␣little␣passages,␣all␣different.", 0, 0);
make_inst(U, 0, diff0); make_inst(D, 0, diff1); make_inst(W, 0, diff2); make_inst(NE, 0, diff4);
make_inst(SW, 0, diff5); make_inst(E, 0, diff6); make_inst(N, 0, diff7); make_inst(NW, 0, diff8);
make_inst(SE, 0, diff9); make_inst(S, 0, diff10);
make_loc(diff4,
"You␣are␣in␣a␣twisting␣little␣maze␣of␣passages,␣all␣different.", 0, 0);
make_inst(NE, 0, diff0); make_inst(N, 0, diff1); make_inst(NW, 0, diff2); make_inst(SE, 0, diff3);
make_inst(E, 0, diff5); make_inst(D, 0, diff6); make_inst(S, 0, diff7); make_inst(U, 0, diff8);
make_inst(W, 0, diff9); make_inst(SW, 0, diff10);
make_loc(diff5,
"You␣are␣in␣a␣twisty␣little␣maze␣of␣passages,␣all␣different.", 0, 0);
make_inst(N, 0, diff0); make_inst(SE, 0, diff1); make_inst(D, 0, diff2); make_inst(S, 0, diff3);
make_inst(E, 0, diff4); make_inst(W, 0, diff6); make_inst(SW, 0, diff7); make_inst(NE, 0, diff8);
make_inst(NW, 0, diff9); make_inst(U, 0, diff10);
make_loc(diff6,
"You␣are␣in␣a␣twisty␣maze␣of␣little␣passages,␣all␣different.", 0, 0);
make_inst(E, 0, diff0); make_inst(W, 0, diff1); make_inst(U, 0, diff2); make_inst(SW, 0, diff3);
make_inst(D, 0, diff4); make_inst(S, 0, diff5); make_inst(NW, 0, diff7); make_inst(SE, 0, diff8);
make_inst(NE, 0, diff9); make_inst(N, 0, diff10);
make_loc(diff7,
"You␣are␣in␣a␣little␣twisty␣maze␣of␣passages,␣all␣different.", 0, 0);
make_inst(SE, 0, diff0); make_inst(NE, 0, diff1); make_inst(S, 0, diff2); make_inst(D, 0, diff3);
make_inst(U, 0, diff4); make_inst(NW, 0, diff5); make_inst(N, 0, diff6); make_inst(SW, 0, diff8);
make_inst(E, 0, diff9); make_inst(W, 0, diff10);
make_loc(diff8,
"You␣are␣in␣a␣maze␣of␣little␣twisting␣passages,␣all␣different.", 0, 0);
```

```
make_inst(D, 0, diff0); make_inst(E, 0, diff1); make_inst(NE, 0, diff2); make_inst(U, 0, diff3);
make_inst(W, 0, diff4); make_inst(N, 0, diff5); make_inst(S, 0, diff6); make_inst(SE, 0, diff7);
make_inst(SW, 0, diff9); make_inst(NW, 0, diff10);
make_loc(diff9,
"You␣are␣in␣a␣maze␣of␣little␣twisty␣passages,␣all␣different.", 0, 0);
make_inst(SW, 0, diff0); make_inst(NW, 0, diff1); make_inst(E, 0, diff2); make_inst(W, 0, diff3);
make_inst(N, 0, diff4); make_inst(D, 0, diff5); make_inst(SE, 0, diff6); make_inst(U, 0, diff7);
make_inst(S, 0, diff8); make_inst(NE, 0, diff10);
make_loc(diff10,
"You␣are␣in␣a␣little␣maze␣of␣twisting␣passages,␣all␣different.", 0, 0);
make_inst(SW, 0, diff1); make_inst(N, 0, diff2); make_inst(E, 0, diff3); make_inst(NW, 0, diff4);
make_inst(SE, 0, diff5); make_inst(NE, 0, diff6); make_inst(W, 0, diff7); make_inst(D, 0, diff8);
make_inst(U, 0, diff9); make_inst(S, 0, pony);
make_loc(pony, dead_end, 0, 0);
make_inst(N, 0, diff10); ditto(OUT);
```

40. Going north of the long hall, we come to the vicinity of another large room, with royal treasures nearby. (You probably first reached this part of the cavern from the east, via the Hall of Mists.) Unfortunately, a vicious snake is here too; the conditional instructions for getting past the snake are worthy of study.

⟨Build the travel table 23⟩ +≡

```
make_loc(cross,
"You␣are␣at␣a␣crossover␣of␣a␣high␣N/S␣passage␣and␣a␣low␣E/W␣one.", 0, 0);
make_inst(W, 0, elong);
make_inst(N, 0, dead0);
make_inst(E, 0, west);
make_inst(S, 0, wlong);
make_loc(hmk,
"You␣are␣in␣the␣Hall␣of␣the␣Mountain␣King,␣with␣passages␣off␣in␣all\ndirections.",
"You're␣in␣Hall␣of␣Mt␣King.", snake_hint);
make_inst(STAIRS, 0, emist); ditto(U); ditto(E);
make_inst(N, not(SNAKE, 0), ns); ditto(L);
make_inst(S, not(SNAKE, 0), south); ditto(R);
make_inst(W, not(SNAKE, 0), west); ditto(FORWARD);
make_inst(N, 0, snaked);
make_inst(SW, 35, secret);
make_inst(SW, sees(SNAKE), snaked);
make_inst(SECRET, 0, secret);
make_loc(west,
"You␣are␣in␣the␣west␣side␣chamber␣of␣the␣Hall␣of␣the␣Mountain␣King.\n\
    A␣passage␣continues␣west␣and␣up␣here.",
"You're␣in␣west␣side␣chamber.", 0);
make_inst(HALL, 0, hmk); ditto(OUT); ditto(E);
make_inst(W, 0, cross); ditto(U);
make_loc(south,
"You␣are␣in␣the␣south␣side␣chamber.", 0, 0);
make_inst(HALL, 0, hmk); ditto(OUT); ditto(N);
```

41. North of the mountain king's domain is a curious shuttle station called Y2, with magic connections to two other places.

(Real-world cave maps often use the symbol Y to stand for an entrance, and Y2 for a secondary entrance.)

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(ns,
"You␣are␣in␣a␣low␣N/S␣passage␣at␣a␣hole␣in␣the␣floor.␣␣The␣hole␣goes\n\
    down␣to␣an␣E/W␣passage.",
"You're␣in␣N/S␣passage.", 0);
make_inst(HALL, 0, hmk); ditto(OUT); ditto(S);
make_inst(N, 0, y2); ditto(Y2);
make_inst(D, 0, dirty); ditto(HOLE);
make_loc(y2,
"You␣are␣in␣a␣large␣room,␣with␣a␣passage␣to␣the␣south,␣a␣passage␣to␣the\n\
    west,␣and␣a␣wall␣of␣broken␣rock␣to␣the␣east.␣␣There␣is␣a␣large␣\"Y2\"␣on\n\
    a␣rock␣in␣the␣room's␣center.",
"You're␣at␣\"Y2\".", 0);
make_inst(PLUGH, 0, house);
make_inst(S, 0, ns);
make_inst(E, 0, jumble); ditto(WALL); ditto(BROKEN);
make_inst(W, 0, windoe);
make_inst(PLOVER, holds(EMERALD), pdrop);
make_inst(PLOVER, 0, proom);
make_loc(jumble,
"You␣are␣in␣a␣jumble␣of␣rock,␣with␣cracks␣everywhere.", 0, 0);
make_inst(D, 0, y2); ditto(Y2);
make_inst(U, 0, emist);
make_loc(windoe,
"You're␣at␣a␣low␣window␣overlooking␣a␣huge␣pit,␣which␣extends␣up␣out␣of\n\
    sight.␣␣A␣floor␣is␣indistinctly␣visible␣over␣50␣feet␣below.␣␣Traces␣of\n\
    white␣mist␣cover␣the␣floor␣of␣the␣pit,␣becoming␣thicker␣to␣the␣right.\n\
    Marks␣in␣the␣dust␣around␣the␣window␣would␣seem␣to␣indicate␣that\n\
    someone␣has␣been␣here␣recently.␣␣Directly␣across␣the␣pit␣from␣you␣and\n\
    25␣feet␣away␣there␣is␣a␣similar␣window␣looking␣into␣a␣lighted␣room.\n\
    A␣shadowy␣figure␣can␣be␣seen␣there␣peering␣back␣at␣you.",
"You're␣at␣window␣on␣pit.", 0);
make_inst(E, 0, y2); ditto(Y2);
make_inst(JUMP, 0, neck);
```

42. Next let's consider the east/west passage below *ns*.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(dirty,
"You␣are␣in␣a␣dirty␣broken␣passage.␣␣To␣the␣east␣is␣a␣crawl.␣␣To␣the\n\
    west␣is␣a␣large␣passage.␣␣Above␣you␣is␣a␣hole␣to␣another␣passage.",
"You're␣in␣dirty␣passage.", 0);
make_inst(E, 0, clean); ditto(CRAWL);
make_inst(U, 0, ns); ditto(HOLE);
make_inst(W, 0, dusty);
make_inst(BEDQUILT, 0, bedquilt);
make_loc(clean,
"You␣are␣on␣the␣brink␣of␣a␣small␣clean␣climbable␣pit.␣␣A␣crawl␣leads\nwest.",
"You're␣by␣a␣clean␣pit.", 0);
make_inst(W, 0, dirty); ditto(CRAWL);
make_inst(D, 0, wet); ditto(PIT); ditto(CLIMB);
make_loc(wet,
"You␣are␣in␣the␣bottom␣of␣a␣small␣pit␣with␣a␣little␣stream,␣which\n\
    enters␣and␣exits␣through␣tiny␣slits.",
"You're␣in␣pit␣by␣stream.", liquid);
make_inst(CLIMB, 0, clean); ditto(U); ditto(OUT);
make_inst(SLIT, 0, slit_rmk); ditto(STREAM); ditto(D); ditto(UPSTREAM); ditto(DOWNSTREAM);
make_loc(dusty,
"You␣are␣in␣a␣large␣room␣full␣of␣dusty␣rocks.␣␣There␣is␣a␣big␣hole␣in\n\
    the␣floor.␣␣There␣are␣cracks␣everywhere,␣and␣a␣passage␣leading␣east.",
"You're␣in␣dusty␣rock␣room.", 0);
make_inst(E, 0, dirty); ditto(PASSAGE);
make_inst(D, 0, complex); ditto(HOLE); ditto(FLOOR);
make_inst(BEDQUILT, 0, bedquilt);
make_loc(complex,
"You␣are␣at␣a␣complex␣junction.␣␣A␣low␣hands-and-knees␣passage␣from␣the\n\
    north␣joins␣a␣higher␣crawl␣from␣the␣east␣to␣make␣a␣walking␣passage\n\
    going␣west.␣␣There␣is␣also␣a␣large␣room␣above.␣␣The␣air␣is␣damp␣here.",
"You're␣at␣complex␣junction.", 0);
make_inst(U, 0, dusty); ditto(CLIMB); ditto(ROOM);
make_inst(W, 0, bedquilt); ditto(BEDQUILT);
make_inst(N, 0, shell); ditto(SHELL);
make_inst(E, 0, ante);
```

43. A more-or-less self-contained cavelet can be found north of the complex passage. Its connections are more vertical than horizontal.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(shell,
"You're␣in␣a␣large␣room␣carved␣out␣of␣sedimentary␣rock.␣␣The␣floor\n\
    and␣walls␣are␣littered␣with␣bits␣of␣shells␣embedded␣in␣the␣stone.\n\
    A␣shallow␣passage␣proceeds␣downward,␣and␣a␣somewhat␣steeper␣one\n\
    leads␣up.␣␣A␣low␣hands-and-knees␣passage␣enters␣from␣the␣south.",
"You're␣in␣Shell␣Room.", 0);
make_inst(U, 0, arch); ditto(HALL);
make_inst(D, 0, ragged);
remark("You␣can't␣fit␣this␣five-foot␣clam␣through␣that␣little␣passage!");
make_inst(S, holds(CLAM), sayit);
remark("You␣can't␣fit␣this␣five-foot␣oyster␣through␣that␣little␣passage!");
make_inst(S, holds(OYSTER), sayit);
make_inst(S, 0, complex);
make_loc(arch,
"You␣are␣in␣an␣arched␣hall.␣␣A␣coral␣passage␣once␣continued␣up␣and␣east\n\
    from␣here,␣but␣is␣now␣blocked␣by␣debris.␣␣The␣air␣smells␣of␣sea␣water.",
"You're␣in␣arched␣hall.", 0);
make_inst(D, 0, shell); ditto(SHELL); ditto(OUT);
make_loc(ragged,
"You␣are␣in␣a␣long␣sloping␣corridor␣with␣ragged␣sharp␣walls.", 0, 0);
make_inst(U, 0, shell); ditto(SHELL);
make_inst(D, 0, sac);
make_loc(sac,
"You␣are␣in␣a␣cul-de-sac␣about␣eight␣feet␣across.", 0, 0);
make_inst(U, 0, ragged); ditto(OUT);
make_inst(SHELL, 0, shell);
```

44. A dangerous section lies east of the complex junction.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(ante,
"You␣are␣in␣an␣anteroom␣leading␣to␣a␣large␣passage␣to␣the␣east.␣␣Small\n\
    passages␣go␣west␣and␣up.␣␣The␣remnants␣of␣recent␣digging␣are␣evident.\n\
    A␣sign␣in␣midair␣here␣says␣\"CAVE␣UNDER␣CONSTRUCTION␣BEYOND␣THIS␣POINT.\n\
    PROCEED␣AT␣OWN␣RISK.␣␣[WITT␣CONSTRUCTION␣COMPANY]\"",
"You're␣in␣anteroom.", 0);
make_inst(U, 0, complex);
make_inst(W, 0, bedquilt);
make_inst(E, 0, witt);
make_loc(witt,
"You␣are␣at␣Witt's␣End.␣␣Passages␣lead␣off␣in␣\"all\"␣directions.",
"You're␣at␣Witt's␣End.", witt_hint);
remark("You␣have␣crawled␣around␣in␣some␣little␣holes␣and␣wound␣up␣back␣in␣the\n\
    main␣passage.");
loop_rmk = sayit;
make_inst(E, 95, sayit); ditto(N); ditto(S);
ditto(NE); ditto(SE); ditto(SW); ditto(NW); ditto(U); ditto(D);
make_inst(E, 0, ante);     /* one chance in 20 */
remark("You␣have␣crawled␣around␣in␣some␣little␣holes␣and␣found␣your␣way\n\
    blocked␣by␣a␣recent␣cave-in.␣␣You␣are␣now␣back␣in␣the␣main␣passage.");
make_inst(W, 0, sayit);
```

45. Will Crowther, who actively explored and mapped many caves in Kentucky before inventing Adventure, named Bedquilt after the Bedquilt Entrance to Colossal Cave. (The real Colossal Cave was discovered near Mammoth Cave in 1895, and its Bedquilt Entrance was found in 1896; see *The Longest Cave* by Brucker and Watson (New York: Knopf, 1976) for further details.)

Random exploration is the name of the game here.

```
⟨ Build the travel table 23 ⟩ +≡
  make_loc(bedquilt,
  "You␣are␣in␣Bedquilt,␣a␣long␣east/west␣passage␣with␣holes␣everywhere.\n\
      To␣explore␣at␣random␣select␣north,␣south,␣up,␣or␣down.",
  "You're␣in␣Bedquilt.", 0);
  make_inst(E, 0, complex);
  make_inst(W, 0, cheese);
  make_inst(S, 80, loop_rmk);
  make_inst(SLAB, 0, slab);
  make_inst(U, 80, loop_rmk);
  make_inst(U, 50, abovep);
  make_inst(U, 0, dusty);
  make_inst(N, 60, loop_rmk);
  make_inst(N, 75, low);
  make_inst(N, 0, sjunc);
  make_inst(D, 80, loop_rmk);
  make_inst(D, 0, ante);
  make_loc(cheese,
  "You␣are␣in␣a␣room␣whose␣walls␣resemble␣Swiss␣cheese.␣␣Obvious␣passages\n\
      go␣west,␣east,␣NE,␣and␣NW.␣␣Part␣of␣the␣room␣is␣occupied␣by␣a␣large\nbedrock␣block.",
  "You're␣in␣Swiss␣cheese␣room.", 0);
  make_inst(NE, 0, bedquilt);
  make_inst(W, 0, e2pit);
  make_inst(S, 80, loop_rmk);
  make_inst(CANYON, 0, tall);
  make_inst(E, 0, soft);
  make_inst(NW, 50, loop_rmk);
  make_inst(ORIENTAL, 0, oriental);
  make_loc(soft,
  "You␣are␣in␣the␣Soft␣Room.␣␣The␣walls␣are␣covered␣with␣heavy␣curtains,\n\
      the␣floor␣with␣a␣thick␣pile␣carpet.␣␣Moss␣covers␣the␣ceiling.",
  "You're␣in␣Soft␣Room.", 0);
  make_inst(W, 0, cheese);  ditto(OUT);
```

46. West of the quilt and the cheese is a room with two pits. Why would you want to descend into the pits? Keep playing and you'll find out.

```
⟨ Build the travel table 23 ⟩ +≡
  make_loc(e2pit,
  "You␣are␣at␣the␣east␣end␣of␣the␣Twopit␣Room.␣␣The␣floor␣here␣is\n\
     littered␣with␣thin␣rock␣slabs,␣which␣make␣it␣easy␣to␣descend␣the␣pits.\n\
     There␣is␣a␣path␣here␣bypassing␣the␣pits␣to␣connect␣passages␣from␣east\n\
     and␣west.␣␣There␣are␣holes␣all␣over,␣but␣the␣only␣big␣one␣is␣on␣the\n\
     wall␣directly␣over␣the␣west␣pit␣where␣you␣can't␣get␣to␣it.",
  "You're␣at␣east␣end␣of␣Twopit␣Room.", 0);
  make_inst(E, 0, cheese);
  make_inst(W, 0, w2pit); ditto(ACROSS);
  make_inst(D, 0, epit); ditto(PIT);
  make_loc(w2pit,
  "You␣are␣at␣the␣west␣end␣of␣the␣Twopit␣Room.␣␣There␣is␣a␣large␣hole␣in\n\
     the␣wall␣above␣the␣pit␣at␣this␣end␣of␣the␣room.",
  "You're␣at␣west␣end␣of␣Twopit␣Room.", 0);
  make_inst(E, 0, e2pit); ditto(ACROSS);
  make_inst(W, 0, slab); ditto(SLAB);
  make_inst(D, 0, wpit); ditto(PIT);
  remark("It␣is␣too␣far␣up␣for␣you␣to␣reach.");
  make_inst(HOLE, 0, sayit);
  make_loc(epit,
  "You␣are␣at␣the␣bottom␣of␣the␣eastern␣pit␣in␣the␣Twopit␣Room.␣␣There␣is\n\
     a␣small␣pool␣of␣oil␣in␣one␣corner␣of␣the␣pit.",
  "You're␣in␣east␣pit.", liquid + oil);
  make_inst(U, 0, e2pit); ditto(OUT);
  make_loc(wpit,
  "You␣are␣at␣the␣bottom␣of␣the␣western␣pit␣in␣the␣Twopit␣Room.␣␣There␣is\n\
     a␣large␣hole␣in␣the␣wall␣about␣25␣feet␣above␣you.",
  "You're␣in␣west␣pit.", 0);
  make_inst(U, 0, w2pit); ditto(OUT);
  make_inst(CLIMB, not(PLANT, 4), check);
  make_inst(CLIMB, 0, climb);
```

47. Oho, you climbed the plant in the west pit! Now you're in another scenic area with rare treasures—if you can get through the door.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(narrow,
"You␣are␣in␣a␣long,␣narrow␣corridor␣stretching␣out␣of␣sight␣to␣the\n\
    west.␣␣At␣the␣eastern␣end␣is␣a␣hole␣through␣which␣you␣can␣see␣a\n\
    profusion␣of␣leaves.",
"You're␣in␣narrow␣corridor.", 0);
make_inst(D, 0, wpit); ditto(CLIMB); ditto(E);
make_inst(JUMP, 0, neck);
make_inst(W, 0, giant); ditto(GIANT);
make_loc(giant,
"You␣are␣in␣the␣Giant␣Room.␣␣The␣ceiling␣here␣is␣too␣high␣up␣for␣your\n\
    lamp␣to␣show␣it.␣␣Cavernous␣passages␣lead␣east,␣north,␣and␣south.␣␣On\n\
    the␣west␣wall␣is␣scrawled␣the␣inscription,␣\"FEE␣FIE␣FOE␣FOO\"␣[sic].",
"You're␣in␣Giant␣Room.", 0);
make_inst(S, 0, narrow);
make_inst(E, 0, block);
make_inst(N, 0, immense);
make_loc(block,
"The␣passage␣here␣is␣blocked␣by␣a␣recent␣cave-in.", 0, 0);
make_inst(S, 0, giant); ditto(GIANT); ditto(OUT);
make_loc(immense,
"You␣are␣at␣one␣end␣of␣an␣immense␣north/south␣passage.", 0, 0);
make_inst(S, 0, giant); ditto(GIANT); ditto(PASSAGE);
make_inst(N, not(DOOR, 0), falls); ditto(ENTER); ditto(CAVERN);
remark("The␣door␣is␣extremely␣rusty␣and␣refuses␣to␣open.");
make_inst(N, 0, sayit);
make_loc(falls,
"You␣are␣in␣a␣magnificent␣cavern␣with␣a␣rushing␣stream,␣which␣cascades\n\
    over␣a␣sparkling␣waterfall␣into␣a␣roaring␣whirlpool␣that␣disappears\n\
    through␣a␣hole␣in␣the␣floor.␣␣Passages␣exit␣to␣the␣south␣and␣west.",
"You're␣in␣cavern␣with␣waterfall.", liquid);
make_inst(S, 0, immense); ditto(OUT);
make_inst(GIANT, 0, giant);
make_inst(W, 0, steep);
make_loc(steep,
"You␣are␣at␣the␣top␣of␣a␣steep␣incline␣above␣a␣large␣room.␣␣You␣could\n\
    climb␣down␣here,␣but␣you␣would␣not␣be␣able␣to␣climb␣up.␣␣There␣is␣a\n\
    passage␣leading␣back␣to␣the␣north.",
"You're␣at␣steep␣incline␣above␣large␣room.", 0);
make_inst(N, 0, falls); ditto(CAVERN); ditto(PASSAGE);
make_inst(D, 0, low); ditto(CLIMB);
```

