1.

```
#define zero_t ( ( math_data * ) mp->math ) -> zero_t
#define number_zero(A) ( ( ( math_data * ) (mp->math) ) -> equal ) (A, zero_t)
#define number_greater(A, B) ( ( ( math_data * ) (mp->math) ) -> greater ) (A, B)
#define number_positive(A) number_greater(A, zero_t)
#define number_to_scaled(A) ( ( ( math_data * ) (mp->math) ) -> to_scaled ) (A)
#define round_unscaled(A) ( ( ( math_data * ) (mp->math) ) -> round_unscaled ) (A)
#define true 1
#define false 0
#define null_font 0
#define null 0
#define unity 1.0 /* 2^16, represents 1.00000 */
#define incr(A) (A) = (A) + 1 /* increase a variable by unity */
#define decr(A) (A) = (A) - 1 /* decrease a variable by unity */
#define negate(A) (A) = -(A) /* change the sign of a variable */
#define odd(A) (abs(A) % 2 == 1)
#define max_quarterword #3FFF /* largest allowable value in a quarterword */
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <assert.h>
#include <math.h>
#include "avl.h"
#include "mplib.h"
#include "mplibps.h" /* external header */
#include "mpmp.h" /* internal header */
#include "mppsout.h" /* internal header */
#include "mpmath.h" /* internal header */
#include "mpstrings.h" /* internal header */
⟨ Preprocessor definitions ⟩
⟨ Declarations 29 ⟩⟨ Static variables in the outer block 24 ⟩
```

2. There is a small bit of code from the backend that bleads through to the frontend because I do not know how to set up the includes properly. That is the **typedef struct** *psout_data_struct* ***psout_data**.

3. ⟨ mppsout.h 3 ⟩ ≡

```
#ifndef MPPSOUT_H
#define MPPSOUT_H 1
#include "avl.h"
#include "mplib.h"
#include "mpmp.h"
#include "mplibps.h"
⟨ Types 18 ⟩
    typedef struct psout_data_struct {
      ⟨ Globals 7 ⟩
    } psout_data_struct; ⟨ Exported function headers 5 ⟩
#endif
```

```
4. static boolean mp_isdigit(int a)
{
  return (a ≥ '0' ∧ a ≤ '9');
}
static int mp_tolower(int a)
{
  if (a ≥ 'A' ∧ a ≤ 'Z') return a − 'A' + 'a';
  return a;
}
static int mp_strcasecmp(const char *s1, const char *s2)
{
  int r;
  char *ss1, *ss2, *c;
  ss1 = mp_strdup(s1);
  c = ss1;
  while (*c ≠ '\0') {
    *c = (char) mp_tolower(*c);
    c++;
  }
  ss2 = mp_strdup(s2);
  c = ss2;
  while (*c ≠ '\0') {
    *c = (char) mp_tolower(*c);
    c++;
  }
  r = strcmp(ss1, ss2);
  free(ss1);
  free(ss2);
  return r;
}
```

```
5. ⟨ Exported function headers 5 ⟩ ≡
void mp_ps_backend_initialize(MP mp);
void mp_ps_backend_free(MP mp);
```

See also sections 55, 59, 65, 72, 104, 106, 109, 189, and 233.

This code is used in section 3.

6.

```
void mp_ps_backend_initialize(MP mp)
{
  mp→ps = mp_xmalloc(mp, 1, sizeof(psout_data_struct));
  memset(mp→ps, 0, sizeof(psout_data_struct));
  ⟨ Set initial values 8 ⟩;
}
void mp_ps_backend_free(MP mp)
{
  ⟨ Dealloc variables 62 ⟩;
  enc_free(mp);
  t1_free(mp);
  fm_free(mp);
  mp_xfree(mp→ps);
  mp→ps = Λ;
}
```

7. Writing to ps files

```
⟨ Globals 7 ⟩ ≡
  integer ps_offset;      /* the number of characters on the current PostScript file line */
```

See also sections 19, 23, 33, 37, 43, 69, 75, 79, 82, 84, 87, 116, and 192.

This code is used in section 3.

8. ⟨ Set initial values 8 ⟩ ≡

```
  mp→ps→ps_offset = 0;
```

See also sections 25, 34, 38, 44, 70, 76, 85, 88, 94, and 193.

This code is used in section 6.

9.

```
#define wps(A) (mp→write_ascii_file)(mp, mp→output_file, (A))
#define wps_chr(A) do
        {
          char ss[2];
          ss[0] = (char)(A);
          ss[1] = 0;
          (mp→write_ascii_file)(mp, mp→output_file, (char *) ss);
        }
        while (0)
#define wps_cr (mp→write_ascii_file)(mp, mp→output_file, "\n")
#define wps_ln(A)
        {
          wterm_cr;
          (mp→write_ascii_file)(mp, mp→output_file, (A));
        }
static void mp_ps_print_ln(MP mp)
{     /* prints an end-of-line */
  wps_cr;
  mp→ps→ps_offset = 0;
}
```

10. **static void** *mp_ps_print_char*(MP *mp*, **int** *s*)

```
{
  /* prints a single character */
  if (s ≡ 13) {
    wps_cr;
    mp→ps→ps_offset = 0;
  }
  else {
    wps_chr(s);
    incr(mp→ps→ps_offset);
  }
}
```

11. **static void** *mp_ps_do_print*(MP *mp*, **const char** ∗*ss*, **size_t** *len*)

```
{
  /* prints string s */
  size_t j = 0;
  if (len > 255) {
    while (j < len) {
      mp_ps_print_char(mp, ss[j]);
      incr(j);
    }
  }
  else {
    static char outbuf[256];
    strncpy(outbuf, ss, len + 1);
    while (j < len) {
      if (∗(outbuf + j) ≡ 13) {
        ∗(outbuf + j) = '\n';
        mp→ps→ps_offset = 0;
      }
      else {
        mp→ps→ps_offset++;
      }
      j++;
    }
    (mp→write_ascii_file)(mp, mp→output_file, outbuf);
  }
}
```

12. Deciding where to break the ps output line.

```
#define ps_room(A)
        if (mp→ps→ps_offset > 0 ∧ (mp→ps→ps_offset + (int)(A)) > mp→max_print_line) {
          mp_ps_print_ln(mp);      /* optional line break */
        }
  static void mp_ps_print(MP mp, const char *ss)
  {
    ps_room(strlen(ss));
    mp_ps_do_print(mp, ss, strlen(ss));
  }
  static void mp_ps_dsc_print(MP mp, const char *dsc, const char *ss)
  {
    ps_room(strlen(ss));
    if (mp→ps→ps_offset ≡ 0) {
      mp_ps_do_print(mp, "%%+␣", 4);
      mp_ps_do_print(mp, dsc, strlen(dsc));
      mp_ps_print_char(mp, '␣');
    }
    mp_ps_do_print(mp, ss, strlen(ss));
  }
```

13. The procedure *print_nl* is like *print*, but it makes sure that the string appears at the beginning of a new line.

```
  static void mp_ps_print_nl(MP mp, const char *s)
  {     /* prints string s at beginning of line */
    if (mp→ps→ps_offset > 0) mp_ps_print_ln(mp);
    mp_ps_print(mp, s);
  }
```

14. The following procedure, which prints out the decimal representation of a given integer n, has been written carefully so that it works properly if $n = 0$ or if $(-n)$ would cause overflow. It does not apply *mod* or *div* to negative arguments, since such operations are not implemented consistently by all Pascal compilers.

```
static void mp_ps_print_int(MP mp, integer n)
{   /* prints an integer in decimal form */
  integer m;   /* used to negate n in possibly dangerous cases */
  char outbuf[24];   /* dig[23], plus terminating T0 */
  unsigned char dig[23];   /* digits in a number, for rounding */
  int k = 0;   /* index to current digit; we assume that n < 10^23 */
  int l = 0;
  if (n < 0) {
    mp_ps_print_char(mp, '-');
    if (n > -100000000) {
      negate(n);
    }
    else {
      m = -1 - n;
      n = m/10;
      m = (m % 10) + 1;
      k = 1;
      if (m < 10) {
        dig[0] = (unsigned char) m;
      }
      else {
        dig[0] = 0;
        incr(n);
      }
    }
  }
  do {
    dig[k] = (unsigned char)(n % 10);
    n = n/10;
    incr(k);
  } while (n ≠ 0);   /* print the digits */
  while (k-- > 0) {
    outbuf[l++] = (char)('0' + dig[k]);
  }
  outbuf[l] = '\0';
  (mp→write_ascii_file)(mp, mp→output_file, outbuf);
}
```

15. METAPOST also makes use of a trivial procedure to print two digits. The following subroutine is usually called with a parameter in the range $0 \le n \le 99$.

```
static void mp_ps_print_dd(MP mp, integer n)
{   /* prints two least significant digits */
  n = abs(n) % 100;
  mp_ps_print_char(mp, '0' + (n/10));
  mp_ps_print_char(mp, '0' + (n % 10));
}
```

16. Conversely, here is a procedure analogous to *print_int*.

There are two versions of this function: *ps_print_double_scaled* is used if metapost runs in scaled (backward compatibility) mode, because that version produces results that are much closer to the old version that exported figures with scaled fields instead of double fields. It is not always the same because a little bit of precision has gone in the scaled to double conversion, but still quite a bit closer than % $.6_{\rm F}$ in the 'double' case.

```
#define unityold 65536
  static void mp_ps_print_double_new(MP mp, double s)
  {
    char *value, *c;
    int i;
    value = mp_xmalloc(mp, 1, 32);
    memset(value, 0, 32);
    mp_snprintf(value, 32, "%.6f", s);
    for (i = 31; i ≥ 0; i--) {
      if (value[i]) {
        if (value[i] ≡ '0') value[i] = '\0';
        else break;
      }
    }
    if (value[i] ≡ '.') value[i] = '\0';
    c = value;
    while (*c) {
      mp_ps_print_char(mp, *c);
      c++;
    }
    free(value);
  }
  static void mp_ps_print_double_scaled(MP mp, double ss)
  {
    int delta;      /* amount of allowable inaccuracy */
    int s = ss * unityold;
    if (s < 0) {
      mp_ps_print_char(mp, '-');
      negate(s);      /* print the sign, if negative */
    }
    mp_ps_print_int(mp, s/unityold);      /* print the integer part */
    s = 10 * (s % unityold) + 5;
    if (s ≠ 5) {
      delta = 10;
      mp_ps_print_char(mp, '.');
      do {
        if (delta > unityold) s = s + °100000 − (delta/2);      /* round the final digit */
        mp_ps_print_char(mp, '0' + (s/unityold));
        s = 10 * (s % unityold);
        delta = delta * 10;
      } while (s > delta);
    }
  }
  static void mp_ps_print_double(MP mp, double s)
  {
```

```
  if (mp→math_mode ≡ mp_math_scaled_mode) {
    mp_ps_print_double_scaled(mp, s);
  }
  else {
    mp_ps_print_double_new(mp, s);
  }
}
```

17. Dealing with font encodings.
First, here are a few helpers for parsing files

```
#define check_buf(size, buf_size)
        if ((unsigned)(size) > (unsigned)(buf_size)) {
          char S[128];
          mp_snprintf(S, 128, "buffer␣overflow:␣(%u,%u)␣at␣file␣%s,␣line␣%d", (unsigned)(size),
              (unsigned)(buf_size), __FILE__, __LINE__);
          mp_fatal_error(mp, S);
        }
#define append_char_to_buf(c, p, buf, buf_size) do
        {
          if (c ≡ 9) c = 32;
          if (c ≡ 13 ∨ c ≡ EOF) c = 10;
          if (c ≠ '␣' ∨ (p > buf ∧ p[−1] ≠ 32)) {
            check_buf(p − buf + 1, (buf_size));
            ∗p++ = (char) c;
          }
        }
        while (0)
#define append_eol(p, buf, buf_size) do
        {
          check_buf(p − buf + 2, (buf_size));
          if (p − buf > 1 ∧ p[−1] ≠ 10) ∗p++ = 10;
          if (p − buf > 2 ∧ p[−2] ≡ 32) {
            p[−2] = 10;
            p−−;
          }
          ∗p = 0;
        }
        while (0)
#define remove_eol(p, buf) do
        {
          p = strend(buf) − 1;
          if (∗p ≡ 10) ∗p = 0;
        }
        while (0)
#define skip(p, c) if (∗p ≡ c) p++
#define strend(s) strchr(s, 0)
#define str_prefix(s1, s2) (strncmp((s1), (s2), strlen(s2)) ≡ 0)
```

18. ⟨ Types 18 ⟩ ≡

```
typedef struct {
  boolean loaded;      /∗ the encoding has been loaded? ∗/
  char ∗file_name;     /∗ encoding file name ∗/
  char ∗enc_name;      /∗ encoding true name ∗/
  integer objnum;      /∗ object number ∗/
  char ∗∗glyph_names;
  integer tounicode;   /∗ object number of associated ToUnicode entry ∗/
} enc_entry;
```

See also sections 36, 68, 81, 83, 91, 95, 102, 115, 138, 174, 187, 191, and 228.

This code is used in section 3.

19.

```
#define ENC_STANDARD 0
#define ENC_BUILTIN 1
⟨ Globals 7 ⟩ +≡
#define ENC_BUF_SIZE #1000
  char enc_line[ENC_BUF_SIZE];
  void *enc_file;
```

20.

```
#define enc_eof() (mp->eof_file)(mp, mp->ps->enc_file)
#define enc_close() (mp->close_file)(mp, mp->ps->enc_file)
  static int enc_getchar(MP mp)
  {
    size_t len = 1;
    unsigned char abyte = 0;
    void *byte_ptr = &abyte;
    (mp->read_binary_file)(mp, mp->ps->enc_file, &byte_ptr, &len);
    return abyte;
  }
```

```
21. static boolean mp_enc_open(MP mp, char *n)
{
  mp->ps->enc_file = (mp->open_file)(mp, n, "r", mp_filetype_encoding);
  if (mp->ps->enc_file ≠ Λ) return true;
  else return false;
}
static void mp_enc_getline(MP mp)
{
  char *p;
  int c;
RESTART:
  if (enc_eof( )) {
    mp_error(mp, "unexpected␣end␣of␣file", Λ, true);
  }
  p = mp->ps->enc_line;
  do {
    c = enc_getchar(mp);
    append_char_to_buf(c, p, mp->ps->enc_line, ENC_BUF_SIZE);
  } while (c ∧ c ≠ 10);
  append_eol(p, mp->ps->enc_line, ENC_BUF_SIZE);
  if (p − mp->ps->enc_line < 2 ∨ *mp->ps->enc_line ≡ '%') goto RESTART;
}
static void mp_load_enc(MP mp, char *enc_name, char **enc_encname, char **glyph_names)
{
  char buf[ENC_BUF_SIZE], *p, *r;
  int names_count;
  char *myname;
  unsigned save_selector = mp->selector;
  if (¬mp_enc_open(mp, enc_name)) {
    char err[256];
    mp_snprintf(err, 255, "cannot␣open␣encoding␣file␣%s␣for␣reading", enc_name);
    mp_print(mp, err);
    return;
  }
  mp_normalize_selector(mp);
  mp_print(mp, "{");
  mp_print(mp, enc_name);
  mp_enc_getline(mp);
  if (*mp->ps->enc_line ≠ '/' ∨ (r = strchr(mp->ps->enc_line, '[')) ≡ Λ) {
    char msg[256];
    remove_eol(r, mp->ps->enc_line);
    mp_snprintf(msg, 256, "invalid␣encoding␣vector␣(a␣name␣or␣`['␣missing):␣`%s'",
        mp->ps->enc_line);
    mp_error(mp, msg, Λ, true);
  }
  while (*(r − 1) ≡ '␣') r−−;      /* strip trailing spaces from encoding name */
  myname = mp_xmalloc(mp, (size_t)(r − mp->ps->enc_line), 1);
  memcpy(myname, (mp->ps->enc_line + 1), (size_t)((r − mp->ps->enc_line) − 1));
  *(myname + (r − mp->ps->enc_line − 1)) = 0;
  *enc_encname = myname;
  while (*r ≠ '[') r++;
```

```
  r++;      /* skip '[' */
  names_count = 0;
  skip(r, ' ');
  for ( ; ; ) {
    while (*r == '/') {
      for (p = buf, r++; *r != ' ' ∧ *r != 10 ∧ *r != ']' ∧ *r != '/'; *p++ = *r++) ;
      *p = 0;
      skip(r, ' ');
      if (names_count > 256) {
        mp_error(mp, "encoding vector contains more than 256 names", Λ, true);
      }
      if (mp_xstrcmp(buf, notdef) != 0) glyph_names[names_count] = mp_xstrdup(mp, buf);
      names_count++;
    }
    if (*r != 10 ∧ *r != '%') {
      if (str_prefix(r, "] def")) goto DONE;
      else {
        char msg[256];
        remove_eol(r, mp→ps→enc_line);
        mp_snprintf(msg, 256, "invalid encoding vector: a name or `] def' expected: `%s'",
            mp→ps→enc_line);
        mp_error(mp, msg, Λ, true);
      }
    }
    mp_enc_getline(mp);
    r = mp→ps→enc_line;
  }
DONE: enc_close( );
  mp_print(mp, "}");
  mp→selector = save_selector;
}
static void mp_read_enc(MP mp, enc_entry *e)
{
  if (e→loaded) return;
  mp_xfree(e→enc_name);
  e→enc_name = Λ;
  mp_load_enc(mp, e→file_name, &e→enc_name, e→glyph_names);
  e→loaded = true;
}
```

22. *write_enc* is used to write either external encoding (given in map file) or internal encoding (read from the font file); the 2nd argument is a pointer to the encoding entry;

```
static void mp_write_enc(MP mp, enc_entry *e)
{
  int i;
  size_t s, foffset;
  char **g;
  if (e→objnum ≠ 0)      /* the encoding has been written already */
    return;
  e→objnum = 1;
  g = e→glyph_names;
  mp_ps_print(mp, "\n%%%%BeginResource: encoding ");
  mp_ps_print(mp, e→enc_name);
  mp_ps_print_nl(mp, "/");
  mp_ps_print(mp, e→enc_name);
  mp_ps_print(mp, " [ ");
  mp_ps_print_ln(mp);
  foffset = strlen(e→file_name) + 3;
  for (i = 0; i < 256; i++) {
    s = strlen(g[i]);
    if (s + 1 + foffset ≥ 80) {
      mp_ps_print_ln(mp);
      foffset = 0;
    }
    foffset += s + 2;
    mp_ps_print_char(mp, '/');
    mp_ps_print(mp, g[i]);
    mp_ps_print_char(mp, ' ');
  }
  if (foffset > 75) mp_ps_print_ln(mp);
  mp_ps_print_nl(mp, "] def\n");
  mp_ps_print(mp, "%%%%EndResource");
}
```

23. All encoding entries go into AVL tree for fast search by name.

```
⟨ Globals 7 ⟩ +≡
  avl_tree enc_tree;
```

24.

```
⟨ Static variables in the outer block 24 ⟩ ≡
  static char notdef[] = ".notdef";
```

See also sections 78 and 86.

This code is used in section 1.

25.

```
⟨ Set initial values 8 ⟩ +≡
  mp→ps→enc_tree = Λ;
```

26. **static int** *comp_enc_entry*(**void** ∗*p*, **const void** ∗*pa*, **const void** ∗*pb*)

```
{
  (void) p;
  return strcmp(((const enc_entry *) pa)→file_name, ((const enc_entry *) pb)→file_name);
}
static void *destroy_enc_entry(void *pa)
{
  enc_entry *p;
  int i;
  p = (enc_entry *) pa;
  mp_xfree(p→file_name);
  if (p→glyph_names ≠ Λ)
    for (i = 0; i < 256; i++)
      if (p→glyph_names[i] ≠ notdef) mp_xfree(p→glyph_names[i]);
  mp_xfree(p→enc_name);
  mp_xfree(p→glyph_names);
  mp_xfree(p);
  return Λ;
}
```

27. Not having an *mp* instance here means that lots of *malloc* and *strdup* checks are needed. Spotted by Peter Breitenlohner.

```
static void *copy_enc_entry(const void *pa)
{
  const enc_entry *p;
  enc_entry *q;
  int i;
  p = (const enc_entry *) pa;
  q = malloc(sizeof(enc_entry));
  if (q ≠ Λ) {
    memset(q, 0, sizeof(enc_entry));
    if (p→enc_name ≠ Λ) {
      q→enc_name = strdup(p→enc_name);
      if (q→enc_name ≡ Λ) return Λ;
    }
    q→loaded = p→loaded;
    if (p→file_name ≠ Λ) {
      q→file_name = strdup(p→file_name);
      if (q→file_name ≡ Λ) return Λ;
    }
    q→objnum = p→objnum;
    q→tounicode = p→tounicode;
    q→glyph_names = malloc(256 * sizeof(char *));
    if (p→glyph_names ≡ Λ) return Λ;
    for (i = 0; i < 256; i++) {
      if (p→glyph_names[i] ≠ Λ) {
        q→glyph_names[i] = strdup(p→glyph_names[i]);
        if (q→glyph_names[i] ≡ Λ) return Λ;
      }
    }
  }
  return (void *) q;
}
static enc_entry *mp_add_enc(MP mp, char *s)
{
  int i;
  enc_entry tmp, *p;
  if (mp→ps→enc_tree ≡ Λ) {
    mp→ps→enc_tree = avl_create(comp_enc_entry, copy_enc_entry, destroy_enc_entry, malloc, free, Λ);
  }
  tmp.file_name = s;
  p = (enc_entry *) avl_find(&tmp, mp→ps→enc_tree);
  if (p ≠ Λ)      /* encoding already registered */
    return p;
  p = mp_xmalloc(mp, 1, sizeof(enc_entry));
  memset(p, 0, sizeof(enc_entry));
  p→loaded = false;
  p→file_name = mp_xstrdup(mp, s);
  p→objnum = 0;
  p→tounicode = 0;
  p→glyph_names = mp_xmalloc(mp, 256, sizeof(char *));
```

```
  for (i = 0; i < 256; i++) {
    p→glyph_names[i] = mp_xstrdup(mp, notdef);
  }
  assert(avl_ins(p, mp→ps→enc_tree, avl_false) > 0);
  destroy_enc_entry(p);
  return avl_find(&tmp, mp→ps→enc_tree);
}
```

28. cleaning up...

29. ⟨ Declarations 29 ⟩ ≡

```
static void enc_free(MP mp);
```

See also sections 31, 39, 41, 63, 92, 100, 110, 112, 117, 119, 121, 123, 125, 127, 129, 131, 133, 139, 141, 143, 145, 150, 155, 159, 163, 165, 167, 169, 171, 177, 184, 196, 204, 209, 212, 218, 220, 222, 224, 226, and 229.

This code is used in section 1.

