1. Introduction. This program reads a binary `mmo` file output by the `MMIXAL` processor and lists it in human-readable form. It lists only the symbol table, if invoked with the `-s` option. It lists also the tetrabytes of input, if invoked with the `-v` option.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>
  ⟨ Prototype preparations 5 ⟩
  ⟨ Type definitions 7 ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 8 ⟩
  int main(argc, argv)
      int argc; char *argv[];
  {
    register int j, delta, postamble = 0;
    register char *p;
    ⟨ Process the command line 2 ⟩;
    ⟨ Initialize everything 3 ⟩;
    ⟨ List the preamble 23 ⟩;
    do ⟨ List the next item 13 ⟩ while (¬postamble);
    ⟨ List the postamble 24 ⟩;
    ⟨ List the symbol table 25 ⟩;
    return 0;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
  listing = 1, verbose = 0;
  for (j = 1; j < argc − 1 ∧ argv[j][0] ≡ '-' ∧ argv[j][2] ≡ '\0'; j++) {
    if (argv[j][1] ≡ 's') listing = 0;
    else if (argv[j][1] ≡ 'v') verbose = 1;
    else break;
  }
  if (j ≠ argc − 1) {
    fprintf(stderr, "Usage: %s [-s] [-v] mmofile\n", argv[0]);
    exit(−1);
  }
```

This code is used in section 1.

3. ⟨ Initialize everything 3 ⟩ ≡

```
  mmo_file = fopen(argv[argc − 1], "rb");
  if (¬mmo_file) {
    fprintf(stderr, "Can't open file %s!\n", argv[argc − 1]);
    exit(−2);
  }
```

See also sections 12 and 17.

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
int listing;      /* are we listing everything? */
int verbose;      /* are we also showing the tetras of input as they are read? */
FILE *mmo_file;   /* the input file */
```

See also sections 11, 16, and 29.

This code is used in section 1.

5. ⟨ Prototype preparations 5 ⟩ ≡

```
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
```

This code is used in section 1.

6. A complete definition of `mmo` format appears in the `MMIXAL` document. Here we need to define only the basic constants used for interpretation.

```
#define mm #98      /* the escape code of mmo format */
#define lop_quote #0      /* the quotation lopcode */
#define lop_loc #1      /* the location lopcode */
#define lop_skip #2      /* the skip lopcode */
#define lop_fixo #3      /* the octabyte-fix lopcode */
#define lop_fixr #4      /* the relative-fix lopcode */
#define lop_fixrx #5      /* extended relative-fix lopcode */
#define lop_file #6      /* the file name lopcode */
#define lop_line #7      /* the file position lopcode */
#define lop_spec #8      /* the special hook lopcode */
#define lop_pre #9      /* the preamble lopcode */
#define lop_post #a      /* the postamble lopcode */
#define lop_stab #b      /* the symbol table lopcode */
#define lop_end #c      /* the end-it-all lopcode */
```

7. Low-level arithmetic. This program is intended to work correctly whenever an **int** has at least 32 bits.

⟨ Type definitions 7 ⟩ ≡

```
typedef unsigned char byte;      /* a monobyte */
typedef unsigned int tetra;      /* a tetrabyte */
typedef struct { tetra h, l;
} octa;      /* an octabyte */
```

This code is used in section 1.

8. The *incr* subroutine adds a signed integer to an (unsigned) octabyte.

⟨ Subroutines 8 ⟩ ≡

```
octa incr ARGS((octa, int));
octa incr(o, delta)
    octa o;
    int delta;
{
  register tetra t;
  octa x;
  if (delta ≥ 0) {
    t = #ffffffff − delta;
    if (o.l ≤ t) x.l = o.l + delta, x.h = o.h;
    else x.l = o.l − t − 1, x.h = o.h + 1;
  }
  else {
    t = −delta;
    if (o.l ≥ t) x.l = o.l − t, x.h = o.h;
    else x.l = o.l + (#ffffffff + delta) + 1, x.h = o.h − 1;
  }
  return x;
}
```

See also sections 9, 10, and 26.