48. Meanwhile let's backtrack to another part of the cave possibly reachable from Bedquilt.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(abovep,
"You␣are␣in␣a␣secret␣N/S␣canyon␣above␣a␣sizable␣passage.", 0, 0);
make_inst(N, 0, sjunc);
make_inst(D, 0, bedquilt); ditto(PASSAGE);
make_inst(S, 0, tite);
make_loc(sjunc,
"You␣are␣in␣a␣secret␣canyon␣at␣a␣junction␣of␣three␣canyons,␣bearing\n\
    north,␣south,␣and␣SE.␣␣The␣north␣one␣is␣as␣tall␣as␣the␣other␣two\ncombined.",
"You're␣at␣junction␣of␣three␣secret␣canyons.", 0);
make_inst(SE, 0, bedquilt);
make_inst(S, 0, abovep);
make_inst(N, 0, window);
make_loc(tite,
"A␣large␣stalactite␣extends␣from␣the␣roof␣and␣almost␣reaches␣the␣floor\n\
    below.␣␣You␣could␣climb␣down␣it,␣and␣jump␣from␣it␣to␣the␣floor,␣but\n\
    having␣done␣so␣you␣would␣be␣unable␣to␣reach␣it␣to␣climb␣back␣up.",
"You're␣on␣top␣of␣stalactite.", 0);
make_inst(N, 0, abovep);
make_inst(D, 40, like6); ditto(JUMP); ditto(CLIMB);
make_inst(D, 50, like9);
make_inst(D, 0, like4);      /* oh dear, you're in a random part of the maze */
make_loc(low,
"You␣are␣in␣a␣large␣low␣room.␣␣Crawls␣lead␣north,␣SE,␣and␣SW.", 0, 0);
make_inst(BEDQUILT, 0, bedquilt);
make_inst(SW, 0, scorr);
make_inst(N, 0, crawl);
make_inst(SE, 0, oriental); ditto(ORIENTAL);
make_loc(crawl,
"Dead␣end␣crawl.", 0, 0);
make_inst(S, 0, low); ditto(CRAWL); ditto(OUT);
```

49. The described view from the west window, *window*, is identical to the view from the east window, *windoe*, except for one word. What on earth do you see from those windows? (Don Woods has confided that the shadowy figure is actually your own reflection, because *mirror* lies between the two window rooms. An intentional false clue.)

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(window,
"You're␣at␣a␣low␣window␣overlooking␣a␣huge␣pit,␣which␣extends␣up␣out␣of\n\
    sight.␣␣A␣floor␣is␣indistinctly␣visible␣over␣50␣feet␣below.␣␣Traces␣of\n\
    white␣mist␣cover␣the␣floor␣of␣the␣pit,␣becoming␣thicker␣to␣the␣left.\n\
    Marks␣in␣the␣dust␣around␣the␣window␣would␣seem␣to␣indicate␣that\n\
    someone␣has␣been␣here␣recently.␣␣Directly␣across␣the␣pit␣from␣you␣and\n\
    25␣feet␣away␣there␣is␣a␣similar␣window␣looking␣into␣a␣lighted␣room.\n\
    A␣shadowy␣figure␣can␣be␣seen␣there␣peering␣back␣to␣you.",
short_desc[windoe], 0);
make_inst(W, 0, sjunc);
make_inst(JUMP, 0, neck);
```

50. More treasures await you via the *low* corridor.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(oriental,
"This␣is␣the␣Oriental␣Room.␣␣Ancient␣oriental␣cave␣drawings␣cover␣the\n\
    walls.␣␣A␣gently␣sloping␣passage␣leads␣upward␣to␣the␣north,␣another\n\
    passage␣leads␣SE,␣and␣a␣hands-and-knees␣crawl␣leads␣west.",
"You're␣in␣Oriental␣Room.", 0);
make_inst(SE, 0, cheese);
make_inst(W, 0, low); ditto(CRAWL);
make_inst(U, 0, misty); ditto(N); ditto(CAVERN);
make_loc(misty,
"You␣are␣following␣a␣wide␣path␣around␣the␣outer␣edge␣of␣a␣large␣cavern.\n\
    Far␣below,␣through␣a␣heavy␣white␣mist,␣strange␣splashing␣noises␣can␣be\n\
    heard.␣␣The␣mist␣rises␣up␣through␣a␣fissure␣in␣the␣ceiling.␣␣The␣path\n\
    exits␣to␣the␣south␣and␣west.",
"You're␣in␣misty␣cavern.", 0);
make_inst(S, 0, oriental); ditto(ORIENTAL);
make_inst(W, 0, alcove);
```

51. One of the darkest secrets is hidden here. You will discover that you must take the emerald from the Plover Room to the alcove. But you don't learn the name of the Plover Room until the second time you've been there, since your first visit will be lampless until you know the secret.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(alcove,
"You␣are␣in␣an␣alcove.␣␣A␣small␣NW␣path␣seems␣to␣widen␣after␣a␣short\n\
    distance.␣␣An␣extremely␣tight␣tunnel␣leads␣east.␣␣It␣looks␣like␣a␣very\n\
    tight␣squeeze.␣␣An␣eerie␣light␣can␣be␣seen␣at␣the␣other␣end.",
"You're␣in␣alcove.", dark_hint);
make_inst(NW, 0, misty); ditto(CAVERN);
make_inst(E, 0, ppass); ditto(PASSAGE);
make_inst(E, 0, proom);     /* never performed, but seen by 'go back' */
make_loc(proom,
"You're␣in␣a␣small␣chamber␣lit␣by␣an␣eerie␣green␣light.␣␣An␣extremely\n\
    narrow␣tunnel␣exits␣to␣the␣west.␣␣A␣dark␣corridor␣leads␣NE.",
"You're␣in␣Plover␣Room.", dark_hint);
make_inst(W, 0, ppass); ditto(PASSAGE); ditto(OUT);
make_inst(W, 0, alcove);     /* never performed, but seen by 'go back' */
make_inst(PLOVER, holds(EMERALD), pdrop);
make_inst(PLOVER, 0, y2);
make_inst(NE, 0, droom); ditto(DARK);
make_loc(droom,
"You're␣in␣the␣Dark-Room.␣␣A␣corridor␣leading␣south␣is␣the␣only␣exit.",
"You're␣in␣Dark-Room.", dark_hint);
make_inst(S, 0, proom); ditto(PLOVER); ditto(OUT);
```

52. We forgot to mention the circuitous passage leading west from the Twopit Room. It winds around and takes you to a somewhat more mundane area, yet not without interest.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(slab,
"You␣are␣in␣a␣large␣low␣circular␣chamber␣whose␣floor␣is␣an␣immense␣slab\n\
    fallen␣from␣the␣ceiling␣(Slab␣Room).␣␣There␣once␣were␣large␣passages\n\
    to␣the␣east␣and␣west,␣but␣they␣are␣now␣filled␣with␣boulders.␣␣Low\n\
    small␣passages␣go␣north␣and␣south,␣and␣the␣south␣one␣quickly␣bends\n\
    west␣around␣the␣boulders.",
"You're␣in␣Slab␣Room.", 0);
make_inst(S, 0, w2pit);
make_inst(U, 0, abover); ditto(CLIMB);
make_inst(N, 0, bedquilt);
make_loc(abover,
"You␣are␣in␣a␣secret␣N/S␣canyon␣above␣a␣large␣room.", 0, 0);
make_inst(D, 0, slab); ditto(SLAB);
make_inst(S, not(DRAGON, 0), scan2);
make_inst(S, 0, scan1);
make_inst(N, 0, mirror);
make_inst(RESERVOIR, 0, res);
make_loc(mirror,
"You␣are␣in␣a␣north/south␣canyon␣about␣25␣feet␣across.␣␣The␣floor␣is\n\
    covered␣by␣white␣mist␣seeping␣in␣from␣the␣north.␣␣The␣walls␣extend\n\
    upward␣for␣well␣over␣100␣feet.␣␣Suspended␣from␣some␣unseen␣point␣far\n\
    above␣you,␣an␣enormous␣two-sided␣mirror␣is␣hanging␣parallel␣to␣and\n\
    midway␣between␣the␣canyon␣walls.␣␣(The␣mirror␣is␣obviously␣provided\n\
    for␣the␣use␣of␣the␣dwarves,␣who␣as␣you␣know␣are␣extremely␣vain.)\n\
    A␣small␣window␣can␣be␣seen␣in␣either␣wall,␣some␣fifty␣feet␣up.",
"You're␣in␣mirror␣canyon.", 0);
make_inst(S, 0, abover);
make_inst(N, 0, res); ditto(RESERVOIR);
make_loc(res,
"You␣are␣at␣the␣edge␣of␣a␣large␣underground␣reservoir.␣␣An␣opaque␣cloud\n\
    of␣white␣mist␣fills␣the␣room␣and␣rises␣rapidly␣upward.␣␣The␣lake␣is\n\
    fed␣by␣a␣stream,␣which␣tumbles␣out␣of␣a␣hole␣in␣the␣wall␣about␣10␣feet\n\
    overhead␣and␣splashes␣noisily␣into␣the␣water␣somewhere␣within␣the\n\
    mist.␣␣The␣only␣passage␣goes␣back␣toward␣the␣south.",
"You're␣at␣reservoir.", liquid);
make_inst(S, 0, mirror); ditto(OUT);
```

53. Four more secret canyons lead back to the Hall of the Mountain King. Three of them are actually the same, but the dragon blocks the connection between the northern passage (to *abover*) and the eastern passage (to *secret*). Once you've vanquished the dragon, *scan2* takes the place of *scan1* and *scan3*.

⟨ Build the travel table 23 ⟩ +≡
```
make_loc(scan1,
"You␣are␣in␣a␣secret␣canyon␣that␣exits␣to␣the␣north␣and␣east.", 0, 0);
make_inst(N, 0, abover); ditto(OUT);
remark("The␣dragon␣looks␣rather␣nasty.␣␣You'd␣best␣not␣try␣to␣get␣by.");
make_inst(E, 0, sayit); ditto(FORWARD);
make_loc(scan2, long_desc[scan1], 0, 0);
make_inst(N, 0, abover);
make_inst(E, 0, secret);
make_loc(scan3, long_desc[scan1], 0, 0);
make_inst(E, 0, secret); ditto(OUT);
make_inst(N, 0, sayit); ditto(FORWARD);
make_loc(secret,
"You␣are␣in␣a␣secret␣canyon,␣which␣here␣runs␣E/W.␣␣It␣crosses␣over␣a\n\
    very␣tight␣canyon␣15␣feet␣below.␣␣If␣you␣go␣down␣you␣may␣not␣be␣able\n\
    to␣get␣back␣up.",
"You're␣in␣secret␣E/W␣canyon␣above␣tight␣canyon.", 0);
make_inst(E, 0, hmk);
make_inst(W, not(DRAGON, 0), scan2);
make_inst(W, 0, scan3);
make_inst(D, 0, wide);
```

54. Below *secret* there's another way to reach the cheese.

⟨ Build the travel table 23 ⟩ +≡
```
make_loc(wide,
"You␣are␣at␣a␣wide␣place␣in␣a␣very␣tight␣N/S␣canyon.", 0, 0);
make_inst(S, 0, tight);
make_inst(N, 0, tall);
make_loc(tight,
"The␣canyon␣here␣becomes␣too␣tight␣to␣go␣further␣south.", 0, 0);
make_inst(N, 0, wide);
make_loc(tall,
"You␣are␣in␣a␣tall␣E/W␣canyon.␣␣A␣low␣tight␣crawl␣goes␣3␣feet␣north␣and\n\
    seems␣to␣open␣up.",
"You're␣in␣tall␣E/W␣canyon.", 0);
make_inst(E, 0, wide);
make_inst(W, 0, boulders);
make_inst(N, 0, cheese); ditto(CRAWL);
make_loc(boulders,
"The␣canyon␣runs␣into␣a␣mass␣of␣boulders␣---␣dead␣end.", 0, 0);
make_inst(S, 0, tall);
```

55. If you aren't having fun yet, wait till you meet the troll. The only way to get here is to crawl southwest from the *low* room. And then you have a new problem to solve; we'll see later that the `TROLL` and the `BRIDGE` are here.

(Don Woods got the idea for the mist-covered bridge after an early morning visit to Mount Diablo; see Steven Levy, *Hackers* (New York: Delta, 1994), Chapter 7.)

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(scorr,
"You␣are␣in␣a␣long␣winding␣corridor␣sloping␣out␣of␣sight␣in␣both\ndirections.",
"You're␣in␣sloping␣corridor.", 0);
make_inst(D, 0, low);
make_inst(U, 0, swside);
make_loc(swside,
"You␣are␣on␣one␣side␣of␣a␣large,␣deep␣chasm.␣␣A␣heavy␣white␣mist␣rising\n\
    up␣from␣below␣obscures␣all␣view␣of␣the␣far␣side.␣␣A␣SW␣path␣leads␣away\n\
    from␣the␣chasm␣into␣a␣winding␣corridor.",
"You're␣on␣SW␣side␣of␣chasm.", 0);
make_inst(SW, 0, scorr);
remark("The␣troll␣refuses␣to␣let␣you␣cross.");
make_inst(OVER, sees(TROLL), sayit); ditto(ACROSS); ditto(CROSS); ditto(NE);
remark("There␣is␣no␣longer␣any␣way␣across␣the␣chasm.");
make_inst(OVER, not(TROLL, 0), sayit);
make_inst(OVER, 0, troll);
make_inst(JUMP, not(TROLL, 0), lose);
make_inst(JUMP, 0, bridge_rmk);
```

56. The only things not yet explored on this side of the troll bridge are a dozen dead ends. They appear at this place in the ordering of all locations because of the pirate logic explained later: The pirate will never go to locations $\geq$ *dead3*.

#define *max_pirate_loc* *dead2*

⟨ Build the travel table 23 ⟩ +≡
make_loc(*dead0*, *dead_end*, 0, 0);
make_inst(S, 0, *cross*); *ditto*(OUT);
make_loc(*dead1*, *dead_end*, 0, *twist_hint*);
make_inst(W, 0, *like11*); *ditto*(OUT);
make_loc(*dead2*, *dead_end*, 0, 0);
make_inst(SE, 0, *like13*);
make_loc(*dead3*, *dead_end*, 0, *twist_hint*);
make_inst(W, 0, *like4*); *ditto*(OUT);
make_loc(*dead4*, *dead_end*, 0, *twist_hint*);
make_inst(E, 0, *like4*); *ditto*(OUT);
make_loc(*dead5*, *dead_end*, 0, *twist_hint*);
make_inst(U, 0, *like3*); *ditto*(OUT);
make_loc(*dead6*, *dead_end*, 0, *twist_hint*);
make_inst(W, 0, *like9*); *ditto*(OUT);
make_loc(*dead7*, *dead_end*, 0, *twist_hint*);
make_inst(U, 0, *like10*); *ditto*(OUT);
make_loc(*dead8*, *dead_end*, 0, 0);
make_inst(E, 0, *brink*); *ditto*(OUT);
make_loc(*dead9*, *dead_end*, 0, *twist_hint*);
make_inst(S, 0, *like3*); *ditto*(OUT);
make_loc(*dead10*, *dead_end*, 0, *twist_hint*);
make_inst(E, 0, *like12*); *ditto*(OUT);
make_loc(*dead11*, *dead_end*, 0, *twist_hint*);
make_inst(U, 0, *like8*); *ditto*(OUT);

57. A whole nuther cave with nine sites and additional treasures is on tuther side of the troll bridge! This cave was inspired in part by J. R. R. Tolkien's stories.

⟨Build the travel table 23⟩ +≡

```
make_loc(neside,
"You␣are␣on␣the␣far␣side␣of␣the␣chasm.␣␣A␣NE␣path␣leads␣away␣from␣the\n\
    chasm␣on␣this␣side.",
"You're␣on␣NE␣side␣of␣chasm.", 0);
make_inst(NE, 0, corr);
make_inst(OVER, sees(TROLL), sayit − 1); ditto(ACROSS); ditto(CROSS); ditto(SW);
make_inst(OVER, 0, troll);
make_inst(JUMP, 0, bridge_rmk);
make_inst(FORK, 0, fork);
make_inst(VIEW, 0, view);
make_inst(BARREN, 0, fbarr);
make_loc(corr,
"You're␣in␣a␣long␣east/west␣corridor.␣␣A␣faint␣rumbling␣noise␣can␣be\n\
    heard␣in␣the␣distance.",
"You're␣in␣corridor.", 0);
make_inst(W, 0, neside);
make_inst(E, 0, fork); ditto(FORK);
make_inst(VIEW, 0, view);
make_inst(BARREN, 0, fbarr);
make_loc(fork,
"The␣path␣forks␣here.␣␣The␣left␣fork␣leads␣northeast.␣␣A␣dull␣rumbling\n\
    seems␣to␣get␣louder␣in␣that␣direction.␣␣The␣right␣fork␣leads␣southeast\n\
    down␣a␣gentle␣slope.␣␣The␣main␣corridor␣enters␣from␣the␣west.",
"You're␣at␣fork␣in␣path.", 0);
make_inst(W, 0, corr);
make_inst(NE, 0, warm); ditto(L);
make_inst(SE, 0, lime); ditto(R); ditto(D);
make_inst(VIEW, 0, view);
make_inst(BARREN, 0, fbarr);
make_loc(warm,
"The␣walls␣are␣quite␣warm␣here.␣␣From␣the␣north␣can␣be␣heard␣a␣steady\n\
    roar,␣so␣loud␣that␣the␣entire␣cave␣seems␣to␣be␣trembling.␣␣Another\n\
    passage␣leads␣south,␣and␣a␣low␣crawl␣goes␣east.",
"You're␣at␣junction␣with␣warm␣walls.", 0);
make_inst(S, 0, fork); ditto(FORK);
make_inst(N, 0, view); ditto(VIEW);
make_inst(E, 0, chamber); ditto(CRAWL);
make_loc(view,
"You␣are␣on␣the␣edge␣of␣a␣breath-taking␣view.␣␣Far␣below␣you␣is␣an\n\
    active␣volcano,␣from␣which␣great␣gouts␣of␣molten␣lava␣come␣surging\n\
    out,␣cascading␣back␣down␣into␣the␣depths.␣␣The␣glowing␣rock␣fills␣the\n\
    farthest␣reaches␣of␣the␣cavern␣with␣a␣blood-red␣glare,␣giving␣every-\n\
    thing␣an␣eerie,␣macabre␣appearance.␣␣The␣air␣is␣filled␣with␣flickering\n\
    sparks␣of␣ash␣and␣a␣heavy␣smell␣of␣brimstone.␣␣The␣walls␣are␣hot␣to\n\
    the␣touch,␣and␣the␣thundering␣of␣the␣volcano␣drowns␣out␣all␣other\n\
    sounds.␣␣Embedded␣in␣the␣jagged␣roof␣far␣overhead␣are␣myriad␣twisted\n\
    formations,␣composed␣of␣pure␣white␣alabaster,␣which␣scatter␣the␣murky\n\
    light␣into␣sinister␣apparitions␣upon␣the␣walls.␣␣To␣one␣side␣is␣a␣deep\n\
```

```
    gorge,␣filled␣with␣a␣bizarre␣chaos␣of␣tortured␣rock␣that␣seems␣to␣have\n\
    been␣crafted␣by␣the␣Devil␣himself.␣␣An␣immense␣river␣of␣fire␣crashes\n\
    out␣from␣the␣depths␣of␣the␣volcano,␣burns␣its␣way␣through␣the␣gorge,\n\
    and␣plummets␣into␣a␣bottomless␣pit␣far␣off␣to␣your␣left.␣␣To␣the\n\
    right,␣an␣immense␣geyser␣of␣blistering␣steam␣erupts␣continuously\n\
    from␣a␣barren␣island␣in␣the␣center␣of␣a␣sulfurous␣lake,␣which␣bubbles\n\
    ominously.␣␣The␣far␣right␣wall␣is␣aflame␣with␣an␣incandescence␣of␣its\n\
    own,␣which␣lends␣an␣additional␣infernal␣splendor␣to␣the␣already\n\
    hellish␣scene.␣␣A␣dark,␣foreboding␣passage␣exits␣to␣the␣south.",
"You're␣at␣breath-taking␣view.", lighted);
make_inst(S, 0, warm); ditto(PASSAGE); ditto(OUT);
make_inst(FORK, 0, fork);
remark(default_msg[EAT]);
make_inst(D, 0, sayit); ditto(JUMP);
make_loc(chamber,
"You␣are␣in␣a␣small␣chamber␣filled␣with␣large␣boulders.␣␣The␣walls␣are\n\
    very␣warm,␣causing␣the␣air␣in␣the␣room␣to␣be␣almost␣stifling␣from␣the\n\
    heat.␣␣The␣only␣exit␣is␣a␣crawl␣heading␣west,␣through␣which␣a␣low\n\
    rumbling␣noise␣is␣coming.",
"You're␣in␣chamber␣of␣boulders.", 0);
make_inst(W, 0, warm); ditto(OUT); ditto(CRAWL);
make_inst(FORK, 0, fork);
make_inst(VIEW, 0, view);
make_loc(lime,
"You␣are␣walking␣along␣a␣gently␣sloping␣north/south␣passage␣lined␣with\n\
    oddly␣shapped␣limestone␣formations.",
"You're␣in␣limestone␣passage.", 0);
make_inst(N, 0, fork); ditto(U); ditto(FORK);
make_inst(S, 0, fbarr); ditto(D); ditto(BARREN);
make_inst(VIEW, 0, view);
make_loc(fbarr,
"You␣are␣standing␣at␣the␣entrance␣to␣a␣large,␣barren␣room.␣␣A␣sign\n\
    posted␣above␣the␣entrance␣reads:␣␣\"CAUTION!␣␣BEAR␣IN␣ROOM!\"",
"You're␣in␣front␣of␣barren␣room.", 0);      /* don't laugh too loud */
make_inst(W, 0, lime); ditto(U);
make_inst(FORK, 0, fork);
make_inst(E, 0, barr); ditto(IN); ditto(BARREN); ditto(ENTER);
make_inst(VIEW, 0, view);
make_loc(barr,
"You␣are␣inside␣a␣barren␣room.␣␣The␣center␣of␣the␣room␣is␣completely\n\
    empty␣except␣for␣some␣dust.␣␣Marks␣in␣the␣dust␣lead␣away␣toward␣the\n\
    far␣end␣of␣the␣room.␣␣The␣only␣exit␣is␣the␣way␣you␣came␣in.",
"You're␣in␣barren␣room.", 0);
make_inst(W, 0, fbarr); ditto(OUT);
make_inst(FORK, 0, fork);
make_inst(VIEW, 0, view);
```