30.

```
static void enc_free(MP mp)
{
  if (mp→ps→enc_tree ≠ Λ) avl_destroy(mp→ps→enc_tree);
}
```

31. ⟨ Declarations 29 ⟩ +≡

```
static void mp_reload_encodings(MP mp);
static void mp_font_encodings(MP mp, font_number lastfnum, boolean encodings_only);
```

```
32. void mp_reload_encodings(MP mp)
{
  font_number f;
  enc_entry *e;
  fm_entry * fm_cur;
  font_number lastfnum = mp->last_fnum;
  for (f = null_font + 1; f ≤ lastfnum; f++) {
    if (mp->font_enc_name[f] ≠ Λ) {
      mp_xfree(mp->font_enc_name[f]);
      mp->font_enc_name[f] = Λ;
    }
    if (mp_has_fm_entry(mp, f, &fm_cur)) {
      if (fm_cur ≠ Λ ∧ fm_cur->ps_name ≠ Λ ∧ is_reencoded(fm_cur)) {
        e = fm_cur->encoding;
        mp_read_enc(mp, e);
      }
    }
  }
}
static void mp_font_encodings(MP mp, font_number lastfnum, boolean encodings_only)
{
  font_number f;
  enc_entry *e;
  fm_entry * fm;
  for (f = null_font + 1; f ≤ lastfnum; f++) {
    if (mp_has_font_size(mp, f) ∧ mp_has_fm_entry(mp, f, &fm)) {
      if (fm ≠ Λ ∧ (fm->ps_name ≠ Λ)) {
        if (is_reencoded(fm)) {
          if (encodings_only ∨ (¬is_subsetted(fm))) {
            e = fm->encoding;
            mp_write_enc(mp, e);      /* clear for next run */
            e->objnum = 0;
          }
        }
      }
    }
  }
}
```

33. Parsing font map files.

```
#define FM_BUF_SIZE 1024
⟨ Globals 7 ⟩ +≡
  void *fm_file;
  size_t fm_byte_waiting;
  size_t fm_byte_length;
  unsigned char *fm_bytes;
```

34. This is comparable to t1 font loading (see below) but because the first thing done is not calling *fm_getchar*( ) but *fm_eof*( ), the initial value of length has to be one more than waiting.

```
⟨ Set initial values 8 ⟩ +≡
  mp→ps→fm_byte_waiting = 0;
  mp→ps→fm_byte_length = 1;
  mp→ps→fm_bytes = Λ;
```

35.

```
#define fm_eof() (mp→ps→fm_byte_waiting ≥ mp→ps→fm_byte_length)
#define fm_close() do
        {
          (mp→close_file)(mp, mp→ps→fm_file);
          mp_xfree(mp→ps→fm_bytes);
          mp→ps→fm_bytes = Λ;
          mp→ps→fm_byte_waiting = 0;
          mp→ps→fm_byte_length = 1;
        }
        while (0)
#define valid_code(c) (c ≥ 0 ∧ c < 256)
#define unwrap_file(ff) ( mp→noninteractive ( ( File * ) ff ) → f: ff )
  static int fm_getchar(MP mp)
  {
    if (mp→ps→fm_bytes ≡ Λ) {
      void *byte_ptr;
      (void) fseek(unwrap_file(mp→ps→fm_file), 0, SEEK_END);
      mp→ps→fm_byte_length = (size_t) ftell(unwrap_file(mp→ps→fm_file));
      (void) fseek(unwrap_file(mp→ps→fm_file), 0, SEEK_SET);
      if (mp→ps→fm_byte_length ≡ 0) return EOF;
      mp→ps→fm_bytes = mp_xmalloc(mp, mp→ps→fm_byte_length, 1);
      byte_ptr = (void *) mp→ps→fm_bytes;
      (mp→read_binary_file)(mp, mp→ps→fm_file, &byte_ptr, &mp→ps→fm_byte_length);
    }
    if (mp→ps→fm_byte_waiting ≥ mp→ps→fm_byte_length) return 10;
    return *(mp→ps→fm_bytes + mp→ps→fm_byte_waiting++);
  }
```

36. ⟨ Types 18 ⟩ +≡

```
enum _mode {
  FM_DUPIGNORE, FM_REPLACE, FM_DELETE
};
enum _ltype {
  MAPFILE, MAPLINE
};
enum _tfmavail {
  TFM_UNCHECKED, TFM_FOUND, TFM_NOTFOUND
};
typedef struct mitem {
  int mode;      /* FM_DUPIGNORE or FM_REPLACE or FM_DELETE */
  int type;      /* map file or map line */
  char *map_line;      /* pointer to map file name or map line */
  int lineno;      /* line number in map file */
} mapitem;
```

37. ⟨ Globals 7 ⟩ +≡

```
mapitem *
    mitem;
fm_entry * fm_cur;
fm_entry * loaded_tfm_found;
fm_entry * avail_tfm_found;
fm_entry * non_tfm_found;
fm_entry * not_avail_tfm_found;
```

38. ⟨ Set initial values 8 ⟩ +≡

```
mp→ps → mitem = Λ;
```

39. ⟨ Declarations 29 ⟩ +≡

```
static const char nontfm[] = "<nontfm>";
```

40.

```
#define read_field(r, q, buf) do
        {
          q = buf;
          while (*r ≠ '␣' ∧ *r ≠ '\0') *q++ = *r++;
          *q = '\0';
          skip(r, '␣');
        }
        while (0)
#define set_field(F) do
        {
          if (q > buf) fm→F = mp_xstrdup(mp, buf);
          if (*r ≡ '\0') goto DONE;
        }
        while (0)
#define cmp_return(a, b)
        if (a > b) return 1;
        if (a < b) return −1
#define do_strdup(a) (a ≡ ΛΛ : strdup(a))
  static fm_entry*new_fm_entry(MP mp)
  {
    fm_entry * fm;
    fm = mp_xmalloc(mp, 1, sizeof (fm_entry));
    fm→tfm_name = Λ;
    fm→ps_name = Λ;
    fm→flags = 4;
    fm→ff_name = Λ;
    fm→subset_tag = Λ;
    fm→encoding = Λ;
    fm→tfm_num = null_font;
    fm→tfm_avail = TFM_UNCHECKED;
    fm→type = 0;
    fm→slant = 0;
    fm→extend = 0;
    fm→ff_objnum = 0;
    fm→fn_objnum = 0;
    fm→fd_objnum = 0;
    fm→charset = Λ;
    fm→all_glyphs = false;
    fm→links = 0;
    fm→pid = −1;
    fm→eid = −1;
    return fm;
  }
  static void *copy_fm_entry(const void *p)
  {
    fm_entry * fm;
    const fm_entry*fp;
    fp = (const fm_entry*) p;
    fm = malloc(sizeof (fm_entry));
    if (fm ≡ Λ) return Λ;
```

```
  memcpy(fm, fp, sizeof (fm_entry));
  fm→tfm_name = do_strdup(fp→tfm_name);
  fm→ps_name = do_strdup(fp→ps_name);
  fm→ff_name = do_strdup(fp→ff_name);
  fm→subset_tag = do_strdup(fp→subset_tag);
  fm→charset = do_strdup(fp→charset);
  return (void *) fm;
}
static void *delete_fm_entry(void *p){ fm_entry * fm = ( fm_entry * ) p;
  mp_xfree(fm→tfm_name);
  mp_xfree(fm→ps_name);
  mp_xfree(fm→ff_name);
  mp_xfree(fm→subset_tag);
  mp_xfree(fm→charset);
  mp_xfree(fm);
  return Λ; } static ff_entry*new_ff_entry(MP mp)
  {
    ff_entry * ff ;
    ff = mp_xmalloc(mp, 1, sizeof (ff_entry));
    ff→ff_name = Λ;
    ff→ff_path = Λ;
    return ff ;
  }
  static void *copy_ff_entry(const void *p){ ff_entry * ff ;
    const ff_entry*fp;
    fp = (const ff_entry*) p; ff = ( ff_entry * ) malloc(sizeof (ff_entry));
    if (ff ≡ Λ) return Λ;
    ff→ff_name = do_strdup(fp→ff_name);
    ff→ff_path = do_strdup(fp→ff_path);
    return ff ; } static void *delete_ff_entry(void *p){ ff_entry * ff = ( ff_entry * ) p;
      mp_xfree(ff→ff_name);
      mp_xfree(ff→ff_path);
      mp_xfree(ff);
      return Λ; } static char *mk_base_tfm(MP mp, char *tfmname, int *i)
      {
        static char buf[SMALL_BUF_SIZE];
        char *p = tfmname, *r = strend(p) − 1, *q = r;
        while (q > p ∧ mp_isdigit(*q)) −−q;
        if (¬(q > p) ∨ q ≡ r ∨ (*q ≠ '+' ∧ *q ≠ '-')) return Λ;
        check_buf(q − p + 1, SMALL_BUF_SIZE);
        strncpy(buf, p, (size_t)(q − p));
        buf[q − p] = '\0';
        *i = atoi(q);
        return buf;
      }
```

41. ⟨ Declarations 29 ⟩ +≡

```
static boolean mp_has_fm_entry(MP mp, font_number f, fm_entry **fm);
```

42. *boolean mp_has_fm_entry*(MP *mp*, *font_number f*, *fm_entry* ∗ ∗*fm*)

```
{
  fm_entry * res = Λ;
  res = mp_fm_lookup(mp, f);
  if (fm ≠ Λ) {
    *fm = res;
  }
  return (res ≠ Λ);
}
```

43. ⟨ Globals 7 ⟩ +≡

```
avl_tree tfm_tree;
avl_tree ps_tree;
avl_tree ff_tree;
```

44. ⟨ Set initial values 8 ⟩ +≡

```
mp→ps→tfm_tree = Λ;
mp→ps→ps_tree = Λ;
mp→ps→ff_tree = Λ;
```

45. AVL sort *fm_entry* into *tfm_tree* by *tfm_name*

```
static int comp_fm_entry_tfm(void *p, const void *pa, const void *pb)
{
  (void) p;
  return strcmp(((const fm_entry*) pa)→tfm_name, ((const fm_entry*) pb)→tfm_name);
}
```

46. AVL sort *fm_entry* into *ps_tree* by *ps_name*, *slant*, and *extend*

```
static int comp_fm_entry_ps(void *p, const void *pa, const void *pb)
{
  int i;
  const fm_entry*p1 = (const fm_entry*) pa;
  const fm_entry*p2 = (const fm_entry*) pb;
  (void) p;
  assert(p1→ps_name ≠ Λ ∧ p2→ps_name ≠ Λ);
  if ((i = strcmp(p1→ps_name, p2→ps_name))) return i;
  cmp_return(p1→slant, p2→slant);
  cmp_return(p1→extend, p2→extend);
  if (p1→tfm_name ≠ Λ ∧ p2→tfm_name ≠ Λ ∧ (i = strcmp(p1→tfm_name, p2→tfm_name))) return i;
  return 0;
}
```

47. AVL sort *ff_entry* into *ff_tree* by *ff_name*

```
static int comp_ff_entry(void *p, const void *pa, const void *pb)
{
  (void) p;
  return strcmp(((const ff_entry*) pa)→ff_name, ((const ff_entry*) pb)→ff_name);
}
```

```
48. static void create_avl_trees(MP mp)
{
  if (mp→ps→tfm_tree ≡ Λ) {
    mp→ps→tfm_tree = avl_create(comp_fm_entry_tfm, copy_fm_entry, delete_fm_entry, malloc, free, Λ);
    assert(mp→ps→tfm_tree ≠ Λ);
  }
  if (mp→ps→ps_tree ≡ Λ) {
    mp→ps→ps_tree = avl_create(comp_fm_entry_ps, copy_fm_entry, delete_fm_entry, malloc, free, Λ);
    assert(mp→ps→ps_tree ≠ Λ);
  }
  if (mp→ps→ff_tree ≡ Λ) {
    mp→ps→ff_tree = avl_create(comp_ff_entry, copy_ff_entry, delete_ff_entry, malloc, free, Λ);
    assert(mp→ps→ff_tree ≠ Λ);
  }
}
```

49. The function *avl_do_entry* is not completely symmetrical with regards to *tfm_name* and *ps_name handling*, e. g. a duplicate *tfm_name* gives a **goto** *exit*, and no *ps_name* link is tried. This is to keep it compatible with the original version.

```
#define LINK_TFM ^01
#define LINK_PS ^02
#define set_tfmlink(fm) ((fm)→links |= LINK_TFM)
#define set_pslink(fm) ((fm)→links |= LINK_PS)
#define has_tfmlink(fm) ((fm)→links & LINK_TFM)
#define has_pslink(fm) ((fm)→links & LINK_PS)
  static int avl_do_entry(MP mp, fm_entry * fp, int mode){ fm_entry * p;
    char s[128]; /* handle tfm_name link */
    if (strcmp(fp→tfm_name, nontfm)) { p = ( fm_entry * ) avl_find(fp, mp→ps→tfm_tree);
    if (p ≠ Λ) {
      if (mode ≡ FM_DUPIGNORE) {
        mp_snprintf(s, 128, "fontmap␣entry␣for␣‘%s’␣already␣exists,␣duplicates␣ignored",
            fp→tfm_name);
        mp_warn(mp, s);
        goto exit;
      }
      else { /* mode == FM_REPLACE / FM_DELETE */
        if (mp_has_font_size(mp, p→tfm_num)) {
          mp_snprintf(s, 128,
              "fontmap␣entry␣for␣‘%s’␣has␣been␣used,␣replace/delete␣not␣allowed",
              fp→tfm_name);
          mp_warn(mp, s);
          goto exit;
        }
        (void) avl_del(p, mp→ps→tfm_tree, Λ);
        p = Λ;
      }
    }
    if (mode ≠ FM_DELETE) {
      if (p ≡ Λ) {
        assert(avl_ins(fp, mp→ps→tfm_tree, avl_false) > 0);
      }
      set_tfmlink(fp);
    }
    } /* handle ps_name link */
    if (fp→ps_name ≠ Λ) { assert(fp→tfm_name ≠ Λ); p = ( fm_entry * ) avl_find(fp, mp→ps→ps_tree);
    if (p ≠ Λ) {
      if (mode ≡ FM_DUPIGNORE) {
        mp_snprintf(s, 128, "ps_name␣entry␣for␣‘%s’␣already␣exists,␣duplicates␣ignored",
            fp→ps_name);
        mp_warn(mp, s);
        goto exit;
      }
      else { /* mode == FM_REPLACE / FM_DELETE */
        if (mp_has_font_size(mp, p→tfm_num)) { /* REPLACE/DELETE not allowed */
          mp_snprintf(s, 128,
              "fontmap␣entry␣for␣‘%s’␣has␣been␣used,␣replace/delete␣not␣allowed",
              p→tfm_name);
```

```
        mp_warn(mp, s);
        goto exit;
      }
      (void) avl_del(p, mp→ps→ps_tree, Λ);
      p = Λ;
    }
  }
  if (mode ≠ FM_DELETE) {
    if (p ≡ Λ) {
      assert(avl_ins(fp, mp→ps→ps_tree, avl_false) > 0);
    }
    set_pslink(fp);
  }
  }
exit:
  if (¬has_tfmlink(fp) ∧ ¬has_pslink(fp))      /* e. g. after FM_DELETE */
    return 1;      /* deallocation of fm_entry structure required */
  else return 0;
  }
```

50. consistency check for map entry, with warn flag

```
static int check_fm_entry(MP mp, fm_entry * fm, boolean warn)
{
  int a = 0;
  char s[128];
  assert(fm ≠ Λ);
  if (fm→ps_name ≠ Λ) {
    if (is_basefont(fm)) {
      if (is_fontfile(fm) ∧ ¬is_included(fm)) {
        if (warn) {
          mp_snprintf(s, 128, "invalid␣entry␣for␣'%s':␣""font␣file␣must␣be␣i\
              ncluded␣or␣omitted␣for␣base␣fonts", fm→tfm_name);
          mp_warn(mp, s);
        }
        a += 1;
      }
    }
    else {      /* not a base font */
      /* if no font file given, drop this entry */      /* if (¬is_fontfile(fm)) { if (warn) {
        mp_snprintf(s, 128, "invalid␣entry␣for␣'%s':␣font␣file␣missing", fm→tfm_name);
        mp_warn(mp, s); } a += 2; } */
    }
  }
  if (is_truetype(fm) ∧ is_reencoded(fm) ∧ ¬is_subsetted(fm)) {
    if (warn) {
      mp_snprintf(s, 128,
          "invalid␣entry␣for␣'%s':␣only␣subsetted␣TrueType␣font␣can␣be␣reencoded",
          fm→tfm_name);
      mp_warn(mp, s);
    }
    a += 4;
  }
  if ((fm→slant ≠ 0 ∨ fm→extend ≠ 0) ∧ (is_truetype(fm))) {
    if (warn) {
      mp_snprintf(s, 128, "invalid␣entry␣for␣'%s':␣""SlantFont/ExtendFont␣can␣be␣used␣only␣w\
          ith␣embedded␣T1␣fonts", fm→tfm_name);
      mp_warn(mp, s);
    }
    a += 8;
  }
  if (abs(fm→slant) > 1000) {
    if (warn) {
      mp_snprintf(s, 128, "invalid␣entry␣for␣'%s':␣too␣big␣value␣of␣SlantFont␣(%d/1000.0)",
          fm→tfm_name, (int) fm→slant);
      mp_warn(mp, s);
    }
    a += 16;
  }
  if (abs(fm→extend) > 2000) {
    if (warn) {
      mp_snprintf(s, 128, "invalid␣entry␣for␣'%s':␣too␣big␣value␣of␣ExtendFont␣(%d/1000.0)",
          fm→tfm_name, (int) fm→extend);
```

```
      mp_warn(mp, s);
    }
    a += 32;
  }
  if (fm→pid ≠ −1 ∧ ¬(is_truetype(fm) ∧ is_included(fm) ∧ is_subsetted(fm) ∧ ¬is_reencoded(fm))) {
    if (warn) {
      mp_snprintf(s, 128, "invalid␣entry␣for␣'%s':␣""PidEid␣can␣be␣used␣only␣with␣subsetted␣\
          non-reencoded␣TrueType␣fonts", fm→tfm_name);
      mp_warn(mp, s);
    }
    a += 64;
  }
  return a;
}
```

51. returns true if s is one of the 14 std. font names; speed-trimmed.

```
static boolean check_basefont(char *s)
{
  static const char *basefont_names[] = {"Courier",      /* 0:7 */
  "Courier-Bold",      /* 1:12 */
  "Courier-Oblique",      /* 2:15 */
  "Courier-BoldOblique",      /* 3:19 */
  "Helvetica",      /* 4:9 */
  "Helvetica-Bold",      /* 5:14 */
  "Helvetica-Oblique",      /* 6:17 */
  "Helvetica-BoldOblique",      /* 7:21 */
  "Symbol",      /* 8:6 */
  "Times-Roman",      /* 9:11 */
  "Times-Bold",      /* 10:10 */
  "Times-Italic",      /* 11:12 */
  "Times-BoldItalic",      /* 12:16 */
  "ZapfDingbats"      /* 13:12 */
  };
  static const int Index[] = {-1,-1,-1,-1,-1,-1,8,0,-1,4,10,9,-1,-1,5,2,12,6,-1,3,-1,7};
  const size_t n = strlen(s);
  int k = -1;
  if (n > 21) return false;
  if (n == 12) {      /* three names have length 12 */
    switch (*s) {
    case 'C': k = 1;      /* Courier-Bold */
      break;
    case 'T': k = 11;      /* Times-Italic */
      break;
    case 'Z': k = 13;      /* ZapfDingbats */
      break;
    default: return false;
    }
  }
  else k = Index[n];
  if (k > -1 ∧ ¬strcmp(basefont_names[k], s)) return true;
  return false;
}
```

52.

```
#define is_cfg_comment(c) (c ≡ 10 ∨ c ≡ '*' ∨ c ≡ '#' ∨ c ≡ ';' ∨ c ≡ '%')
  static void fm_scan_line(MP mp){ int a, b, c, j, u = 0, v = 0;
    float d;
    fm_entry * fm;
    char fm_line[FM_BUF_SIZE], buf[FM_BUF_SIZE];
    char *p, *q, *s;
    char warn_s[128];
    char *r = Λ; switch ( mp→ps → mitem→type ) {
  case MAPFILE: p = fm_line;
    do {
      c = fm_getchar(mp);
      append_char_to_buf(c, p, fm_line, FM_BUF_SIZE);
    } while (c ≠ 10);
    *(--p) = '\0';
    r = fm_line;
    break; case MAPLINE: r = mp→ps → mitem→map_line;
    break;
  default: assert(0); }
    if (*r ≡ '\0' ∨ is_cfg_comment(*r)) return;
    fm = new_fm_entry(mp);
    read_field(r, q, buf);
    set_field(tfm_name);
    p = r;
    read_field(r, q, buf);
    if (*buf ≠ '<' ∧ *buf ≠ '"') set_field(ps_name);
    else r = p;      /* unget the field */
    if (mp_isdigit(*r)) {      /* font flags given */
      fm→flags = atoi(r);
      while (mp_isdigit(*r)) r++;
    }
    if (fm→ps_name ≡ Λ) fm→ps_name = xstrdup(fm→tfm_name);
    while (1) {      /* loop through "specials", encoding, font file */
      skip(r, ' ');
      switch (*r) {
      case '\0': goto DONE;
      case '"':      /* opening quote */
        r++;
        u = v = 0;
        do {
          skip(r, ' ');
          if (sscanf(r, "%f␣%n", &d, &j) > 0) {
            s = r + j;      /* jump behind number, eat also blanks, if any */
            if (*(s − 1) ≡ 'E' ∨ *(s − 1) ≡ 'e') s−−;      /* e. g. 0.5ExtendFont: %f = 0.5E */
            if (str_prefix(s, "SlantFont")) {
              d *= (float) 1000.0;      /* correct rounding also for neg. numbers */
              fm→slant = (short int)(d > 0d + 0.5 : d − 0.5);
              r = s + strlen("SlantFont");
            }
            else if (str_prefix(s, "ExtendFont")) {
              d *= (float) 1000.0;
```

```
        fm→extend = (short int)(d > 0d + 0.5 : d − 0.5);
        if (fm→extend ≡ 1000) fm→extend = 0;
        r = s + strlen("ExtendFont");
      }
      else {      /* unknown name */
        for (r = s; *r ≠ '␣' ∧ *r ≠ '"' ∧ *r ≠ '\0'; r++) ;      /* jump over name */
        c = *r;      /* remember char for temporary end of string */
        *r = '\0';
        mp_snprintf(warn_s, 128, "invalid␣entry␣for␣`%s':␣unknown␣name␣`%s'␣ignored",
            fm→tfm_name, s);
        mp_warn(mp, warn_s);
        *r = (char) c;
      }
    }
    else
      for ( ; *r ≠ '␣' ∧ *r ≠ '"' ∧ *r ≠ '\0'; r++) ;
  } while (*r ≡ '␣');
  if (*r ≡ '"')      /* closing quote */
    r++;
  else {
    mp_snprintf(warn_s, 128, "invalid␣entry␣for␣`%s':␣closing␣quote␣missing",
        fm→tfm_name);
    mp_warn(mp, warn_s);
    goto bad_line;
  }
  break;
case 'P':      /* handle cases for subfonts like 'PidEid=3,1' */
  if (sscanf(r, "PidEid=%i,␣%i␣%n", &a, &b, &c) ≥ 2) {
    fm→pid = (short int) a;
    fm→eid = (short int) b;
    r += c;
    break;
  }      /* fallthrough */
default:      /* encoding or font file specification */
  a = b = 0;
  if (*r ≡ '<') {
    a = *r++;
    if (*r ≡ '<' ∨ *r ≡ '[') b = *r++;
  }
  read_field(r, q, buf);      /* encoding, formats: '8r.enc' or '¡8r.enc' or '¡[8r.enc' */
  if (strlen(buf) > 4 ∧ mp_strcasecmp(strend(buf) − 4, ".enc") ≡ 0) {
    fm→encoding = mp_add_enc(mp, buf);
    u = v = 0;      /* u, v used if intervening blank: "¡¡ foo" */
  }
  else if (strlen(buf) > 0) {      /* file name given */      /* font file, formats: * subsetting:
      '¡cmr10.pfa' * no subsetting: '¡¡cmr10.pfa' * no embedding: 'cmr10.pfa' */
    if (a ≡ '<' ∨ u ≡ '<') {
      set_included(fm);
      if ((a ≡ '<' ∧ b ≡ 0) ∨ (a ≡ 0 ∧ v ≡ 0)) set_subsetted(fm);
        /* otherwise b == '¡' (or '[') =¿ no subsetting */
    }
    set_field(ff_name);
```

```
        u = v = 0;
      }
      else {
        u = a;
        v = b;
      }
    }
  }
DONE:
  if (fm→ps_name ≠ Λ ∧ check_basefont(fm→ps_name)) set_basefont(fm);
  if (is_fontfile(fm) ∧ mp_strcasecmp(strend(fm_fontfile(fm)) − 4, ".ttf") ≡ 0) set_truetype(fm);
  if (check_fm_entry(mp, fm, true) ≠ 0) goto bad_line;
    /* Until here the map line has been completely scanned without errors; fm points to a valid,
      freshly filled-out fm_entry structure. Now follows the actual work of registering/deleting. */
  if ( avl_do_entry (mp, fm, mp→ps → mitem→mode ) ≡ 0 )
  {   /* if success */
    delete_fm_entry(fm);
    return;
  }
bad_line: delete_fm_entry(fm); }
```

53.

```
static void fm_read_info(MP mp){ char *n;
  char s[256];
  if (mp→ps→tfm_tree ≡ Λ) create_avl_trees(mp);
  if ( mp→ps → mitem→map_line ≡ Λ )   /* nothing to do */
  return; mp→ps → mitem→lineno = 1; switch ( mp→ps → mitem→type ) { case MAPFILE: n = mp→ps
      → mitem→map_line;
  mp→ps→fm_file = (mp→open_file)(mp, n, "r", mp_filetype_fontmap);
  if (¬mp→ps→fm_file) {
    mp_snprintf(s, 256, "cannot␣open␣font␣map␣file␣%s", n);
    mp_warn(mp, s);
  }
  else { unsigned save_selector = mp→selector;
  mp_normalize_selector(mp);
  mp_print(mp, "{");
  mp_print(mp, n); while (¬fm_eof( )) { fm_scan_line(mp); mp→ps → mitem→lineno++; } fm_close( );
  mp_print(mp, "}");
  mp→selector = save_selector;
  mp→ps→fm_file = Λ; }   /* mp_xfree(n); */
  break;
case MAPLINE: fm_scan_line(mp);
  break;
default: assert(0); } mp→ps → mitem→map_line = Λ;   /* done with this line */
  return; }
```

54. **static void** *init_fm*(*fm_entry* ∗ *fm*, *font_number f*)
{
if (*fm*→*tfm_num* ≡ *null_font*) {
fm→*tfm_num* = *f*;
fm→*tfm_avail* = TFM_FOUND;
}
}

55. ⟨ Exported function headers 5 ⟩ +≡
fm_entry ∗ *mp_fm_lookup*(MP *mp*, *font_number f*);

56. *fm_entry* ∗ *mp_fm_lookup*(MP *mp*, *font_number f*){ **char** ∗*tfm*;
fm_entry ∗ *fm*;
fm_entry tmp;
int *e*;
if (*mp*→*ps*→*tfm_tree* ≡ Λ) *mp_read_psname_table*(*mp*); /∗ only to read default map file ∗/
tfm = *mp*→*font_name*[*f*];
assert(*strcmp*(*tfm*, *nontfm*)); /∗ Look up for full ¡tfmname¿[+-]¡expand¿ ∗/
tmp.*tfm_name* = *tfm*; *fm* = (*fm_entry* ∗) *avl_find*(&*tmp*, *mp*→*ps*→*tfm_tree*); **if** (*fm* ≠ Λ) {
init_fm(*fm*, *f*); **return** (*fm_entry* ∗) *fm*; } *tfm* = *mk_base_tfm*(*mp*, *mp*→*font_name*[*f*], &*e*);
if (*tfm* ≡ Λ) /∗ not an expanded font, nothing to do ∗/
return Λ;
tmp.*tfm_name* = *tfm*; *fm* = (*fm_entry* ∗) *avl_find*(&*tmp*, *mp*→*ps*→*tfm_tree*); **if** (*fm* ≠ Λ) {
/∗ found an entry with the base tfm name, e.g. cmr10 ∗/
return (*fm_entry* ∗) *fm*; /∗ font expansion uses the base font ∗/
} **return** Λ; }

57. Early check whether a font file exists. Used e. g. for replacing fonts of embedded PDF files: Without font file, the font within the embedded PDF-file is used. Search tree *ff_tree* is used in 1st instance, as it may be faster than the *kpse_find_file*( ), and *kpse_find_file*( ) is called only once per font file name + expansion parameter. This might help keeping speed, if many PDF pages with same fonts are to be embedded.