This code is used in section 1.

9. Low-level input. The tetrabytes of an `mmo` file are stored in friendly big-endian fashion, but this program is supposed to work also on computers that are little-endian. Therefore we read four successive bytes and pack them into a tetrabyte, instead of reading a single tetrabyte.

```
⟨ Subroutines 8 ⟩ +≡
  void read_tet ARGS((void));
  void read_tet()
  {
    if (fread(buf, 1, 4, mmo_file) ≠ 4) {
      fprintf(stderr, "Unexpected␣end␣of␣file␣after␣%d␣tetras!\n", count);
      exit(−3);
    }
    yz = (buf[2] ≪ 8) + buf[3];
    tet = (((buf[0] ≪ 8) + buf[1]) ≪ 16) + yz;
    if (verbose) printf("␣␣%08x\n", tet);
    count++;
  }
```

10.
```
⟨ Subroutines 8 ⟩ +≡
  byte read_byte ARGS((void));
  byte read_byte()
  {
    register byte b;
    if (¬byte_count) read_tet();
    b = buf[byte_count];
    byte_count = (byte_count + 1) & 3;
    return b;
  }
```

11.
```
⟨ Global variables 4 ⟩ +≡
  int count;      /* the number of tetrabytes we've read */
  int byte_count;      /* index of the next-to-be-read byte */
  byte buf[4];      /* the most recently read bytes */
  int yz;      /* the two least significant bytes */
  tetra tet;      /* buf bytes packed big-endianwise */
```

12.
```
⟨ Initialize everything 3 ⟩ +≡
  count = byte_count = 0;
```

13. The main loop. Now for the bread-and-butter part of this program.

```
⟨ List the next item 13 ⟩ ≡
  {
    read_tet();
  loop: if (buf[0] ≡ mm)
      switch (buf[1]) {
      case lop_quote: if (yz ≠ 1) err("YZ␣field␣of␣lop_quote␣should␣be␣1");
        read_tet(); break;
      ⟨ Cases for lopcodes in the main loop 18 ⟩
      default: err("Unknown␣lopcode");
      }
    if (listing) ⟨ List tet as a normal item 15 ⟩;
  }
```

This code is used in section 1.

14. We want to catch all cases where the rules of `mmo` format are not obeyed. The *err* macro ameliorates this somewhat tedious chore.

```
#define err(m)
        { fprintf(stderr, "Error␣in␣tetra␣%d:␣%s!\n", count, m); continue; }
```

15. In a normal situation, the newly read tetrabyte is simply supposed to be loaded into the current location. We list not only the current location but also the current file position, if *cur_line* is nonzero and *cur_loc* belongs to segment 0.

```
⟨ List tet as a normal item 15 ⟩ ≡
  {
    printf("%08x%08x:␣%08x", cur_loc.h, cur_loc.l, tet);
    if (¬cur_line) printf("\n");
    else {
      if (cur_loc.h & #e0000000) printf("\n");
      else {
        if (cur_file ≡ listed_file) printf("␣(line␣%d)\n", cur_line);
        else {
          printf("␣(\"%s\",␣line␣%d)\n", file_name[cur_file], cur_line);
          listed_file = cur_file;
        }
      }
      cur_line++;
    }
    cur_loc = incr(cur_loc, 4); cur_loc.l &= −4;
  }
```

This code is used in section 13.