58. The two storage locations are accessible only from each other, and they lead only to each other.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(neend,
"You␣are␣at␣the␣northeast␣end␣of␣an␣immense␣room,␣even␣larger␣than␣the\n\
    Giant␣Room.␣␣It␣appears␣to␣be␣a␣repository␣for␣the␣\"Adventure\"\n\
    program.␣␣Massive␣torches␣far␣overhead␣bathe␣the␣room␣with␣smoky\n\
    yellow␣light.␣␣Scattered␣about␣you␣can␣be␣seen␣a␣pile␣of␣bottles␣(all\n\
    of␣them␣empty),␣a␣nursery␣of␣young␣beanstalks␣murmuring␣quietly,␣a␣bed\n\
    of␣oysters,␣a␣bundle␣of␣black␣rods␣with␣rusty␣stars␣on␣their␣ends,␣and\n\
    a␣collection␣of␣brass␣lanterns.␣␣Off␣to␣one␣side␣a␣great␣many␣dwarves\n\
    are␣sleeping␣on␣the␣floor,␣snoring␣loudly.␣␣A␣sign␣nearby␣reads:␣\"DO\n\
    NOT␣DISTURB␣THE␣DWARVES!\"␣␣An␣immense␣mirror␣is␣hanging␣against␣one\n\
    wall,␣and␣stretches␣to␣the␣other␣end␣of␣the␣room,␣where␣various␣other\n\
    sundry␣objects␣can␣be␣glimpsed␣dimly␣in␣the␣distance.",
"You're␣at␣NE␣end.", lighted);
make_inst(SW, 0, swend);
make_loc(swend,
"You␣are␣at␣the␣southwest␣end␣of␣the␣repository.␣␣To␣one␣side␣is␣a␣pit\n\
    full␣of␣fierce␣green␣snakes.␣␣On␣the␣other␣side␣is␣a␣row␣of␣small\n\
    wicker␣cages,␣each␣of␣which␣contains␣a␣little␣sulking␣bird.␣␣In␣one\n\
    corner␣is␣a␣bundle␣of␣black␣rods␣with␣rusty␣marks␣on␣their␣ends.\n\
    A␣large␣number␣of␣velvet␣pillows␣are␣scattered␣about␣on␣the␣floor.\n\
    A␣vast␣mirror␣stretches␣off␣to␣the␣northeast.␣␣At␣your␣feet␣is␣a\n\
    large␣steel␣grate,␣next␣to␣which␣is␣a␣sign␣that␣reads,␣\"TREASURE\n\
    VAULT.␣␣KEYS␣IN␣MAIN␣OFFICE.\"",
"You're␣at␣SW␣end.", lighted);
make_inst(NE, 0, neend);
make_inst(D, 0, grate_rmk);
```

59. When the current location is *crack* or higher, it's a pseudo-location. In such cases we don't ask you for input; we assume that you have told us to force another instruction through. For example, if you try to go through the crack by the small pit in the upper cave (location *spit*), the instruction there sends you to *crack*, which immediately sends you back to *spit*.

```
#define forced_move(loc) (loc ≥ min_forced_loc)
#define FORCE 0    /* actually any value will do here */
```

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(crack,
"The␣crack␣is␣far␣too␣small␣for␣you␣to␣follow.", 0, 0);
make_inst(FORCE, 0, spit);
```

60. Here are some forced actions that are less pleasant.

⟨ Build the travel table 23 ⟩ +≡

```
make_loc(neck,
"You␣are␣at␣the␣bottom␣of␣the␣pit␣with␣a␣broken␣neck.", 0, 0);
make_inst(FORCE, 0, limbo);
make_loc(lose, "You␣didn't␣make␣it.", 0, 0);
make_inst(FORCE, 0, limbo);
```

61. The rest are more-or-less routine, except for *check*—which executes a *conditional* forced command.

```
⟨ Build the travel table 23 ⟩ +≡
  make_loc(cant,
  "The␣dome␣is␣unclimbable.", 0, 0);
  make_inst(FORCE, 0, emist);
  make_loc(climb,
  "You␣clamber␣up␣the␣plant␣and␣scurry␣through␣the␣hole␣at␣the␣top.", 0, 0);
  make_inst(FORCE, 0, narrow);
  make_loc(check, "", 0, 0);
  make_inst(FORCE, not(PLANT, 2), upnout);
  make_inst(FORCE, 0, didit);
  make_loc(snaked,
  "You␣can't␣get␣by␣the␣snake.", 0, 0);
  make_inst(FORCE, 0, hmk);
  make_loc(thru,
  "You␣have␣crawled␣through␣a␣very␣low␣wide␣passage␣parallel␣to␣and␣north\n\
      of␣the␣Hall␣of␣Mists.", 0, 0);
  make_inst(FORCE, 0, wmist);
  make_loc(duck, long_desc[thru], 0, 0);
  make_inst(FORCE, 0, wfiss);
  make_loc(sewer,
  "The␣stream␣flows␣out␣through␣a␣pair␣of␣1-foot-diameter␣sewer␣pipes.\n\
      It␣would␣be␣advisable␣to␣use␣the␣exit.", 0, 0);
  make_inst(FORCE, 0, house);
  make_loc(upnout,
  "There␣is␣nothing␣here␣to␣climb.␣␣Use␣\"up\"␣or␣\"out\"␣to␣leave␣the␣pit.", 0, 0);
  make_inst(FORCE, 0, wpit);
  make_loc(didit,
  "You␣have␣climbed␣up␣the␣plant␣and␣out␣of␣the␣pit.", 0, 0);
  make_inst(FORCE, 0, w2pit);
```

62. The table of instructions ends here; the remaining "locations" *ppass*, *pdrop*, and *troll* are special.

```
⟨ Build the travel table 23 ⟩ +≡
  start[ppass] = q;
  if (q > &travels[travel_size] ∨ rem_count > rem_size) {
    printf("Oops,␣I'm␣broken!\n"); exit(−1);
  }
```

63. Data structures for objects. A fixed universe of objects was enumerated in the vocabulary section. Most of the objects can move or be moved from place to place; so we maintain linked lists of the objects at each location. The first object at location l is *first*[l], then comes *link*[*first*[l]], then *link*[*link*[*first*[l]]], etc., ending with 0 (which is the "object" called `NOTHING`).

Some of the objects are placed in *groups* of one or more objects. In such cases *base*[t] is the smallest object in the group containing object t. Objects that belong to groups are immovable; they always stay in the same location. Other objects have *base*[t] = `NOTHING` and they are free to leave one list and join another. For example, it turns out that the `KEYS` are movable, but the `SNAKE` is always in the Hall of the Mountain King; we set *base*[`KEYS`] = `NOTHING` and *base*[`SNAKE`] = `SNAKE`. Several groups, such as the `GRATE` and `GRATE_`, consist of two objects. This program supports operations on groups of more than two objects, but no such objects actually occur.

Each movable or base object t has a current property *prop*[t], which is initially -1 for treasures, otherwise initially 0. We change *prop*[t] to 0 when you first see treasure t; and property values often change further as the game progresses. For example, the `PLANT` can grow. When you see an object, we usually print a message that corresponds to its current property value. That message is the string *note*[*prop*[t] + *offset*[t]].

(Exception: When you first see the `RUG` or the `CHAIN`, its property value is set to 1, not 0. The reason for this hack is that you get maximum score only if the property values of all treasures are zero when you finish.)

Each object is in at most one list, *place*[t]. If you are carrying object t, the value of *place*[t] is *inhand*, which is negative. The special location *limbo* has value 0; we don't maintain a list *first*[*limbo*] for objects that have *place*[t] = *limbo*. Thus object t is in a list if and only if *place*[t] > 0. The global variable *holding* counts how many objects you are carrying.

One more array completes our set of data structures: Objects that appear in inventory reports have a name, *name*[t].

```
#define toting(t) (place[t] < 0)
```

⟨ Global variables 7 ⟩ +≡

```
object first[max_loc + 1];   /* the first object present at a location */
object link[max_obj + 1];   /* the next object present in the same location */
object base[max_obj + 2];   /* the smallest object in each object's group, if any */
int prop[max_obj + 1];   /* each object's current property value */
location place[max_obj + 1];   /* each object's current location */
char *name[max_obj + 1];   /* name of object for inventory listing */
char *note[100];   /* descriptions of object properties */
int offset[max_obj + 1];   /* where notes for each object start */
int holding;   /* how many objects have prop[t] < 0? */
int note_ptr = 0;   /* how many notes have we stored? */
```

64. Here then is a simple subroutine to place an object at a given location, when the object isn't presently in a list.

```
⟨ Subroutines 6 ⟩ +≡
  void drop ARGS((object, location));
  void drop(t, l)
      object t;
      location l;
  {
    if (toting(t)) holding--;
    place[t] = l;
    if (l < 0) holding++;
    else if (l > 0) {
      link[t] = first[l];
      first[l] = t;
    }
  }
```

65. Similarly, we need a subroutine to pick up an object.

```
#define move(t, l) { carry(t); drop(t, l); }
#define destroy(t) move(t, limbo)

⟨ Subroutines 6 ⟩ +≡
  void carry ARGS((object));
  void carry(t)
      object t;
  { register location l = place[t];
    if (l ≥ limbo) {
      place[t] = inhand;
      holding++;
      if (l > limbo) {
        register object r, s;
        for (r = 0, s = first[l]; s ≠ t; r = s, s = link[s]) ;
        if (r ≡ 0) first[l] = link[s];
        else link[r] = link[s];    /* remove t from list */
      }
    }
  }
```

66. The *is_at_loc* subroutine tests if a possibly multipart object is at a particular place. It uses the fact that multipart objects have consecutive values, and $base[max\_obj + 1] \equiv$ `NOTHING`.

```
⟨ Subroutines 6 ⟩ +≡
  boolean is_at_loc ARGS((object));
  boolean is_at_loc(t)
      object t;
  {
    register object tt;
    if (base[t] ≡ NOTHING) return place[t] ≡ loc;
    for (tt = t; base[tt] ≡ t; tt++)
      if (place[tt] ≡ loc) return true;
    return false;
  }
```

67. A few macros make it easy to get each object started.

#define $new\_obj(t, n, b, l)$
{ /* object t named n with base b starts at l */
$name[t] = n;$
$base[t] = b;$
$offset[t] = note\_ptr;$
$prop[t] = (is\_treasure(t) \ ? -1 : 0);$
$drop(t, l);$
}
#define $new\_note(n)$ $note[note\_ptr{++}] = n$

68. ⟨ Additional local registers 22 ⟩ +≡
register object t;

69. Object data. Now it's time to build the object structures just defined.

We put the objects into their initial locations backwards, that is, highest first; moreover, we place all two-part objects before placing the others. Then low-numbered objects will appear first in the list, and two-part objects will appear last.

Here are the two-part objects, which are mostly unnamed because you won't be picking them up.

⟨ Build the object tables 69 ⟩ ≡

```
new_obj(RUG_, 0, RUG, scan3);
new_obj(RUG, "Persian␣rug", RUG, scan1);
new_note("There␣is␣a␣Persian␣rug␣spread␣out␣on␣the␣floor!");
new_note("The␣dragon␣is␣sprawled␣out␣on␣a␣Persian␣rug!!");
new_obj(TROLL2_, 0, TROLL2, limbo);
new_obj(TROLL2, 0, TROLL2, limbo);
new_note("The␣troll␣is␣nowhere␣to␣be␣seen.");
new_obj(TROLL_, 0, TROLL, neside);
new_obj(TROLL, 0, TROLL, swside);
new_note("A␣burly␣troll␣stands␣by␣the␣bridge␣and␣insists␣you␣throw␣him␣a\n\
    treasure␣before␣you␣may␣cross.");
new_note("The␣troll␣steps␣out␣from␣beneath␣the␣bridge␣and␣blocks␣your␣way.");
new_note(0);
new_obj(BRIDGE_, 0, BRIDGE, neside);
new_obj(BRIDGE, 0, BRIDGE, swside);
new_note("A␣rickety␣wooden␣bridge␣extends␣across␣the␣chasm,␣vanishing␣into␣the\n\
    mist.␣␣A␣sign␣posted␣on␣the␣bridge␣reads,␣\"STOP!␣␣PAY␣TROLL!\"");
new_note("The␣wreckage␣of␣a␣bridge␣(and␣a␣dead␣bear)␣can␣be␣seen␣at␣the␣bottom\n\
    of␣the␣chasm.");
new_obj(DRAGON_, 0, DRAGON, scan3);
new_obj(DRAGON, 0, DRAGON, scan1);
new_note("A␣huge␣green␣fierce␣dragon␣bars␣the␣way!");
new_note("Congratulations!␣␣You␣have␣just␣vanquished␣a␣dragon␣with␣your␣bare\n\
    hands!␣(Unbelievable,␣isn't␣it?)");
new_note("The␣body␣of␣a␣huge␣green␣dead␣dragon␣is␣lying␣off␣to␣one␣side.");
new_obj(SHADOW_, 0, SHADOW, window);
new_obj(SHADOW, 0, SHADOW, windoe);
new_note("The␣shadowy␣figure␣seems␣to␣be␣trying␣to␣attract␣your␣attention.");
new_obj(PLANT2_, 0, PLANT2, e2pit);
new_obj(PLANT2, 0, PLANT2, w2pit);
new_note(0);
new_note("The␣top␣of␣a␣12-foot-tall␣beanstalk␣is␣poking␣out␣of␣the␣west␣pit.");
new_note("There␣is␣a␣huge␣beanstalk␣growing␣out␣of␣the␣west␣pit␣up␣to␣the␣hole.");
new_obj(CRYSTAL_, 0, CRYSTAL, wfiss);
new_obj(CRYSTAL, 0, CRYSTAL, efiss);
new_note(0);
new_note("A␣crystal␣bridge␣now␣spans␣the␣fissure.");
new_note("The␣crystal␣bridge␣has␣vanished!");
new_obj(TREADS_, 0, TREADS, emist);
new_obj(TREADS, 0, TREADS, spit);
new_note("Rough␣stone␣steps␣lead␣down␣the␣pit.");
new_note("Rough␣stone␣steps␣lead␣up␣the␣dome.");
new_obj(GRATE_, 0, GRATE, inside);
new_obj(GRATE, 0, GRATE, outside);
new_note("The␣grate␣is␣locked.");
new_note("The␣grate␣is␣open.");
```

new_obj(`MIRROR_`, 0, `MIRROR`, *limbo*); /* joins up with `MIRROR` later */

See also section 70.

This code is used in section 200.