The *ff_tree* contains only font files, which are actually needed, so this tree typically is much smaller than the *tfm_tree* or *ps_tree*.

static *ff_entry*∗*check_ff_exist*(MP *mp*, *fm_entry* ∗ *fm*){ *ff_entry* ∗ *ff*;
ff_entry tmp;
assert(*fm*→*ff_name* ≠ Λ);
tmp.*ff_name* = *fm*→*ff_name*; *ff* = (*ff_entry* ∗) *avl_find*(&*tmp*, *mp*→*ps*→*ff_tree*); **if** (*ff* ≡ Λ) {
/∗ not yet in database ∗/
ff = *new_ff_entry*(*mp*);
ff→*ff_name* = *mp_xstrdup*(*mp*, *fm*→*ff_name*);
ff→*ff_path* = *mp_xstrdup*(*mp*, *fm*→*ff_name*);
assert(*avl_ins*(*ff*, *mp*→*ps*→*ff_tree*, *avl_false*) > 0);
delete_ff_entry(*ff*); *ff* = (*ff_entry* ∗) *avl_find*(&*tmp*, *mp*→*ps*→*ff_tree*); } **return** *ff*; }

58. Process map file given by its name or map line contents. Items not beginning with [+-=] flush default map file, if it has not yet been read. Leading blanks and blanks immediately following [+-=] are ignored.

```
static void mp_process_map_item(MP mp, char *s, int type){ char *p;
  int mode;
  if (*s ≡ '␣') s++;  /* ignore leading blank */
  switch (*s) {
case '+':  /* +mapfile.map, +mapline */
  mode = FM_DUPIGNORE;  /* insert entry, if it is not duplicate */
  s++;
  break;
case '=':  /* =mapfile.map, =mapline */
  mode = FM_REPLACE;  /* try to replace earlier entry */
  s++;
  break;
case '-':  /* -mapfile.map, -mapline */
  mode = FM_DELETE;  /* try to delete entry */
  s++;
  break;
default: mode = FM_DUPIGNORE;  /* like +, but also: */
  mp_xfree ( mp→ps → mitem→map_line ) ; mp→ps → mitem→map_line = Λ;
    /* flush default map file name */
  }
  if (*s ≡ '␣') s++;  /* ignore blank after [+-=] */
  p = s;  /* map item starts here */
  switch (type) {
  case MAPFILE:  /* remove blank at end */
    while (*p ≠ '\0' ∧ *p ≠ '␣') p++;
    *p = '\0';
    break;
  case MAPLINE:  /* blank at end allowed */
    break;
  default: assert(0);
  }
  if ( mp→ps → mitem→map_line ≠ Λ )  /* read default map file first */
  fm_read_info(mp); if (*s ≠ '\0') {  /* only if real item to process */
  mp→ps → mitem→mode = mode; mp→ps → mitem→type = type; mp→ps → mitem→map_line = s;
  fm_read_info(mp); } }
```

59. ⟨ Exported function headers 5 ⟩ +≡

```
void mp_map_file(MP mp, mp_stringt);
void mp_map_line(MP mp, mp_stringt);
void mp_init_map_file(MP mp, int is_troff );
```

60.
```
void mp_map_file(MP mp, mp_stringt)
{
  char *ss = mp_str(mp,t);
  char *s = mp_xstrdup(mp, ss);
  mp_process_map_item(mp, s, MAPFILE);
}
void mp_map_line(MP mp, mp_stringt)
{
  char *ss = mp_str(mp,t);
  char *s = mp_xstrdup(mp, ss);
  mp_process_map_item(mp, s, MAPLINE);
  mp_xfree(s);
}
```

61.
```
void mp_init_map_file(MP mp, int is_troff){ char *r; mp->ps -> mitem = mp_xmalloc(mp, 1,
    sizeof(mapitem)); mp->ps -> mitem->mode = FM_DUPIGNORE; mp->ps -> mitem->type = MAPFILE;
    mp->ps -> mitem->map_line = Λ;
  r = (mp->find_file)(mp, "mpost.map", "r", mp_filetype_fontmap); if (r ≠ Λ) { mp_xfree(r);
    mp->ps -> mitem->map_line = mp_xstrdup(mp, "mpost.map"); } else { if (is_troff)
    { mp->ps -> mitem->map_line = mp_xstrdup(mp, "troff.map"); } else { mp->ps ->
    mitem->map_line = mp_xstrdup(mp, "pdftex.map"); } } }
```

62. ⟨ Dealloc variables 62 ⟩ ≡
```
if ( mp->ps -> mitem ≠ Λ ) { mp_xfree ( mp->ps -> mitem->map_line ) ; mp_xfree ( mp->ps -> mitem ) ; }
```
See also sections 73 and 194.

This code is used in section 6.

63. ⟨ Declarations 29 ⟩ +≡
```
static void fm_free(MP mp);
```

64.
```
static void fm_free(MP mp)
{
  if (mp->ps->tfm_tree ≠ Λ) avl_destroy(mp->ps->tfm_tree);
  if (mp->ps->ps_tree ≠ Λ) avl_destroy(mp->ps->ps_tree);
  if (mp->ps->ff_tree ≠ Λ) avl_destroy(mp->ps->ff_tree);
}
```

65. The file *ps_tab_file* gives a table of TEX font names and corresponding PostScript names for fonts that do not have to be downloaded, i.e., fonts that can be used when *internal*[*prologues*] > 0. Each line consists of a TEX name, one or more spaces, a PostScript name, and possibly a space and some other junk. This routine reads the table, updates *font_ps_name* entries starting after *last_ps_fnum*, and sets *last_ps_fnum* := *last_fnum*.

```
#define ps_tab_name "psfonts.map" /* locates font name translation table */
```
⟨ Exported function headers 5 ⟩ +≡
```
void mp_read_psname_table(MP mp);
```

66. **void** *mp_read_psname_table*(MP *mp*){ *font_number k*;

```
char *s;
static int isread = 0; if ( mp->ps -> mitem ≡ Λ ) { mp->ps -> mitem = mp_xmalloc(mp, 1,
    sizeof(mapitem)); mp->ps -> mitem->mode = FM_DUPIGNORE; mp->ps -> mitem->type = MAPFILE;
    mp->ps -> mitem->map_line = Λ; } s = mp_xstrdup(mp, ps_tab_name); mp->ps ->
    mitem->map_line = s;
if (isread ≡ 0) {
  isread ++;
  fm_read_info(mp);
}
for (k = mp->last_ps_fnum + 1; k ≤ mp->last_fnum; k++) {
  if (mp_has_fm_entry(mp, k, Λ)) {
    mp_xfree(mp->font_ps_name[k]);
    mp->font_ps_name[k] = mp_fm_font_name(mp, k);
  }
}
mp->last_ps_fnum = mp->last_fnum; }
```

67. The traditional function is a lot shorter now.

68. Helper functions for Type1 fonts.
Avoid to redefine *Byte* and *Bytef* from < *zlib.h* >.

```
⟨ Types 18 ⟩ +≡
  typedef char char_entry;
#ifndef ZCONF_H
  typedef unsigned char Byte;
  typedef Byte Bytef;
#endif
```

69. ⟨ Globals 7 ⟩ +≡

```
  char_entry *char_ptr, *char_array;
  size_t char_limit;
  char *job_id_string;
```

70. ⟨ Set initial values 8 ⟩ +≡

```
  mp->ps->char_array = Λ;
  mp->ps->job_id_string = Λ;
```

71.

```
#define SMALL_ARRAY_SIZE 256
#define Z_NULL 0
  void mp_set_job_id(MP mp)
  {
    char *name_string, *s;
    size_t slen;
    if (mp->ps->job_id_string ≠ Λ) return;
    if (mp->job_name ≡ Λ) mp->job_name = mp_xstrdup(mp, "mpout");
    name_string = mp_xstrdup(mp, mp->job_name);
    slen = SMALL_BUF_SIZE + strlen(name_string);
    s = mp_xmalloc(mp, slen, sizeof(char));
    | /* @ − buf f erover f lowhigh@ */ sprintf(s, "%.4u/%.2u/%.2u␣%.2u:%.2u␣%s", ((unsigned)
        number_to_scaled(internal_value(mp_year)) ≫ 16),
        ((unsigned) number_to_scaled(internal_value(mp_month)) ≫ 16), ((unsigned)
        number_to_scaled(internal_value(mp_day)) ≫ 16),
        ((unsigned) number_to_scaled(internal_value(mp_time)) ≫ 16)/60, ((unsigned)
        number_to_scaled(internal_value(mp_time)) ≫ 16) % 60, name_string);
    | /* @ = buf f erover f lowhigh@ */ mp->ps->job_id_string = mp_xstrdup(mp, s);
    mp_xfree(s);
    mp_xfree(name_string);
  }
  static void fnstr_append(MP mp, const char *ss)
  {
    size_t n = strlen(ss) + 1;
    alloc_array(char, n, SMALL_ARRAY_SIZE);
    strcat(mp->ps->char_ptr, ss);
    mp->ps->char_ptr = strend(mp->ps->char_ptr);
  }
```

72. ⟨ Exported function headers 5 ⟩ +≡

```
  void mp_set_job_id(MP mp);
```

73. ⟨ Dealloc variables 62 ⟩ +≡
mp_xfree(*mp→ps→job_id_string*);

74. this is not really a true crc32, but it should be just enough to keep subsets prefixes somewhat disjunct

```
static unsigned long crc32(unsigned long oldcrc, const Byte *buf, size_t len)
{
  unsigned long ret = 0;
  size_t i;
  if (oldcrc ≡ 0) ret = (unsigned long)((23 ≪ 24) + (45 ≪ 16) + (67 ≪ 8) + 89);
  else
    for (i = 0; i < len; i++) ret = (ret ≪ 2) + buf[i];
  return ret;
}
static boolean mp_char_marked(MP mp, font_number f, eight_bits c)
{
  integer b;      /* char_base[f] */
  b = mp→char_base[f];
  if ((c ≥ mp→font_bc[f]) ∧ (c ≤ mp→font_ec[f]) ∧ (mp→font_info[b + c].qqqq.b3 ≠ 0)) return true;
  else return false;
}
static void make_subset_tag(MP mp, fm_entry * fm_cur, char **glyph_names, font_number tex_font)
{
  char tag[7];
  unsigned long crc;
  int i;
  size_t l;
  if (mp→ps→job_id_string ≡ Λ) mp_fatal_error(mp, "no␣job␣id!");
  l = strlen(mp→ps→job_id_string) + 1;
  alloc_array(char, l, SMALL_ARRAY_SIZE);
  strcpy(mp→ps→char_array, mp→ps→job_id_string);
  mp→ps→char_ptr = strend(mp→ps→char_array);
  if (fm_cur→tfm_name ≠ Λ) {
    fnstr_append(mp, "␣TFM␣name:␣");
    fnstr_append(mp, fm_cur→tfm_name);
  }
  fnstr_append(mp, "␣PS␣name:␣");
  if (fm_cur→ps_name ≠ Λ) fnstr_append(mp, fm_cur→ps_name);
  fnstr_append(mp, "␣Encoding:␣");
  if (fm_cur→encoding ≠ Λ ∧ (fm_cur→encoding)→file_name ≠ Λ)
    fnstr_append(mp, (fm_cur→encoding)→file_name);
  else fnstr_append(mp, "built-in");
  fnstr_append(mp, "␣CharSet:␣");
  for (i = 0; i < 256; i++)
    if (mp_char_marked(mp, tex_font, (eight_bits)i) ∧ glyph_names[i] ≠ notdef ∧ strcmp(glyph_names[i],
        notdef) ≠ 0) {
      if (glyph_names[i] ≠ Λ) {
        fnstr_append(mp, "/");
        fnstr_append(mp, glyph_names[i]);
      }
    }
  if (fm_cur→charset ≠ Λ) {
    fnstr_append(mp, "␣Extra␣CharSet:␣");
    fnstr_append(mp, fm_cur→charset);
  }
```

```
    crc = crc32(0_L, Z_NULL, 0);
    crc = crc32(crc, (Bytef *) mp→ps→char_array, strlen(mp→ps→char_array));
      /* we need to fit a 32-bit number into a string of 6 uppercase chars long; * there are 26 uppercase
        chars ==¿ each char represents a number in range * 0..25. The maximal number that can be
        represented by the tag is * 26^6 − 1, which is a number between 2^28 and 2^29. Thus the bits 29..31 *
        of the CRC must be dropped out. */
    for (i = 0; i < 6; i++) {
      tag[i] = (char)('A' + crc % 26);
      crc /= 26;
    }
    tag[6] = 0;
    mp_xfree(fm_cur→subset_tag);
    fm_cur→subset_tag = mp_xstrdup(mp, tag);
  }
```

75.

```
#define external_enc() (fm_cur→encoding)→glyph_names
#define is_used_char(c) mp_char_marked(mp, tex_font, (eight_bits)c)
#define end_last_eexec_line() mp→ps→hexline_length = HEXLINE_WIDTH;
        end_hexline(mp); mp→ps→t1_eexec_encrypt = false
#define t1_log(s) mp_print(mp, s)
#define t1_putchar(c) wps_chr(c)
#define embed_all_glyphs(tex_font) false
#define t1_char(c) c
#define extra_charset() mp→ps→dvips_extra_charset
#define update_subset_tag()
#define fixedcontent true
⟨ Globals 7 ⟩ +≡
#define PRINTF_BUF_SIZE 1024
  char *dvips_extra_charset;
  char *cur_enc_name;
  unsigned char *grid;
  char *ext_glyph_names[256];
  char print_buf[PRINTF_BUF_SIZE];
  size_t t1_byte_waiting;
  size_t t1_byte_length;
  unsigned char *t1_bytes;
```

76. ⟨ Set initial values 8 ⟩ +≡

```
  mp→ps→dvips_extra_charset = Λ;
  mp→ps→t1_byte_waiting = 0;
  mp→ps→t1_byte_length = 0;
  mp→ps→t1_bytes = Λ;
```

77.

```
#define t1_ungetchar() mp->ps->t1_byte_waiting--
#define t1_eof() (mp->ps->t1_byte_waiting ≥ mp->ps->t1_byte_length)
#define t1_close() do
        {
          (mp->close_file)(mp, mp->ps->t1_file);
          mp_xfree(mp->ps->t1_bytes);
          mp->ps->t1_bytes = Λ;
          mp->ps->t1_byte_waiting = 0;
          mp->ps->t1_byte_length = 0;
        }
        while (0)
#define valid_code(c) (c ≥ 0 ∧ c < 256)
  static int t1_getchar(MP mp)
  {
    if (mp->ps->t1_bytes ≡ Λ) {
      void *byte_ptr;
      (void) fseek(unwrap_file(mp->ps->t1_file), 0, SEEK_END);
      mp->ps->t1_byte_length = (size_t) ftell(unwrap_file(mp->ps->t1_file));
      (void) fseek(unwrap_file(mp->ps->t1_file), 0, SEEK_SET);
      mp->ps->t1_bytes = mp_xmalloc(mp, mp->ps->t1_byte_length, 1);
      byte_ptr = (void *) mp->ps->t1_bytes;
      (mp->read_binary_file)(mp, mp->ps->t1_file, &byte_ptr, &mp->ps->t1_byte_length);
    }
    return *(mp->ps->t1_bytes + mp->ps->t1_byte_waiting++);
  }
```

78. ⟨ Static variables in the outer block 24 ⟩ +≡

```
static const char *standard_glyph_names[256] = {notdef, notdef, notdef, notdef, notdef, notdef,
    notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef,
    notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef,
    "space", "exclam", "quotedbl", "numbersign", "dollar", "percent", "ampersand",
    "quoteright", "parenleft", "parenright", "asterisk", "plus", "comma", "hyphen", "period",
    "slash", "zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine",
    "colon", "semicolon", "less", "equal", "greater", "question", "at", "A", "B", "C", "D", "E", "F",
    "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z",
    "bracketleft", "backslash", "bracketright", "asciicircum", "underscore", "quoteleft", "a",
    "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v",
    "w", "x", "y", "z", "braceleft", "bar", "braceright", "asciitilde", notdef, notdef, notdef,
    notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef,
    notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef,
    notdef, notdef, notdef, notdef, notdef, "exclamdown", "cent", "sterling", "fraction", "yen",
    "florin", "section", "currency", "quotesingle", "quotedblleft", "guillemotleft",
    "guilsinglleft", "guilsinglright", "fi", "fl", notdef, "endash", "dagger", "daggerdbl",
    "periodcentered", notdef, "paragraph", "bullet", "quotesinglbase", "quotedblbase",
    "quotedblright", "guillemotright", "ellipsis", "perthousand", notdef, "questiondown",
    notdef, "grave", "acute", "circumflex", "tilde", "macron", "breve", "dotaccent", "dieresis",
    notdef, "ring", "cedilla", notdef, "hungarumlaut", "ogonek", "caron", "emdash", notdef, notdef,
    notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef, notdef,
    notdef, "AE", notdef, "ordfeminine", notdef, notdef, notdef, notdef, "Lslash", "Oslash", "OE",
    "ordmasculine", notdef, notdef, notdef, notdef, notdef, "ae", notdef, notdef, notdef, "dotlessi",
    notdef, notdef, "lslash", "oslash", "oe", "germandbls", notdef, notdef, notdef, notdef};
static const char charstringname[] = "/CharStrings";
```

79. ⟨ Globals 7 ⟩ +≡

```
char **t1_glyph_names;
char *t1_builtin_glyph_names[256];
char charsetstr[#4000];
boolean read_encoding_only;
int t1_encoding;
```

80.

```
#define T1_BUF_SIZE #100
#define CS_HSTEM 1
#define CS_VSTEM 3
#define CS_VMOVETO 4
#define CS_RLINETO 5
#define CS_HLINETO 6
#define CS_VLINETO 7
#define CS_RRCURVETO 8
#define CS_CLOSEPATH 9
#define CS_CALLSUBR 10
#define CS_RETURN 11
#define CS_ESCAPE 12
#define CS_HSBW 13
#define CS_ENDCHAR 14
#define CS_RMOVETO 21
#define CS_HMOVETO 22
#define CS_VHCURVETO 30
#define CS_HVCURVETO 31
#define CS_1BYTE_MAX (CS_HVCURVETO + 1)
#define CS_DOTSECTIONCS_1BYTE_MAX + 0
#define CS_VSTEM3CS_1BYTE_MAX + 1
#define CS_HSTEM3CS_1BYTE_MAX + 2
#define CS_SEACCS_1BYTE_MAX + 6
#define CS_SBWCS_1BYTE_MAX + 7
#define CS_DIVCS_1BYTE_MAX + 12
#define CS_CALLOTHERSUBRCS_1BYTE_MAX + 16
#define CS_POPCS_1BYTE_MAX + 17
#define CS_SETCURRENTPOINTCS_1BYTE_MAX + 33
#define CS_2BYTE_MAX (CS_SETCURRENTPOINT + 1)
#define CS_MAX CS_2BYTE_MAX
```

81. ⟨ Types 18 ⟩ +≡

```
typedef unsigned char byte;
typedef struct {
  byte nargs;      /* number of arguments */
  boolean bottom;      /* take arguments from bottom of stack? */
  boolean clear;      /* clear stack? */
  boolean valid;
} cc_entry;      /* CharString Command */
typedef struct {
  char *glyph_name;      /* glyph name (or notdef for Subrs entry) */
  byte *data;
  unsigned short len;      /* length of the whole string */
  unsigned short cslen;      /* length of the encoded part of the string */
  boolean is_used;
  boolean valid;
} cs_entry;
```

82.

```
#define t1_c1 52845
#define t1_c2 22719
⟨ Globals 7 ⟩ +≡
  unsigned short t1_dr, t1_er;
  unsigned short t1_cslen;
  short t1_lenIV;
```

83. ⟨ Types 18 ⟩ +≡

```
typedef char t1_line_entry;
typedef char t1_buf_entry;
```

84. ⟨ Globals 7 ⟩ +≡

```
t1_line_entry *t1_line_ptr, *t1_line_array;
size_t t1_line_limit;
t1_buf_entry *t1_buf_ptr, *t1_buf_array;
size_t t1_buf_limit;
int cs_start;
cs_entry *cs_tab, *cs_ptr, *cs_notdef;
char *cs_dict_start, *cs_dict_end;
int cs_count, cs_size, cs_size_pos;
cs_entry *subr_tab;
char *subr_array_start, *subr_array_end;
int subr_max, subr_size, subr_size_pos;
```

85. ⟨ Set initial values 8 ⟩ +≡

```
mp→ps→t1_line_array = Λ;
mp→ps→t1_buf_array = Λ;
```