16.

```
⟨ Global variables 4 ⟩ +≡
  octa cur_loc;      /* the current location */
  int listed_file;   /* the most recently listed file number */
  int cur_file;      /* the most recently selected file number */
  int cur_line;      /* the current position in cur_file */
  char *file_name[256];   /* file names seen */
  octa tmp;      /* an octabyte of temporary interest */
```

17. ⟨ Initialize everything 3 ⟩ +≡

```
cur_loc.h = cur_loc.l = 0;
listed_file = cur_file = -1;
cur_line = 0;
```

18. The simple lopcodes. We have already implemented *lop_quote*, which falls through to the normal case after reading an extra tetrabyte. Now let's consider the other lopcodes in turn.

```
#define y buf[2]    /* the next-to-least significant byte */
#define z buf[3]    /* the least significant byte */
⟨ Cases for lopcodes in the main loop 18 ⟩ ≡
case lop_loc: if (z ≡ 2) {
    j = y; read_tet(); cur_loc.h = (j ≪ 24) + tet;
  } else if (z ≡ 1) cur_loc.h = y ≪ 24;
  else err("Z␣field␣of␣lop_loc␣should␣be␣1␣or␣2");
  read_tet(); cur_loc.l = tet;
  continue;
case lop_skip: cur_loc = incr(cur_loc, yz); continue;
```

See also sections 19, 20, 21, and 22.

This code is used in section 13.

19. Fixups load information out of order, when future references have been resolved. The current file name and line number are not considered relevant.

```
⟨ Cases for lopcodes in the main loop 18 ⟩ +≡
case lop_fixo: if (z ≡ 2) {
    j = y; read_tet(); tmp.h = (j ≪ 24) + tet;
  } else if (z ≡ 1) tmp.h = y ≪ 24;
  else err("Z␣field␣of␣lop_fixo␣should␣be␣1␣or␣2");
  read_tet(); tmp.l = tet;
  if (listing) printf("%08x%08x:␣%08x%08x\n", tmp.h, tmp.l, cur_loc.h, cur_loc.l);
  continue;
case lop_fixr: delta = yz;
  goto fixr;
case lop_fixrx: j = yz; if (j ≠ 16 ∧ j ≠ 24) err("YZ␣field␣of␣lop_fixrx␣should␣be␣16␣or␣24");
  read_tet();
  delta = tet;
  if (delta & #fe000000) err("increment␣of␣lop_fixrx␣is␣too␣large");
fixr: tmp = incr(cur_loc, −(delta ≥ #1000000 ? (delta & #ffffff) − (1 ≪ j) : delta) ≪ 2);
  if (listing) printf("%08x%08x:␣%08x\n", tmp.h, tmp.l, delta);
  continue;
```

20. The space for file names isn't allocated until we are sure we need it.

```
⟨ Cases for lopcodes in the main loop 18 ⟩ +≡
case lop_file: if (file_name[y]) {
    for (j = z; j > 0; j--) read_tet();
    cur_file = y;
    if (z) err("Two␣file␣names␣with␣the␣same␣number");
  } else {
    if (¬z) err("No␣name␣given␣for␣newly␣selected␣file");
    file_name[y] = (char *) calloc(4 * z + 1, 1);
    if (¬file_name[y]) {
      fprintf(stderr, "No␣room␣to␣store␣the␣file␣name!\n"); exit(−4);
    }
    cur_file = y;
    for (j = z, p = file_name[y]; j > 0; j--, p += 4) {
      read_tet();
      *p = buf[0]; *(p + 1) = buf[1]; *(p + 2) = buf[2]; *(p + 3) = buf[3];
    }
  }
  cur_line = 0; continue;
case lop_line: if (cur_file < 0) err("No␣file␣was␣selected␣for␣lop_line");
  cur_line = yz; continue;
```

21. Special bytes in the file might be in synch with the current location and/or the current file position, so we list those parameters too.

```
⟨ Cases for lopcodes in the main loop 18 ⟩ +≡
case lop_spec: if (listing) {
    printf("Special␣data␣%d␣at␣loc␣%08x%08x", yz, cur_loc.h, cur_loc.l);
    if (¬cur_line) printf("\n");
    else if (cur_file ≡ listed_file) printf("␣(line␣%d)\n", cur_line);
    else {
      printf("␣(\"%s\",␣line␣%d)\n", file_name[cur_file], cur_line);
      listed_file = cur_file;
    }
  }
  while (1) {
    read_tet();
    if (buf[0] ≡ mm) {
      if (buf[1] ≠ lop_quote ∨ yz ≠ 1) goto loop;      /* end of special data */
      read_tet();
    }
    if (listing) printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%08x\n", tet);
  }
```