70. And here are the one-place objects, some of which are immovable (because they are in a group of size one).

```
⟨ Build the object tables 69 ⟩ +≡
  new_obj(CHAIN, "Golden␣chain", CHAIN, barr);
  new_note("There␣is␣a␣golden␣chain␣lying␣in␣a␣heap␣on␣the␣floor!");
  new_note("The␣bear␣is␣locked␣to␣the␣wall␣with␣a␣golden␣chain!");
  new_note("There␣is␣a␣golden␣chain␣locked␣to␣the␣wall!");
  new_obj(SPICES, "Rare␣spices", 0, chamber);
  new_note("There␣are␣rare␣spices␣here!");
  new_obj(PEARL, "Glistening␣pearl", 0, limbo);
  new_note("Off␣to␣one␣side␣lies␣a␣glistening␣pearl!");
  new_obj(PYRAMID, "Platinum␣pyramid", 0, droom);
  new_note("There␣is␣a␣platinum␣pyramid␣here,␣8␣inches␣on␣a␣side!");
  new_obj(EMERALD, "Egg-sized␣emerald", 0, proom);
  new_note("There␣is␣an␣emerald␣here␣the␣size␣of␣a␣plover's␣egg!");
  new_obj(VASE, "Ming␣vase", 0, oriental);
  new_note("There␣is␣a␣delicate,␣precious,␣Ming␣vase␣here!");
  new_note("The␣vase␣is␣now␣resting,␣delicately,␣on␣a␣velvet␣pillow.");
  new_note("The␣floor␣is␣littered␣with␣worthless␣shards␣of␣pottery.");
  new_note("The␣Ming␣vase␣drops␣with␣a␣delicate␣crash.");
  new_obj(TRIDENT, "Jeweled␣trident", 0, falls);
  new_note("There␣is␣a␣jewel-encrusted␣trident␣here!");
  new_obj(EGGS, "Golden␣eggs", 0, giant);
  new_note("There␣is␣a␣large␣nest␣here,␣full␣of␣golden␣eggs!");
  new_note("The␣nest␣of␣golden␣eggs␣has␣vanished!");
  new_note("Done!");
  new_obj(CHEST, "Treasure␣chest", 0, limbo);
  new_note("The␣pirate's␣treasure␣chest␣is␣here!");
  new_obj(COINS, "Rare␣coins", 0, west);
  new_note("There␣are␣many␣coins␣here!");
  new_obj(JEWELS, "Precious␣jewelry", 0, south);
  new_note("There␣is␣precious␣jewelry␣here!");
  new_obj(SILVER, "Bars␣of␣silver", 0, ns);
  new_note("There␣are␣bars␣of␣silver␣here!");
  new_obj(DIAMONDS, "Several␣diamonds", 0, wfiss);
  new_note("There␣are␣diamonds␣here!");
  new_obj(GOLD, "Large␣gold␣nugget", 0, nugget);
  new_note("There␣is␣a␣large␣sparkling␣nugget␣of␣gold␣here!");
  new_obj(MOSS, 0, MOSS, soft);
  new_note(0);
  new_obj(BATTERIES, "Batteries", 0, limbo);
  new_note("There␣are␣fresh␣batteries␣here.");
  new_note("Some␣worn-out␣batteries␣have␣been␣discarded␣nearby.");
  new_obj(PONY, 0, PONY, pony);
  new_note("There␣is␣a␣massive␣vending␣machine␣here.␣␣The␣instructions␣on␣it␣read:\n\
      \"Drop␣coins␣here␣to␣receive␣fresh␣batteries.\"");
  new_obj(GEYSER, 0, GEYSER, view);
  new_note(0);
  new_obj(MESSAGE, 0, MESSAGE, limbo);
  new_note("There␣is␣a␣message␣scrawled␣in␣the␣dust␣in␣a␣flowery␣script,␣reading:\n\
      \"This␣is␣not␣the␣maze␣where␣the␣pirate␣hides␣his␣treasure␣chest.\"");
  new_obj(BEAR, 0, BEAR, barr);
```

```
new_note("There␣is␣a␣ferocious␣cave␣bear␣eying␣you␣from␣the␣far␣end␣of␣the␣room!");
new_note("There␣is␣a␣gentle␣cave␣bear␣sitting␣placidly␣in␣one␣corner.");
new_note("There␣is␣a␣contented-looking␣bear␣wandering␣about␣nearby.");
new_note(0);
new_obj(PIRATE, 0, PIRATE, limbo);
new_note(0);
new_obj(ART, 0, ART, oriental);
new_note(0);
new_obj(AXE, "Dwarf's␣axe", 0, limbo);
new_note("There␣is␣a␣little␣axe␣here.");
new_note("There␣is␣a␣little␣axe␣lying␣beside␣the␣bear.");
new_obj(STALACTITE, 0, STALACTITE, tite);
new_note(0);
new_obj(PLANT, 0, PLANT, wpit);
new_note("There␣is␣a␣tiny␣little␣plant␣in␣the␣pit,␣murmuring␣\"Water,␣water,␣...\"");
new_note("The␣plant␣spurts␣into␣furious␣growth␣for␣a␣few␣seconds.");
new_note("There␣is␣a␣12-foot-tall␣beanstalk␣stretching␣up␣out␣of␣the␣pit,\n\
    bellowing␣\"Water!!␣␣Water!!\"");
new_note("The␣plant␣grows␣explosively,␣almost␣filling␣the␣bottom␣of␣the␣pit.");
new_note("There␣is␣a␣gigantic␣beanstalk␣stretching␣all␣the␣way␣up␣to␣the␣hole.");
new_note("You've␣over-watered␣the␣plant!␣␣It's␣shriveling␣up!␣␣It's,␣it's...");
new_obj(MIRROR, 0, MIRROR, mirror);
new_note(0);
new_obj(OIL, "Oil␣in␣the␣bottle", 0, limbo);
new_obj(WATER, "Water␣in␣the␣bottle", 0, limbo);
new_obj(BOTTLE, "Small␣bottle", 0, house);
new_note("There␣is␣a␣bottle␣of␣water␣here.");
new_note("There␣is␣an␣empty␣bottle␣here.");
new_note("There␣is␣a␣bottle␣of␣oil␣here.");
new_obj(FOOD, "Tasty␣food", 0, house);
new_note("There␣is␣food␣here.");
new_obj(KNIFE, 0, 0, limbo);
new_obj(DWARF, 0, DWARF, limbo);
new_obj(MAG, "\"Spelunker␣Today\"", 0, ante);
new_note("There␣are␣a␣few␣recent␣issues␣of␣\"Spelunker␣Today\"␣magazine␣here.");
new_obj(OYSTER, "Giant␣oyster␣>GROAN!<", 0, limbo);
new_note("There␣is␣an␣enormous␣oyster␣here␣with␣its␣shell␣tightly␣closed.");
new_note("Interesting.␣␣There␣seems␣to␣be␣something␣written␣on␣the␣underside␣of\n\
    the␣oyster.");
new_obj(CLAM, "Giant␣clam␣>GRUNT!<", 0, shell);
new_note("There␣is␣an␣enormous␣clam␣here␣with␣its␣shell␣tightly␣closed.");
new_obj(TABLET, 0, TABLET, droom);
new_note("A␣massive␣stone␣tablet␣embedded␣in␣the␣wall␣reads:\n\
    \"CONGRATULATIONS␣ON␣BRINGING␣LIGHT␣INTO␣THE␣DARK-ROOM!\"");
new_obj(SNAKE, 0, SNAKE, hmk);
new_note("A␣huge␣green␣fierce␣snake␣bars␣the␣way!");
new_note(0);
new_obj(PILLOW, "Velvet␣pillow", 0, soft);
new_note("A␣small␣velvet␣pillow␣lies␣on␣the␣floor.");
new_obj(DOOR, 0, DOOR, immense);
new_note("The␣way␣north␣is␣barred␣by␣a␣massive,␣rusty,␣iron␣door.");
new_note("The␣way␣north␣leads␣through␣a␣massive,␣rusty,␣iron␣door.");
```

```
new_obj(BIRD, "Little␣bird␣in␣cage", 0, bird);
new_note("A␣cheerful␣little␣bird␣is␣sitting␣here␣singing.");
new_note("There␣is␣a␣little␣bird␣in␣the␣cage.");
new_obj(ROD2, "Black␣rod", 0, limbo);
new_note("A␣three-foot␣black␣rod␣with␣a␣rusty␣mark␣on␣an␣end␣lies␣nearby.");
new_obj(ROD, "Black␣rod", 0, debris);
new_note("A␣three-foot␣black␣rod␣with␣a␣rusty␣star␣on␣an␣end␣lies␣nearby.");
new_obj(CAGE, "Wicker␣cage", 0, cobbles);
new_note("There␣is␣a␣small␣wicker␣cage␣discarded␣nearby.");
new_obj(LAMP, "Brass␣lantern", 0, house);
new_note("There␣is␣a␣shiny␣brass␣lamp␣nearby.");
new_note("There␣is␣a␣lamp␣shining␣nearby.");
new_obj(KEYS, "Set␣of␣keys", 0, house);
new_note("There␣are␣some␣keys␣on␣the␣ground␣here.");
```

71. Low-level input. Sometimes we need to ask you a question, for which the answer is either yes or no. The subroutine *yes*(*q*, *y*, *n*) prints *q*, waits for you to answer, and then prints *y* or *n* depending on your answer. It returns a nonzero value if your answer was affirmative.

```
⟨ Subroutines 6 ⟩ +≡
  boolean yes ARGS((char *, char *, char *));
  boolean yes(q, y, n)
      char *q, *y, *n;
  {
    while (1) {
      printf("%s\n**␣", q); fflush(stdout);
      fgets(buffer, buf_size, stdin);
      if (tolower(*buffer) ≡ 'y') {
        if (y) printf("%s\n", y); return true;
      }
      else if (tolower(*buffer) ≡ 'n') {
        if (n) printf("%s\n", n); return false;
      }
      else printf("␣Please␣answer␣Yes␣or␣No.\n");
    }
  }
```

72. The only other kind of input is almost as simple. You are supposed to tell us what to do next in your adventure, by typing one- or two-word commands. We put the first word in *word1* and the (possibly null) second word in *word2*. Words are separated by white space; otherwise white space is ignored.

```
⟨ Subroutines 6 ⟩ +≡
  void listen ARGS((void));
  void listen() {
    register char *p, *q;
    while (1) {
      printf("*␣"); fflush(stdout);
      fgets(buffer, buf_size, stdin);
      for (p = buffer; isspace(*p); p++) ;
      if (*p ≡ 0) {
        printf("␣Tell␣me␣to␣do␣something.\n"); continue;
      }
      for (q = word1; *p; p++, q++) {
        if (isspace(*p)) break;
        *q = tolower(*p);
      }
      *q = '\0';    /* end of word1 */
      for (p++; isspace(*p); p++) ;
      if (*p ≡ 0) {
        *word2 = '\0'; return;
      }
      for (q = word2; *p; p++, q++) {
        if (isspace(*p)) break;
        *q = tolower(*p);
      }
      *q = '\0';    /* end of word2 */
      for (p++; isspace(*p); p++) ;
      if (*p ≡ 0) return;
      printf("␣Please␣stick␣to␣1-␣and␣2-word␣commands.\n");
    }
  }
```

73. A 20-character buffer would probably be big enough, but what the heck.

```
#define buf_size 72

⟨ Global variables 7 ⟩ +≡
  char buffer[buf_size];    /* your input goes here */
  char word1[buf_size], word2[buf_size];    /* and then we snarf it to here */
```

74. The main control loop. Now we've got enough low-level mechanisms in place to start thinking of the program from the top down, and to specify the high-level control.

A global variable *loc* represents where you currently live in the simulated cave. Another variable *newloc* represents where you will go next, unless something like a dwarf blocks you. We also keep track of *oldloc* (the previous value of *loc*) and *oldoldloc* (the previous previous value), for use when you ask to '`go back`'.

#define *here*(t) (*toting*(t) $\vee$ *place*[t] $\equiv$ *loc*) /∗ is object t present? ∗/
#define *water_here* ((*flags*[*loc*] & (*liquid* + *oil*)) $\equiv$ *liquid*)
#define *oil_here* ((*flags*[*loc*] & (*liquid* + *oil*)) $\equiv$ *liquid* + *oil*)
#define *no_liquid_here* ((*flags*[*loc*] & *liquid*) $\equiv$ 0)

⟨ Global variables 7 ⟩ +≡
  location *oldoldloc*, *oldloc*, *loc*, *newloc*; /∗ recent and future locations ∗/

75. Here is our overall strategy for administering the game. It is understood that the program might **goto** *quit* from within any of the subsections named here, even though the section names don't mention this explicitly. For example, while checking for interference we might find out that time has run out, or that a dwarf has killed you and no more reincarnations are possible.

The execution consists of two nested loops: There are "minor cycles" inside of "major cycles." Actions define minor cycles in which you stay in the same place and we tell you the result of your action. Motions define major cycles in which you move and we tell you what you can see at the new place.

⟨ Simulate an adventure, going to *quit* when finished 75 ⟩ ≡
  while (1) {
    ⟨ Check for interference with the proposed move to *newloc* 153 ⟩;
    loc = *newloc*; /∗ hey, we actually moved you ∗/
    ⟨ Possibly move dwarves and the pirate 161 ⟩;
  commence: ⟨ Report the current state 86 ⟩;
    while (1) {
      ⟨ Get user input; **goto** *try_move* if motion is requested 76 ⟩;
      ⟨ Perform an action in the current place 79 ⟩;
    }
  try_move: ⟨ Handle special motion words 140 ⟩;
    oldoldloc = *oldloc*;
    oldloc = *loc*;
  go_for_it: ⟨ Determine the next location, *newloc* 146 ⟩;
  }

This code is used in section 2.

76. Our main task in the simulation loop is to parse your input. Depending on the kind of command you give, the following section of the program will exit in one of four ways:

- **goto** *try_move* with *mot* set to a desired motion.
- **goto** *transitive* with *verb* set to a desired action and *obj* set to the object of that motion.
- **goto** *intransitive* with *verb* set to a desired action and *obj* = `NOTHING`; no object has been specified.
- **goto** *speakit* with *hash_table*[*k*].*meaning* the index of a message for a vocabulary word of *message_type*.

Sometimes we have to ask you to complete an ambiguous command before we know both a verb and its object. In most cases the words can be in either order; for example, `take rod` is equivalent to `rod take`. A motion word overrides a previously given action or object.

Lots of special cases make the program a bit messy. For example, if the verb is `say`, we don't want to look up the object in our vocabulary; we simply want to "say" it.

⟨ Get user input; **goto** *try_move* if motion is requested 76 ⟩ ≡
  verb = *oldverb* = `ABSTAIN`;
  oldobj = *obj*;
  obj = `NOTHING`;
cycle: ⟨ Check if a hint applies, and give it if requested 195 ⟩;
  ⟨ Make special adjustments before looking at new input 85 ⟩;
  listen( );
pre_parse: *turns*++;
  ⟨ Handle special cases of input 82 ⟩;
  ⟨ Check the clocks and the lamp 178 ⟩;
  ⟨ Handle additional special cases of input 83 ⟩;
parse: ⟨ Give advice about going `WEST` 80 ⟩;
  ⟨ Look at *word1* and exit to the right place if it completes a command 78 ⟩;
shift: *strcpy*(*word1*, *word2*); ∗*word2* = `'\0'`; **goto** *parse*;

This code is used in section 75.

77. ⟨ Global variables 7 ⟩ +≡
  motion *mot*; /∗ currently specified motion, if any ∗/
  action *verb*; /∗ currently specified action, if any ∗/
  action *oldverb*; /∗ *verb* before it was changed ∗/
  object *obj*; /∗ currently specified object, if any ∗/
  object *oldobj*; /∗ former value of *obj* ∗/
  wordtype *command_type*; /∗ type of word found in hash table ∗/
  int *turns*; /∗ how many times we've read your commands ∗/

78. The *try_motion* macro is often used to end a major cycle.

```
#define try_motion(m) { mot = m; goto try_move; }
#define stay_put try_motion(NOWHERE)
⟨ Look at word1 and exit to the right place if it completes a command 78 ⟩ ≡
  k = lookup(word1);
  if (k < 0) {    /* Gee, I don't understand */
    printf("Sorry, I don't know the word \"%s\".\n", word1); goto cycle;
  }
branch: command_type = hash_table[k].word_type;
  switch (command_type) {
  case motion_type: try_motion(hash_table[k].meaning);
  case object_type: obj = hash_table[k].meaning;
    ⟨ Make sure obj is meaningful at the current location 90 ⟩;
    if (*word2) break;
    if (verb) goto transitive;
    printf("What do you want to do with the %s?\n", word1); goto cycle;
  case action_type: verb = hash_table[k].meaning;
    if (verb ≡ SAY) obj = *word2;
    else if (*word2) break;
    if (obj) goto transitive; else goto intransitive;
  case message_type: goto speakit;
  }
```

This code is used in section 76.

79. Here is the multiway branch where many kinds of actions can be launched.

If a verb can only be transitive, but no object has been given, we must go back and ask for an object.

If a verb can only be intransitive, but an object has been given, we issue the default message for that verb and start over.

The variable k, initially zero, is used to count various things in several of the action routines.

The *report* macro is often used to end a minor cycle.

```
#define report(m) { printf("%s\n", m); continue; }
#define default_to(v) report(default_msg[v])
#define change_to(v) { oldverb = verb; verb = v; goto transitive; }
⟨ Perform an action in the current place 79 ⟩ ≡
intransitive: k = 0;
  switch (verb) {
  case GO: case RELAX: goto report_default;
  case ON: case OFF: case POUR: case FILL: case DRINK: case BLAST: case KILL: goto transitive;
    ⟨ Handle cases of intransitive verbs and continue 92 ⟩;
  default: goto get_object;
  }
transitive: k = 0;
  switch (verb) {
    ⟨ Handle cases of transitive verbs and continue 97 ⟩;
  default: goto report_default;
  }
speakit: report(message[hash_table[k].meaning]);
report_default: if (default_msg[verb]) report(default_msg[verb]) else continue;
get_object: printf("%s␣what?\n", word1); goto cycle;
cant_see_it: if ((verb ≡ FIND ∨ verb ≡ INVENTORY) ∧ *word2 ≡ '\0') goto transitive;
  printf("I␣see␣no␣%s␣here.\n", word1); continue;
```

This code is used in section 75.

80. Here's a freely offered hint that may save you typing.

```
⟨ Give advice about going WEST 80 ⟩ ≡
  if (streq(word1, "west")) {
    west_count++;
    if (west_count ≡ 10) printf("␣If␣you␣prefer,␣simply␣type␣W␣rather␣than␣WEST.\n");
  }
```

This code is used in section 76.

81. ⟨ Global variables 7 ⟩ +≡

```
  int west_count;    /* how many times have we parsed the word 'west'? */
```

82. Maybe you said '`say`' and we said '`say what?`' and you replied with two things to say. Then we assume you don't really want us to say anything.

```
⟨ Handle special cases of input 82 ⟩ ≡
  if (verb ≡ SAY) {
    if (*word2) verb = ABSTAIN; else goto transitive;
  }
```

See also section 138.

This code is used in section 76.

83. The verb '`enter`' is listed in our vocabulary as a motion rather than an action. Here we deal with cases where you try to use it as an action. Notice that '`H2O`' is not a synonym for '`water`' in this context.

```
⟨ Handle additional special cases of input 83 ⟩ ≡
  if (streq(word1, "enter")) {
    if (streq(word2, "water") ∨ streq(word2, "strea")) {
      if (water_here) report("Your␣feet␣are␣now␣wet.");
      default_to(GO);
    }
    else if (∗word2) goto shift;
  }
```

See also section 105.

This code is used in section 76.

84. Cavers can become cadavers if they don't have light. We keep a variable *was_dark* to remember how dark things were when you gave your last command.

```
#define dark ((flags[loc] & lighted) ≡ 0 ∧ (prop[LAMP] ≡ 0 ∨ ¬here(LAMP)))
⟨ Global variables 7 ⟩ +≡
  boolean was_dark;    /∗ you've recently been in the dark ∗/
```

85. ⟨ Make special adjustments before looking at new input 85 ⟩ ≡

```
  was_dark = dark;
```

See also sections 158, 169, and 182.

This code is used in section 76.

86. After moving to *newloc*, we act as your eyes. We print the long description of *newloc* if you haven't been there before; but when you return to a previously seen place, we often use a short form. The long form is used every 5th time, unless you say '`brief`', in which case we use the shortest form we know. You can always ask for the long form by saying '`look`'.

```
⟨ Report the current state 86 ⟩ ≡
  if (loc ≡ limbo) goto death;
  if (dark ∧ ¬forced_move(loc)) {
    if (was_dark ∧ pct(35)) goto pitch_dark;
    p = pitch_dark_msg;
  }
  else if (short_desc[loc] ≡ 0 ∨ visits[loc] % interval ≡ 0) p = long_desc[loc];
  else p = short_desc[loc];
  if (toting(BEAR)) printf("You␣are␣being␣followed␣by␣a␣very␣large,␣tame␣bear.\n");
  printf("\n%s\n", p);
  if (forced_move(loc)) goto try_move;
  ⟨ Give optional plugh hint 157 ⟩;
  if (¬dark) ⟨ Describe the objects at this location 88 ⟩;
```

This code is used in section 75.

87. ⟨ Global variables 7 ⟩ +≡

```
  int interval = 5;    /∗ will change to 10000 if you want us to be BRIEF ∗/
  char pitch_dark_msg[] =
      "It␣is␣now␣pitch␣dark.␣␣If␣you␣proceed␣you␣will␣most␣likely␣fall␣into␣a␣pit.";
```

88. If TREADS are present but you have a heavy load, we don't describe them. The treads never actually get property value 1; we use the *note* for property 1 only when they are seen from above.

The global variable *tally* counts the number of treasures you haven't seen. Another variable, *lost_treasures*, counts those you never will see.

```
⟨ Describe the objects at this location 88 ⟩ ≡
  { register object tt;
    visits[loc]++;
    for (t = first[loc]; t; t = link[t]) {
      tt = (base[t] ? base[t] : t);
      if (prop[tt] < 0) {      /* you've spotted a treasure */
        if (closed) continue;      /* no automatic prop change after hours */
        prop[tt] = (tt ≡ RUG ∨ tt ≡ CHAIN);      /* initialize the property value */
        tally−−;
        ⟨ Zap the lamp if the remaining treasures are too elusive 183 ⟩;
      }
      if (tt ≡ TREADS ∧ toting(GOLD)) continue;
      p = note[prop[tt] + offset[tt] + (tt ≡ TREADS ∧ loc ≡ emist)];
      if (p) printf("%s\n", p);
    }
  }
```

This code is used in section 86.

89. ⟨ Global variables 7 ⟩ +≡

```
  int tally = 15;      /* treasures awaiting you */
  int lost_treasures;      /* treasures that you won't find */
```

90. When you specify an object, it must be at the current location, unless the verb is already known to be FIND or INVENTORY. A few other special cases also are permitted; for example, water and oil are funny, since they are never actually dropped at any location, but they might be present inside the bottle or as a feature of the location.

```
#define object_in_bottle ((obj ≡ WATER ∧ prop[BOTTLE] ≡ 0) ∨ (obj ≡ OIL ∧ prop[BOTTLE] ≡ 2))

⟨ Make sure obj is meaningful at the current location 90 ⟩ ≡
  if (¬toting(obj) ∧ ¬is_at_loc(obj))
    switch (obj) {
    case GRATE: ⟨ If GRATE is actually a motion word, move to it 91 ⟩;
      goto cant_see_it;
    case DWARF: if (dflag ≥ 2 ∧ dwarf()) break; else goto cant_see_it;
    case PLANT: if (is_at_loc(PLANT2) ∧ prop[PLANT2]) {
        obj = PLANT2; break;
      }
      else goto cant_see_it;
    case KNIFE: if (loc ≠ knife_loc) goto cant_see_it;
      knife_loc = −1;
      report("The␣dwarves'␣knives␣vanish␣as␣they␣strike␣the␣walls␣of␣the␣cave.");
    case ROD: if (¬here(ROD2)) goto cant_see_it;
      obj = ROD2; break;
    case WATER: case OIL: if (here(BOTTLE) ∧ object_in_bottle) break;
      if ((obj ≡ WATER ∧ water_here) ∨ (obj ≡ OIL ∧ oil_here)) break;
    default: goto cant_see_it;
    }
```

This code is used in section 78.