86. This list contains the begin/end tokens commonly used in the /Subrs array of a Type 1 font.

⟨ Static variables in the outer block 24 ⟩ +≡

```
static const char *cs_token_pairs_list[][2] = {{"␣RD","NP"},{"␣-|","|"},{"␣RD","noaccess␣put"},
    {"␣-|","noaccess␣put"},{Λ,Λ}};
```

87. ⟨ Globals 7 ⟩ +≡

```
const char **cs_token_pair;
boolean t1_pfa, t1_cs, t1_scan, t1_eexec_encrypt, t1_synthetic;
int t1_in_eexec;      /* 0 before eexec-encrypted, 1 during, 2 after */
int t1_block_length;
int last_hexbyte;
void *t1_file;
int hexline_length;
```

88.

```
#define HEXLINE_WIDTH 64
⟨ Set initial values 8 ⟩ +≡
  mp→ps→hexline_length = 0;
```

89.

```
#define t1_prefix(s) str_prefix(mp→ps→t1_line_array, s)
#define t1_buf_prefix(s) str_prefix(mp→ps→t1_buf_array, s)
#define t1_suffix(s) str_suffix(mp→ps→t1_line_array, mp→ps→t1_line_ptr, s)
#define t1_buf_suffix(s) str_suffix(mp→ps→t1_buf_array, mp→ps→t1_buf_ptr, s)
#define t1_charstrings() strstr(mp→ps→t1_line_array, charstringname)
#define t1_subrs() t1_prefix("/Subrs")
#define t1_end_eexec() t1_suffix("mark␣currentfile␣closefile")
#define t1_cleartomark() t1_prefix("cleartomark")
  static void end_hexline(MP mp)
  {
    if (mp→ps→hexline_length ≥ HEXLINE_WIDTH) {
      wps_cr;
      mp→ps→hexline_length = 0;
    }
  }
  static void t1_check_pfa(MP mp)
  {
    const int c = t1_getchar(mp);
    mp→ps→t1_pfa = (c ≠ 128)true : false;
    t1_ungetchar( );
  }
  static int t1_getbyte(MP mp)
  {
    int c = t1_getchar(mp);
    if (mp→ps→t1_pfa) return c;
    if (mp→ps→t1_block_length ≡ 0) {
      if (c ≠ 128) mp_fatal_error(mp, "invalid␣marker");
      c = t1_getchar(mp);
      if (c ≡ 3) {
        while (¬t1_eof( )) (void) t1_getchar(mp);
        return EOF;
      }
      mp→ps→t1_block_length = t1_getchar(mp) & #ff;
      mp→ps→t1_block_length |= (int)(((unsigned) t1_getchar(mp) & #ff) ≪ 8);
      mp→ps→t1_block_length |= (int)(((unsigned) t1_getchar(mp) & #ff) ≪ 16);
      mp→ps→t1_block_length |= (int)(((unsigned) t1_getchar(mp) & #ff) ≪ 24);
      c = t1_getchar(mp);
    }
    mp→ps→t1_block_length−−;
    return c;
  }
  static int hexval(int c)
  {
    if (c ≥ 'A' ∧ c ≤ 'F') return c − 'A' + 10;
    else if (c ≥ 'a' ∧ c ≤ 'f') return c − 'a' + 10;
    else if (c ≥ '0' ∧ c ≤ '9') return c − '0';
    else return −1;
  }
  static byte edecrypt(MP mp, byte cipher)
```

```
{
  byte plain;
  if (mp→ps→t1_pfa) {
    while (cipher ≡ 10 ∨ cipher ≡ 13) cipher = (byte) t1_getbyte(mp);
    mp→ps→last_hexbyte = cipher = (byte)(((byte) hexval(cipher) ≪ 4) + hexval(t1_getbyte(mp)));
  }
  plain = (byte)(cipher ⊕ (mp→ps→t1_dr ≫ 8));
  mp→ps→t1_dr = (unsigned short)((cipher + mp→ps→t1_dr) * t1_c1 + t1_c2);
  return plain;
}
static byte cdecrypt(byte cipher, unsigned short *cr)
{
  const byte plain = (byte)(cipher ⊕ (*cr ≫ 8));
  *cr = (unsigned short)((cipher + *cr) * t1_c1 + t1_c2);
  return plain;
}
static byte eencrypt(MP mp, byte plain)
{
  const byte cipher = (byte)(plain ⊕ (mp→ps→t1_er ≫ 8));
  mp→ps→t1_er = (unsigned short)((cipher + mp→ps→t1_er) * t1_c1 + t1_c2);
  return cipher;
}
static byte cencrypt(byte plain, unsigned short *cr)
{
  const byte cipher = (byte)(plain ⊕ (*cr ≫ 8));
  *cr = (unsigned short)((cipher + *cr) * t1_c1 + t1_c2);
  return cipher;
}
static char *eol(char *s)
{
  char *p = strend(s);
  if (p ≠ Λ ∧ p − s > 1 ∧ p[−1] ≠ 10) {
    *p++ = 10;
    *p = 0;
  }
  return p;
}
static float t1_scan_num(MP mp, char *p, char **r)
{
  float f;
  char s[128];
  skip(p, '␣');
  if (sscanf(p, "%g", &f) ≠ 1) {
    remove_eol(p, mp→ps→t1_line_array);
    mp_snprintf(s, 128, "a␣number␣expected:␣‘%s’", mp→ps→t1_line_array);
    mp_fatal_error(mp, s);
  }
  if (r ≠ Λ) {
    for ( ; mp_isdigit(*p) ∨ *p ≡ '.' ∨ *p ≡ 'e' ∨ *p ≡ 'E' ∨ *p ≡ '+' ∨ *p ≡ '-'; p++) ;
```

```
    *r = p;
  }
  return f;
}
static boolean str_suffix(const char *begin_buf, const char *end_buf, const char *s)
{
  const char *s1 = end_buf - 1, *s2 = strend(s) - 1;
  if (*s1 ≡ 10) s1 --;
  while (s1 ≥ begin_buf ∧ s2 ≥ s) {
    if (*s1 -- ≠ *s2 --) return false;
  }
  return s2 < s;
}
```

90.

```
#define alloc_array(T,n,s) do
        {
          size_t nn = (size_t) n;
          if (mp→ps→T##_array ≡ Λ) {
            mp→ps→T##_limit = s;
            if (nn > mp→ps→T##_limit) mp→ps→T##_limit = nn;
            mp→ps→T##_array = mp_xmalloc(mp, mp→ps→T##_limit, sizeof (T##_entry));
            mp→ps→T##_ptr = mp→ps→T##_array;
          }
          else if ((size_t)(mp→ps→T##_ptr − mp→ps→T##_array) + nn > mp→ps→T##_limit) {
            size_t last_ptr_index;
            last_ptr_index = (size_t)(mp→ps→T##_ptr − mp→ps→T##_array);
            mp→ps→T##_limit ∗= 2;
            mp→ps→T##_limit += s;
            if ((size_t)(mp→ps→T##_ptr − mp→ps→T##_array) + nn > mp→ps→T##_limit)
              mp→ps→T##_limit = (size_t)(mp→ps→T##_ptr − mp→ps→T##_array) + nn;
            mp→ps→T##_array = mp_xrealloc(mp, mp→ps→T##_array, mp→ps→T##_limit, sizeof
                (T##_entry));
            mp→ps→T##_ptr = mp→ps→T##_array + last_ptr_index;
          }
        }
        while (0)
static void t1_getline(MP mp)
{
  int c, l, eexec_scan;
  char ∗p;
  static const char eexec_str[] = "currentfile␣eexec";
  static int eexec_len = 17;      /∗ strlen(eexec_str) ∗/
RESTART:
  if (t1_eof()) mp_fatal_error(mp, "unexpected␣end␣of␣file");
  mp→ps→t1_line_ptr = mp→ps→t1_line_array;
  alloc_array(t1_line, 1, T1_BUF_SIZE);
  mp→ps→t1_cslen = 0;
  eexec_scan = 0;
  c = t1_getbyte(mp);
  if (c ≡ EOF) goto EXIT;
  while (¬t1_eof()) {
    if (mp→ps→t1_in_eexec ≡ 1) c = edecrypt(mp, (byte) c);
    alloc_array(t1_line, 1, T1_BUF_SIZE);
    append_char_to_buf(c, mp→ps→t1_line_ptr, mp→ps→t1_line_array, mp→ps→t1_line_limit);
    if (mp→ps→t1_in_eexec ≡ 0 ∧ eexec_scan ≥ 0 ∧ eexec_scan < eexec_len) {
      if (mp→ps→t1_line_array[eexec_scan] ≡ eexec_str[eexec_scan]) eexec_scan++;
      else eexec_scan = −1;
    }
    if (c ≡ 10 ∨ (mp→ps→t1_pfa ∧ eexec_scan ≡ eexec_len ∧ c ≡ 32)) break;
    if (mp→ps→t1_cs ∧ mp→ps→t1_cslen ≡ 0 ∧ (mp→ps→t1_line_ptr − mp→ps→t1_line_array >
        4) ∧ (t1_suffix("␣RD␣") ∨ t1_suffix("␣-|␣"))) {
      p = mp→ps→t1_line_ptr − 5;
      while (∗p ≠ '␣') p−−;
      l = (int) t1_scan_num(mp, p + 1, 0);
```

```
    mp→ps→t1_cslen = (unsigned short) l;
    mp→ps→cs_start = (int)(mp→ps→t1_line_ptr − mp→ps→t1_line_array);
      /* mp→ps→cs_start is an index now */
    alloc_array(t1_line, l, T1_BUF_SIZE);
    while (l−− > 0) {
      *mp→ps→t1_line_ptr = (t1_line_entry) edecrypt(mp, (byte) t1_getbyte(mp));
      mp→ps→t1_line_ptr++;
    }
  }
  c = t1_getbyte(mp);
}
alloc_array(t1_line, 2, T1_BUF_SIZE);    /* append_eol can append 2 chars */
append_eol(mp→ps→t1_line_ptr, mp→ps→t1_line_array, mp→ps→t1_line_limit);
if (mp→ps→t1_line_ptr − mp→ps→t1_line_array < 2) goto RESTART;
if (eexec_scan ≡ eexec_len) mp→ps→t1_in_eexec = 1;
EXIT:    /* ensure that mp→ps→t1_buf_array has as much room as t1_line_array */
mp→ps→t1_buf_ptr = mp→ps→t1_buf_array;
alloc_array(t1_buf, mp→ps→t1_line_limit, mp→ps→t1_line_limit);
}
static void t1_putline(MP mp)
{
  char ss[256];
  int ss_cur = 0;
  static const char *hexdigits = "0123456789ABCDEF";
  char *p = mp→ps→t1_line_array;
  if (mp→ps→t1_line_ptr − mp→ps→t1_line_array ≤ 1) return;
  if (mp→ps→t1_eexec_encrypt) {
    while (p < mp→ps→t1_line_ptr) {
      byte b = eencrypt(mp, (byte) *p++);
      if (ss_cur ≥ 253) {
        ss[ss_cur] = '\0';
        (mp→write_ascii_file)(mp, mp→output_file, (char *) ss);
        ss_cur = 0;
      }
      ss[ss_cur++] = hexdigits[b/16];
      ss[ss_cur++] = hexdigits[b % 16];
      mp→ps→hexline_length += 2;
      if (mp→ps→hexline_length ≥ HEXLINE_WIDTH) {
        ss[ss_cur++] = '\n';
        mp→ps→hexline_length = 0;
      }
    }
  }
  else {
    while (p < mp→ps→t1_line_ptr) {
      if (ss_cur ≥ 255) {
        ss[ss_cur] = '\0';
        (mp→write_ascii_file)(mp, mp→output_file, (char *) ss);
        ss_cur = 0;
      }
      ss[ss_cur++] = (char)(*p++);
```

```
    }
  }
  ss[ss_cur] = '\0';
  (mp→write_ascii_file)(mp, mp→output_file, (char *) ss);
}
static void t1_puts(MP mp, const char *s)
{
  if (s ≠ mp→ps→t1_line_array) strcpy(mp→ps→t1_line_array, s);
  mp→ps→t1_line_ptr = strend(mp→ps→t1_line_array);
  t1_putline(mp);
}
static void t1_init_params(MP mp, const char *open_name_prefix, char *cur_file_name)
{
  if ((open_name_prefix ≠ Λ) ∧ strlen(open_name_prefix)) {
    t1_log(open_name_prefix);
    t1_log(cur_file_name);
  }
  mp→ps→t1_lenIV = 4;
  mp→ps→t1_dr = 55665;
  mp→ps→t1_er = 55665;
  mp→ps→t1_in_eexec = 0;
  mp→ps→t1_cs = false;
  mp→ps→t1_scan = true;
  mp→ps→t1_synthetic = false;
  mp→ps→t1_eexec_encrypt = false;
  mp→ps→t1_block_length = 0;
  t1_check_pfa(mp);
}
static void t1_close_font_file(MP mp, const char *close_name_suffix)
{
  if ((close_name_suffix ≠ Λ) ∧ strlen(close_name_suffix)) {
    t1_log(close_name_suffix);
  }
  t1_close( );
}
static void t1_check_block_len(MP mp, boolean decrypt)
{
  int l, c;
  char s[128];
  if (mp→ps→t1_block_length ≡ 0) return;
  c = t1_getbyte(mp);
  if (decrypt) c = edecrypt(mp, (byte) c);
  l = mp→ps→t1_block_length;
  if (¬(l ≡ 0 ∧ (c ≡ 10 ∨ c ≡ 13))) {
    mp_snprintf(s, 128, "%i␣bytes␣more␣than␣expected␣were␣ignored", l + 1);
    mp_warn(mp, s);
    while (l-- > 0) (void) t1_getbyte(mp);
  }
}
static void t1_start_eexec(MP mp, fm_entry * fm_cur)
{
```

```
  int i;
  if (¬mp→ps→t1_pfa) t1_check_block_len(mp, false);
  for (mp→ps→t1_line_ptr = mp→ps→t1_line_array, i = 0; i < 4; i++) {
    (void) edecrypt(mp, (byte) t1_getbyte(mp));
    *mp→ps→t1_line_ptr++ = 0;
  }
  mp→ps→t1_eexec_encrypt = true;
  if (¬mp→ps→read_encoding_only)
    if (is_included(fm_cur)) t1_putline(mp);      /* to put the first four bytes */
}
static void t1_stop_eexec(MP mp)
{
  int c;
  end_last_eexec_line( );
  if (¬mp→ps→t1_pfa) t1_check_block_len(mp, true);
  else {
    c = edecrypt(mp, (byte) t1_getbyte(mp));
    if (¬(c ≡ 10 ∨ c ≡ 13)) {
      if (mp→ps→last_hexbyte ≡ 0) t1_puts(mp, "00");
      else mp_warn(mp, "unexpected␣data␣after␣eexec");
    }
  }
  mp→ps→t1_cs = false;
  mp→ps→t1_in_eexec = 2;
}
static void t1_modify_fm(MP mp)
{
  mp→ps→t1_line_ptr = eol(mp→ps→t1_line_array);
}
static void t1_modify_italic(MP mp)
{
  mp→ps→t1_line_ptr = eol(mp→ps→t1_line_array);
}
```

91. ⟨ Types 18 ⟩ +≡

```
typedef struct {
  const char *pdfname;
  const char *t1name;
  float value;
  boolean valid;
} key_entry;
```

92.

```
#define FONT_KEYS_NUM 11
```

⟨ Declarations 29 ⟩ +≡

```
static key_entry font_keys[FONT_KEYS_NUM] = {{"Ascent", "Ascender", 0, false}, {"CapHeight",
    "CapHeight", 0, false}, {"Descent", "Descender", 0, false}, {"FontName", "FontName", 0, false},
    {"ItalicAngle", "ItalicAngle", 0, false}, {"StemV", "StdVW", 0, false}, {"XHeight", "XHeight", 0,
    false}, {"FontBBox", "FontBBox", 0, false}, {"", "", 0, false}, {"", "", 0, false}, {"", "", 0, false}};
```

93.

```
#define ASCENT_CODE 0
#define CAPHEIGHT_CODE 1
#define DESCENT_CODE 2
#define FONTNAME_CODE 3
#define ITALIC_ANGLE_CODE 4
#define STEMV_CODE 5
#define XHEIGHT_CODE 6
#define FONTBBOX1_CODE 7
#define FONTBBOX2_CODE 8
#define FONTBBOX3_CODE 9
#define FONTBBOX4_CODE 10
#define MAX_KEY_CODE (FONTBBOX1_CODE + 1)
  static void t1_scan_keys(MP mp, font_number tex_font, fm_entry * fm_cur)
  {
    int i, k;
    char *p, *r;
    key_entry *key;
    if (fm_extend(fm_cur) ≠ 0 ∨ fm_slant(fm_cur) ≠ 0) {
      if (t1_prefix("/FontMatrix")) {
        t1_modify_fm(mp);
        return;
      }
      if (t1_prefix("/ItalicAngle")) {
        t1_modify_italic(mp);
        return;
      }
    }
    if (t1_prefix("/FontType")) {
      p = mp→ps→t1_line_array + strlen("FontType") + 1;
      if ((i = (int) t1_scan_num(mp, p, 0)) ≠ 1) {
        char s[128];
        mp_snprintf(s, 125, "Type%d␣fonts␣unsupported␣by␣metapost", i);
        mp_fatal_error(mp, s);
      }
      return;
    }
    for (key = font_keys; key - font_keys < MAX_KEY_CODE; key++)
      if (str_prefix(mp→ps→t1_line_array + 1, key→t1name)) break;
    if (key - font_keys ≡ MAX_KEY_CODE) return;
    key→valid = true;
    p = mp→ps→t1_line_array + strlen(key→t1name) + 1;
    skip(p, '␣');
    if ((k = (int)(key - font_keys)) ≡ FONTNAME_CODE) {
      if (*p ≠ '/') {
        char s[128];
        remove_eol(p, mp→ps→t1_line_array);
        mp_snprintf(s, 128, "a␣name␣expected:␣`%s'", mp→ps→t1_line_array);
        mp_fatal_error(mp, s);
      }
      r = ++p;      /* skip the slash */
```

```
    if (is_included(fm_cur)) {      /* save the fontname */
      strncpy(mp→ps→fontname_buf, p, FONTNAME_BUF_SIZE);
      for (i = 0; mp→ps→fontname_buf[i] ≠ 10; i++) ;
      mp→ps→fontname_buf[i] = 0;
      if (is_subsetted(fm_cur)) {
        if (fm_cur→encoding ≠ Λ ∧ fm_cur→encoding→glyph_names ≠ Λ)
          make_subset_tag(mp, fm_cur, fm_cur→encoding→glyph_names, tex_font);
        else make_subset_tag(mp, fm_cur, mp→ps→t1_builtin_glyph_names, tex_font);
        alloc_array(t1_line, (size_t)(r − mp→ps→t1_line_array) + 6 + 1 + strlen(mp→ps→fontname_buf) + 1,
            T1_BUF_SIZE);
        strncpy(r, fm_cur→subset_tag, 6);
        *(r + 6) = '-';
        strncpy(r + 7, mp→ps→fontname_buf, strlen(mp→ps→fontname_buf) + 1);
        mp→ps→t1_line_ptr = eol(r);
      }
      else {      /* for (q = p; *q ≠ '␣' ∧ *q ≠ 10; *q++) ; */      /* *q = 0; */
        mp→ps→t1_line_ptr = eol(r);
      }
    }
    return;
  }
  if ((k ≡ STEMV_CODE ∨ k ≡ FONTBBOX1_CODE) ∧ (*p ≡ '[' ∨ *p ≡ '{')) p++;
  if (k ≡ FONTBBOX1_CODE) {
    for (i = 0; i < 4; i++) {
      key[i].value = t1_scan_num(mp, p, &r);
      p = r;
    }
    return;
  }
  key→value = t1_scan_num(mp, p, 0);
}
static void t1_scan_param(MP mp, font_number tex_font, fm_entry * fm_cur)
{
  static const char *lenIV = "/lenIV";
  if (¬mp→ps→t1_scan ∨ *mp→ps→t1_line_array ≠ '/') return;
  if (t1_prefix(lenIV)) {
    mp→ps→t1_lenIV = (short int) t1_scan_num(mp, mp→ps→t1_line_array + strlen(lenIV), 0);
    return;
  }
  t1_scan_keys(mp, tex_font, fm_cur);
}
static void copy_glyph_names(MP mp, char **glyph_names, int a, int b)
{
  if (glyph_names[b] ≠ notdef) mp_xfree(glyph_names[b]);
  glyph_names[b] = mp_xstrdup(mp, glyph_names[a]);
}
static void t1_builtin_enc(MP mp)
{
  int i, a, b, c, counter = 0;
  char *r, *p;      /* * At this moment "/Encoding" is the prefix of mp→ps→t1_line_array */
  if (t1_suffix("def")) {      /* predefined encoding */
```

```
  (void) sscanf(mp→ps→t1_line_array + strlen("/Encoding"), "%255s", mp→ps→t1_buf_array);
  if (strcmp(mp→ps→t1_buf_array, "StandardEncoding") ≡ 0) {
    for (i = 0; i < 256; i++) {
      if (mp→ps→t1_builtin_glyph_names[i] ≠ notdef) mp_xfree(mp→ps→t1_builtin_glyph_names[i]);
      mp→ps→t1_builtin_glyph_names[i] = mp_xstrdup(mp, standard_glyph_names[i]);
    }
    mp→ps→t1_encoding = ENC_STANDARD;
  }
  else {
    char s[128];
    mp_snprintf(s, 128, "cannot␣subset␣font␣(unknown␣predefined␣encoding␣‘%s’)",
        mp→ps→t1_buf_array);
    mp_fatal_error(mp, s);
  }
  return;
}
else mp→ps→t1_encoding = ENC_BUILTIN;
    /* * At this moment "/Encoding" is the prefix of mp→ps→t1_line_array, and the encoding is * not
      a predefined encoding * * We have two possible forms of Encoding vector. The first case is * *
      /Encoding [/a /b /c...] readonly def * * and the second case can look like * * /Encoding 256 array
      0 1 255 1 index exch /.notdef put for * dup 0 /x put * dup 1 /y put * ... * readonly def */
for (i = 0; i < 256; i++) {
  if (mp→ps→t1_builtin_glyph_names[i] ≠ notdef) {
    mp_xfree(mp→ps→t1_builtin_glyph_names[i]);
    mp→ps→t1_builtin_glyph_names[i] = mp_xstrdup(mp, notdef);
  }
}
if (t1_prefix("/Encoding␣[") ∨ t1_prefix("/Encoding[")) {      /* the first case */
  r = strchr(mp→ps→t1_line_array, '[') + 1;
  skip(r, '␣');
  for ( ; ; ) {
    while (*r ≡ '/') {
      for (p = mp→ps→t1_buf_array, r++; *r ≠ 32 ∧ *r ≠ 10 ∧ *r ≠ ']' ∧ *r ≠ '/'; *p++ = *r++) ;
      *p = 0;
      skip(r, '␣');
      if (counter > 255) {
        mp_fatal_error(mp, "encoding␣vector␣contains␣more␣than␣256␣names");
      }
      if (strcmp(mp→ps→t1_buf_array, notdef) ≠ 0) {
        if (mp→ps→t1_builtin_glyph_names[counter] ≠ notdef)
          mp_xfree(mp→ps→t1_builtin_glyph_names[counter]);
        mp→ps→t1_builtin_glyph_names[counter] = mp_xstrdup(mp, mp→ps→t1_buf_array);
      }
      counter++;
    }
    if (*r ≠ 10 ∧ *r ≠ '%') {
      if (str_prefix(r, "]␣def") ∨ str_prefix(r, "]␣readonly␣def")) break;
      else {
        char s[128];
        remove_eol(r, mp→ps→t1_line_array);
        mp_snprintf(s, 128, "a␣name␣or␣‘]␣def’␣or␣‘]␣readonly␣def’␣expected:␣‘%s’",
            mp→ps→t1_line_array);
```

```
        mp_fatal_error(mp, s);
      }
    }
    t1_getline(mp);
    r = mp→ps→t1_line_array;
  }
}
else {      /* the second case */
  p = strchr(mp→ps→t1_line_array, 10);
  for ( ; p ≠ Λ; ) {
    if (*p ≡ 10) {
      t1_getline(mp);
      p = mp→ps→t1_line_array;
    }      /* check for 'dup ¡index¿ ¡glyph¿ put' */
    if (sscanf(p, "dup␣%i%255s␣put", &i,
          mp→ps→t1_buf_array) ≡ 2 ∧ *mp→ps→t1_buf_array ≡ '/' ∧ valid_code(i)) {
      if (strcmp(mp→ps→t1_buf_array + 1, notdef) ≠ 0) {
        if (mp→ps→t1_builtin_glyph_names[i] ≠ notdef) mp_xfree(mp→ps→t1_builtin_glyph_names[i]);
        mp→ps→t1_builtin_glyph_names[i] = mp_xstrdup(mp, mp→ps→t1_buf_array + 1);
      }
      p = strstr(p, "␣put") + strlen("␣put");
      skip(p, '␣');
    }      /* check for 'dup dup ¡to¿ exch ¡from¿ get put' */
    else if (sscanf(p, "dup␣dup␣%i␣exch␣%i␣get␣put", &b, &a) ≡ 2 ∧ valid_code(a) ∧ valid_code(b)) {
      copy_glyph_names(mp, mp→ps→t1_builtin_glyph_names, a, b);
      p = strstr(p, "␣get␣put") + strlen("␣get␣put");
      skip(p, '␣');
    }      /* check for 'dup dup ¡from¿ ¡size¿ getinterval ¡to¿ exch putinterval' */
    else if (sscanf(p, "dup␣dup␣%i␣%i␣getinterval␣%i␣exch␣putinterval", &a, &c,
          &b) ≡ 3 ∧ valid_code(a) ∧ valid_code(b) ∧ valid_code(c)) {
      for (i = 0; i < c; i++) copy_glyph_names(mp, mp→ps→t1_builtin_glyph_names, a + i, b + i);
      p = strstr(p, "␣putinterval") + strlen("␣putinterval");
      skip(p, '␣');
    }      /* check for 'def' or 'readonly def' */
    else if ((p ≡ mp→ps→t1_line_array ∨ (p > mp→ps→t1_line_array ∧ p[−1] ≡ '␣')) ∧ strcmp(p, "def\n") ≡ 0)
      return;      /* skip an unrecognizable word */
    else {
      while (*p ≠ '␣' ∧ *p ≠ 10) p++;
      skip(p, '␣');
    }
  }
}
}
static void t1_check_end(MP mp)
{
  if (t1_eof( )) return;
  t1_getline(mp);
  if (t1_prefix("{restore}")) t1_putline(mp);
}
```