22. The other cases shouldn't appear in the main loop.

```
⟨ Cases for lopcodes in the main loop 18 ⟩ +≡
case lop_pre: err("Can't␣have␣another␣preamble");
case lop_post: postamble = 1;
  if (y) err("Y␣field␣of␣lop_post␣should␣be␣zero");
  if (z < 32) err("Z␣field␣of␣lop_post␣must␣be␣32␣or␣more");
  continue;
case lop_stab: err("Symbol␣table␣must␣follow␣postamble");
case lop_end: err("Symbol␣table␣can't␣end␣before␣it␣begins");
```

23. The preamble and postamble. Now here's what we do before and after the main loop.

```
⟨ List the preamble 23 ⟩ ≡
  read_tet( );      /* read the first tetrabyte of input */
  if (buf[0] ≠ mm ∨ buf[1] ≠ lop_pre) {
    fprintf(stderr, "Input is not an MMO file (first two bytes are wrong)!\n");
    exit(−5);
  }
  if (y ≠ 1)
    fprintf(stderr, "Warning: I'm reading this file as version 1, not version %d!\n", y);
  if (z > 0) {
    j = z;
    read_tet( );
    if (listing) {
      time_t t = tet;
      printf("File was created %s", asctime(localtime(&t)));
    }
    for (j−−; j > 0; j−−) {
      read_tet( );
      if (listing) printf("Preamble data %08x\n", tet);
    }
  }
```

This code is used in section 1.

24.

```
⟨ List the postamble 24 ⟩ ≡
  for (j = z; j < 256; j++) {
    read_tet( ); tmp.h = tet; read_tet( );
    if (listing) {
      if (tmp.h ∨ tet) printf("g%03d: %08x%08x\n", j, tmp.h, tet);
      else printf("g%03d: 0\n", j);
    }
  }
```

This code is used in section 1.

25. The symbol table. Finally we come to the symbol table, which is the most interesting part of this program because it recursively traces an implicit ternary trie structure.

```
⟨ List the symbol table 25 ⟩ ≡
  read_tet();
  if (buf[0] ≠ mm ∨ buf[1] ≠ lop_stab) {
    fprintf(stderr, "Symbol table does not follow the postamble!\n");
    exit(-6);
  }
  if (yz) fprintf(stderr, "YZ field of lop_stab should be zero!\n");
  printf("Symbol table (beginning at tetra %d):\n", count);
  stab_start = count;
  sym_ptr = sym_buf;
  print_stab();
  ⟨ Check the lop_end 30 ⟩;
```

This code is used in section 1.

26. The main work is done by a recursive subroutine called *print_stab*, which manipulates a global array *sym_buf* containing the current symbol prefix; the global variable *sym_ptr* points to the first unfilled character of that array.

```
⟨ Subroutines 8 ⟩ +≡
  void print_stab ARGS((void));
  void print_stab()
  {
    register int m = read_byte();      /* the master control byte */
    register int c;      /* the character at the current trie node */
    register int j, k;
    if (m & #40) print_stab();      /* traverse the left subtrie, if it is nonempty */
    if (m & #2f) {
      ⟨ Read the character c 27 ⟩;
      *sym_ptr++ = c;
      if (sym_ptr ≡ &sym_buf[sym_length_max]) {
        fprintf(stderr, "Oops, the symbol is too long!\n"); exit(-7);
      }
      if (m & #f) ⟨ Print the current symbol with its equivalent and serial number 28 ⟩;
      if (m & #20) print_stab();      /* traverse the middle subtrie */
      sym_ptr--;
    }
    if (m & #10) print_stab();      /* traverse the right subtrie, if it is nonempty */
  }
```

27. The present implementation doesn't support Unicode; characters with more than 8-bit codes are printed as '?'. However, the changes for 16-bit codes would be quite easy if proper fonts for Unicode output were available. In that case, *sym_buf* would be an array of wyde characters.

```
⟨ Read the character c 27 ⟩ ≡
  if (m & #80) j = read_byte();      /* 16-bit character */
  else j = 0;
  c = read_byte();
  if (j) c = '?';      /* oops, we can't print (j ≪ 8) + c easily at this time */
```

This code is used in section 26.