91. Henning Makholm has pointed out that the logic here makes `GRATE` a motion word regardless of the verb. For example, you can get to the grate by saying '`wave grate`' from the *road* or the *valley* (but curiously not from the *slit*).

⟨ If `GRATE` is actually a motion word, move to it 91 ⟩ ≡

```
if (loc < min_lower_loc)
  switch (loc) {
  case road: case valley: case slit: try_motion(DEPRESSION);
  case cobbles: case debris: case awk: case bird: case spit: try_motion(ENTRANCE);
  default: break;
  }
```

This code is used in section 90.

92. Simple verbs. Let's get experience implementing the actions by dispensing with the easy cases first.

First there are several "intransitive" verbs that reduce to transitive when we identify an appropriate object. For example, '`take`' makes sense by itself if there's only one possible thing to take.

⟨ Handle cases of intransitive verbs and **continue** 92 ⟩ ≡

```
case TAKE: if (first[loc] ≡ 0 ∨ link[first[loc]] ∨ dwarf( )) goto get_object;
  obj = first[loc]; goto transitive;
case EAT: if (¬here(FOOD)) goto get_object;
  obj = FOOD; goto transitive;
```

See also sections 93, 94, 95, and 136.

This code is used in section 79.

93. Only the objects `GRATE`, `DOOR`, `CLAM`/`OYSTER`, and `CHAIN` can be opened or closed. And only a few objects can be read.

⟨ Handle cases of intransitive verbs and **continue** 92 ⟩ +≡

```
case OPEN: case CLOSE: if (place[GRATE] ≡ loc ∨ place[GRATE_] ≡ loc) obj = GRATE;
  else if (place[DOOR] ≡ loc) obj = DOOR;
  else if (here(CLAM)) obj = CLAM;
  else if (here(OYSTER)) obj = OYSTER;
  if (here(CHAIN)) {
    if (obj) goto get_object; else obj = CHAIN;
  }
  if (obj) goto transitive;
  report("There␣is␣nothing␣here␣with␣a␣lock!");
case READ: if (dark) goto get_object;      /* can't read in the dark */
  if (here(MAG)) obj = MAG;
  if (here(TABLET)) {
    if (obj) goto get_object; else obj = TABLET;
  }
  if (here(MESSAGE)) {
    if (obj) goto get_object; else obj = MESSAGE;
  }
  if (closed ∧ toting(OYSTER)) obj = OYSTER;
  if (obj) goto transitive; else goto get_object;
```

94. A request for an inventory is pretty simple too.

⟨ Handle cases of intransitive verbs and **continue** 92 ⟩ +≡

```
case INVENTORY:
  for (t = 1; t ≤ max_obj; t++)
    if (toting(t) ∧ (base[t] ≡ NOTHING ∨ base[t] ≡ t) ∧ t ≠ BEAR) {
      if (k ≡ 0) k = 1, printf("You␣are␣currently␣holding␣the␣following:\n");
      printf("␣%s\n", name[t]);
    }
  if (toting(BEAR)) report("You␣are␣being␣followed␣by␣a␣very␣large,␣tame␣bear.");
  if (k ≡ 0) report("You're␣not␣carrying␣anything.");
  continue;
```

95. Here are other requests about the mechanics of the game.

```
⟨Handle cases of intransitive verbs and continue 92⟩ +≡
case BRIEF: interval = 10000;
  look_count = 3;
  report("Okay,␣from␣now␣on␣I'll␣only␣describe␣a␣place␣in␣full␣the␣first␣time\n\
      you␣come␣to␣it.␣␣To␣get␣the␣full␣description,␣say␣\"LOOK\".");
case SCORE: printf("If␣you␣were␣to␣quit␣now,␣you␣would␣score␣%d\nout␣of␣a␣possible␣%d.\n",
      score() − 4, max_score);
  if (¬yes("Do␣you␣indeed␣wish␣to␣quit␣now?", ok, ok)) continue;
  goto give_up;
case QUIT: if (¬yes("Do␣you␣really␣wish␣to␣quit␣now?", ok, ok)) continue;
give_up: gave_up = true; goto quit;
```

96.

```
⟨Global variables 7⟩ +≡
  boolean gave_up;    /* did you quit while you were alive? */
```

97. The SAY routine is just an echo unless you say a magic word.

```
⟨Handle cases of transitive verbs and continue 97⟩ ≡
case SAY: if (*word2) strcpy(word1, word2);
  k = lookup(word1);
  switch (hash_table[k].meaning) {
  case XYZZY: case PLUGH: case PLOVER: case FEEFIE: *word2 = '\0'; obj = NOTHING; goto branch;
  default: printf("Okay,␣\"%s\".\n", word1); continue;
  }
```

See also sections 98, 99, 100, 101, 102, 106, 107, 110, 112, 117, 122, 125, 129, 130, and 135.

This code is used in section 79.

98. Hungry?

```
⟨Handle cases of transitive verbs and continue 97⟩ +≡
case EAT:
  switch (obj) {
  case FOOD: destroy(FOOD);
    report("Thank␣you,␣it␣was␣delicious!");
  case BIRD: case SNAKE: case CLAM: case OYSTER: case DWARF: case DRAGON: case TROLL: case BEAR:
    report("I␣think␣I␣just␣lost␣my␣appetite.");
  default: goto report_default;
  }
```

99. Waving to the shadowy figure has no effect; but you might wave a rod at the fissure. Blasting has no effect unless you've got dynamite, which is a neat trick! Rubbing yields only snide remarks.

```
⟨Handle cases of transitive verbs and continue 97⟩ +≡
case WAVE: if (obj ≠ ROD ∨ (loc ≠ efiss ∧ loc ≠ wfiss) ∨
          ¬toting(obj) ∨ closing) {
      if (toting(obj) ∨ (obj ≡ ROD ∧ toting(ROD2))) goto report_default;
      default_to(DROP);
  }
  prop[CRYSTAL] = 1 − prop[CRYSTAL];
  report(note[offset[CRYSTAL] + 2 − prop[CRYSTAL]]);
case BLAST: if (closed ∧ prop[ROD2] ≥ 0) {
      bonus = (here(ROD2) ? 25 : loc ≡ neend ? 30 : 45);
      printf("%s\n", message[bonus/5]); goto quit;
  }
  else goto report_default;
case RUB: if (obj ≡ LAMP) goto report_default;
  default_to(TOSS);
```

100. If asked to find an object that isn't visible, we give a caveat.

```
⟨Handle cases of transitive verbs and continue 97⟩ +≡
case FIND: case INVENTORY: if (toting(obj)) default_to(TAKE);
  if (closed) report("I␣daresay␣whatever␣you␣want␣is␣around␣here␣somewhere.");
  if (is_at_loc(obj) ∨ (object_in_bottle ∧ place[BOTTLE] ≡ loc) ∨
          (obj ≡ WATER ∧ water_here) ∨ (obj ≡ OIL ∧ oil_here) ∨
          (obj ≡ DWARF ∧ dwarf())) report("I␣believe␣what␣you␣want␣is␣right␣here␣with␣you.");
  goto report_default;
```

101. Breaking and/or waking have no effect until the cave is closed, except of course that you might break the vase. The dwarves like mirrors and hate being awakened.

```
⟨Handle cases of transitive verbs and continue 97⟩ +≡
case BREAK: if (obj ≡ VASE ∧ prop[VASE] ≡ 0) {
      if (toting(VASE)) drop(VASE, loc);      /* crash */
      printf("You␣have␣taken␣the␣vase␣and␣hurled␣it␣delicately␣to␣the␣ground.\n");
  smash: prop[VASE] = 2; base[VASE] = VASE;      /* it's no longer movable */
      continue;
  }
  else if (obj ≠ MIRROR) goto report_default;
  if (closed) {
      printf("You␣strike␣the␣mirror␣a␣resounding␣blow,␣whereupon␣it␣shatters␣into␣a\n\
          myriad␣tiny␣fragments.");
      goto dwarves_upset;
  }
  report("It␣is␣too␣far␣up␣for␣you␣to␣reach.");
case WAKE: if (closed ∧ obj ≡ DWARF) {
      printf("You␣prod␣the␣nearest␣dwarf,␣who␣wakes␣up␣grumpily,␣takes␣one␣look␣at\n\
          you,␣curses,␣and␣grabs␣for␣his␣axe.\n");
      goto dwarves_upset;
  }
  else goto report_default;
```

102. Here we deal with lighting or extinguishing the lamp. The variable *limit* tells how much juice you've got left.

⟨ Handle cases of transitive verbs and **continue** 97 ⟩ +≡

```
case ON: if (¬here(LAMP)) goto report_default;
  if (limit < 0) report("Your␣lamp␣has␣run␣out␣of␣power.");
  prop[LAMP] = 1;
  printf("Your␣lamp␣is␣now␣on.\n");
  if (was_dark) goto commence;
  continue;
case OFF: if (¬here(LAMP)) goto report_default;
  prop[LAMP] = 0;
  printf("Your␣lamp␣is␣now␣off.\n");
  if (dark) printf("%s\n", pitch_dark_msg);
  continue;
```

103. ⟨ Global variables 7 ⟩ +≡

```
  int limit;    /* countdown till darkness */
```

104. Liquid assets. Readers of this program will already have noticed that the `BOTTLE` is a rather complicated object, since it can be empty or filled with either water or oil. Let's consider now the main actions that involve liquids.

When you are carrying a bottle full of water, *place*[`WATER`] will be *inhand*; hence both *toting*(`WATER`) and *toting*(`BOTTLE`) are true. A similar remark applies to a bottle full of oil.

The value of *prop*[`BOTTLE`] is 0 if it holds water, 2 if it holds oil, otherwise either 1 or −2. (The value −2 is used after closing the cave.)

```
#define bottle_empty (prop[BOTTLE] ≡ 1 ∨ prop[BOTTLE] < 0)
```

105. Sometimes '`water`' and '`oil`' are used as verbs.

```
⟨Handle additional special cases of input 83⟩ +≡
  if ((streq(word1, "water") ∨ streq(word1, "oil")) ∧
        (streq(word2, "plant") ∨ streq(word2, "door")) ∧
        (loc = place[hash_table[lookup(word2)].meaning])) strcpy(word2, "pour");
```

106. If you ask simply to drink, we assume that you want water. If there's water in the bottle, you drink that; otherwise you must be at a water location.

```
⟨Handle cases of transitive verbs and continue 97⟩ +≡
case DRINK: if (obj ≡ NOTHING) {
      if (¬water_here ∧ ¬(here(BOTTLE) ∧ prop[BOTTLE] ≡ 0)) goto get_object;
  }
  else if (obj ≠ WATER) default_to(EAT);
  if (¬(here(BOTTLE) ∧ prop[BOTTLE] ≡ 0)) goto report_default;
  prop[BOTTLE] = 1; place[WATER] = limbo;
  report("The␣bottle␣of␣water␣is␣now␣empty.");
```

107. Pouring involves liquid from the bottle.

```
⟨Handle cases of transitive verbs and continue 97⟩ +≡
case POUR: if (obj ≡ NOTHING ∨ obj ≡ BOTTLE) {
      obj = (prop[BOTTLE] ≡ 0 ? WATER : prop[BOTTLE] ≡ 2 ? OIL : 0);
      if (obj ≡ NOTHING) goto get_object;
  }
  if (¬toting(obj)) goto report_default;
  if (obj ≠ WATER ∧ obj ≠ OIL) report("You␣can't␣pour␣that.");
  prop[BOTTLE] = 1; place[obj] = limbo;
  if (loc ≡ place[PLANT]) ⟨Try to water the plant 108⟩;
  if (loc ≡ place[DOOR]) ⟨Pour water or oil on the door 109⟩;
  goto report_default;
```

108. ⟨Try to water the plant 108⟩ ≡

```
  {
      if (obj ≠ WATER)
        report("The␣plant␣indignantly␣shakes␣the␣oil␣off␣its␣leaves␣and␣asks,␣\"Water?\"");
      printf("%s\n", note[prop[PLANT] + 1 + offset[PLANT]]);
      prop[PLANT] += 2; if (prop[PLANT] > 4) prop[PLANT] = 0;
      prop[PLANT2] = prop[PLANT] ≫ 1;
      stay_put;
  }
```

This code is used in section 107.

109. ⟨ Pour water or oil on the door 109 ⟩ ≡

```
switch (obj) {
case WATER: prop[DOOR] = 0;
  report("The␣hinges␣are␣quite␣thoroughly␣rusted␣now␣and␣won't␣budge.");
case OIL: prop[DOOR] = 1;
  report("The␣oil␣has␣freed␣up␣the␣hinges␣so␣that␣the␣door␣will␣now␣open.");
}
```

This code is used in section 107.

110. You can fill the bottle only when it's empty and liquid is available. You can't fill the lamp with oil.

⟨ Handle cases of transitive verbs and **continue** 97 ⟩ +≡

```
case FILL: if (obj ≡ VASE) ⟨ Try to fill the vase 111 ⟩;
  if (¬here(BOTTLE)) {
    if (obj ≡ NOTHING) goto get_object; else goto report_default;
  }
  else if (obj ≠ NOTHING ∧ obj ≠ BOTTLE) goto report_default;
  if (¬bottle_empty) report("Your␣bottle␣is␣already␣full.");
  if (no_liquid_here) report("There␣is␣nothing␣here␣with␣which␣to␣fill␣the␣bottle.");
  prop[BOTTLE] = flags[loc] & oil;
  if (toting(BOTTLE)) place[prop[BOTTLE] ? OIL : WATER] = inhand;
  printf("Your␣bottle␣is␣now␣full␣of␣%s.\n", prop[BOTTLE] ? "oil" : "water");
  continue;
```

111. Filling the vase is a nasty business.

⟨ Try to fill the vase 111 ⟩ ≡

```
{
  if (no_liquid_here) report("There␣is␣nothing␣here␣with␣which␣to␣fill␣the␣vase.\n");
  if (¬toting(VASE)) report(default_msg[DROP]);
  printf("The␣sudden␣change␣in␣temperature␣has␣delicately␣shattered␣the␣vase.\n");
  goto smash;
}
```

This code is used in section 110.

112. Picking up a liquid depends, of course, on the status of the bottle. Other objects need special handling, too, because of various side effects and the fact that we can't take bird and cage separately when the bird is in the cage.

```
⟨ Handle cases of transitive verbs and continue 97 ⟩ +≡
case TAKE: if (toting(obj)) goto report_default;    /* already carrying it */
  if (base[obj]) {    /* it is immovable */
    if (obj ≡ CHAIN ∧ prop[BEAR]) report("The␣chain␣is␣still␣locked.");
    if (obj ≡ BEAR ∧ prop[BEAR] ≡ 1) report("The␣bear␣is␣still␣chained␣to␣the␣wall.");
    if (obj ≡ PLANT ∧ prop[PLANT] ≤ 0)
      report("The␣plant␣has␣exceptionally␣deep␣roots␣and␣cannot␣be␣pulled␣free.");
    report("You␣can't␣be␣serious!");
  }
  if (obj ≡ WATER ∨ obj ≡ OIL) ⟨ Check special cases for taking a liquid 113 ⟩;
  if (holding ≥ 7)
    report("You␣can't␣carry␣anything␣more.␣␣You'll␣have␣to␣drop␣something␣first.");
  if (obj ≡ BIRD ∧ prop[BIRD] ≡ 0) ⟨ Check special cases for taking a bird 114 ⟩;
  if (obj ≡ BIRD ∨ (obj ≡ CAGE ∧ prop[BIRD])) carry(BIRD + CAGE − obj);
  carry(obj);
  if (obj ≡ BOTTLE ∧ ¬bottle_empty) place[prop[BOTTLE] ? OIL : WATER] = inhand;
  default_to(RELAX);    /* OK, we've taken it */
```

113. ⟨ Check special cases for taking a liquid 113 ⟩ ≡

```
  if (here(BOTTLE) ∧ object_in_bottle) obj = BOTTLE;
  else {
    obj = BOTTLE;
    if (toting(BOTTLE)) change_to(FILL);
    report("You␣have␣nothing␣in␣which␣to␣carry␣it.");
  }
```

This code is used in section 112.

114. ⟨ Check special cases for taking a bird 114 ⟩ ≡

```
  {
    if (toting(ROD))
      report("The␣bird␣was␣unafraid␣when␣you␣entered,␣but␣as␣you␣approach␣it␣becomes\n\
          disturbed␣and␣you␣cannot␣catch␣it.");
    if (toting(CAGE)) prop[BIRD] = 1;
    else report("You␣can␣catch␣the␣bird,␣but␣you␣cannot␣carry␣it.");
  }
```

This code is used in section 112.

115. Similarly, when dropping the bottle we must drop also its liquid contents, if any.

```
⟨ Check special cases for dropping a liquid 115 ⟩ ≡
  if (object_in_bottle) obj = BOTTLE;
  if (obj ≡ BOTTLE ∧ ¬bottle_empty) place[prop[BOTTLE] ? OIL : WATER] = limbo;
```

This code is used in section 117.

116. The other actions. Now that we understand how to write action routines, we're ready to complete the set.

117. Dropping an object has special cases for the bird (which might attack the snake or the dragon), the cage, the vase, etc. The verb THROW also reduces to DROP for most objects.

```
⟨ Handle cases of transitive verbs and continue 97 ⟩ +≡
case DROP: if (obj ≡ ROD ∧ toting(ROD2) ∧ ¬toting(ROD)) obj = ROD2;
  if (¬toting(obj)) goto report_default;
  if (obj ≡ COINS ∧ here(PONY)) ⟨ Put coins in the vending machine 118 ⟩;
  if (obj ≡ BIRD) ⟨ Check special cases for dropping the bird 120 ⟩;
  if (obj ≡ VASE ∧ loc ≠ soft) ⟨ Check special cases for dropping the vase 121 ⟩;
  if (obj ≡ BEAR ∧ is_at_loc(TROLL)) ⟨ Chase the troll away 119 ⟩;
  ⟨ Check special cases for dropping a liquid 115 ⟩;
  if (obj ≡ BIRD) prop[BIRD] = 0;
  else if (obj ≡ CAGE ∧ prop[BIRD]) drop(BIRD, loc);
  drop(obj, loc);
  if (k) continue; else default_to(RELAX);
```

118.

```
⟨ Put coins in the vending machine 118 ⟩ ≡
  {
    destroy(COINS);
    drop(BATTERIES, loc);
    prop[BATTERIES] = 0;
    report(note[offset[BATTERIES]]);
  }
```

This code is used in section 117.

119. TROLL2 is the absent troll. We move the troll bridge up to first in the list of things at its location.

```
⟨ Chase the troll away 119 ⟩ ≡
  {
    printf("The␣bear␣lumbers␣toward␣the␣troll,␣who␣lets␣out␣a␣startled␣shriek␣and\n\
        scurries␣away.␣␣The␣bear␣soon␣gives␣up␣the␣pursuit␣and␣wanders␣back.\n");
    k = 1;   /* suppress the "OK" message */
    destroy(TROLL); destroy(TROLL_);
    drop(TROLL2, swside); drop(TROLL2_, neside);
    prop[TROLL] = 2;
    move(BRIDGE, swside); move(BRIDGE_, neside);   /* put first in their lists */
  }
```

This code is used in section 117.

120. ⟨ Check special cases for dropping the bird 120 ⟩ ≡

```
{
  if (here(SNAKE)) {
    printf("The little bird attacks the green snake, and in an astounding flurry\n\
        drives the snake away.\n");
    k = 1;
    if (closed) goto dwarves_upset;
    destroy(SNAKE);
    prop[SNAKE] = 1;    /* used in conditional instructions */
  }
  else if (is_at_loc(DRAGON) ∧ prop[DRAGON] ≡ 0) {
    destroy(BIRD); prop[BIRD] = 0;
    if (place[SNAKE] ≡ hmk) lost_treasures++;
    report("The little bird attacks the green dragon, and in an astounding flurry\n\
        gets burnt to a cinder.  The ashes blow away.");
  }
}
```

This code is used in section 117.

121. ⟨ Check special cases for dropping the vase 121 ⟩ ≡

```
{
  prop[VASE] = (place[PILLOW] ≡ loc ? 0 : 2);
  printf("%s\n", note[offset[VASE] + 1 + prop[VASE]]); k = 1;
  if (prop[VASE]) base[VASE] = VASE;
}
```

This code is used in section 117.