94. ⟨ Set initial values 8 ⟩ +≡

```
{
  int i;
  for (i = 0; i < 256; i++) {
    mp→ps→t1_builtin_glyph_names[i] = strdup(notdef);
    assert(mp→ps→t1_builtin_glyph_names[i]);
  }
}
```

95. ⟨ Types 18 ⟩ +≡

```
typedef struct {
  char *ff_name;      /* base name of font file */
  char *ff_path;      /* full path to font file */
} ff_entry;
```

96. **static** *boolean t1_open_fontfile*(MP *mp*, *fm_entry* ∗ *fm_cur*, **const char** ∗*open_name_prefix*)

```
{
  ff_entry *ff;
  ff = check_ff_exist(mp, fm_cur);
  mp->ps->t1_file = Λ;
  if (ff->ff_path ≠ Λ) {
    mp->ps->t1_file = (mp->open_file)(mp, ff->ff_path, "r", mp_filetype_font);
  }
  if (mp->ps->t1_file ≡ Λ) {
    char err[256];
    mp_snprintf(err, 255, "cannot␣open␣Type␣1␣font␣file␣%s␣for␣reading", ff->ff_path);
    mp_warn(mp, err);
    return false;
  }
  t1_init_params(mp, open_name_prefix, fm_cur->ff_name);
  mp->ps->fontfile_found = true;
  return true;
}
static void t1_scan_only(MP mp, font_number tex_font, fm_entry * fm_cur)
{
  do {
    t1_getline(mp);
    t1_scan_param(mp, tex_font, fm_cur);
  } while (mp->ps->t1_in_eexec ≡ 0);
  t1_start_eexec(mp, fm_cur);
  do {
    t1_getline(mp);
    t1_scan_param(mp, tex_font, fm_cur);
  } while (¬(t1_charstrings( ) ∨ t1_subrs( )));
}
static void t1_include(MP mp, font_number tex_font, fm_entry * fm_cur)
{
  do {
    t1_getline(mp);
    t1_scan_param(mp, tex_font, fm_cur);
    t1_putline(mp);
  } while (mp->ps->t1_in_eexec ≡ 0);
  t1_start_eexec(mp, fm_cur);
  do {
    t1_getline(mp);
    t1_scan_param(mp, tex_font, fm_cur);
    t1_putline(mp);
  } while (¬(t1_charstrings( ) ∨ t1_subrs( )));
  mp->ps->t1_cs = true;
  do {
    t1_getline(mp);
    t1_putline(mp);
  } while (¬t1_end_eexec( ));
  t1_stop_eexec(mp);
  if (fixedcontent) {      /* copy 512 zeros (not needed for PDF) */
    do {
```

```
      t1_getline(mp);
      t1_putline(mp);
    } while (¬t1_cleartomark());
    t1_check_end(mp);      /* write "restoreif" if found */
  }
}
```

97.

```
#define check_subr(SUBR)
        if (SUBR ≥ mp→ps→subr_size ∨ SUBR < 0) {
          char s[128];
          mp_snprintf(s, 128, "Subrs␣array:␣entry␣index␣out␣of␣range␣(%i)", SUBR);
          mp_fatal_error(mp, s);
        }
  static const char **check_cs_token_pair(MP mp)
  {
    const char **p = (const char **) cs_token_pairs_list;
    for ( ; p[0] ≠ Λ; ++p)
      if (t1_buf_prefix(p[0]) ∧ t1_buf_suffix(p[1])) return p;
    return Λ;
  }
  static void cs_store(MP mp, boolean is_subr)
  {
    char *p;
    cs_entry *ptr;
    int subr;
    for (p = mp→ps→t1_line_array, mp→ps→t1_buf_ptr = mp→ps→t1_buf_array; *p ≠ '␣';
          *mp→ps→t1_buf_ptr++ = *p++) ;
    *mp→ps→t1_buf_ptr = 0;
    if (is_subr) {
      subr = (int) t1_scan_num(mp, p + 1, 0);
      check_subr(subr);
      ptr = mp→ps→subr_tab + subr;
    }
    else {
      ptr = mp→ps→cs_ptr++;
      if (mp→ps→cs_ptr − mp→ps→cs_tab > mp→ps→cs_size) {
        char s[128];
        mp_snprintf(s, 128, "CharStrings␣dict:␣more␣entries␣than␣dict␣size␣(%i)", mp→ps→cs_size);
        mp_fatal_error(mp, s);
      }
      ptr→glyph_name = mp_xstrdup(mp, mp→ps→t1_buf_array + 1);
    }     /* copy " RD " + cs data to mp→ps→t1_buf_array */
    memcpy(mp→ps→t1_buf_array, mp→ps→t1_line_array + mp→ps→cs_start − 4, (size_t)(mp→ps→t1_cslen + 4));
       /* copy the end of cs data to mp→ps→t1_buf_array */
    for (p = mp→ps→t1_line_array + mp→ps→cs_start + mp→ps→t1_cslen,
          mp→ps→t1_buf_ptr = mp→ps→t1_buf_array + mp→ps→t1_cslen + 4; *p ≠ 10;
          *mp→ps→t1_buf_ptr++ = *p++) ;
    *mp→ps→t1_buf_ptr++ = 10;
    if (is_subr ∧ mp→ps→cs_token_pair ≡ Λ) mp→ps→cs_token_pair = check_cs_token_pair(mp);
    ptr→len = (unsigned short)(mp→ps→t1_buf_ptr − mp→ps→t1_buf_array);
    ptr→cslen = mp→ps→t1_cslen;
    ptr→data = mp_xmalloc(mp, (size_t) ptr→len, sizeof(byte));
    memcpy(ptr→data, mp→ps→t1_buf_array, (size_t) ptr→len);
    ptr→valid = true;
  }
#define store_subr(mp) cs_store (mp, true)
```

```
#define store_cs(mp) cs_store (mp, false)
#define CC_STACK_SIZE 24
  static double cc_stack[CC_STACK_SIZE], *stack_ptr = cc_stack;
  static cc_entry cc_tab[CS_MAX];
  static boolean is_cc_init = false;
#define cc_pop(N)
  if ((stack_ptr - cc_stack < (N)) stack_error(N);
  stack_ptr -= N
#define stack_error(N)
  {
    char s[256];
    mp_snprintf(s, 255, "CharString: invalid access (%i) to stack (%i entries)", (int)
        N, (int)(stack_ptr - cc_stack));
    mp_warn(mp, s);
    goto cs_error;
  }
#define cc_get(N) ((N) < 0*(stack_ptr + (N)) : *(cc_stack + (N)))
#define cc_push(V) * stack_ptr++ = (double)(V)
#define cc_clear( ) stack_ptr = cc_stack
#define set_cc(N, B, A, C) cc_tab[N].nargs = A;
  cc_tab[N].bottom = B;
  cc_tab[N].clear = C; cc_tab[N].valid = true
  static void cc_init(void)
  {
    int i;
    if (is_cc_init) return;
    for (i = 0; i < CS_MAX; i++) cc_tab[i].valid = false;
    set_cc(CS_HSTEM, true, 2, true);
    set_cc(CS_VSTEM, true, 2, true);
    set_cc(CS_VMOVETO, true, 1, true);
    set_cc(CS_RLINETO, true, 2, true);
    set_cc(CS_HLINETO, true, 1, true);
    set_cc(CS_VLINETO, true, 1, true);
    set_cc(CS_RRCURVETO, true, 6, true);
    set_cc(CS_CLOSEPATH, false, 0, true);
    set_cc(CS_CALLSUBR, false, 1, false);
    set_cc(CS_RETURN, false, 0, false);       /* set_cc(CS_ESCAPE, false, 0, false); */
    set_cc(CS_HSBW, true, 2, true);
    set_cc(CS_ENDCHAR, false, 0, true);
    set_cc(CS_RMOVETO, true, 2, true);
    set_cc(CS_HMOVETO, true, 1, true);
    set_cc(CS_VHCURVETO, true, 4, true);
    set_cc(CS_HVCURVETO, true, 4, true);
    set_cc(CS_DOTSECTION, false, 0, true);
    set_cc(CS_VSTEM3, true, 6, true);
    set_cc(CS_HSTEM3, true, 6, true);
    set_cc(CS_SEAC, true, 5, true);
    set_cc(CS_SBW, true, 4, true);
    set_cc(CS_DIV, false, 2, false);
    set_cc(CS_CALLOTHERSUBR, false, 0, false);
    set_cc(CS_POP, false, 0, false);
```

```
    set_cc(CS_SETCURRENTPOINT, true, 2, true);
    is_cc_init = true;
  }
```

98.

```
#define cs_getchar(mp) cdecrypt(*data++,&cr)
#define mark_subr(mp,n) cs_mark(mp,0,n)
#define mark_cs(mp,s) cs_mark(mp,s,0)
#define SMALL_BUF_SIZE 256
  static void cs_warn(MP mp,const char *cs_name,int subr,const char *fmt,...)
  {
    char buf[SMALL_BUF_SIZE];
    char s[300];
    va_list args;
    va_start(args,fmt);
    |/*@-bufferoverflowhigh@*/ (void)vsprintf(buf,fmt,args);
    |/*@=bufferoverflowhigh@*/ va_end(args);
    if (cs_name≡Λ) {
      mp_snprintf(s,299,"Subr␣(%i):␣%s",(int) subr,buf);
    }
    else {
      mp_snprintf(s,299,"CharString␣(/%s):␣%s",cs_name,buf);
    }
    mp_warn(mp,s);
  }
  static void cs_mark(MP mp,const char *cs_name,int subr)
  {
    byte *data;
    int i, b, cs_len;
    integer a, a1, a2;
    unsigned short cr;
    static integer lastargOtherSubr3 = 3;  /* the argument of last call to OtherSubrs[3] */
    cs_entry *ptr;
    cc_entry *cc;
    if (cs_name≡Λ) {
      check_subr(subr);
      ptr = mp→ps→subr_tab + subr;
      if (¬ptr→valid) return;
    }
    else {
      if (mp→ps→cs_notdef ≠ Λ ∧ (cs_name≡notdef ∨ strcmp(cs_name,notdef)≡0))
        ptr = mp→ps→cs_notdef;
      else {
        for (ptr = mp→ps→cs_tab; ptr < mp→ps→cs_ptr; ptr++)
          if (strcmp(ptr→glyph_name,cs_name)≡0) break;
        if (ptr≡mp→ps→cs_ptr) {
          char s[128];
          mp_snprintf(s,128,"glyph␣‘%s’␣undefined",cs_name);
          mp_warn(mp,s);
          return;
        }
        if (ptr→glyph_name≡notdef) mp→ps→cs_notdef = ptr;
      }
```

```
} /* only marked CharString entries and invalid entries can be skipped; valid marked subrs must
     be parsed to keep the stack in sync */
if (¬ptr→valid ∨ (ptr→is_used ∧ cs_name ≠ Λ)) return;
ptr→is_used = true;
cr = 4330;
cs_len = (int) ptr→cslen;
data = ptr→data + 4;
for (i = 0; i < mp→ps→t1_lenIV; i++, cs_len−−) (void) cs_getchar(mp);
while (cs_len > 0) {
  −−cs_len;
  b = cs_getchar(mp);
  if (b ≥ 32) {
    if (b ≤ 246) a = b − 139;
    else if (b ≤ 250) {
      −−cs_len;
      a = (int)((unsigned)(b − 247) ≪ 8) + 108 + cs_getchar(mp);
    }
    else if (b ≤ 254) {
      −−cs_len;
      a = −(int)((unsigned)(b − 251) ≪ 8) − 108 − cs_getchar(mp);
    }
    else {
      cs_len −= 4;
      a = (cs_getchar(mp) & #ff) ≪ 24;
      a |= (cs_getchar(mp) & #ff) ≪ 16;
      a |= (cs_getchar(mp) & #ff) ≪ 8;
      a |= (cs_getchar(mp) & #ff) ≪ 0;
      if (sizeof (integer) > 4 ∧ (a & #80000000)) a |= ∼#7FFFFFFF;
    }
    cc_push(a);
  }
  else {
    if (b ≡ CS_ESCAPE) {
      b = cs_getchar(mp) + CS_1BYTE_MAX;
      cs_len−−;
    }
    if (b ≥ CS_MAX) {
      cs_warn(mp, cs_name, subr, "command␣value␣out␣of␣range:␣%i", (int) b);
      goto cs_error;
    }
    cc = cc_tab + b;
    if (¬cc→valid) {
      cs_warn(mp, cs_name, subr, "command␣not␣valid:␣%i", (int) b);
      goto cs_error;
    }
    if (cc→bottom) {
      if (stack_ptr − cc_stack < cc→nargs)
        cs_warn(mp, cs_name, subr, "less␣arguments␣on␣stack␣(%i)␣than␣required␣(%i)",
            (int)(stack_ptr − cc_stack), (int) cc→nargs);
      else if (stack_ptr − cc_stack > cc→nargs)
        cs_warn(mp, cs_name, subr, "more␣arguments␣on␣stack␣(%i)␣than␣required␣(%i)",
            (int)(stack_ptr − cc_stack), (int) cc→nargs);
```

```
    }
    switch (cc − cc_tab) {
    case CS_CALLSUBR: a1 = (integer)cc_get(−1);
      cc_pop(1);
      mark_subr(mp, a1);
      if (¬mp→ps→subr_tab[a1].valid) {
        cs_warn(mp, cs_name, subr, "cannot␣call␣subr␣(%i)", (int) a1);
        goto cs_error;
      }
      break;
    case CS_DIV: cc_pop(2);
      cc_push(0);
      break;
    case CS_CALLOTHERSUBR: a1 = (integer)cc_get(−1);
      if (a1 ≡ 3) lastargOtherSubr3 = (integer)cc_get(−3);
      a1 = (integer)cc_get(−2) + 2;
      cc_pop(a1);
      break;
    case CS_POP: cc_push(lastargOtherSubr3);
        /* the only case when we care about the value being pushed onto stack is when POP follows
          CALLOTHERSUBR (changing hints by OtherSubrs[3]) */
      break;
    case CS_SEAC: a1 = (integer)cc_get(3);
      a2 = (integer)cc_get(4);
      cc_clear( );
      mark_cs(mp, standard_glyph_names[a1]);
      mark_cs(mp, standard_glyph_names[a2]);
      break;
    default:
      if (cc→clear) cc_clear( );
    }
  }
 }
 return;
cs_error: /* an error occured during parsing */
 cc_clear( );
 ptr→valid = false;
 ptr→is_used = false;
}
static void t1_subset_ascii_part(MP mp, font_number tex_font, fm_entry * fm_cur)
{
 int i, j;
 t1_getline(mp);
 while (¬t1_prefix("/Encoding")) {
  t1_scan_param(mp, tex_font, fm_cur); /* Patch the initial font directory cacheing mechanism
      found in some * pfb fonts. * * Even though the T1 spec does not explicitly state that
      'FontDirectory' * should appear at the start of a line, luckily this is standard practise. */
  if (t1_prefix("FontDirectory")) {
   char *endloc, *p;
   char new_line[T1_BUF_SIZE] = {0};
   p = mp→ps→t1_line_array;
```

```
    while ((endloc = strstr(p, fm_cur→ps_name)) ≠ Λ) {
      int n = (endloc − mp→ps→t1_line_array)+strlen(fm_cur→subset_tag)+2+strlen(fm_cur→ps_name);
      if (n ≥ T1_BUF_SIZE) {
        mp_fatal_error(mp, "t1_subset_ascii_part: buffer overrun detected.");
      }
      strncat(new_line, p, (endloc − p));
      strcat(new_line, fm_cur→subset_tag);
      strcat(new_line, "-");
      strcat(new_line, fm_cur→ps_name);
      p = endloc + strlen(fm_cur→ps_name);
    }
    if (strlen(new_line) + strlen(p) + 1 ≥ T1_BUF_SIZE) {
      mp_fatal_error(mp, "t1_subset_ascii_part: buffer overrun detected.");
    }
    strcat(new_line, p);
    strcpy(mp→ps→t1_line_array, new_line);
    mp→ps→t1_line_ptr = mp→ps→t1_line_array + strlen(mp→ps→t1_line_array);
    t1_putline(mp);
  }
  else {
    t1_putline(mp);
  }
  t1_getline(mp);
}
t1_builtin_enc(mp);
if (is_reencoded(fm_cur)) mp→ps→t1_glyph_names = external_enc( );
else mp→ps→t1_glyph_names = mp→ps→t1_builtin_glyph_names;
if ((¬is_subsetted(fm_cur)) ∧ mp→ps→t1_encoding ≡ ENC_STANDARD)
  t1_puts(mp, "/Encoding StandardEncoding def\n");
else {
  t1_puts(mp, "/Encoding 256 array\n0 1 255 {1 index exch /.notdef put} for\n");
  for (i = 0, j = 0; i < 256; i++) {
    if (is_used_char(i) ∧ mp→ps→t1_glyph_names[i] ≠ notdef ∧ strcmp(mp→ps→t1_glyph_names[i],
        notdef) ≠ 0) {
      j++;
      mp_snprintf(mp→ps→t1_line_array, (int) mp→ps→t1_line_limit, "dup %i /%s put\n", (int)
          t1_char(i), mp→ps→t1_glyph_names[i]);
      t1_puts(mp, mp→ps→t1_line_array);
    }
  }   /* We didn't mark anything for the Encoding array. */
      /* We add "dup 0 /.notdef put" for compatibility */   /* with Acrobat 5.0. */
  if (j ≡ 0) t1_puts(mp, "dup 0 /.notdef put\n");
  t1_puts(mp, "readonly def\n");
}
do {
  t1_getline(mp);
  t1_scan_param(mp, tex_font, fm_cur);
  if (¬t1_prefix("/UniqueID"))   /* ignore UniqueID for subsetted fonts */
    t1_putline(mp);
} while (mp→ps→t1_in_eexec ≡ 0);
}
#define t1_subr_flush(mp) t1_flush_cs (mp, true)
```

```
#define t1_cs_flush(mp) t1_flush_cs (mp, false)
  static void cs_init(MP mp)
  {
    mp→ps→cs_ptr = mp→ps→cs_tab = Λ;
    mp→ps→cs_dict_start = mp→ps→cs_dict_end = Λ;
    mp→ps→cs_count = mp→ps→cs_size = mp→ps→cs_size_pos = 0;
    mp→ps→cs_token_pair = Λ;
    mp→ps→subr_tab = Λ;
    mp→ps→subr_array_start = mp→ps→subr_array_end = Λ;
    mp→ps→subr_max = mp→ps→subr_size = mp→ps→subr_size_pos = 0;
  }
  static void init_cs_entry(cs_entry *cs)
  {
    cs→data = Λ;
    cs→glyph_name = Λ;
    cs→len = 0;
    cs→cslen = 0;
    cs→is_used = false;
    cs→valid = false;
  }
  static void t1_mark_glyphs(MP mp, font_number tex_font);
  static void t1_read_subrs(MP mp, font_number tex_font, fm_entry * fm_cur, int read_only)
  {
    int i, s;
    cs_entry *ptr;
    t1_getline(mp);
    while (¬(t1_charstrings() ∨ t1_subrs())) {
      t1_scan_param(mp, tex_font, fm_cur);
      if (¬read_only) t1_putline(mp);
      t1_getline(mp);
    }
  FOUND: mp→ps→t1_cs = true;
    mp→ps→t1_scan = false;
    if (¬t1_subrs()) return;
    mp→ps→subr_size_pos = (int)(strlen("/Subrs") + 1);
      /* subr_size_pos points to the number indicating dict size after "/Subrs" */
    mp→ps→subr_size = (int) t1_scan_num(mp, mp→ps→t1_line_array + mp→ps→subr_size_pos, 0);
    if (mp→ps→subr_size ≡ 0) {
      while (¬t1_charstrings()) t1_getline(mp);
      return;
    }     /* subr_tab = xtalloc(subr_size, cs_entry); */
    mp→ps→subr_tab = (cs_entry *) mp_xmalloc(mp, (size_t) mp→ps→subr_size, sizeof(cs_entry));
    for (ptr = mp→ps→subr_tab; ptr − mp→ps→subr_tab < mp→ps→subr_size; ptr++) init_cs_entry(ptr);
    mp→ps→subr_array_start = mp_xstrdup(mp, mp→ps→t1_line_array);
    t1_getline(mp);
    while (mp→ps→t1_cslen) {
      store_subr(mp);
      t1_getline(mp);
    }     /* mark the first four entries without parsing */
    for (i = 0; i < mp→ps→subr_size ∧ i < 4; i++)
      mp→ps→subr_tab[i].is_used = true;     /* the end of the Subrs array might have more than one line
```