28. ⟨ Print the current symbol with its equivalent and serial number 28 ⟩ ≡

```
{
  *sym_ptr = '\0';
  j = m & #f;
  if (j ≡ 15) sprintf(equiv_buf, "$%03d", read_byte());
  else if (j ≤ 8) {
    strcpy(equiv_buf, "#");
    for ( ; j > 0; j--) sprintf(equiv_buf + strlen(equiv_buf), "%02x", read_byte());
    if (strcmp(equiv_buf, "#0000") ≡ 0) strcpy(equiv_buf, "?");      /* undefined */
  } else {
    strncpy(equiv_buf, "#20000000000000", 33 − 2 * j);
    equiv_buf[33 − 2 * j] = '\0';
    for ( ; j > 8; j--) sprintf(equiv_buf + strlen(equiv_buf), "%02x", read_byte());
  }
  for (j = k = read_byte(); ; k = read_byte(), j = (j ≪ 7) + k)
    if (k ≥ 128) break;      /* the serial number is now j − 128 */
  printf("␣␣␣␣%s␣=␣%s␣(%d)\n", sym_buf + 1, equiv_buf, j − 128);
}
```

This code is used in section 26.

29. **#define** *sym_length_max* 1000

⟨ Global variables 4 ⟩ +≡

```
int stab_start;      /* where the symbol table began */
char sym_buf[sym_length_max];      /* the characters on middle transitions to current node */
char *sym_ptr;      /* the character in sym_buf following the current prefix */
char equiv_buf[20];      /* equivalent of the current symbol */
```

30. ⟨ Check the *lop_end* 30 ⟩ ≡

```
while (byte_count)
  if (read_byte()) fprintf(stderr, "Nonzero␣byte␣follows␣the␣symbol␣table!\n");
read_tet();
if (buf[0] ≠ mm ∨ buf[1] ≠ lop_end)
  fprintf(stderr, "The␣symbol␣table␣isn't␣followed␣by␣lop_end!\n");
else if (count − stab_start − 1 ≠ yz)
  fprintf(stderr, "YZ␣field␣at␣lop_end␣should␣have␣been␣%d!\n", count − stab_start − 1);
else {
  if (verbose) printf("Symbol␣table␣ends␣at␣tetra␣%d.\n", count);
  if (fread(buf, 1, 1, mmo_file)) fprintf(stderr, "Extra␣bytes␣follow␣the␣lop_end!\n");
}
```

This code is used in section 25.

31. Index.

⟨ Cases for lopcodes in the main loop 18, 19, 20, 21, 22 ⟩ Used in section 13.
⟨ Check the *lop_end* 30 ⟩ Used in section 25.
⟨ Global variables 4, 11, 16, 29 ⟩ Used in section 1.
⟨ Initialize everything 3, 12, 17 ⟩ Used in section 1.
⟨ List the next item 13 ⟩ Used in section 1.
⟨ List the postamble 24 ⟩ Used in section 1.
⟨ List the preamble 23 ⟩ Used in section 1.
⟨ List the symbol table 25 ⟩ Used in section 1.
⟨ List *tet* as a normal item 15 ⟩ Used in section 13.
⟨ Print the current symbol with its equivalent and serial number 28 ⟩ Used in section 26.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ Prototype preparations 5 ⟩ Used in section 1.
⟨ Read the character *c* 27 ⟩ Used in section 26.
⟨ Subroutines 8, 9, 10, 26 ⟩ Used in section 1.
⟨ Type definitions 7 ⟩ Used in section 1.

January 30, 2023 at 17:35

MMOTYPE