122. Throwing is like dropping, except that it covers a few more cases.

⟨ Handle cases of transitive verbs and **continue** 97 ⟩ +≡

```
case TOSS: if (obj ≡ ROD ∧ toting(ROD2) ∧ ¬toting(ROD)) obj = ROD2;
  if (¬toting(obj)) goto report_default;
  if (is_treasure(obj) ∧ is_at_loc(TROLL)) ⟨ Snarf a treasure for the troll 124 ⟩;
  if (obj ≡ FOOD ∧ here(BEAR)) {
    obj = BEAR; change_to(FEED);
  }
  if (obj ≠ AXE) change_to(DROP);
  if (dwarf()) ⟨ Throw the axe at a dwarf 163 ⟩;
  if (is_at_loc(DRAGON) ∧ prop[DRAGON] ≡ 0)
    printf("The axe bounces harmlessly off the dragon's thick scales.\n");
  else if (is_at_loc(TROLL))
    printf("The troll deftly catches the axe, examines it carefully, and tosses it\n\
      back, declaring, \"Good workmanship, but it's not valuable enough.\n");
  else if (here(BEAR) ∧ prop[BEAR]) ⟨ Throw the axe at the bear 123 ⟩
  else {
    obj = NOTHING;
    change_to(KILL);
  }
  drop(AXE, loc); stay_put;
```

123. This'll teach you a lesson.

```
⟨ Throw the axe at the bear 123 ⟩ ≡
  {
    drop(AXE, loc);
    prop[AXE] = 1;  base[AXE] = AXE;    /* it becomes immovable */
    if (place[BEAR] ≡ loc) move(BEAR, loc);    /* put bear first in its list */
    report("The␣axe␣misses␣and␣lands␣near␣the␣bear␣where␣you␣can't␣get␣at␣it");
  }
```

This code is used in section 122.

124. If you toss the vase, the skillful troll will catch it before it breaks.

```
⟨ Snarf a treasure for the troll 124 ⟩ ≡
  {
    drop(obj, limbo);
    destroy(TROLL);  destroy(TROLL_);
    drop(TROLL2, swside);  drop(TROLL2_, neside);
    move(BRIDGE, swside);  move(BRIDGE_, neside);
    report("The␣troll␣catches␣your␣treasure␣and␣scurries␣away␣out␣of␣sight");
  }
```

This code is used in section 122.

125. When you try to attack, the action becomes violent.

```
⟨ Handle cases of transitive verbs and continue 97 ⟩ +≡
case KILL: if (obj ≡ NOTHING) ⟨ See if there's a unique object to attack 126 ⟩;
  switch (obj) {
  case 0: report("There␣is␣nothing␣here␣to␣attack.");
  case BIRD: ⟨ Dispatch the poor bird 127 ⟩;
  case DRAGON: if (prop[DRAGON] ≡ 0) ⟨ Fun stuff for dragon 128 ⟩;
  cry: report("For␣crying␣out␣loud,␣the␣poor␣thing␣is␣already␣dead!");
  case CLAM: case OYSTER: report("The␣shell␣is␣very␣strong␣and␣impervious␣to␣attack.");
  case SNAKE: report("Attacking␣the␣snake␣both␣doesn't␣work␣and␣is␣very␣dangerous.");
  case DWARF: if (closed) goto dwarves_upset;
    report("With␣what?␣␣Your␣bare␣hands?");
  case TROLL: report("Trolls␣are␣close␣relatives␣with␣the␣rocks␣and␣have␣skin␣as␣tough␣as\n\
        a␣rhinoceros␣hide.␣␣The␣troll␣fends␣off␣your␣blows␣effortlessly.");
  case BEAR:
    switch (prop[BEAR]) {
    case 0: report("With␣what?␣␣Your␣bare␣hands?␣␣Against␣HIS␣bear␣hands?");
    case 3: goto cry;
    default: report("The␣bear␣is␣confused;␣he␣only␣wants␣to␣be␣your␣friend.");
    }
  default: goto report_default;
  }
```

126. Attackable objects fall into two categories: enemies (snake, dwarf, etc.) and others (bird, clam).
We might get here when you threw an axe; you can't attack the bird with an axe.

⟨ See if there's a unique object to attack 126 ⟩ ≡

```
{
  if (dwarf()) k++, obj = DWARF;
  if (here(SNAKE)) k++, obj = SNAKE;
  if (is_at_loc(DRAGON) ∧ prop[DRAGON] ≡ 0) k++, obj = DRAGON;
  if (is_at_loc(TROLL)) k++, obj = TROLL;
  if (here(BEAR) ∧ prop[BEAR] ≡ 0) k++, obj = BEAR;
  if (k ≡ 0) {    /* no enemies present */
    if (here(BIRD) ∧ oldverb ≠ TOSS) k++, obj = BIRD;
    if (here(CLAM) ∨ here(OYSTER)) k++, obj = CLAM;
        /* no harm done to call the oyster a clam in this case */
  }
  if (k > 1) goto get_object;
}
```

This code is used in section 125.

127. ⟨ Dispatch the poor bird 127 ⟩ ≡

```
{
  if (closed) report("Oh,␣leave␣the␣poor␣unhappy␣bird␣alone.");
  destroy(BIRD); prop[BIRD] = 0;
  if (place[SNAKE] ≡ hmk) lost_treasures++;
  report("The␣little␣bird␣is␣now␣dead.␣␣Its␣body␣disappears.");
}
```

This code is used in section 125.

128. Here we impersonate the main dialog loop. If you insist on attacking the dragon, you win! He dies, the Persian rug becomes free, and *scan2* takes the place of *scan1* and *scan3*.

⟨ Fun stuff for dragon 128 ⟩ ≡

```
{
  printf("With␣what?␣␣Your␣bare␣hands?\n");
  verb = ABSTAIN; obj = NOTHING;
  listen();
  if (¬(streq(word1, "yes") ∨ streq(word1, "y"))) goto pre_parse;
  printf(note[offset[DRAGON] + 1]);
  prop[DRAGON] = 2;    /* dead */
  prop[RUG] = 0; base[RUG] = NOTHING;    /* now it's a usable treasure */
  base[DRAGON_] = DRAGON_;
  destroy(DRAGON_);    /* inaccessible */
  base[RUG_] = RUG_;
  destroy(RUG_);    /* inaccessible */
  for (t = 1; t ≤ max_obj; t++)
    if (place[t] ≡ scan1 ∨ place[t] ≡ scan3) move(t, scan2);
  loc = scan2; stay_put;
}
```

This code is used in section 125.

129. Feeding various animals leads to various quips. Feeding a dwarf is a bad idea. The bear is special.

⟨ Handle cases of transitive verbs and **continue** 97 ⟩ +≡

```
case FEED:
  switch (obj) {
  case BIRD: report("It's␣not␣hungry␣(it's␣merely␣pinin'␣for␣the␣fjords).␣␣Besides,␣you\n\
        have␣no␣bird␣seed.");
  case TROLL: report("Gluttony␣is␣not␣one␣of␣the␣troll's␣vices.␣␣Avarice,␣however,␣is.");
  case DRAGON: if (prop[DRAGON]) report(default_msg[EAT]);
    break;
  case SNAKE: if (closed ∨ ¬here(BIRD)) break;
    destroy(BIRD); prop[BIRD] = 0; lost_treasures++;
    report("The␣snake␣has␣now␣devoured␣your␣bird.");
  case BEAR: if (¬here(FOOD)) {
      if (prop[BEAR] ≡ 0) break;
      if (prop[BEAR] ≡ 3) change_to(EAT);
      goto report_default;
    }
    destroy(FOOD); prop[BEAR] = 1;
    prop[AXE] = 0; base[AXE] = NOTHING;     /* axe is movable again */
    report("The␣bear␣eagerly␣wolfs␣down␣your␣food,␣after␣which␣he␣seems␣to␣calm\n\
        down␣considerably␣and␣even␣becomes␣rather␣friendly.");
  case DWARF: if (¬here(FOOD)) goto report_default;
    dflag++;
    report("You␣fool,␣dwarves␣eat␣only␣coal!␣␣Now␣you've␣made␣him␣REALLY␣mad!");
  default: report(default_msg[CALM]);
  }
  report("There's␣nothing␣here␣it␣wants␣to␣eat␣(except␣perhaps␣you).");
```

130. Locking and unlocking involves several interesting special cases.

⟨ Handle cases of transitive verbs and **continue** 97 ⟩ +≡

```
case OPEN: case CLOSE:
  switch (obj) {
  case OYSTER: k = 1;
  case CLAM: ⟨ Open/close clam/oyster 134 ⟩;
  case GRATE: case CHAIN: if (¬here(KEYS)) report("You␣have␣no␣keys!");
    ⟨ Open/close grate/chain 131 ⟩;
  case KEYS: report("You␣can't␣lock␣or␣unlock␣the␣keys.");
  case CAGE: report("It␣has␣no␣lock.");
  case DOOR: if (prop[DOOR]) default_to(RELAX);
    report("The␣door␣is␣extremely␣rusty␣and␣refuses␣to␣open.");
  default: goto report_default;
  }
```

131. ⟨ Open/close grate/chain 131 ⟩ ≡

```
if (obj ≡ CHAIN) ⟨ Open/close chain 132 ⟩;
if (closing) {
  ⟨ Panic at closing time 180 ⟩; continue;
}
k = prop[GRATE];
prop[GRATE] = (verb ≡ OPEN);
switch (k + 2 * prop[GRATE]) {
case 0: report("It␣was␣already␣locked.");
case 1: report("The␣grate␣is␣now␣locked.");
case 2: report("The␣grate␣is␣now␣unlocked.");
case 3: report("It␣was␣already␣unlocked.");
}
```

This code is used in section 130.

132. ⟨ Open/close chain 132 ⟩ ≡

```
{
  if (verb ≡ OPEN) ⟨ Open chain 133 ⟩;
  if (loc ≠ barr) report("There␣is␣nothing␣here␣to␣which␣the␣chain␣can␣be␣locked.");
  if (prop[CHAIN]) report("It␣was␣already␣locked.");
  prop[CHAIN] = 2, base[CHAIN] = CHAIN;
  if (toting(CHAIN)) drop(CHAIN, loc);
  report("The␣chain␣is␣now␣locked.");
}
```

This code is used in section 131.

133. ⟨ Open chain 133 ⟩ ≡

```
{
  if (prop[CHAIN] ≡ 0) report("It␣was␣already␣unlocked.");
  if (prop[BEAR] ≡ 0)
    report("There␣is␣no␣way␣to␣get␣past␣the␣bear␣to␣unlock␣the␣chain,␣which␣is\n\
        probably␣just␣as␣well.");
  prop[CHAIN] = 0, base[CHAIN] = NOTHING;     /* chain is free */
  if (prop[BEAR] ≡ 3) base[BEAR] = BEAR;
  else prop[BEAR] = 2, base[BEAR] = NOTHING;
  report("The␣chain␣is␣now␣unlocked.");
}
```

This code is used in section 132.

134. The clam/oyster is extremely heavy to carry, although not as heavy as the gold.

```
#define clam_oyster (obj ≡ CLAM ? "clam" : "oyster")
⟨ Open/close clam/oyster 134 ⟩ ≡
  if (verb ≡ CLOSE) report("What?");
  if (¬toting(TRIDENT)) {
    printf("You␣don't␣have␣anything␣strong␣enough␣to␣open␣the␣%s", clam_oyster);
    report(".");
  }
  if (toting(obj)) {
    printf("I␣advise␣you␣to␣put␣down␣the␣%s␣before␣opening␣it.␣␣", clam_oyster);
    report(obj ≡ CLAM ? ">STRAIN!<" : ">WRENCH!<");
  }
  if (obj ≡ CLAM) {
    destroy(CLAM); drop(OYSTER, loc); drop(PEARL, sac);
    report("A␣glistening␣pearl␣falls␣out␣of␣the␣clam␣and␣rolls␣away.␣␣Goodness,\n\
        this␣must␣really␣be␣an␣oyster.␣␣(I␣never␣was␣very␣good␣at␣identifying\n\
        bivalves.)␣␣Whatever␣it␣is,␣it␣has␣now␣snapped␣shut␣again.");
  } else report("The␣oyster␣creaks␣open,␣revealing␣nothing␣but␣oyster␣inside.\n\
        It␣promptly␣snaps␣shut␣again.");
```

This code is used in section 130.

135. You get little satisfaction from asking us to read, unless you hold the oyster—*after* the cave is closed.

```
⟨ Handle cases of transitive verbs and continue 97 ⟩ +≡
case READ: if (dark) goto cant_see_it;
  switch (obj) {
  case MAG: report("I'm␣afraid␣the␣magazine␣is␣written␣in␣dwarvish.");
  case TABLET: report("\"CONGRATULATIONS␣ON␣BRINGING␣LIGHT␣INTO␣THE␣DARK-ROOM!\"");
  case MESSAGE:
    report("\"This␣is␣not␣the␣maze␣where␣the␣pirate␣hides␣his␣treasure␣chest.\"");
  case OYSTER: if (hinted[1]) {
      if (toting(OYSTER)) report("It␣says␣the␣same␣thing␣it␣did␣before.");
    }
    else if (closed ∧ toting(OYSTER)) {
      offer(1); continue;
    }
  default: goto report_default;
  }
```

136. OK, that just about does it. We're left with only one more "action verb" to handle, and it is intransitive. In order to penetrate this puzzle, you must pronounce the magic incantation in its correct order, as it appears on the wall of the Giant Room. A global variable *foobar* records your progress.

```
⟨ Handle cases of intransitive verbs and continue 92 ⟩ +≡
case FEEFIE:
  while (¬streq(word1, incantation[k])) k++;
  if (foobar ≡ −k) ⟨ Proceed foobarically 139 ⟩;
  if (foobar ≡ 0) goto nada_sucede;
  report("What's␣the␣matter,␣can't␣you␣read?␣␣Now␣you'd␣best␣start␣over.");
```

137. ⟨ Global variables 7 ⟩ +≡

```
  char *incantation[] = {"fee", "fie", "foe", "foo", "fum"};
  int foobar;    /* current incantation progress */
```

138. Just after every command you give, we make the *foobar* counter negative if you're on track, otherwise we zero it.

⟨ Handle special cases of input 82 ⟩ +≡
  if ($foobar > 0$) $foobar = -foobar$;
  else $foobar = 0$;

139. If you get all the way through, we zip the eggs back to the Giant Room, unless they're already there. The troll returns if you've stolen the eggs back from him.

⟨ Proceed foobarically 139 ⟩ ≡

```
{
  foobar = k + 1;
  if (foobar ≠ 4) default_to(RELAX);
  foobar = 0;
  if (place[EGGS] ≡ giant ∨ (toting(EGGS) ∧ loc ≡ giant))
  nada_sucede: report(default_msg[WAVE]);
  if (place[EGGS] ≡ limbo ∧ place[TROLL] ≡ limbo ∧ prop[TROLL] ≡ 0) prop[TROLL] = 1;
  k = (loc ≡ giant ? 0 : here(EGGS) ? 1 : 2);
  move(EGGS, giant);
  report(note[offset[EGGS] + k]);
}
```

This code is used in section 136.

140. Motions. A major cycle comes to an end when a motion verb *mot* has been given and we have computed the appropriate *newloc* accordingly.

First, we deal with motions that don't refer directly to the travel table.

```
⟨Handle special motion words 140⟩ ≡
  newloc = loc;     /* by default we will stay put */
  if (mot ≡ NOWHERE) continue;
  if (mot ≡ BACK) ⟨Try to go back 143⟩;
  if (mot ≡ LOOK) ⟨Repeat the long description and continue 141⟩;
  if (mot ≡ CAVE) {
    if (loc < min_in_cave)
      printf("I␣can't␣see␣where␣the␣cave␣is,␣but␣hereabouts␣no␣stream␣can␣run␣on\n\
          the␣surface␣for␣long.␣␣I␣would␣try␣the␣stream.\n");
    else printf("I␣need␣more␣detailed␣instructions␣to␣do␣that.\n");
    continue;
  }
```

This code is used in section 75.

141. When looking around, we pretend that it wasn't dark (though it may *now* be dark), so you won't fall into a pit while staring into the gloom.

```
⟨Repeat the long description and continue 141⟩ ≡
  {
    if (++look_count ≤ 3)
      printf("Sorry,␣but␣I␣am␣not␣allowed␣to␣give␣more␣detail.␣␣I␣will␣repeat␣the\n\
          long␣description␣of␣your␣location.\n");
    was_dark = false;
    visits[loc] = 0;
    continue;
  }
```

This code is used in section 140.

142. ⟨Global variables 7⟩ +≡

```
  int look_count;     /* how many times you've asked us to look */
```

143. If you ask us to go back, we look for a motion that goes from *loc* to *oldloc*, or to *oldoldloc* if *oldloc* has forced motion. Otherwise we can't take you back.

⟨ Try to go back 143 ⟩ ≡

```
{
  l = (forced_move(oldloc) ? oldoldloc : oldloc);
  oldoldloc = oldloc;
  oldloc = loc;
  if (l ≡ loc) ⟨ Apologize for inability to backtrack 145 ⟩;
  for (q = start[loc], qq = Λ; q < start[loc + 1]; q++) {
    ll = q→dest;
    if (ll ≡ l) goto found;
    if (ll ≤ max_loc ∧ forced_move(ll) ∧ start[ll]→dest ≡ l) qq = q;
  }
  if (qq ≡ Λ) {
    printf("You can't get there from here.\n"); continue;
  }
  else q = qq;
found: mot = q→mot;
  goto go_for_it;
}
```

This code is used in section 140.

144. ⟨ Additional local registers 22 ⟩ +≡

```
register location l, ll;
```

145. ⟨ Apologize for inability to backtrack 145 ⟩ ≡

```
{
  printf("Sorry, but I no longer seem to remember how you got here.\n");
  continue;
}
```

This code is used in section 143.

146. Now we are ready to interpret the instructions in the travel table. The following code implements the conventions of section 19.

⟨ Determine the next location, *newloc* 146 ⟩ ≡

```
for (q = start[loc]; q < start[loc + 1]; q++) {
  if (forced_move(loc) ∨ q→mot ≡ mot) break;
}
if (q ≡ start[loc + 1]) ⟨ Report on inapplicable motion and continue 148 ⟩;
⟨ If the condition of instruction q isn't satisfied, advance q 147 ⟩;
newloc = q→dest;
if (newloc ≤ max_loc) continue;
if (newloc > max_spec) {
  printf("%s\n", remarks[newloc − max_spec]);
stay: newloc = loc; continue;
}
switch (newloc) {
case ppass: ⟨ Choose newloc via plover-alcove passage 149 ⟩;
case pdrop: ⟨ Drop the emerald during plover transportation 150 ⟩; goto no_good;
case troll: ⟨ Cross troll bridge if possible 151 ⟩;
}
```

This code is used in section 75.

147. ⟨ If the condition of instruction q isn't satisfied, advance q 147 ⟩ ≡

```
while (1) {
  j = q→cond;
  if (j > 300) {
    if (prop[j % 100] ≠ (int)((j − 300)/100)) break;
  } else if (j ≤ 100) {
    if (j ≡ 0 ∨ pct(j)) break;
  } else if (toting(j % 100) ∨ (j ≥ 200 ∧ is_at_loc(j % 100))) break;
no_good:
  for (qq = q++;
  q→dest ≡ qq→dest ∧ q→cond ≡ qq→cond;
  q++) ;
}
```

This code is used in section 146.

148. Here we look at *verb* just in case you asked us to '`find gully`' or something like that.

⟨ Report on inapplicable motion and **continue** 148 ⟩ ≡

```
{
  if (mot ≡ CRAWL) printf("Which␣way?");
  else if (mot ≡ XYZZY ∨ mot ≡ PLUGH) printf(default_msg[WAVE]);
  else if (verb ≡ FIND ∨ verb ≡ INVENTORY) printf(default_msg[FIND]);
  else if (mot ≤ FORWARD)
    switch (mot) {
    case IN: case OUT:
      printf("I␣don't␣know␣in␣from␣out␣here.␣␣␣␣Use␣compass␣points␣or␣name␣something\n\
          in␣the␣general␣direction␣you␣want␣to␣go.");
      break;
    case FORWARD: case L: case R:
      printf("I␣am␣unsure␣how␣you␣are␣facing.␣␣␣␣Use␣compass␣points␣or␣nearby␣objects.");
      break;
    default: printf("There␣is␣no␣way␣to␣go␣in␣that␣direction.");
    } else printf("I␣don't␣know␣how␣to␣apply␣that␣word␣here.");
  printf("\n"); continue;     /* newloc = loc */
}
```

This code is used in section 146.

149. Only the emerald can be toted through the plover-alcove passage — not even the lamp.

⟨ Choose *newloc* via plover-alcove passage 149 ⟩ ≡

```
if (holding ≡ 0 ∨ (toting(EMERALD) ∧ holding ≡ 1)) {
  newloc = alcove + proom − loc; continue;     /* move through the passage */
} else {
  printf("Something␣you're␣carrying␣won't␣fit␣through␣the␣tunnel␣with␣you.\n\
      You'd␣best␣take␣inventory␣and␣drop␣something.\n");
  goto stay;
}
```

This code is used in section 146.

150. The *pdrop* command applies only when you're carrying the emerald. We make you drop it, thereby forcing you to use the plover-alcove passage if you want to get it out. We don't actually tell you that it was dropped; we just pretend you weren't carrying it after all.

⟨ Drop the emerald during plover transportation 150 ⟩ ≡

```
drop(EMERALD, loc);
```

This code is used in section 146.

151. Troll bridge crossing is treated as a special motion so that dwarves won't wander across and encounter the bear.

You can get here only if TROLL is in limbo but TROLL2 has taken its place. Moreover, if you're on the southwest side, *prop*[TROLL] will be nonzero. If *prop*[TROLL] is 1, you've crossed since paying, or you've stolen away the payment. Special stuff involves the bear.