so we need to concatnate them to *subr_array_end*. Unfortunately some fonts don't have the Subrs array followed by the CharStrings dict immediately (synthetic fonts). If we cannot find CharStrings in next `POST_SUBRS_SCAN` lines then we will treat the font as synthetic and ignore everything until next Subrs is found */

```
#define POST_SUBRS_SCAN 5
  s = 0;
  *mp->ps->t1_buf_array = 0;
  for (i = 0; i < POST_SUBRS_SCAN; i++) {
    if (t1_charstrings()) break;
    s += (int)(mp->ps->t1_line_ptr - mp->ps->t1_line_array);
    alloc_array(t1_buf, s, T1_BUF_SIZE);
    strcat(mp->ps->t1_buf_array, mp->ps->t1_line_array);
    t1_getline(mp);
  }
  mp->ps->subr_array_end = mp_xstrdup(mp, mp->ps->t1_buf_array);
  if (i ≡ POST_SUBRS_SCAN) {      /* CharStrings not found; suppose synthetic font */
    for (ptr = mp->ps->subr_tab; ptr - mp->ps->subr_tab < mp->ps->subr_size; ptr++)
      if (ptr->valid) mp_xfree(ptr->data);
    mp_xfree(mp->ps->subr_tab);
    mp_xfree(mp->ps->subr_array_start);
    mp_xfree(mp->ps->subr_array_end);
    cs_init(mp);
    mp->ps->t1_cs = false;
    mp->ps->t1_synthetic = true;
    while (¬(t1_charstrings() ∨ t1_subrs())) t1_getline(mp);
    goto FOUND;
  }
}
```

```
99. static void t1_flush_cs(MP mp, boolean is_subr)
{
  char *p;
  byte *r, *return_cs = Λ;
  cs_entry *tab, *end_tab, *ptr;
  char *start_line, *line_end;
  int count, size_pos;
  unsigned short cr, cs_len = 0;      /* to avoid warning about uninitialized use of cs_len */
  if (is_subr) {
    start_line = mp->ps->subr_array_start;
    line_end = mp->ps->subr_array_end;
    size_pos = mp->ps->subr_size_pos;
    tab = mp->ps->subr_tab;
    count = mp->ps->subr_max + 1;
    end_tab = mp->ps->subr_tab + count;
  }
  else {
    start_line = mp->ps->cs_dict_start;
    line_end = mp->ps->cs_dict_end;
    size_pos = mp->ps->cs_size_pos;
    tab = mp->ps->cs_tab;
    end_tab = mp->ps->cs_ptr;
    count = mp->ps->cs_count;
  }
  mp->ps->t1_line_ptr = mp->ps->t1_line_array;
  for (p = start_line; p - start_line < size_pos; ) *mp->ps->t1_line_ptr++ = *p++;
  while (mp_isdigit(*p)) p++;
  mp_snprintf(mp->ps->t1_line_ptr, (int) mp->ps->t1_line_limit, "%u", (unsigned) count);
  strcat(mp->ps->t1_line_ptr, p);
  mp->ps->t1_line_ptr = eol(mp->ps->t1_line_array);
  t1_putline(mp);      /* create return_cs to replace unsused subr's */
  if (is_subr) {
    cr = 4330;
    cs_len = 0;
    return_cs = mp_xmalloc(mp, (size_t)(mp->ps->t1_lenIV + 1), sizeof(byte));
    if (mp->ps->t1_lenIV ≥ 0) {
      for (cs_len = 0, r = return_cs; cs_len < (unsigned short) mp->ps->t1_lenIV; cs_len++, r++)
        *r = cencrypt(#00, &cr);
      *r = cencrypt(CS_RETURN, &cr);
    }
    else {
      *return_cs = CS_RETURN;
    }
    cs_len++;
  }
  for (ptr = tab; ptr < end_tab; ptr++) {
    if (ptr->is_used) {
      if (is_subr) mp_snprintf(mp->ps->t1_line_array, (int) mp->ps->t1_line_limit, "dup␣%i␣%u",
          (int)(ptr - tab), ptr->cslen);
      else mp_snprintf(mp->ps->t1_line_array, (int) mp->ps->t1_line_limit, "/%s␣%u", ptr->glyph_name,
          ptr->cslen);
      p = strend(mp->ps->t1_line_array);
```

```
    memcpy(p, ptr→data, (size_t) ptr→len);
    mp→ps→t1_line_ptr = p + ptr→len;
    t1_putline(mp);
  }
  else {      /* replace unsused subr's by return_cs */
    if (is_subr) {
      mp_snprintf(mp→ps→t1_line_array, (int) mp→ps→t1_line_limit, "dup␣%i␣%u%s␣", (int)(ptr − tab),
          cs_len, mp→ps→cs_token_pair[0]);
      p = strend(mp→ps→t1_line_array);
      memcpy(p, return_cs, (size_t) cs_len);
      mp→ps→t1_line_ptr = p + cs_len;
      t1_putline(mp);
      mp_snprintf(mp→ps→t1_line_array, (int) mp→ps→t1_line_limit, "␣%s", mp→ps→cs_token_pair[1]);
      mp→ps→t1_line_ptr = eol(mp→ps→t1_line_array);
      t1_putline(mp);
    }
  }
  mp_xfree(ptr→data);
  if (ptr→glyph_name ≠ notdef) mp_xfree(ptr→glyph_name);
}
mp_snprintf(mp→ps→t1_line_array, (int) mp→ps→t1_line_limit, "%s", line_end);
mp_xfree(mp→ps→t1_line_ptr = eol(mp→ps→t1_line_array);
t1_putline(mp);
if (is_subr) mp_xfree(return_cs);
mp_xfree(tab);
mp_xfree(start_line);
mp_xfree(line_end);
if (is_subr) {
  mp→ps→subr_array_start = Λ;
  mp→ps→subr_array_end = Λ;
  mp→ps→subr_tab = Λ;
}
else {
  mp→ps→cs_dict_start = Λ;
  mp→ps→cs_dict_end = Λ;
  mp→ps→cs_tab = Λ;
}
}
static void t1_mark_glyphs(MP mp, font_number tex_font)
{
  int i;
  char *charset = extra_charset( );
  char *g, *s, *r;
  cs_entry *ptr;
  if (mp→ps→t1_synthetic ∨ embed_all_glyphs(tex_font)) {      /* mark everything */
    if (mp→ps→cs_tab ≠ Λ)
      for (ptr = mp→ps→cs_tab; ptr < mp→ps→cs_ptr; ptr++)
        if (ptr→valid) ptr→is_used = true;
    if (mp→ps→subr_tab ≠ Λ) {
      for (ptr = mp→ps→subr_tab; ptr − mp→ps→subr_tab < mp→ps→subr_size; ptr++)
        if (ptr→valid) ptr→is_used = true;
      mp→ps→subr_max = mp→ps→subr_size − 1;
```

```
    }
    return;
  }
  mark_cs(mp, notdef);
  for (i = 0; i < 256; i++)
    if (is_used_char(i)) {
      if (mp→ps→t1_glyph_names[i] ≡ notdef ∨ strcmp(mp→ps→t1_glyph_names[i], notdef) ≡ 0) {
        char S[128];
        mp_snprintf(S, 128, "character␣%i␣is␣mapped␣to␣%s", i, notdef);
        mp_warn(mp, S);
      }
      else mark_cs(mp, mp→ps→t1_glyph_names[i]);
    }
  if (charset ≡ Λ) goto SET_SUBR_MAX;
  g = s = charset + 1;      /* skip the first '/' */
  r = strend(g);
  while (g < r) {
    while (*s ≠ '/' ∧ s < r) s++;
    *s = 0;      /* terminate g by rewriting '/' to 0 */
    mark_cs(mp, g);
    g = s + 1;
  }
SET_SUBR_MAX:
  if (mp→ps→subr_tab ≠ Λ)
    for (mp→ps→subr_max = −1, ptr = mp→ps→subr_tab; ptr − mp→ps→subr_tab < mp→ps→subr_size;
        ptr++)
      if (ptr→is_used ∧ ptr − mp→ps→subr_tab > mp→ps→subr_max)
        mp→ps→subr_max = (int)(ptr − mp→ps→subr_tab);
}
static void t1_do_subset_charstrings(MP mp, font_number tex_font)
{
  cs_entry *ptr;
  mp→ps→cs_size_pos = (int)(strstr(mp→ps→t1_line_array,
      charstringname) + strlen(charstringname) − mp→ps→t1_line_array + 1);
    /* cs_size_pos points to the number indicating dict size after "/CharStrings" */
  mp→ps→cs_size = (int) t1_scan_num(mp, mp→ps→t1_line_array + mp→ps→cs_size_pos, 0);
  mp→ps→cs_ptr = mp→ps→cs_tab = mp_xmalloc(mp, (size_t) mp→ps→cs_size, sizeof(cs_entry));
  for (ptr = mp→ps→cs_tab; ptr − mp→ps→cs_tab < mp→ps→cs_size; ptr++) init_cs_entry(ptr);
  mp→ps→cs_notdef = Λ;
  mp→ps→cs_dict_start = mp_xstrdup(mp, mp→ps→t1_line_array);
  t1_getline(mp);
  while (mp→ps→t1_cslen) {
    store_cs(mp);
    t1_getline(mp);
  }
  mp→ps→cs_dict_end = mp_xstrdup(mp, mp→ps→t1_line_array);
  t1_mark_glyphs(mp, tex_font);
}
static void t1_subset_charstrings(MP mp, font_number tex_font)
{
  cs_entry *ptr;
```

```
  t1_do_subset_charstrings(mp, tex_font);
  if (mp→ps→subr_tab ≠ Λ) {
    if (mp→ps→cs_token_pair ≡ Λ) mp_fatal_error(mp,
        "This␣Type␣1␣font␣uses␣mismatched␣subroutine␣begin/end␣token␣pairs.");
    t1_subr_flush(mp);
  }
  for (mp→ps→cs_count = 0, ptr = mp→ps→cs_tab; ptr < mp→ps→cs_ptr; ptr++)
    if (ptr→is_used) mp→ps→cs_count++;
  t1_cs_flush(mp);
}
static void t1_subset_end(MP mp)
{
  if (mp→ps→t1_synthetic) {      /* copy to "dup /FontName get exch definefont pop" */
    while (¬strstr(mp→ps→t1_line_array, "definefont")) {
      t1_getline(mp);
      t1_putline(mp);
    }
    while (¬t1_end_eexec()) t1_getline(mp);      /* ignore the rest */
    t1_putline(mp);      /* write "mark currentfile closefile" */
  }
  else
    while (¬t1_end_eexec()) {      /* copy to "mark currentfile closefile" */
      t1_getline(mp);
      t1_putline(mp);
    }
  t1_stop_eexec(mp);
  if (fixedcontent) {      /* copy 512 zeros (not needed for PDF) */
    while (¬t1_cleartomark()) {
      t1_getline(mp);
      t1_putline(mp);
    }
    if (¬mp→ps→t1_synthetic)      /* don't check "restoreif" for synthetic fonts */
      t1_check_end(mp);      /* write "restoreif" if found */
  }
}
static int t1_updatefm(MP mp, font_number f, fm_entry * fm)
{
  char *s, *p;
  mp→ps→read_encoding_only = true;
  if (¬t1_open_fontfile(mp, fm, Λ)) {
    return 0;
  }
  t1_scan_only(mp, f, fm);
  s = mp_xstrdup(mp, mp→ps→fontname_buf);
  p = s;
  while (*p ≠ '␣' ∧ *p ≠ 0) p++;
  *p = 0;
  mp_xfree(fm→ps_name);
  fm→ps_name = s;
  t1_close_font_file(mp, "");
  return 1;
```

```
}
static void writet1(MP mp, font_number tex_font, fm_entry * fm_cur)
{
  unsigned save_selector = mp->selector;
  mp_normalize_selector(mp);
  mp->ps->read_encoding_only = false;
  if (¬is_included(fm_cur)) {      /* scan parameters from font file */
    if (¬t1_open_fontfile(mp, fm_cur, "{")) return;
    t1_scan_only(mp, tex_font, fm_cur);
    t1_close_font_file(mp, "}");
    return;
  }
  if (¬is_subsetted(fm_cur)) {      /* include entire font */
    if (¬t1_open_fontfile(mp, fm_cur, "<<")) return;
    t1_include(mp, tex_font, fm_cur);
    t1_close_font_file(mp, ">>");
    return;
  }      /* partial downloading */
  if (¬t1_open_fontfile(mp, fm_cur, "<")) return;
  t1_subset_ascii_part(mp, tex_font, fm_cur);
  t1_start_eexec(mp, fm_cur);
  cc_init( );
  cs_init(mp);
  t1_read_subrs(mp, tex_font, fm_cur, false);
  t1_subset_charstrings(mp, tex_font);
  t1_subset_end(mp);
  t1_close_font_file(mp, ">");
  mp->selector = save_selector;
}
```

100. ⟨ Declarations 29 ⟩ +≡

```
static void t1_free(MP mp);
```

101. **static void** *t1_free*(MP *mp*)

```
{
  int k;
  mp_xfree(mp→ps→subr_array_start);
  mp_xfree(mp→ps→subr_array_end);
  mp_xfree(mp→ps→cs_dict_start);
  mp_xfree(mp→ps→cs_dict_end);
  cs_init(mp);
  mp_xfree(mp→ps→t1_line_array);
  mp_xfree(mp→ps→char_array);
  mp→ps→char_array = Λ;
  mp→ps→t1_line_array = mp→ps→t1_line_ptr = Λ;
  mp→ps→t1_line_limit = 0;
  mp_xfree(mp→ps→t1_buf_array);
  mp→ps→t1_buf_array = mp→ps→t1_buf_ptr = Λ;
  mp→ps→t1_buf_limit = 0;
  for (k = 0; k ≤ 255; k++) {
    if (mp→ps→t1_builtin_glyph_names[k] ≠ notdef) mp_xfree(mp→ps→t1_builtin_glyph_names[k]);
    mp→ps→t1_builtin_glyph_names[k] = notdef;
  }
}
```

102. Embedding Charstrings.

The SVG backend uses some routines that use an ascii representation of a type1 font. First, here is the type associated with it:

```
⟨ Types 18 ⟩ +≡
  typedef struct mp_ps_font {
    int font_num;      /* just to put something in */
    char **t1_glyph_names;
    cs_entry *cs_tab;
    cs_entry *cs_ptr;
    cs_entry *subr_tab;
    int subr_size;
    int t1_lenIV;
    int slant;
    int extend;
    ⟨ Variables for the charstring parser 107 ⟩
  } mp_ps_font;
```

103. The parser creates a structure and fills it.

```
mp_ps_font *mp_ps_font_parse(MP mp, int tex_font)
{
  mp_ps_font *f;
  fm_entry * fm_cur;
  char msg[128];
  (void) mp_has_fm_entry(mp, (font_number)tex_font, &fm_cur);
  if (fm_cur ≡ Λ) {
    mp_snprintf(msg, 128, "fontmap␣entry␣for␣‘%s’␣not␣found", mp→font_name[tex_font]);
    mp_warn(mp, msg);
    return Λ;
  }
  if (is_truetype(fm_cur) ∨ (fm_cur→ps_name ≡ Λ ∧ fm_cur→ff_name ≡ Λ) ∨ (¬is_included(fm_cur))) {
    mp_snprintf(msg, 128, "font␣‘%s’␣cannot␣be␣embedded", mp→font_name[tex_font]);
    mp_warn(mp, msg);
    return Λ;
  }
  if (¬t1_open_fontfile(mp, fm_cur, "<")) {      /* message handled there */
    return Λ;
  }
  f = mp_xmalloc(mp, 1, sizeof(struct mp_ps_font));
  f→font_num = tex_font;
  f→t1_glyph_names = Λ;
  f→cs_tab = Λ;
  f→cs_ptr = Λ;
  f→subr_tab = Λ;
  f→orig_x = f→orig_y = 0.0;
  f→slant = (int) fm_cur→slant;
  f→extend = (int) fm_cur→extend;
  t1_getline(mp);
  while (¬t1_prefix("/Encoding")) {
    t1_scan_param(mp, (font_number)tex_font, fm_cur);
    t1_getline(mp);
  }
  t1_builtin_enc(mp);
  if (is_reencoded(fm_cur)) {
    mp_read_enc(mp, fm_cur→encoding);
    ;
    f→t1_glyph_names = external_enc( );
  }
  else {
    f→t1_glyph_names = mp→ps→t1_builtin_glyph_names;
  }
  do {
    t1_getline(mp);
    t1_scan_param(mp, (font_number)tex_font, fm_cur);
  } while (mp→ps→t1_in_eexec ≡ 0);      /* t1_start_eexec(mp, fm_cur); */
  cc_init( );
  cs_init(mp);      /* the boolean is needed to make sure that t1_read_subrs doesn't output stuff */
  t1_read_subrs(mp, (font_number)tex_font, fm_cur, true);
  mp→ps→t1_synthetic = true;
```

```
  t1_do_subset_charstrings(mp, (font_number)tex_font);
  f→cs_tab = mp→ps→cs_tab;
  mp→ps→cs_tab = Λ;
  f→cs_ptr = mp→ps→cs_ptr;
  mp→ps→cs_ptr = Λ;
  f→subr_tab = mp→ps→subr_tab;
  mp→ps→subr_tab = Λ;
  f→subr_size = mp→ps→subr_size;
  mp→ps→subr_size = mp→ps→subr_size_pos = 0;
  f→t1_lenIV = mp→ps→t1_lenIV;
  t1_close_font_file(mp, ">");
  return f;
}
```

104. ⟨ Exported function headers 5 ⟩ +≡

```
mp_ps_font *mp_ps_font_parse(MP mp, int tex_font);
```

105. Freeing the structure

```
void mp_ps_font_free(MP mp, mp_ps_font *f)
{
  cs_entry *ptr;
  for (ptr = f→cs_tab; ptr < f→cs_ptr; ptr++) {
    if (ptr→glyph_name ≠ notdef) mp_xfree(ptr→glyph_name);
    mp_xfree(ptr→data);
  }
  mp_xfree(f→cs_tab);
  f→cs_tab = Λ;
  for (ptr = f→subr_tab; ptr − f→subr_tab < f→subr_size; ptr++) {
    if (ptr→glyph_name ≠ notdef) mp_xfree(ptr→glyph_name);
    mp_xfree(ptr→data);
  }
  mp_xfree(f→subr_tab);
  f→subr_tab = Λ;
  t1_free(mp);
  mp_xfree(f);
}
```

106. ⟨ Exported function headers 5 ⟩ +≡

```
void mp_ps_font_free(MP mp, mp_ps_font *f);
```

107. Parsing Charstrings.

⟨ Variables for the charstring parser 107 ⟩ ≡

```
double flex_hint_data[14];      /* store temp. coordinates of flex hints */
unsigned int flex_hint_index;   /* index for flex_hint_data */
boolean ignore_flex_hint;       /* skip hint for flex */
double cur_x, cur_y;            /* current point */
double orig_x, orig_y;          /* origin (for seac) */
mp_edge_object *h;              /* the whole picture */
mp_graphic_object *p;           /* the current subpath in the picture */
mp_gr_knot pp;                  /* the last known knot in the subpath */
```

This code is used in section 102.

108.
```
mp_edge_object * mp_ps_do_font_charstring(MP mp, mp_ps_font *f, char *nam)
{
  mp_edge_object * h = Λ;
  f→h = Λ;
  f→p = Λ;
  f→pp = Λ;
  f→ignore_flex_hint = 0;
  f→flex_hint_index = 0;      /* just in case */
  f→cur_x = f→cur_y = 0.0;
  f→orig_x = f→orig_y = 0.0;
  if (nam ≡ Λ) {
    mp_warn(mp, "nonexistant␣glyph␣requested");
    return h;
  }
  if (cs_parse(mp, f, nam, 0)) {
    h = f→h;
  }
  else {
    char err[256];
    mp_snprintf(err, 255, "Glyph␣interpreter␣failed␣(missing␣glyph␣'%s'?)", nam);
    mp_warn(mp, err);
    if (f→h ≠ Λ) {
      finish_subpath(mp, f);
      mp_gr_toss_objects(f→h);
    }
  }
  f→h = Λ;
  f→p = Λ;
  f→pp = Λ;
  return h;
}
mp_edge_object * mp_ps_font_charstring(MP mp, mp_ps_font *f, int c)
{
  char *s = Λ;
  if (f ≠ Λ ∧ f→t1_glyph_names ≠ Λ ∧ c ≥ 0 ∧ c < 256) s = f→t1_glyph_names[c];
  return mp_ps_do_font_charstring(mp, f, s);
}
```

109.
```
⟨ Exported function headers 5 ⟩ +≡
mp_edge_object * mp_ps_font_charstring(MP mp, mp_ps_font *f, int c);
mp_edge_object * mp_ps_do_font_charstring(MP mp, mp_ps_font *f, char *n);
```

110.
```
⟨ Declarations 29 ⟩ +≡
boolean cs_parse(MP mp, mp_ps_font *f, const char *cs_name, int subr);
```

111.

```
static void start_subpath(MP mp, mp_ps_font *f, double dx, double dy){ assert(f->pp ≡ Λ);
    assert(f->p ≡ Λ);
    f->pp = mp_xmalloc(mp, 1, sizeof(struct mp_gr_knot_data));
    f->pp->data.types.left_type = mp_explicit;
    f->pp->data.types.right_type = mp_explicit;
    f->pp->x_coord = (f->cur_x + dx);
    f->pp->y_coord = (f->cur_y + dy);
    f->pp->left_x = f->pp->right_x = f->pp->x_coord;
    f->pp->left_y = f->pp->right_y = f->pp->y_coord;
    f->pp->next = Λ;
    f->cur_x += dx;
    f->cur_y += dy;
    f->p = mp_new_graphic_object(mp, mp_fill_code); gr_path_p ( ( mp_fill_object * ) f->p ) = f->pp; } static
        void add_line_segment(MP mp, mp_ps_font *f, double dx, double dy){ mp_gr_knot n;
      assert(f->pp ≠ Λ);
      n = mp_xmalloc(mp, 1, sizeof(struct mp_gr_knot_data));
      n->data.types.left_type = mp_explicit;
      n->data.types.right_type = mp_explicit; n->next = gr_path_p ( ( mp_fill_object * ) f->p ) ;
        /* loop */
      n->x_coord = (f->cur_x + dx);
      n->y_coord = (f->cur_y + dy);
      n->right_x = n->x_coord;
      n->right_y = n->y_coord;
      n->left_x = n->x_coord;
      n->left_y = n->y_coord;
      f->pp->next = n;
      f->pp = n;
      f->cur_x += dx;
      f->cur_y += dy; } static void add_curve_segment(MP mp, mp_ps_font *f, double dx1, double
          dy1, double dx2, double dy2, double dx3, double dy3){ mp_gr_knot n;
        n = mp_xmalloc(mp, 1, sizeof(struct mp_gr_knot_data));
        n->data.types.left_type = mp_explicit;
        n->data.types.right_type = mp_explicit; n->next = gr_path_p ( ( mp_fill_object * ) f->p ) ;
          /* loop */
        n->x_coord = (f->cur_x + dx1 + dx2 + dx3);
        n->y_coord = (f->cur_y + dy1 + dy2 + dy3);
        n->right_x = n->x_coord;
        n->right_y = n->y_coord;
        n->left_x = (f->cur_x + dx1 + dx2);
        n->left_y = (f->cur_y + dy1 + dy2);
        f->pp->right_x = (f->cur_x + dx1);
        f->pp->right_y = (f->cur_y + dy1);
        f->pp->next = n;
        f->pp = n;
        f->cur_x += dx1 + dx2 + dx3;
        f->cur_y += dy1 + dy2 + dy3; } static void finish_subpath(MP mp, mp_ps_font *f){
          if (f->p ≠ Λ) {
            if (f->h->body ≡ Λ) {
              f->h->body = f->p;
            }
            else {
```

```
      mp_graphic_object * q = f->h->body;
      while (gr_link(q) ≠ Λ) q = gr_link(q);
      q->next = f->p;
    }
  }
  if (f->p ≠ Λ) { mp_gr_knot r, rr;
  assert(f->pp ≠ Λ); r = gr_path_p ( ( mp_fill_object * ) f->p ) ;
  rr = r;
  if (r) {
    if (r ≡ f->pp) {
      r->next = r;
    }
    else if (r->x_coord ≡ f->pp->x_coord ∧ r->y_coord ≡ f->pp->y_coord) {
      while (rr->next ≠ f->pp) rr = rr->next;
      rr->next = r;
      r->left_x = f->pp->left_x;
      r->left_y = f->pp->left_y;
      mp_xfree(f->pp);
    }
  }
  } f->p = Λ;
  f->pp = Λ; }
```

112.

```
#define cs_no_debug(A) cs_do_debug(mp, f, A, #A)
#define cs_debug(A)
```