⟨ Cross troll bridge if possible 151 ⟩ ≡

```
if (prop[TROLL] ≡ 1) ⟨ Block the troll bridge and stay put 152 ⟩;
newloc = neside + swside − loc;    /* cross it */
if (prop[TROLL] ≡ 0) prop[TROLL] = 1;
if (¬toting(BEAR)) continue;
printf("Just␣as␣you␣reach␣the␣other␣side,␣the␣bridge␣buckles␣beneath␣the\n\
    weight␣of␣the␣bear,␣who␣was␣still␣following␣you␣around.␣␣You\n\
    scrabble␣desperately␣for␣support,␣but␣as␣the␣bridge␣collapses␣you\n\
    stumble␣back␣and␣fall␣into␣the␣chasm.\n");
prop[BRIDGE] = 1; prop[TROLL] = 2;
drop(BEAR, newloc); base[BEAR] = BEAR; prop[BEAR] = 3;    /* the bear is dead */
if (prop[SPICES] < 0 ∧ place[SPICES] ≥ neside) lost_treasures++;
if (prop[CHAIN] < 0 ∧ place[CHAIN] ≥ neside) lost_treasures++;
oldoldloc = newloc;    /* if you are revived, you got across */
goto death;
```

This code is used in section 146.

152. ⟨ Block the troll bridge and stay put 152 ⟩ ≡

```
{
  move(TROLL, swside); move(TROLL_, neside); prop[TROLL] = 0;
  destroy(TROLL2); destroy(TROLL2_);
  move(BRIDGE, swside); move(BRIDGE_, neside);
  printf("%s\n", note[offset[TROLL] + 1]);
  goto stay;
}
```

This code is used in section 151.

153. Obstacles might still arise after the choice of *newloc* has been made. The following program is executed at the beginning of each major cycle.

⟨ Check for interference with the proposed move to *newloc* 153 ⟩ ≡

```
if (closing ∧ newloc < min_in_cave ∧ newloc ≠ limbo) {
  ⟨ Panic at closing time 180 ⟩; newloc = loc;
} else if (newloc ≠ loc) ⟨ Stay in loc if a dwarf is blocking the way to newloc 176 ⟩;
```

This code is used in section 75.

154. Random numbers. You won't realize it until you have played the game for awhile, but adventures in Colossal Cave are not deterministic. Lots of things can happen differently when you give the same input, because caves are continually changing, and the dwarves don't have consistent aim, etc.

A simple linear congruential method is used to provide numbers that are random enough for our purposes.

```
⟨ Subroutines 6 ⟩ +≡
  int ran ARGS((int));
  int ran(range)
      int range;    /* for uniform integers between 0 and range − 1 */
  {
    rx = (1021 * rx) & #fffff;    /* multiply by 1021, modulo 2^20 */
    return (range * rx) ≫ 20;
  }
```

155. ⟨ Global variables 7 ⟩ +≡

```
  int rx;    /* the last random value generated */
```

156. Each run is different.

```
⟨ Initialize the random number generator 156 ⟩ ≡
  rx = (((int) time(Λ)) & #fffff) + 1;
```

This code is used in section 200.

157. The *pct* macro returns true a given percentage of the time.

```
#define pct(r) (ran(100) < r)
⟨ Give optional plugh hint 157 ⟩ ≡
  if (loc ≡ y2 ∧ pct(25) ∧ ¬closing) printf("A␣hollow␣voice␣says␣\"PLUGH\".\n");
```

This code is used in section 86.

158. We kick the random number generator often, just to add variety to the chase.

```
⟨ Make special adjustments before looking at new input 85 ⟩ +≡
  k = ran(0);
```

159. Dwarf stuff. We've said a lot of vague stuff about dwarves; now is the time to be explicit. Five dwarves roam about the cave. Initially they are dormant but eventually they each walk about at random. A global variable called *dflag* governs their level of activity:

0	no dwarf stuff yet (we wait until you reach the Hall of Mists)
1	you've reached that hall, but haven't met the first dwarf
2	you've met one; the others start moving, but no knives thrown yet
3	a knife has been thrown, but it misses
4	knives will hit you with probability .095
5	knives will hit you with probability .190
6	knives will hit you with probability .285

and so on. Dwarves get madder and madder as *dflag* increases; this increases their accuracy.

A pirate stalks the cave too. He acts a lot like a dwarf with respect to random walks, so we call him *dwarf*[0], but actually he is quite different. He starts at the location of his treasure chest; you won't see that chest until after you've spotted him.

The present location of *dwarf*[*i*] is *dloc*[*i*]; initially no two dwarves are adjacent. The value of *dseen*[*i*] records whether or not dwarf *i* is following you.

```
#define nd 5   /* this many dwarves */
#define chest_loc dead2
#define message_loc pony
```

⟨ Global variables 7 ⟩ +≡

```
  int dflag;   /* how angry are the dwarves? */
  int dkill;   /* how many of them have you killed? */
  location dloc[nd + 1] = {chest_loc, hmk, wfiss, y2, like3, complex};   /* dwarf locations */
  location odloc[nd + 1];   /* prior locations */
  boolean dseen[nd + 1];   /* have you been spotted? */
```

160. The following subroutine is often useful.

⟨ Subroutines 6 ⟩ +≡

```
  int dwarf ARGS((void));
  int dwarf()   /* is a dwarf present? */
  {
    register int j;
    if (dflag < 2) return 0;
    for (j = 1; j ≤ nd; j++)
      if (dloc[j] ≡ loc) return 1;
    return 0;
  }
```

161. Just after you've moved to a new *loc*, we move the other guys. But we bypass all dwarf motion if you are in a place forbidden to the pirate, or if your next motion is forced. In particular, this means that the pirate can't steal the return toll, and dwarves can't meet the bear. It also means that dwarves won't follow you into a dead end of the maze, but c'est la vie; they'll wait for you outside the dead end.

⟨ Possibly move dwarves and the pirate 161 ⟩ ≡

```
if (loc ≤ max_pirate_loc ∧ loc ≠ limbo) {
   if (dflag ≡ 0) {
      if (loc ≥ min_lower_loc) dflag = 1;
   }
   else if (dflag ≡ 1) {
      if (loc ≥ min_lower_loc ∧ pct(5)) ⟨ Advance dflag to 2 162 ⟩;
   }
   else ⟨ Move dwarves and the pirate 164 ⟩;
}
```

This code is used in section 75.

162. When level 2 is reached, we silently kill 0, 1, or 2 of the dwarves. Then if any of the survivors is in the current location, we move him to *nugget*; thus no dwarf is presently tracking you. Another dwarf does, however, toss an axe and grumpily leave the scene.

(The grumpy dwarf might throw the axe while you're in the maze of all-different twists, even though other dwarves never go in there!)

⟨ Advance dflag to 2 162 ⟩ ≡

```
{
   dflag = 2;
   for (j = 0; j < 2; j++)
      if (pct(50)) dloc[1 + ran(nd)] = limbo;
   for (j = 1; j ≤ nd; j++) {
      if (dloc[j] ≡ loc) dloc[j] = nugget;
      odloc[j] = dloc[j];
   }
   printf("A␣little␣dwarf␣just␣walked␣around␣a␣corner,␣saw␣you,␣threw␣a␣little\n\
      axe␣at␣you,␣cursed,␣and␣ran␣away.␣␣(The␣axe␣missed.)\n");
   drop(AXE, loc);
}
```

This code is used in section 161.

163. It turns out that the only way you can get rid of a dwarf is to attack him with the axe. You'll hit him 2/3 of the time; in either case, the axe will be available for reuse.

```
⟨ Throw the axe at a dwarf 163 ⟩ ≡
  {
    for (j = 1; j ≤ nd; j++)
      if (dloc[j] ≡ loc) break;
    if (ran(3) < 2) {
      dloc[j] = limbo; dseen[j] = 0; dkill++;
      if (dkill ≡ 1)
        printf("You␣killed␣a␣little␣dwarf.␣␣The␣body␣vanishes␣in␣a␣cloud␣of␣greasy\n\
            black␣smoke.\n");
      else printf("You␣killed␣a␣little␣dwarf.\n");
    } else printf("You␣attack␣a␣little␣dwarf,␣but␣he␣dodges␣out␣of␣the␣way.\n");
    drop(AXE, loc); stay_put;
  }
```

This code is used in section 122.

164. Now things are in full swing. Dead dwarves don't do much of anything, but each live dwarf tends to stay with you if he's seen you. Otherwise he moves at random, never backing up unless there's no alternative.

```
⟨ Move dwarves and the pirate 164 ⟩ ≡
  {
    dtotal = attack = stick = 0;    /* initialize totals for possible battles */
    for (j = 0; j ≤ nd; j++)
      if (dloc[j] ≠ limbo) {
        register int i;
        ⟨ Make a table of all potential exits, ploc[0] through ploc[i − 1] 166 ⟩;
        if (i ≡ 0) i = 1, ploc[0] = odloc[j];
        odloc[j] = dloc[j];
        dloc[j] = ploc[ran(i)];    /* this is the random walk */
        dseen[j] = (dloc[j] ≡ loc ∨ odloc[j] ≡ loc ∨ (dseen[j] ∧ loc ≥ min_lower_loc));
        if (dseen[j]) ⟨ Make dwarf j follow 167 ⟩;
      }
    if (dtotal) ⟨ Make the threatening dwarves attack 170 ⟩;
  }
```

This code is used in section 161.

165. ⟨ Global variables 7 ⟩ +≡

```
  int dtotal;    /* this many dwarves are in the room with you */
  int attack;    /* this many have had time to draw their knives */
  int stick;    /* this many have hurled their knives accurately */
  location ploc[19];    /* potential locations for the next random step */
```

166. Random-moving dwarves think *scan1*, *scan2*, and *scan3* are three different locations, although you will never have that perception.

⟨ Make a table of all potential exits, $ploc[0]$ through $ploc[i-1]$ 166 ⟩ ≡

```
for (i = 0, q = start[dloc[j]]; q < start[dloc[j] + 1]; q++) {
   newloc = q->dest;
   if (newloc ≥ min_lower_loc ∧ newloc ≠ odloc[j] ∧ newloc ≠ dloc[j] ∧
         (i ≡ 0 ∨ newloc ≠ ploc[i − 1]) ∧ i < 19 ∧ q->cond ≠ 100 ∧
         newloc ≤ (j ≡ 0 ? max_pirate_loc : min_forced_loc − 1)) ploc[i++] = newloc;
}
```

This code is used in section 164.

167. A global variable *knife_loc* is used to remember where dwarves have most recently thrown knives at you. But as soon as you try to refer to the knife, we tell you it's pointless to do so; *knife_loc* is −1 thereafter.

⟨ Make dwarf j follow 167 ⟩ ≡

```
{
   dloc[j] = loc;
   if (j ≡ 0) ⟨ Make the pirate track you 172 ⟩
   else {
      dtotal++;
      if (odloc[j] ≡ dloc[j]) {
         attack++;
         if (knife_loc ≥ 0) knife_loc = loc;
         if (ran(1000) < 95 * (dflag − 2)) stick++;
      }
   }
}
```

This code is used in section 164.

168. ⟨ Global variables 7 ⟩ +≡

```
int knife_loc;    /* place where knife was mentioned, or −1 */
```

169. ⟨ Make special adjustments before looking at new input 85 ⟩ +≡

```
if (knife_loc > limbo ∧ knife_loc ≠ loc) knife_loc = limbo;
```

170. We actually know the results of the attack already; this is where we inform you of the outcome, pretending that the battle is now taking place.

```
⟨ Make the threatening dwarves attack 170 ⟩ ≡
  {
    if (dtotal ≡ 1) printf("There␣is␣a␣threatening␣little␣dwarf");
    else printf("There␣are␣%d␣threatening␣little␣dwarves", dtotal);
    printf("␣in␣the␣room␣with␣you!\n");
    if (attack) {
      if (dflag ≡ 2) dflag = 3;
      if (attack ≡ 1) k = 0, printf("One␣sharp␣nasty␣knife␣is␣thrown");
      else k = 2, printf("␣%d␣of␣them␣throw␣knives", attack);
      printf("␣at␣you␣---␣");
      if (stick ≤ 1) printf("%s!\n", attack_msg[k + stick]);
      else printf("%d␣of␣them␣get␣you!\n", stick);
      if (stick) {
        oldoldloc = loc; goto death;
      }
    }
  }
```

This code is used in section 164.

171. ⟨ Global variables 7 ⟩ +≡

```
  char *attack_msg[ ] = {"it␣misses", "it␣gets␣you",
  "none␣of␣them␣hit␣you", "one␣of␣them␣gets␣you"};
```

172. The pirate leaves you alone once you have found the chest. Otherwise he steals one of the treasures you're carrying, unless it's too easy. (The pyramid is too easy, if you're in the Plover Room or the Dark-Room.)

You spot the pirate if he robs you, or when you have seen all of the possible treasures (except, of course, the chest) and the current location has no treasures that still can't be moved. Before you've spotted him, we may give you a vague indication of his movements.

We use the value of *place*[MESSAGE] to determine whether the pirate has been seen; the condition of *place*[CHEST] is not a reliable indicator, since the chest might be in limbo if you've thrown it to the troll.

```
#define pirate_not_spotted (place[MESSAGE] ≡ limbo)
#define too_easy(i) (i ≡ PYRAMID ∧ (loc ≡ proom ∨ loc ≡ droom))

⟨ Make the pirate track you 172 ⟩ ≡
  {
    if (loc ≠ max_pirate_loc ∧ prop[CHEST] < 0) {
      for (i = min_treasure, k = 0; i ≤ max_obj; i++)
        if (¬too_easy(i)) {
          if (toting(i)) ⟨ Take booty and hide it in the chest 173 ⟩;
          if (place[i] ≡ loc) k = 1;     /* a treasure with base[i] ≠ NOTHING */
        }
      if (tally ≡ lost_treasures + 1 ∧ k ≡ 0 ∧ pirate_not_spotted ∧ prop[LAMP] ∧ here(LAMP))
        ⟨ Let the pirate be spotted 175 ⟩;
      if (odloc[0] ≠ dloc[0] ∧ pct(20))
        printf("There␣are␣faint␣rustling␣noises␣from␣the␣darkness␣behind␣you.\n");
    }
  }
```

This code is used in section 167.

173. The pirate isn't secretive about the fact that his chest is somewhere in a maze. However, he doesn't say which maze he means. Nor does he explain why he is interested in treasures only when you are carrying them; evidently he just likes to see you squirm.

⟨ Take booty and hide it in the chest 173 ⟩ ≡

```
{
  printf("Out␣from␣the␣shadows␣behind␣you␣pounces␣a␣bearded␣pirate!␣␣\"Har,␣har,\"\n\
      he␣chortles,␣\"I'll␣just␣take␣all␣this␣booty␣and␣hide␣it␣away␣with␣me\n\
      chest␣deep␣in␣the␣maze!\"␣␣He␣snatches␣your␣treasure␣and␣vanishes␣into\n\
      the␣gloom.\n");
  ⟨ Snatch all treasures that are snatchable here 174 ⟩;
  if (pirate_not_spotted) {
  move_chest: move(CHEST, chest_loc); move(MESSAGE, message_loc);
  }
  dloc[0] = odloc[0] = chest_loc; dseen[0] = false;
}
```

This code is used in section 172.

174. ⟨ Snatch all treasures that are snatchable here 174 ⟩ ≡

```
for (i = min_treasure; i ≤ max_obj; i++)
  if (¬too_easy(i)) {
    if (base[i] ≡ NOTHING ∧ place[i] ≡ loc) carry(i);
    if (toting(i)) drop(i, chest_loc);
  }
```

This code is used in section 173.

175. The window rooms are slightly lighted, but we don't spot the pirate there unless our lamp is on. Regardless of where we are, we do spot him even if the lighted lamp is on the ground.

⟨ Let the pirate be spotted 175 ⟩ ≡

```
{
  printf("There␣are␣faint␣rustling␣noises␣from␣the␣darkness␣behind␣you.␣␣As␣you\n\
      turn␣toward␣them,␣the␣beam␣of␣your␣lamp␣falls␣across␣a␣bearded␣pirate.\n\
      He␣is␣carrying␣a␣large␣chest.␣␣\"Shiver␣me␣timbers!\"␣he␣cries,␣\"I've\n\
      been␣spotted!␣␣I'd␣best␣hie␣meself␣off␣to␣the␣maze␣to␣hide␣me␣chest!\"\n\
      With␣that,␣he␣vanishes␣into␣the␣gloom.\n");
  goto move_chest;
}
```

This code is used in section 172.

176. One more loose end related to dwarfs needs to be addressed here.

⟨ Stay in *loc* if a dwarf is blocking the way to *newloc* 176 ⟩ ≡

```
if (newloc ≤ max_pirate_loc) {
  for (j = 1; j ≤ nd; j++)
    if (odloc[j] ≡ newloc ∧ dseen[j]) {
      printf("A␣little␣dwarf␣with␣a␣big␣knife␣blocks␣your␣way.\n");
      newloc = loc; break;
    }
}
```

This code is used in section 153.

177. Closing the cave. You get to wander around until you've located all fifteen treasures, although you need not have taken them yet. After that, you enter a new level of complexity: A global variable called *clock1* starts ticking downwards, every time you take a turn inside the cave. When it hits zero, we start closing the cave; then we sit back and wait for you to try to get out, letting *clock2* do the ticking. The initial value of *clock1* is large enough for you to get outside.

#define *closing* $(clock1 < 0)$

⟨ Global variables 7 ⟩ +≡
```
int clock1 = 15, clock2 = 30;   /* clocks that govern closing time */
boolean panic, closed;   /* various stages of closedness */
```

178. Location Y2 is virtually outside the cave, so *clock1* doesn't tick there. If you stay outside the cave with all your treasures, and with the lamp switched off, the game might go on forever; but you wouldn't be having any fun.

There's an interesting hack by which you can keep *tally* positive until you've taken all the treasures out of the cave. Namely, if your first moves are

`in`, `take lamp`, `plugh`, `on`, `drop lamp`, `s`, `take silver`,
`back`, `take lamp`, `plugh`, `out`, `drop silver`, `in`,

the silver bars will be at *road*; but *prop*[SILVER] will still be −1 and *tally* will still be 15. You can bring the other 14 treasures to the *house* at your leisure; then the *tally* will drop to zero when you step outside and actually see the silver for the first time.

⟨ Check the clocks and the lamp 178 ⟩ ≡
```
if (tally ≡ 0 ∧ loc ≥ min_lower_loc ∧ loc ≠ y2) clock1 −−;
if (clock1 ≡ 0) ⟨ Warn that the cave is closing 179 ⟩
else {
  if (clock1 < 0) clock2 −−;
  if (clock2 ≡ 0) ⟨ Close the cave 181 ⟩
  else ⟨ Check the lamp 184 ⟩;
}
```
This code is used in section 76.

179. At the time of first warning, we lock the grate, destroy the crystal bridge, kill all the dwarves (and the pirate), remove the troll and the bear (unless dead), and set *closing* to true. It's too much trouble to move the dragon, so we leave it. From now on until *clock2* runs out, you cannot unlock the grate, move to any location outside the cave, or create the bridge. Nor can you be resurrected if you die.

⟨ Warn that the cave is closing 179 ⟩ ≡
```
{
  printf("A␣sepulchral␣voice,␣reverberating␣through␣the␣cave,␣says,␣\"Cave\n\
      closing␣soon.␣␣All␣adventurers␣exit␣immediately␣through␣main␣office.\"\n");
  clock1 = −1;
  prop[GRATE] = 0;
  prop[CRYSTAL] = 0;
  for (j = 0; j ≤ nd; j++) dseen[j] = 0, dloc[j] = limbo;
  destroy(TROLL); destroy(TROLL_);
  move(TROLL2, swside); move(TROLL2_, neside);
  move(BRIDGE, swside); move(BRIDGE_, neside);
  if (prop[BEAR] ≠ 3) destroy(BEAR);
  prop[CHAIN] = 0; base[CHAIN] = NOTHING;
  prop[AXE] = 0; base[AXE] = NOTHING;
}
```
This code is used in section 178.

180. If you try to get out while the cave is closing, we assume that you panic; we give you a few additional turns to get frantic before we close.

⟨ Panic at closing time 180 ⟩ ≡

```
{
  if (¬panic) clock2 = 15, panic = true;
  printf("A␣mysterious␣recorded␣voice␣groans␣into␣life␣and␣announces:\n\
      ␣␣␣\"This␣exit␣is␣closed.␣␣Please␣leave␣via␣main␣office.\"\n");
}
```

This code is used in sections 131 and 153.

181. Finally, after *clock2* hits zero, we transport you to the final puzzle, which takes place in the previously inaccessible storage room. We have to set everything up anew, in order to use the existing machinery instead of writing a special program. We are careful not to include keys in the room, since we don't want to allow you to unlock the grate that separates you from your treasures. There is no water; otherwise we would need special code for watering the beanstalks.