⟨ Declarations 29 ⟩ +≡

```
  void cs_do_debug(MP mp, mp_ps_font *f, int i, char *s);
  static void finish_subpath(MP mp, mp_ps_font *f);
  static void add_curve_segment(MP mp, mp_ps_font *f, double dx1, double dy1, double dx2, double
      dy2, double dx3, double dy3);
  static void add_line_segment(MP mp, mp_ps_font *f, double dx, double dy);
  static void start_subpath(MP mp, mp_ps_font *f, double dx, double dy);
```

113. **void** *cs_do_debug*(MP *mp*, **mp_ps_font** ∗*f*, **int** *i*, **char** ∗*s*)

```
{
  int n = cc_tab[i].nargs;
  (void) mp;      /* for -Wall */
  (void) f;       /* for -Wall */
  while (n > 0) {
    fprintf(stdout, "%d ", (int) cc_get((−n)));
    n−−;
  }
  fprintf(stdout, "%s\n", s);
}
boolean cs_parse(MP mp, mp_ps_font *f, const char *cs_name, int subr)
{
  byte *data;
  int i, b, cs_len;
  integer a, a1, a2;
  unsigned short cr;
  static integer lastargOtherSubr3 = 3;
  cs_entry *ptr;
  cc_entry *cc;
  if (cs_name ≡ Λ) {
    ptr = f→subr_tab + subr;
  }
  else {
    i = 0;
    for (ptr = f→cs_tab; ptr < f→cs_ptr; ptr++, i++) {
      if (strcmp(ptr→glyph_name, cs_name) ≡ 0) break;
    }
    ptr = f→cs_tab + i;      /* this is the right charstring */
  }
  if (ptr ≡ f→cs_ptr) return false;
  data = ptr→data + 4;
  cr = 4330;
  cs_len = (int) ptr→cslen;
  for (i = 0; i < f→t1_lenIV; i++, cs_len−−) (void) cs_getchar(mp);
  while (cs_len > 0) {
    −−cs_len;
    b = cs_getchar(mp);
    if (b ≥ 32) {
      if (b ≤ 246) a = b − 139;
      else if (b ≤ 250) {
        −−cs_len;
        a = (int)((unsigned)(b − 247) ≪ 8) + 108 + cs_getchar(mp);
      }
      else if (b ≤ 254) {
        −−cs_len;
        a = −(int)((unsigned)(b − 251) ≪ 8) − 108 − cs_getchar(mp);
      }
      else {
        cs_len −= 4;
        a = (cs_getchar(mp) & #ff) ≪ 24;
```

```
    a |= (cs_getchar(mp) & #ff) ≪ 16;
    a |= (cs_getchar(mp) & #ff) ≪ 8;
    a |= (cs_getchar(mp) & #ff) ≪ 0;
    if (sizeof (integer) > 4 ∧ (a & #80000000)) a |= ∼#7FFFFFFF;
  }
  cc_push(a);
}
else {
  if (b ≡ CS_ESCAPE) {
    b = cs_getchar(mp) + CS_1BYTE_MAX;
    cs_len −−;
  }
  if (b ≥ CS_MAX) {
    cs_warn(mp, cs_name, subr, "command␣value␣out␣of␣range:␣%i", (int) b);
    goto cs_error;
  }
  cc = cc_tab + b;
  if (¬cc→valid) {
    cs_warn(mp, cs_name, subr, "command␣not␣valid:␣%i", (int) b);
    goto cs_error;
  }
  if (cc→bottom) {
    if (stack_ptr − cc_stack < cc→nargs)
      cs_warn(mp, cs_name, subr, "less␣arguments␣on␣stack␣(%i)␣than␣required␣(%i)",
          (int)(stack_ptr − cc_stack), (int) cc→nargs);
    else if (stack_ptr − cc_stack > cc→nargs)
      cs_warn(mp, cs_name, subr, "more␣arguments␣on␣stack␣(%i)␣than␣required␣(%i)",
          (int)(stack_ptr − cc_stack), (int) cc→nargs);
  }
  switch (cc − cc_tab) {
  case CS_CLOSEPATH: /* −CLOSEPATH- */
    cs_debug(CS_CLOSEPATH);
    finish_subpath(mp, f);
    cc_clear( );
    break;
  case CS_HLINETO: /* −dxHLINETO- */
    cs_debug(CS_HLINETO);
    add_line_segment(mp, f, cc_get(−1), 0);
    cc_clear( );
    break;
  case CS_HVCURVETO: /* −dx1 dx2 dy2 dy3HVCURVETO- */
    cs_debug(CS_HVCURVETO);
    add_curve_segment(mp, f, cc_get(−4), 0, cc_get(−3), cc_get(−2), 0, cc_get(−1));
    cc_clear( );
    break;
  case CS_RLINETO: /* −dx dyRLINETO- */
    cs_debug(CS_RLINETO);
    add_line_segment(mp, f, cc_get(−2), cc_get(−1));
    cc_clear( );
    break;
  case CS_RRCURVETO: /* −dx1 dy1 dx2 dy2 dx3 dy3RRCURVETO- */
    cs_debug(CS_RRCURVETO);
```

```
    add_curve_segment(mp, f, cc_get(−6), cc_get(−5), cc_get(−4), cc_get(−3), cc_get(−2), cc_get(−1));
    cc_clear( );
    break;
  case CS_VHCURVETO:    /* −dy1 dx2 dy2 dx3 VHCURVETO- */
    cs_debug(CS_VHCURVETO);
    add_curve_segment(mp, f, 0, cc_get(−4), cc_get(−3), cc_get(−2), cc_get(−1), 0);
    cc_clear( );
    break;
  case CS_VLINETO:    /* −dy VLINETO- */
    cs_debug(CS_VLINETO);
    add_line_segment(mp, f, 0, cc_get(−1));
    cc_clear( );
    break;
  case CS_HMOVETO:    /* −dx HMOVETO- */
    cs_debug(CS_HMOVETO);    /* treating in-line moves as 'line segments' work better than
        attempting to split the path up in two separate sections, at least for now. */
    if (f→pp ≡ Λ) {    /* this is the first */
      start_subpath(mp, f, cc_get(−1), 0);
    }
    else {
      add_line_segment(mp, f, cc_get(−1), 0);
    }
    cc_clear( );
    break;
  case CS_RMOVETO:    /* −dx dy RMOVETO- */
    cs_debug(CS_RMOVETO);
    if (f→ignore_flex_hint ≡ 1) {
      f→flex_hint_data[f→flex_hint_index++] = cc_get(−2);
      f→flex_hint_data[f→flex_hint_index++] = cc_get(−1);
    }
    else {
      if (f→pp ≡ Λ) {    /* this is the first */
        start_subpath(mp, f, cc_get(−2), cc_get(−1));
      }
      else {
        add_line_segment(mp, f, cc_get(−2), cc_get(−1));
      }
    }
    cc_clear( );
    break;
  case CS_VMOVETO:    /* −dy VMOVETO- */
    cs_debug(CS_VMOVETO);
    if (f→pp ≡ Λ) {    /* this is the first */
      start_subpath(mp, f, 0, cc_get(−1));
    }
    else {
      add_line_segment(mp, f, 0, cc_get(−1));
    }
    cc_clear( );
    break;    /* hinting commands */
  case CS_DOTSECTION:    /* −DOTSECTION- */
    cs_debug(CS_DOTSECTION);
```

```
    cc_clear( );
    break;
  case CS_HSTEM:     /* -y dy HSTEM- */
    cs_debug(CS_HSTEM);
    cc_clear( );
    break;
  case CS_HSTEM3:     /* -y0 dy0 y1 dy1 y2 dy2 HSTEM3- */
    cs_debug(CS_HSTEM3);
    cc_clear( );
    break;
  case CS_VSTEM:     /* -x dx VSTEM- */
    cs_debug(CS_VSTEM);
    cc_clear( );
    break;
  case CS_VSTEM3:     /* -x0 dx0 x1 dx1 x2 dx2 VSTEM3- */
    cs_debug(CS_VSTEM3);
    cc_clear( );
    break;     /* start and close commands */
  case CS_SEAC:     /* -asb adx ady bchar achar SEAC- */
    cs_debug(CS_SEAC);
    {
      double adx, ady, asb;
      asb = cc_get(0);
      adx = cc_get(1);
      ady = cc_get(2);
      a1 = (integer)cc_get(3);
      a2 = (integer)cc_get(4);
      cc_clear( );
      (void) cs_parse(mp, f, standard_glyph_names[a1], 0);     /* base */
      f→orig_x += (adx - asb);
      f→orig_y += ady;
      (void) cs_parse(mp, f, standard_glyph_names[a2], 0);
    }
    break;
  case CS_ENDCHAR:     /* -ENDCHAR- */
    cs_debug(CS_ENDCHAR);
    cc_clear( );
    return true;
    break;
  case CS_HSBW:     /* -sbx wx HSBW- */
    cs_debug(CS_HSBW);
    if (¬f→h) {
      f→h = mp_xmalloc(mp, 1, sizeof (mp_edge_object));
      f→h→body = Λ;
      f→h→next = Λ;
      f→h→parent = mp;
      f→h→filename = Λ;
      f→h→minx = f→h→miny = f→h→maxx = f→h→maxy = 0.0;
    }
    f→cur_x = cc_get(-2) + f→orig_x;
    f→cur_y = 0.0 + f→orig_y;
    f→orig_x = f→cur_x;
```

```
    f→orig_y = f→cur_y;
    cc_clear( );
    break;
  case CS_SBW:      /* -sbx sby wx wy SBW- */
    cs_debug(CS_SBW);
    if (¬f→h) {
      f→h = mp_xmalloc(mp, 1, sizeof (mp_edge_object));
      f→h→body = Λ;
      f→h→next = Λ;
      f→h→parent = mp;
      f→h→filename = Λ;
      f→h→minx = f→h→miny = f→h→maxx = f→h→maxy = 0.0;
    }
    f→cur_x = cc_get(−4) + f→orig_x;
    f→cur_y = cc_get(−3) + f→orig_y;
    f→orig_x = f→cur_x;
    f→orig_y = f→cur_y;
    cc_clear( );
    break;      /* arithmetic */
  case CS_DIV:      /* num1 num2 DIV quotient */
    cs_debug(CS_DIV);
    {
      double num, den, res;
      num = cc_get(−2);
      den = cc_get(−1);
      res = num/den;
      cc_pop(2);
      cc_push(res);
      break;
    }      /* subrs */
  case CS_CALLSUBR:      /* subr CALLSUBR - */
    cs_debug(CS_CALLSUBR);
    a1 = (integer)cc_get(−1);
    if (a1 ≡ 1) f→ignore_flex_hint = 1;
    if (a1 ≡ 0) {      /*double first_x, first_y, first_r_x, first_r_y;*/
      /*double join_x, join_y, join_l_x, join_l_y, join_r_x, join_r_y;*/
      /*double last_x, last_y, last_l_x, last_l_y;*/      /*// a := glyph "q" of "cmti12";*/
      /*first_x = 206.0; first_y = −194.0;*/      /*double ref_x, ref_y;*/
      /*ref_x = first_x + f→flex_hint_data[0];*/      /*ref_y = first_y + f→flex_hint_data[1];*/
        /*printf ("1:(%f,␣%f)␣2:(%f,%f)␣3:(%f,%f)␣4:(%f,%f)␣5:(%f,%f)␣6:(%f,%f\
        )␣7:(%f,%f)\n",*/      /*f→flex_hint_data[0], f→flex_hint_data[1] ,*/
      /*f→flex_hint_data[2], f→flex_hint_data[3] ,*/      /*f→flex_hint_data[4], f→flex_hint_data[5]
        ,*/      /*f→flex_hint_data[6], f→flex_hint_data[7] ,*/
      /*f→flex_hint_data[8], f→flex_hint_data[9] ,*/      /*f→flex_hint_data[10], f→flex_hint_data[11]
        ,*/      /*f→flex_hint_data[12], f→flex_hint_data[13] ) ;*/
      /*printf("Reference=(%f,%f)\n", ref_x, ref_y);*/
      /*first_r_x = ref_x + f→flex_hint_data[2]; first_r_y = ref_y + f→flex_hint_data[3];*/
      /*join_l_x = first_r_x + f→flex_hint_data[4]; join_l_y = first_r_y + f→flex_hint_data[5];*/
      /*join_x = join_l_x + f→flex_hint_data[6]; join_y = join_l_y + f→flex_hint_data[7];*/
      /*join_r_x = join_x + f→flex_hint_data[8]; join_r_y = join_y + f→flex_hint_data[9];*/
      /*last_l_x = join_r_x + f→flex_hint_data[10]; last_l_y = join_r_y + f→flex_hint_data[11];*/
       /*last_x = last_l_x + f→flex_hint_data[12]; last_y = last_l_y + f→flex_hint_data[13];*/
```

```
                /* printf ("(%f,%f) . . (%f,%f) and (%f,%f) . . (%f,%f) . . (%f,%f) and (%\
                f,%f) . . (%f,%f)\n", */
              /* first_x, first_y, first_r_x, first_r_y, join_l_x, join_l_y, join_x, join_y, join_r_x, join_r_y , */
              /* last_l_x, last_l_y, last_x, last_y ) ; */
            f→ignore_flex_hint = 0;
            f→flex_hint_index = 0;
            add_curve_segment(mp, f, f→flex_hint_data[0] + f→flex_hint_data[2],
                f→flex_hint_data[1] + f→flex_hint_data[3], f→flex_hint_data[4], f→flex_hint_data[5],
                f→flex_hint_data[6], f→flex_hint_data[7]);
            add_curve_segment(mp, f, f→flex_hint_data[8], f→flex_hint_data[9], f→flex_hint_data[10],
                f→flex_hint_data[11], f→flex_hint_data[12], f→flex_hint_data[13]);
          }
          cc_pop(1);
          (void) cs_parse(mp, f, Λ, a1);
          break;
        case CS_RETURN: /* - RETURN - */
          cs_debug(CS_RETURN);
          return true;
          break;
        case CS_CALLOTHERSUBR: /* arg1 ... argn n othersubr CALLOTHERSUBR - */
          cs_debug(CS_CALLOTHERSUBR);
          a1 = (integer)cc_get(−1);
          if (a1 ≡ 3) lastargOtherSubr3 = (integer)cc_get(−3);
          a1 = (integer)cc_get(−2) + 2;
          cc_pop(a1);
          break;
        case CS_POP: /* - POP number */
          cc_push(lastargOtherSubr3);
          break;
        case CS_SETCURRENTPOINT: /* −xy SETCURRENTPOINT- */
          cs_debug(CS_SETCURRENTPOINT);
            /* totally ignoring setcurrentpoint actually works better for most fonts ? */
          cc_clear( );
          break;
        default:
          if (cc→clear) cc_clear( );
        }
      }
    }
    return true;
  cs_error: /* an error occured during parsing */
    cc_clear( );
    ptr→valid = false;
    ptr→is_used = false;
    return false;
  }
```

114. Embedding fonts.

115. The *tfm_num* is officially of type *font_number*, but that type does not exist yet at this point in the output order.

```
⟨ Types 18 ⟩ +≡
  typedef struct {
    char *tfm_name;      /* TFM file name */
    char *ps_name;       /* PostScript name */
    integer flags;       /* font flags */
    char *ff_name;       /* font file name */
    char *subset_tag;    /* pseudoUniqueTag for subsetted font */
    enc_entry *encoding;     /* pointer to corresponding encoding */
    unsigned int tfm_num;    /* number of the TFM refering this entry */
    unsigned short type;     /* font type (T1/TTF/...) */
    short slant;     /* SlantFont */
    short extend;    /* ExtendFont */
    integer ff_objnum;       /* FontFile object number */
    integer fn_objnum;       /* FontName/BaseName object number */
    integer fd_objnum;       /* FontDescriptor object number */
    char *charset;       /* string containing used glyphs */
    boolean all_glyphs;      /* embed all glyphs? */
    unsigned short links;    /* link flags from tfm_tree and ps_tree */
    short tfm_avail;     /* flags whether a tfm is available */
    short pid;       /* Pid for truetype fonts */
    short eid;       /* Eid for truetype fonts */
  } fm_entry;
```

116.

```
⟨ Globals 7 ⟩ +≡
#define FONTNAME_BUF_SIZE  128
  boolean fontfile_found;
  boolean is_otf_font;
  char fontname_buf[FONTNAME_BUF_SIZE];
```

117.

```
#define F_INCLUDED ^01
#define F_SUBSETTED ^02
#define F_TRUETYPE ^04
#define F_BASEFONT ^08
#define set_included(fm) ((fm)→type |= F_INCLUDED)
#define set_subsetted(fm) ((fm)→type |= F_SUBSETTED)
#define set_truetype(fm) ((fm)→type |= F_TRUETYPE)
#define set_basefont(fm) ((fm)→type |= F_BASEFONT)
#define is_included(fm) ((fm)→type & F_INCLUDED)
#define is_subsetted(fm) ((fm)→type & F_SUBSETTED)
#define is_truetype(fm) ((fm)→type & F_TRUETYPE)
#define is_basefont(fm) ((fm)→type & F_BASEFONT)
#define is_reencoded(fm) ((fm)→encoding ≠ Λ)
#define is_fontfile(fm) (fm_fontfile(fm) ≠ Λ)
#define is_t1fontfile(fm) (is_fontfile(fm) ∧ ¬is_truetype(fm))
#define fm_slant(fm) (fm)→slant
#define fm_extend(fm) (fm)→extend
#define fm_fontfile(fm) (fm)→ff_name
```

⟨ Declarations 29 ⟩ +≡

```
static boolean mp_font_is_reencoded(MP mp, font_number f);
static boolean mp_font_is_included(MP mp, font_number f);
static boolean mp_font_is_subsetted(MP mp, font_number f);
```

118.

```
boolean mp_font_is_reencoded(MP mp, font_number f)
{
  fm_entry *fm;
  if (mp_has_font_size(mp, f) ∧ mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ) ∧ is_reencoded(fm)) return true;
  }
  return false;
}
boolean mp_font_is_included(MP mp, font_number f)
{
  fm_entry *fm;
  if (mp_has_font_size(mp, f) ∧ mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ ∧ fm→ff_name ≠ Λ) ∧ is_included(fm)) return true;
  }
  return false;
}
boolean mp_font_is_subsetted(MP mp, font_number f)
{
  fm_entry *fm;
  if (mp_has_font_size(mp, f) ∧ mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ ∧ fm→ff_name ≠ Λ) ∧ is_included(fm) ∧ is_subsetted(fm))
      return true;
  }
  return false;
}
```

119. ⟨ Declarations 29 ⟩ +≡

```
static char *mp_fm_encoding_name(MP mp, font_number f);
static char *mp_fm_font_name(MP mp, font_number f);
static char *mp_fm_font_subset_name(MP mp, font_number f);
```

120.

```
char *mp_fm_encoding_name(MP mp, font_number f)
{
  enc_entry *e;
  fm_entry *fm;
  if (mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ)) {
      if (is_reencoded(fm)) {
        e = fm→encoding;
        if (e→enc_name ≠ Λ) return mp_xstrdup(mp, e→enc_name);
      }
      else {
        return Λ;
      }
    }
  }
  {
    char msg[256];
    mp_snprintf(msg, 256, "fontmap encoding problems for font %s", mp→font_name[f]);
    mp_error(mp, msg, Λ, true);
  }
  return Λ;
}
char *mp_fm_font_name(MP mp, font_number f)
{
  fm_entry *fm;
  if (mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ)) {
      if (mp_font_is_included(mp, f) ∧ ¬mp→font_ps_name_fixed[f]) {
          /* find the real fontname, and update ps_name and subset_tag if needed */
        if (t1_updatefm(mp, f, fm)) {
          mp→font_ps_name_fixed[f] = true;
        }
        else {
          char msg[256];
          mp_snprintf(msg, 256, "font loading problems for font %s", mp→font_name[f]);
          mp_error(mp, msg, Λ, true);
        }
      }
      return mp_xstrdup(mp, fm→ps_name);
    }
  }
  {
    char msg[256];
    mp_snprintf(msg, 256, "fontmap name problems for font %s", mp→font_name[f]);
    mp_error(mp, msg, Λ, true);
  }
  return Λ;
}
static char *mp_fm_font_subset_name(MP mp, font_number f)
```

```
{
  fm_entry *fm;
  if (mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ)) {
      if (is_subsetted(fm)) {
        char *s = mp_xmalloc(mp, strlen(fm→ps_name) + 8, 1);
        mp_snprintf(s, (int) strlen(fm→ps_name) + 8, "%s-%s", fm→subset_tag, fm→ps_name);
        return s;
      }
      else {
        return mp_xstrdup(mp, fm→ps_name);
      }
    }
  }
  {
    char msg[256];
    mp_snprintf(msg, 256, "fontmap name problems for font %s", mp→font_name[f]);
    mp_error(mp, msg, Λ, true);
  }
  return Λ;
}
```

121. ⟨ Declarations 29 ⟩ +≡

```
static integer mp_fm_font_slant(MP mp, font_number f);
static integer mp_fm_font_extend(MP mp, font_number f);
```

122.

```
static integer mp_fm_font_slant(MP mp, font_number f)
{
  fm_entry *fm;
  if (mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ)) {
      return fm→slant;
    }
  }
  return 0;
}
static integer mp_fm_font_extend(MP mp, font_number f)
{
  fm_entry *fm;
  if (mp_has_fm_entry(mp, f, &fm)) {
    if (fm ≠ Λ ∧ (fm→ps_name ≠ Λ)) {
      return fm→extend;
    }
  }
  return 0;
}
```

123. ⟨ Declarations 29 ⟩ +≡

```
static boolean mp_do_ps_font(MP mp, font_number f);
```

124. **static** *boolean mp_do_ps_font* (MP *mp*, *font_number f*)

```
{
  fm_entry *fm_cur;
  (void) mp_has_fm_entry(mp, f, &fm_cur);      /* for side effects */
  if (fm_cur ≡ Λ) return true;
  if (is_truetype(fm_cur) ∨ (fm_cur→ps_name ≡ Λ ∧ fm_cur→ff_name ≡ Λ)) {
    return false;
  }
  if (is_included(fm_cur)) {
    mp_ps_print_nl(mp, "%%BeginResource: font ");
    if (is_subsetted(fm_cur)) {
      mp_ps_print(mp, fm_cur→subset_tag);
      mp_ps_print_char(mp, '-');
    }
    mp_ps_print(mp, fm_cur→ps_name);
    mp_ps_print_ln(mp);
    writet1(mp, f, fm_cur);
    mp_ps_print_nl(mp, "%%EndResource");
    mp_ps_print_ln(mp);
  }
  return true;
}
```

125. Included subset fonts do not need and encoding vector, make sure we skip that case.

⟨ Declarations 29 ⟩ +≡

```
static void mp_list_used_resources(MP mp, int prologues, int procset);
```