The storage room has two locations, *neend* and *swend*. At the northeast end, we place empty bottles, a nursery of plants, a bed of oysters, a pile of lamps, rods with stars, sleeping dwarves, and you. At the southwest end we place a grate, a snake pit, a covey of caged birds, more rods, and pillows. A mirror stretches across one wall. But we destroy all objects you might be carrying, lest you have some that could cause trouble, such as the keys. We describe the flash of light and trundle back.

From the fact that you've seen all the treasures, we can infer that the snake is already gone, since the jewels are accessible only from the Hall of the Mountain King. We also know that you've been in the Giant Room (to get eggs); you've discovered that the clam is an oyster (because of the pearl); the dwarves have been activated, since you've found the chest. Therefore the long descriptions of *neend* and *swend* will make sense to you when you see them.

Dear reader, all the clues to this final puzzle are presented in the program itself, so you should have no trouble finding the solution.

⟨ Close the cave 181 ⟩ ≡

```
{
  printf("The␣sepulchral␣voice␣intones,␣\"The␣cave␣is␣now␣closed.\"␣␣As␣the␣echoes\n\
      fade,␣there␣is␣a␣blinding␣flash␣of␣light␣(and␣a␣small␣puff␣of␣orange\n\
      smoke).␣.␣.␣.␣␣␣␣Then␣your␣eyes␣refocus;␣you␣look␣around␣and␣find...\n");
  move(BOTTLE, neend); prop[BOTTLE] = −2;
  move(PLANT, neend); prop[PLANT] = −1;
  move(OYSTER, neend); prop[OYSTER] = −1;
  move(LAMP, neend); prop[LAMP] = −1;
  move(ROD, neend); prop[ROD] = −1;
  move(DWARF, neend); prop[DWARF] = −1;
  move(MIRROR, neend); prop[MIRROR] = −1;
  loc = oldloc = neend;
  move(GRATE, swend);    /* prop[GRATE] still zero */
  move(SNAKE, swend); prop[SNAKE] = −2;
  move(BIRD, swend); prop[BIRD] = −2;
  move(CAGE, swend); prop[CAGE] = −1;
  move(ROD2, swend); prop[ROD2] = −1;
  move(PILLOW, swend); prop[PILLOW] = −1;
  move(MIRROR_, swend);
  for (j = 1; j ≤ max_obj; j++)
    if (toting(j)) destroy(j);
  closed = true;
  bonus = 10;
  stay_put;
}
```

This code is used in section 178.

182. After the cave is closed, we look for objects being toted with *prop* < 0; their property value is changed to $-1 - prop$. This means they won't be described until they've been picked up and put down, separate from their respective piles.

⟨ Make special adjustments before looking at new input 85 ⟩ +≡
```
if (closed) {
   if (prop[OYSTER] < 0 ∧ toting(OYSTER)) printf("%s\n", note[offset[OYSTER] + 1]);
   for (j = 1; j ≤ max_obj; j++)
      if (toting(j) ∧ prop[j] < 0) prop[j] = −1 − prop[j];
}
```

183. Death and resurrection. Only the most persistent adventurers get to see the closing of the cave, because their lamp gives out first. For example, if you have lost the ability to find any treasures, *tally* will never go to zero.

```
⟨ Zap the lamp if the remaining treasures are too elusive 183 ⟩ ≡
  if (tally ≡ lost_treasures ∧ tally > 0 ∧ limit > 35) limit = 35;
```

This code is used in section 88.

184. On every turn, we check to see if you are in trouble lampwise.

```
⟨ Check the lamp 184 ⟩ ≡
  {
    if (prop[LAMP] ≡ 1) limit−−;
    if (limit ≤ 30 ∧ here(BATTERIES) ∧ prop[BATTERIES] ≡ 0 ∧ here(LAMP)) ⟨ Replace the batteries 186 ⟩
    else if (limit ≡ 0) ⟨ Extinguish the lamp 187 ⟩
    else if (limit < 0 ∧ loc < min_in_cave) {
      printf("There's␣not␣much␣point␣in␣wandering␣around␣out␣here,␣and␣you␣can't\n\
          explore␣the␣cave␣without␣a␣lamp.␣␣So␣let's␣just␣call␣it␣a␣day.\n");
      goto give_up;
    } else if (limit ≤ 30 ∧ ¬warned ∧ here(LAMP)) {
      printf("Your␣lamp␣is␣getting␣dim");
      if (prop[BATTERIES] ≡ 1) printf(",␣and␣you're␣out␣of␣spare␣batteries.␣␣You'd\n\
            best␣start␣wrapping␣this␣up.\n");
      else if (place[BATTERIES] ≡ limbo) printf(".␣␣You'd␣best␣start␣wrapping␣this␣up,␣unless\n\
            you␣can␣find␣some␣fresh␣batteries.␣␣I␣seem␣to␣recall␣that␣there's\n\
            a␣vending␣machine␣in␣the␣maze.␣␣Bring␣some␣coins␣with␣you.\n");
      else printf(".␣␣You'd␣best␣go␣back␣for␣those␣batteries.\n");
      warned = true;
    }
  }
```

This code is used in section 178.

185.

```
⟨ Global variables 7 ⟩ +≡
  boolean warned;    /* have you been warned about the low power supply? */
```

186. The batteries hold a pretty hefty charge.

```
⟨ Replace the batteries 186 ⟩ ≡
  {
    printf("Your␣lamp␣is␣getting␣dim.␣␣I'm␣taking␣the␣liberty␣of␣replacing\n\
        the␣batteries.\n");
    prop[BATTERIES] = 1;
    if (toting(BATTERIES)) drop(BATTERIES, loc);
    limit = 2500;
  }
```

This code is used in section 184.

187.

```
⟨ Extinguish the lamp 187 ⟩ ≡
  {
    limit = −1; prop[LAMP] = 0;
    if (here(LAMP)) printf("Your␣lamp␣has␣run␣out␣of␣power.");
  }
```

This code is used in section 184.

188. The easiest way to get killed is to fall into a pit in pitch darkness.

⟨ Deal with death and resurrection 188 ⟩ ≡

```
pitch_dark: printf("You␣fell␣into␣a␣pit␣and␣broke␣every␣bone␣in␣your␣body!\n");
  oldoldloc = loc;
```

See also sections 189, 191, and 192.

This code is used in section 2.

189. "You're dead, Jim."

When you die, *newloc* is undefined (often *limbo*) and *oldloc* is what killed you. So we look at *oldoldloc*, the last place you were safe.

We generously allow you to die up to three times; *death_count* is the number of deaths you have had so far.

```
#define max_deaths 3
```

⟨ Deal with death and resurrection 188 ⟩ +≡

```
death: death_count++;
  if (closing) {
    printf("It␣looks␣as␣though␣you're␣dead.␣␣Well,␣seeing␣as␣how␣it's␣so␣close\n\
        to␣closing␣time␣anyway,␣let's␣just␣call␣it␣a␣day.\n");
    goto quit;
  }
  if (¬yes(death_wishes[2∗death_count−2], death_wishes[2∗death_count−1], ok) ∨ death_count ≡ max_deaths)
    goto quit;
```

190. ⟨ Global variables 7 ⟩ +≡

```
  int death_count;    /* how often have you kicked the bucket? */
  char *death_wishes[2 * max_deaths] = {
  "Oh␣dear,␣you␣seem␣to␣have␣gotten␣yourself␣killed.␣␣I␣might␣be␣able␣to\n\
     help␣you␣out,␣but␣I've␣never␣really␣done␣this␣before.␣␣Do␣you␣want␣me\n\
     to␣try␣to␣reincarnate␣you?",
  "All␣right.␣␣But␣don't␣blame␣me␣if␣something␣goes␣wr......\n\
     ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣---␣POOF!!␣---\n\
     You␣are␣engulfed␣in␣a␣cloud␣of␣orange␣smoke.␣␣Coughing␣and␣gasping,\n\
     you␣emerge␣from␣the␣smoke␣and␣find....",
  "You␣clumsy␣oaf,␣you've␣done␣it␣again!␣␣I␣don't␣know␣how␣long␣I␣can\n\
     keep␣this␣up.␣␣Do␣you␣want␣me␣to␣try␣reincarnating␣you␣again?",
  "Okay,␣now␣where␣did␣I␣put␣my␣resurrection␣kit?....␣␣>POOF!<\n\
     Everything␣disappears␣in␣a␣dense␣cloud␣of␣orange␣smoke.",
  "Now␣you've␣really␣done␣it!␣␣I'm␣out␣of␣orange␣smoke!␣␣You␣don't␣expect\n\
     me␣to␣do␣a␣decent␣reincarnation␣without␣any␣orange␣smoke,␣do␣you?",
  "Okay,␣if␣you're␣so␣smart,␣do␣it␣yourself!␣␣I'm␣leaving!"};
```

191. At this point you are reborn. All objects you were carrying are dropped at *oldoldloc* (presumably your last place prior to being killed), with their properties unchanged. The loop runs backwards, so that the bird is dropped before the cage. The lamp is a special case, because we wouldn't want to leave it underground; we turn it off and leave it outside the building—only if you were carrying it, of course. You yourself are left *inside* the building. (Heaven help you if you try to xyzzy back into the cave without the lamp.) We zap *oldloc* so that you can't just go back.

```
⟨ Deal with death and resurrection 188 ⟩ +≡
  for (j = max_obj; j > 0; j--)
    if (toting(j)) drop(j, j ≡ LAMP ? road : oldoldloc);
  if (toting(LAMP)) prop[LAMP] = 0;
  place[WATER] = limbo; place[OIL] = limbo;
  loc = oldloc = house;
  goto commence;
```

192. Oh dear, you've disturbed the dwarves.

```
⟨ Deal with death and resurrection 188 ⟩ +≡
dwarves_upset:
  printf("The␣resulting␣ruckus␣has␣awakened␣the␣dwarves.␣␣There␣are␣now␣several\n\
      threatening␣little␣dwarves␣in␣the␣room␣with␣you!␣␣Most␣of␣them␣throw\n\
      knives␣at␣you!␣␣All␣of␣them␣get␣you!\n");
```

193. Scoring. Here is the scoring algorithm we use:

Objective	*Points*	*Total possible*
Getting well into cave	25	25
Each treasure < chest	12	60
Treasure chest itself	14	14
Each treasure > chest	16	144
Each unused death	10	30
Not quitting	4	4
Reaching Witt's End	1	1
Getting to *closing*	25	25
Various additional bonuses		45
Round out the total	2	2
	Total:	350

Points can also be deducted for using hints. One consequence of these rules is that you get 32 points just for quitting on your first turn. And there's a way to get 57 points in just three turns.

Full points for treasures are awarded only if they aren't broken and you have deposited them in the building. But we give you 2 points just for seeing a treasure.

#define *max_score* 350

⟨ Global variables 7 ⟩ +≡
```
int bonus;   /* extra points awarded for exceptional adventuring skills */
```

194. The hints are table driven, using several arrays:

- *hint_count*[*j*] is the number of recent turns whose location is relevant to hint *j*;
- *hint_thresh*[*j*] is the number of such turns before we consider offering that hint;
- *hint_cost*[*j*] is the number of points you pay for it;
- *hint_prompt*[*j*] is the way we offer it;
- *hint*[*j*] is the hint;
- *hinted*[*j*] is true after we've given it.

Hint 0 is for instructions at the beginning; it costs you 5 points, but you get extra power in the lamp. The other hints also usually extend the lamp's power. Hint 1 is for reading the oyster. And hints 2 through 7 are for the *cave_hint*, *bird_hint*, *snake_hint*, *twist_hint*, *dark_hint*, and *witt_hint*, respectively.

Here's the subroutine that handles all eight kinds of hints.

⟨ Subroutines 6 ⟩ +≡
```
void offer ARGS((int));
void offer(j)
    int j;
{
  if (j > 1) {
    if (¬yes(hint_prompt[j], "␣I␣am␣prepared␣to␣give␣you␣a␣hint,", ok)) return;
    printf("␣but␣it␣will␣cost␣you␣%d␣points.␣␣", hint_cost[j]);
    hinted[j] = yes("Do␣you␣want␣the␣hint?", hint[j], ok);
  } else hinted[j] = yes(hint_prompt[j], hint[j], ok);
  if (hinted[j] ∧ limit > 30) limit += 30 * hint_cost[j];
}
```

195. ⟨ Check if a hint applies, and give it if requested 195 ⟩ ≡

```
for (j = 2, k = cave_hint; j ≤ 7; j++, k += k)
  if (¬hinted[j]) {
    if ((flags[loc] & k) ≡ 0) hint_count[j] = 0;
    else if (++hint_count[j] ≥ hint_thresh[j]) {
      switch (j) {
      case 2: if (prop[GRATE] ≡ 0 ∧ ¬here(KEYS)) break; else goto bypass;
      case 3: if (here(BIRD) ∧ oldobj ≡ BIRD ∧ toting(ROD)) break;
        else continue;
      case 4: if (here(SNAKE) ∧ ¬here(BIRD)) break; else goto bypass;
      case 5: if (first[loc] ≡ 0 ∧ first[oldloc] ≡ 0 ∧ first[oldoldloc] ≡ 0 ∧ holding > 1) break;
        else goto bypass;
      case 6: if (prop[EMERALD] ≠ −1 ∧ prop[PYRAMID] ≡ −1) break;
        else goto bypass;
      case 7: break;
      }
      offer(j);
    bypass: hint_count[j] = 0;
    }
  }
```

This code is used in section 76.

196. **#define** *n_hints* 8

⟨ Global variables 7 ⟩ +≡

```
int hint_count[n_hints];   /* how long you have needed this hint */
int hint_thresh[n_hints] = {0, 0, 4, 5, 8, 75, 25, 20};   /* how long we will wait */
int hint_cost[n_hints] = {5, 10, 2, 2, 2, 4, 5, 3};   /* how much we will charge */
char *hint_prompt[n_hints] = {
"Welcome␣to␣Adventure!!␣␣Would␣you␣like␣instructions?",
"Hmmm,␣this␣looks␣like␣a␣clue,␣which␣means␣it'll␣cost␣you␣10␣points␣to\n\
    read␣it.␣␣Should␣I␣go␣ahead␣and␣read␣it␣anyway?",
"Are␣you␣trying␣to␣get␣into␣the␣cave?",
"Are␣you␣trying␣to␣catch␣the␣bird?",
"Are␣you␣trying␣to␣deal␣somehow␣with␣the␣snake?",
"Do␣you␣need␣help␣getting␣out␣of␣the␣maze?",
"Are␣you␣trying␣to␣explore␣beyond␣the␣Plover␣Room?",
"Do␣you␣need␣help␣getting␣out␣of␣here?"};
char *hint[n_hints] = {
"Somewhere␣nearby␣is␣Colossal␣Cave,␣where␣others␣have␣found␣fortunes␣in\n\
    treasure␣and␣gold,␣though␣it␣is␣rumored␣that␣some␣who␣enter␣are␣never\n\
    seen␣again.␣␣Magic␣is␣said␣to␣work␣in␣the␣cave.␣␣I␣will␣be␣your␣eyes\n\
    and␣hands.␣␣Direct␣me␣with␣commands␣of␣one␣or␣two␣words.␣␣I␣should\n\
    warn␣you␣that␣I␣look␣at␣only␣the␣first␣five␣letters␣of␣each␣word,␣so\n\
    you'll␣have␣to␣enter␣\"NORTHEAST\"␣as␣\"NE\"␣to␣distinguish␣it␣from\n\
    \"NORTH\".␣␣Should␣you␣get␣stuck,␣type␣\"HELP\"␣for␣some␣general␣hints.\n\
    For␣information␣on␣how␣to␣end␣your␣adventure,␣etc.,␣type␣\"INFO\".\n\
    ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣-␣␣-␣␣-\n\
    The␣first␣adventure␣program␣was␣developed␣by␣Willie␣Crowther.\n\
    Most␣of␣the␣features␣of␣the␣current␣program␣were␣added␣by␣Don␣Woods;\n\
    all␣of␣its␣bugs␣were␣added␣by␣Don␣Knuth.",
"It␣says,␣\"There␣is␣something␣strange␣about␣this␣place,␣such␣that␣one\n\
    of␣the␣words␣I've␣always␣known␣now␣has␣a␣new␣effect.\"",
"The␣grate␣is␣very␣solid␣and␣has␣a␣hardened␣steel␣lock.␣␣You␣cannot\n\
    enter␣without␣a␣key,␣and␣there␣are␣no␣keys␣in␣sight.␣␣I␣would␣recommend\n\
    looking␣elsewhere␣for␣the␣keys.",
"Something␣seems␣to␣be␣frightening␣the␣bird␣just␣now␣and␣you␣cannot\n\
    catch␣it␣no␣matter␣what␣you␣try.␣␣Perhaps␣you␣might␣try␣later.",
"You␣can't␣kill␣the␣snake,␣or␣drive␣it␣away,␣or␣avoid␣it,␣or␣anything\n\
    like␣that.␣␣There␣is␣a␣way␣to␣get␣by,␣but␣you␣don't␣have␣the␣necessary\n\
    resources␣right␣now.",
"You␣can␣make␣the␣passages␣look␣less␣alike␣by␣dropping␣things.",
"There␣is␣a␣way␣to␣explore␣that␣region␣without␣having␣to␣worry␣about\n\
    falling␣into␣a␣pit.␣␣None␣of␣the␣objects␣available␣is␣immediately\n\
    useful␣for␣discovering␣the␣secret.",
"Don't␣go␣west."};
boolean hinted[n_hints];   /* have you seen the hint? */
```

197. Here's a subroutine that computes the current score.

⟨ Subroutines 6 ⟩ +≡
```
int score ARGS((void));
int score()
{
  register int j, s = 2;
  register object k;
  if (dflag) s += 25;   /* you've gotten well inside */
  for (k = min_treasure; k ≤ max_obj; k++) {
    if (prop[k] ≥ 0) {
      s += 2;
      if (place[k] ≡ house ∧ prop[k] ≡ 0) s += (k < CHEST ? 10 : k ≡ CHEST ? 12 : 14);
    }
  }
  s += 10 * (max_deaths − death_count);
  if (¬gave_up) s += 4;
  if (place[MAG] ≡ witt) s++;   /* proof of your visit */
  if (closing) s += 25;
  s += bonus;
  for (j = 0; j < n_hints; j++)
    if (hinted[j]) s −= hint_cost[j];
  return s;
}
```

198. **#define** *highest_class* 8

⟨ Print the score and say adieu 198 ⟩ ≡
```
k = score();
printf("You␣scored␣%d␣points␣out␣of␣a␣possible␣%d,␣using␣%d␣turn%s.\n", k, max_score, turns,
    turns ≡ 1 ? "" : "s");
for (j = 0; class_score[j] ≤ k; j++) ;
printf("%s\nTo␣achieve␣the␣next␣higher␣rating", class_message[j]);
if (j < highest_class)
  printf(",␣you␣need␣%d␣more␣point%s.\n", class_score[j] − k, class_score[j] ≡ k + 1 ? "" : "s");
else printf("␣would␣be␣a␣neat␣trick!\nCongratulations!!\n");
```
This code is used in section 2.

199. ⟨ Global variables 7 ⟩ +≡
```
int class_score[] = {35, 100, 130, 200, 250, 300, 330, 349, 9999};
char *class_message[] = {
"You␣are␣obviously␣a␣rank␣amateur.␣␣Better␣luck␣next␣time.",
"Your␣score␣qualifies␣you␣as␣a␣novice␣class␣adventurer.",
"You␣have␣achieved␣the␣rating␣\"Experienced␣Adventurer\".",
"You␣may␣now␣consider␣yourself␣a␣\"Seasoned␣Adventurer\".",
"You␣have␣reached␣\"Junior␣Master\"␣status.",
"Your␣score␣puts␣you␣in␣Master␣Adventurer␣Class␣C.",
"Your␣score␣puts␣you␣in␣Master␣Adventurer␣Class␣B.",
"Your␣score␣puts␣you␣in␣Master␣Adventurer␣Class␣A.",
"All␣of␣Adventuredom␣gives␣tribute␣to␣you,␣Adventure␣Grandmaster!"};
```

200. Launching the program. The program is now complete; all we must do is put a few of the pieces together.

Most of the initialization takes place while you are reading the opening message.

⟨ Initialize all tables 200 ⟩ ≡
  ⟨ Initialize the random number generator 156 ⟩;
  offer(0); /* Give the welcome message and possible instructions */
  limit = (*hinted*[0] ? 1000 : 330); /* set lifetime of lamp */
  ⟨ Build the vocabulary 10 ⟩;
  ⟨ Build the travel table 23 ⟩;
  ⟨ Build the object tables 69 ⟩;
  oldoldloc = *oldloc* = *loc* = *newloc* = *road*;

This code is used in section 2.

201. Index. A large cloud of green smoke appears in front of you. It clears away to reveal a tall wizard, clothed in grey. He fixes you with a steely glare and declares, "This adventure has lasted too long." With that he makes a single pass over you with his hands, and everything around you fades away into a grey nothingness.

ADVENTURE