126. **static void** *mp_list_used_resources*(MP *mp*, **int** *prologues*, **int** *procset*)

```
{
  font_number f;      /* fonts used in a text node or as loop counters */
  int ff;      /* a loop counter */
  int ldf;      /* the last DocumentFont listed (otherwise null_font) */
  boolean firstitem;
  if (procset > 0) mp_ps_print_nl(mp, "%%DocumentResources: procset mpost");
  else mp_ps_print_nl(mp, "%%DocumentResources: procset mpost-minimal");
  ldf = null_font;
  firstitem = true;
  for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
    if ((mp_has_font_size(mp, f)) ∧ (mp_font_is_reencoded(mp, f))) {
      for (ff = ldf; ff ≥ null_font; ff--) {
        if (mp_has_font_size(mp, (font_number)ff))
          if (mp_xstrcmp(mp→font_enc_name[f], mp→font_enc_name[ff]) ≡ 0) goto FOUND;
      }
      if (mp_font_is_subsetted(mp, f)) goto FOUND;
      if ((size_t) mp→ps→ps_offset + 1 + strlen(mp→font_enc_name[f]) > (size_t) mp→max_print_line)
        mp_ps_print_nl(mp, "%%+ encoding");
      if (firstitem) {
        firstitem = false;
        mp_ps_print_nl(mp, "%%+ encoding");
      }
      mp_ps_print_char(mp, ' ');
      mp_ps_dsc_print(mp, "encoding", mp→font_enc_name[f]);
      ldf = (int) f;
    }
  FOUND: ;
  }
  ldf = null_font;
  firstitem = true;
  for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
    if (mp_has_font_size(mp, f)) {
      for (ff = ldf; ff ≥ null_font; ff--) {
        if (mp_has_font_size(mp, (font_number)ff))
          if (mp_xstrcmp(mp→font_name[f], mp→font_name[ff]) ≡ 0) goto FOUND2;
      }
      if ((size_t) mp→ps→ps_offset + 1 + strlen(mp→font_ps_name[f]) > (size_t) mp→max_print_line)
        mp_ps_print_nl(mp, "%%+ font");
      if (firstitem) {
        firstitem = false;
        mp_ps_print_nl(mp, "%%+ font");
      }
      mp_ps_print_char(mp, ' ');
      if ((prologues ≡ 3) ∧ (mp_font_is_subsetted(mp, f))) {
        char *s = mp_fm_font_subset_name(mp, f);
        mp_ps_dsc_print(mp, "font", s);
        mp_xfree(s);
      }
      else {
        mp_ps_dsc_print(mp, "font", mp→font_ps_name[f]);
```

```
        }
        ldf = (int) f;
      }
    FOUND2: ;
    }
    mp_ps_print_ln(mp);
  }
```

127. ⟨ Declarations 29 ⟩ +≡

```
static void mp_list_supplied_resources(MP mp, int prologues, int procset);
```

128. **static void** *mp_list_supplied_resources*(MP *mp*, **int** *prologues*, **int** *procset*)

```
{
  font_number f;      /* fonts used in a text node or as loop counters */
  int ff;      /* a loop counter */
  int ldf;      /* the last DocumentFont listed (otherwise null_font) */
  boolean firstitem;
  if (procset > 0) mp_ps_print_nl(mp, "%%DocumentSuppliedResources: procset mpost");
  else mp_ps_print_nl(mp, "%%DocumentSuppliedResources: procset mpost-minimal");
  ldf = null_font;
  firstitem = true;
  for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
    if ((mp_has_font_size(mp, f)) ∧ (mp_font_is_reencoded(mp, f))) {
      for (ff = ldf; ff ≥ null_font; ff ++) {
        if (mp_has_font_size(mp, (font_number)ff))
          if (mp_xstrcmp(mp→font_enc_name[f], mp→font_enc_name[ff]) ≡ 0) goto FOUND;
      }
      if ((prologues ≡ 3) ∧ (mp_font_is_subsetted(mp, f))) goto FOUND;
      if ((size_t) mp→ps→ps_offset + 1 + strlen(mp→font_enc_name[f]) > (size_t) mp→max_print_line)
        mp_ps_print_nl(mp, "%%+ encoding");
      if (firstitem) {
        firstitem = false;
        mp_ps_print_nl(mp, "%%+ encoding");
      }
      mp_ps_print_char(mp, ' ');
      mp_ps_dsc_print(mp, "encoding", mp→font_enc_name[f]);
      ldf = (int) f;
    }
  FOUND: ;
  }
  ldf = null_font;
  firstitem = true;
  if (prologues ≡ 3) {
    for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
      if (mp_has_font_size(mp, f)) {
        for (ff = ldf; ff ≥ null_font; ff --) {
          if (mp_has_font_size(mp, (font_number)ff))
            if (mp_xstrcmp(mp→font_name[f], mp→font_name[ff]) ≡ 0) goto FOUND2;
        }
        if (¬mp_font_is_included(mp, f)) goto FOUND2;
        if ((size_t) mp→ps→ps_offset + 1 + strlen(mp→font_ps_name[f]) > (size_t) mp→max_print_line)
          mp_ps_print_nl(mp, "%%+ font");
        if (firstitem) {
          firstitem = false;
          mp_ps_print_nl(mp, "%%+ font");
        }
        mp_ps_print_char(mp, ' ');
        if (mp_font_is_subsetted(mp, f)) {
          char *s = mp_fm_font_subset_name(mp, f);
          mp_ps_dsc_print(mp, "font", s);
          mp_xfree(s);
        }
```

```
        else {
          mp_ps_dsc_print(mp, "font", mp→font_ps_name[f]);
        }
        ldf = (int) f;
      }
    FOUND2: ;
    }
    mp_ps_print_ln(mp);
  }
}
```

129. ⟨ Declarations 29 ⟩ +≡

```
static void mp_list_needed_resources(MP mp, int prologues);
```

130. **static void** *mp_list_needed_resources*(MP *mp*, **int** *prologues*)

```
{
  font_number f;      /* fonts used in a text node or as loop counters */
  int ff;      /* a loop counter */
  int ldf;      /* the last DocumentFont listed (otherwise null_font) */
  boolean firstitem;
  ldf = null_font;
  firstitem = true;
  for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
    if (mp_has_font_size(mp, f)) {
      for (ff = ldf; ff ≥ null_font; ff--) {
        if (mp_has_font_size(mp, (font_number)ff))
          if (mp_xstrcmp(mp→font_name[f], mp→font_name[ff]) ≡ 0) goto FOUND;
      }
      ;
      if ((prologues ≡ 3) ∧ (mp_font_is_included(mp, f))) goto FOUND;
      if ((size_t) mp→ps→ps_offset + 1 + strlen(mp→font_ps_name[f]) > (size_t) mp→max_print_line)
        mp_ps_print_nl(mp, "%%+ font");
      if (firstitem) {
        firstitem = false;
        mp_ps_print_nl(mp, "%%DocumentNeededResources: font");
      }
      mp_ps_print_char(mp, ' ');
      mp_ps_dsc_print(mp, "font", mp→font_ps_name[f]);
      ldf = (int) f;
    }
  FOUND: ;
  }
  if (¬firstitem) {
    mp_ps_print_ln(mp);
    ldf = null_font;      /* clang: never read: firstitem=true; */
    for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
      if (mp_has_font_size(mp, f)) {
        for (ff = ldf; ff ≥ null_font; ff--) {
          if (mp_has_font_size(mp, (font_number)ff))
            if (mp_xstrcmp(mp→font_name[f], mp→font_name[ff]) ≡ 0) goto FOUND2;
        }
        if ((prologues ≡ 3) ∧ (mp_font_is_included(mp, f))) goto FOUND2;
        mp_ps_print(mp, "%%IncludeResource: font ");
        mp_ps_print(mp, mp→font_ps_name[f]);
        mp_ps_print_ln(mp);
        ldf = (int) f;
      }
    FOUND2: ;
    }
  }
}
```

131. ⟨ Declarations 29 ⟩ +≡

```
static void mp_write_font_definition(MP mp, font_number f, int prologues);
```

132.

```
#define applied_reencoding(A)
        ((mp_font_is_reencoded(mp,(A)))∧((¬mp_font_is_subsetted(mp,(A)))∨(prologues≡2)))
  static void mp_write_font_definition(MP mp,font_number f,int prologues)
  {
    if ((applied_reencoding(f))∨(mp_fm_font_slant(mp,f)≠0)∨(mp_fm_font_extend(mp,
          f)≠0)∨(mp_xstrcmp(mp→font_name[f],"psyrgo")≡0)∨(mp_xstrcmp(mp→font_name[f],
          "zpzdr-reversed")≡0)) {
      if ((mp_font_is_subsetted(mp,f))∧(mp_font_is_included(mp,f))∧(prologues≡3)) {
        char *s=mp_fm_font_subset_name(mp,f);
        mp_ps_name_out(mp,s,true);
        mp_xfree(s);
      }
      else {
        mp_ps_name_out(mp,mp→font_ps_name[f],true);
      }
      mp_ps_print(mp,"␣fcp");
      mp_ps_print_ln(mp);
      if (applied_reencoding(f)) {
        mp_ps_print(mp,"/Encoding␣");
        mp_ps_print(mp,mp→font_enc_name[f]);
        mp_ps_print(mp,"␣def␣");
      }
      ;
      if (mp_fm_font_slant(mp,f)≠0) {
        mp_ps_print_int(mp,mp_fm_font_slant(mp,f));
        mp_ps_print(mp,"␣SlantFont␣");
      }
      ;
      if (mp_fm_font_extend(mp,f)≠0) {
        mp_ps_print_int(mp,mp_fm_font_extend(mp,f));
        mp_ps_print(mp,"␣ExtendFont␣");
      }
      ;
      if (mp_xstrcmp(mp→font_name[f],"psyrgo")≡0) {
        mp_ps_print(mp,"␣890␣ScaleFont␣");
        mp_ps_print(mp,"␣277␣SlantFont␣");
      }
      ;
      if (mp_xstrcmp(mp→font_name[f],"zpzdr-reversed")≡0) {
        mp_ps_print(mp,
            "␣FontMatrix␣[-1␣0␣0␣1␣0␣0]␣matrix␣concatmatrix␣/FontMatrix␣exch␣def␣");
        mp_ps_print(mp,"/Metrics␣2␣dict␣dup␣begin␣");
        mp_ps_print(mp,"/space[0␣-278]def␣");
        mp_ps_print(mp,"/a12[-904␣-939]def␣");
        mp_ps_print(mp,"end␣def␣");
      }
      ;
      mp_ps_print(mp,"currentdict␣end");
      mp_ps_print_ln(mp);
      mp_ps_print_defined_name(mp,f,prologues);
```

```
    mp_ps_print(mp, "␣exch␣definefont␣pop");
    mp_ps_print_ln(mp);
  }
}
```

133. ⟨ Declarations 29 ⟩ +≡

```
static void mp_ps_print_defined_name(MP mp, font_number f, int prologues);
```

134.

```
static void mp_ps_print_defined_name(MP mp, font_number f, int prologues)
{
  mp_ps_print(mp, "␣/");
  if ((mp_font_is_subsetted(mp, f)) ∧ (mp_font_is_included(mp, f)) ∧ (prologues ≡ 3)) {
    char *s = mp_fm_font_subset_name(mp, f);
    mp_ps_print(mp, s);
    mp_xfree(s);
  }
  else {
    mp_ps_print(mp, mp→font_ps_name[f]);
  }
  if (mp_xstrcmp(mp→font_name[f], "psyrgo") ≡ 0) mp_ps_print(mp, "-Slanted");
  if (mp_xstrcmp(mp→font_name[f], "zpzdr-reversed") ≡ 0) mp_ps_print(mp, "-Reverse");
  if (applied_reencoding(f)) {
    mp_ps_print(mp, "-");
    mp_ps_print(mp, mp→font_enc_name[f]);
  }
  if (mp_fm_font_slant(mp, f) ≠ 0) {
    mp_ps_print(mp, "-Slant_");
    mp_ps_print_int(mp, mp_fm_font_slant(mp, f));
  }
  if (mp_fm_font_extend(mp, f) ≠ 0) {
    mp_ps_print(mp, "-Extend_");
    mp_ps_print_int(mp, mp_fm_font_extend(mp, f));
  }
}
```

135. ⟨ Include encodings and fonts for edge structure h 135 ⟩ ≡

```
mp_font_encodings(mp, mp→last_fnum, (prologues ≡ 2)); ⟨ Embed fonts that are available 136 ⟩
```

This code is used in section 144.

136. ⟨ Embed fonts that are available 136 ⟩ ≡

```
{
  next_size = 0;
  ⟨ Make cur_fsize a copy of the font_sizes array 147 ⟩;
  do {
    done_fonts = true;
    for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
      if (cur_fsize[f] ≠ null) {
        if (prologues ≡ 3) {
          if (¬mp_do_ps_font(mp, f)) {
            if (mp_has_fm_entry(mp, f, Λ)) {
              mp_error(mp, "Font␣embedding␣failed", Λ, true);
            }
          }
        }
        if (cur_fsize[f] ≡ mp_void) cur_fsize[f] = null;
        else cur_fsize[f] = mp_link(cur_fsize[f]);
        if (cur_fsize[f] ≠ null) {
          mp_unmark_font(mp, f);
          done_fonts = false;
        }
      }
    }
    if (¬done_fonts)
      ⟨ Increment next_size and apply mark_string_chars to all text nodes with that size index 137 ⟩;
  } while (¬done_fonts);
}
```

This code is used in section 135.

137. ⟨ Increment *next_size* and apply *mark_string_chars* to all text nodes with that size index 137 ⟩ ≡

```
{
  next_size++;
  mp_apply_mark_string_chars(mp, h, next_size);
}
```

This code is used in sections 136 and 146.

138. We also need to keep track of which characters are used in text nodes in the edge structure that is being shipped out. This is done by procedures that use the left-over *b3* field in the *char_info* words; i.e., $char\_info(f)(c).b3$ gives the status of character c in font f.

⟨ Types 18 ⟩ +≡

```
enum mp_char_mark_state {
  mp_unused = 0, mp_used
};
```

139. ⟨ Declarations 29 ⟩ +≡

```
static void mp_mark_string_chars(MP mp, font_number f, char *s, size_t l);
```

140. **void** *mp_mark_string_chars*(MP *mp*, *font_number f*, **char** **s*, **size_t** *l*)
{
integer b; /* *char_base*[*f*] */
int *bc*, *ec*; /* only characters between these bounds are marked */
unsigned char **k*; /* an index into string *s* */
b = *mp*→*char_base*[*f*];
bc = (**int**) *mp*→*font_bc*[*f*];
ec = (**int**) *mp*→*font_ec*[*f*];
k = (**unsigned char** ***) *s*;
while (*l*−− > 0) {
if ((**k* ≥ *bc*) ∧ (**k* ≤ *ec*)) *mp*→*font_info*[*b* + **k*].*qqqq*.*b3* = *mp_used*;
k++;
}
}

141. ⟨ Declarations 29 ⟩ +≡
static void *mp_unmark_font*(MP *mp*, *font_number f*);

142. **void** *mp_unmark_font*(MP *mp*, *font_number f*)
{
int *k*; /* an index into *font_info* */
for (*k* = *mp*→*char_base*[*f*] + *mp*→*font_bc*[*f*]; *k* ≤ *mp*→*char_base*[*f*] + *mp*→*font_ec*[*f*]; *k*++)
mp→*font_info*[*k*].*qqqq*.*b3* = *mp_unused*;
}

143. ⟨ Declarations 29 ⟩ +≡
static void *mp_print_improved_prologue*(MP *mp*, *mp_edge_object* ** *h*, **int** *p1*, **int** *procset*);

144. **void** *mp_print_improved_prologue*(MP *mp*, *mp_edge_object* ∗ *h*, **int** *prologues*, **int** *procset*)

```
{
  quarterword next_size;      /* the size index for fonts being listed */
  mp_node * cur_fsize;      /* current positions in font_sizes */
  boolean done_fonts;      /* have we finished listing the fonts in the header? */
  font_number f;      /* a font number for loops */
  cur_fsize = mp_xmalloc(mp, (size_t)(mp->font_max + 1), sizeof (mp_node));
  mp_list_used_resources(mp, prologues, procset);
  mp_list_supplied_resources(mp, prologues, procset);
  mp_list_needed_resources(mp, prologues);
  mp_ps_print_nl(mp, "%%EndComments");
  mp_ps_print_nl(mp, "%%BeginProlog");
  if (procset > 0) mp_ps_print_nl(mp, "%%BeginResource: procset mpost");
  else mp_ps_print_nl(mp, "%%BeginResource: procset mpost-minimal");
  mp_ps_print_nl(mp,
      "/bd{bind def}bind def""/fshow {exch findfont exch scalefont setfont show}bd");
  if (procset > 0) ⟨ Print the procset 158 ⟩;
  mp_ps_print_nl(mp, "/fcp{findfont dup length dict begin""{1 index/FID ne{def\
      }{pop pop}ifelse}forall}bd");
  mp_ps_print_nl(mp,
      "/fmc{FontMatrix dup length array copy dup dup}bd""/fmd{/FontMatrix exch def}bd");
  mp_ps_print_nl(mp, "/Amul{4 -1 roll exch mul 1000 div}bd""/ExtendFont{fmc 0 g\
      et Amul 0 exch put fmd}bd");
  mp_ps_print_nl(mp,
      "/ScaleFont{dup fmc 0 get"" Amul 0 exch put dup dup 3 get Amul 3 exch put fmd}bd");
  mp_ps_print_nl(mp, "/SlantFont{fmc 2 get dup 0 eq{pop 1}if"" Amul FontMatrix 0 \
      get mul 2 exch put fmd}bd");
  mp_ps_print_nl(mp, "%%EndResource");
  ⟨ Include encodings and fonts for edge structure h 135 ⟩;
  mp_ps_print_nl(mp, "%%EndProlog");
  mp_ps_print_nl(mp, "%%BeginSetup");
  mp_ps_print_ln(mp);
  for (f = null_font + 1; f ≤ mp->last_fnum; f++) {
    if (mp_has_font_size(mp, f)) {
      if (mp_has_fm_entry(mp, f, Λ)) {
        mp_write_font_definition(mp, f, prologues);
        mp_ps_name_out(mp, mp->font_name[f], true);
        mp_ps_print_defined_name(mp, f, prologues);
        mp_ps_print(mp, " def");
      }
      else {
        char s[256];
        mp_snprintf(s, 256, "font %s cannot be found in any fontmapfile!", mp->font_name[f]);
        mp_warn(mp, s);
        mp_ps_name_out(mp, mp->font_name[f], true);
        mp_ps_name_out(mp, mp->font_name[f], true);
        mp_ps_print(mp, " def");
      }
      mp_ps_print_ln(mp);
    }
  }
  mp_ps_print_nl(mp, "%%EndSetup");
```

```
  mp_ps_print_nl(mp, "%%Page: 1 1");
  mp_ps_print_ln(mp);
  mp_xfree(cur_fsize);
}
```

145. ⟨ Declarations 29 ⟩ +≡

```
static font_number mp_print_font_comments(MP mp, mp_edge_object * h, int prologues);
```

146.

```
static font_number mp_print_font_comments(MP mp, mp_edge_object * h, int prologues)
{
  quarterword next_size;      /* the size index for fonts being listed */
  mp_node * cur_fsize;      /* current positions in font_sizes */
  int ff;      /* a loop counter */
  boolean done_fonts;      /* have we finished listing the fonts in the header? */
  font_number f;      /* a font number for loops */
  int ds;      /* design size and scale factor for a text node, scaled */
  int ldf = 0;      /* the last DocumentFont listed (otherwise null_font) */
  cur_fsize = mp_xmalloc(mp, (size_t)(mp->font_max + 1), sizeof (mp_node));
  if (prologues > 0) {
    ⟨ Give a DocumentFonts comment listing all fonts with non-null font_sizes and eliminate
        duplicates 148 ⟩;
  }
  else {
    next_size = 0;
    ⟨ Make cur_fsize a copy of the font_sizes array 147 ⟩;
    do {
      done_fonts = true;
      for (f = null_font + 1; f ≤ mp->last_fnum; f++) {
        if (cur_fsize[f] ≠ null) {
          ⟨ Print the %*Font comment for font f and advance cur_fsize[f] 157 ⟩;
        }
        if (cur_fsize[f] ≠ null) {
          mp_unmark_font(mp, f);
          done_fonts = false;
        }
        ;
      }
      if (¬done_fonts) {
        ⟨ Increment next_size and apply mark_string_chars to all text nodes with that size index 137 ⟩;
      }
    } while (¬done_fonts);
  }
  mp_xfree(cur_fsize);
  return (font_number)ldf;
}
```

147. ⟨ Make cur_fsize a copy of the font_sizes array 147 ⟩ ≡

```
for (f = null_font + 1; f ≤ mp->last_fnum; f++) cur_fsize[f] = mp->font_sizes[f]
```

This code is used in sections 136 and 146.

148. It's not a good idea to make any assumptions about the *font_ps_name* entries, so we carefully remove duplicates. There is no harm in using a slow, brute-force search.

```
⟨ Give a DocumentFonts comment listing all fonts with non-null font_sizes and eliminate duplicates 148 ⟩ ≡
  {
    ldf = null_font;
    for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
      if (mp→font_sizes[f] ≠ null) {
        if (ldf ≡ null_font) mp_ps_print_nl(mp, "%%DocumentFonts:");
        for (ff = ldf; ff ≥ null_font; ff--) {
          if (mp→font_sizes[ff] ≠ null)
            if (mp_xstrcmp(mp→font_ps_name[f], mp→font_ps_name[ff]) ≡ 0) goto FOUND;
        }
        if ((size_t) mp→ps→ps_offset + 1 + strlen(mp→font_ps_name[f]) > (size_t) mp→max_print_line)
          mp_ps_print_nl(mp, "%%+");
        mp_ps_print_char(mp, '␣');
        mp_ps_print(mp, mp→font_ps_name[f]);
        ldf = (int) f;
      FOUND: ;
      }
    }
  }
```

This code is used in section 146.

149.

```
  static void mp_hex_digit_out(MP mp, quarterword d)
  {
    if (d < 10) mp_ps_print_char(mp, d + '0');
    else mp_ps_print_char(mp, d + 'a' − 10);
  }
```

150. We output the marks as a hexadecimal bit string starting at *font_bc*[*f*].

```
⟨ Declarations 29 ⟩ +≡
  static halfword mp_ps_marks_out(MP mp, font_number f);
```

151.

```
  static halfword mp_ps_marks_out(MP mp, font_number f)
  {
    eight_bits bc, ec;      /* only encode characters between these bounds */
    int p;      /* font_info index for the current character */
    int d;      /* used to construct a hexadecimal digit */
    unsigned b;      /* used to construct a hexadecimal digit */
    bc = mp→font_bc[f];
    ec = mp→font_ec[f];
    ⟨ Restrict the range bc .. ec so that it contains no unused characters at either end 152 ⟩;
    ⟨ Print the initial label indicating that the bitmap starts at bc 153 ⟩;
    ⟨ Print a hexadecimal encoding of the marks for characters bc .. ec 154 ⟩;
    while ((ec < mp→font_ec[f]) ∧ (mp→font_info[p].qqqq.b3 ≡ mp_unused)) {
      p++;
      ec++;
    }
    return (ec + 1);
  }
```

152. We could save time by setting the return value before the loop that decrements *ec*, but there is no point in being so tricky.

```
⟨ Restrict the range bc .. ec so that it contains no unused characters at either end 152 ⟩ ≡
  p = mp→char_base[f] + bc;
  while ((mp→font_info[p].qqqq.b3 ≡ mp_unused) ∧ (bc < ec)) {
    p++;
    bc++;
  }
  p = mp→char_base[f] + ec;
  while ((mp→font_info[p].qqqq.b3 ≡ mp_unused) ∧ (bc < ec)) {
    p−−;
    ec−−;
  }
```

This code is used in section 151.

153. ⟨ Print the initial label indicating that the bitmap starts at *bc* 153 ⟩ ≡

```
  mp_ps_print_char(mp, '␣');
  mp_hex_digit_out(mp, (quarterword)(bc/16));
  mp_hex_digit_out(mp, (quarterword)(bc % 16)); mp_ps_print_char(mp, ':')
```

This code is used in section 151.

154.

```
⟨ Print a hexadecimal encoding of the marks for characters bc .. ec 154 ⟩ ≡
  b = 8;
  d = 0;
  for (p = mp→char_base[f] + bc; p ≤ mp→char_base[f] + ec; p++) {
    if (b ≡ 0) {
      mp_hex_digit_out(mp, (quarterword)d);
      d = 0;
      b = 8;
    }
    if (mp→font_info[p].qqqq.b3 ≠ mp_unused) d += (int) b;
    b = b ≫ 1;
  }
  mp_hex_digit_out(mp, (quarterword)d)
```

This code is used in section 151.

155. Here is a simple function that determines whether there are any marked characters in font *f*.

```
⟨ Declarations 29 ⟩ +≡
  static boolean mp_check_ps_marks(MP mp, font_number f);
```

156.

```
static boolean mp_check_ps_marks(MP mp, font_number f)
{
  int p;      /* font_info index for the current character */
  for (p = mp→char_base[f]; p ≤ mp→char_base[f] + mp→font_ec[f]; p++) {
    if (mp→font_info[p].qqqq.b3 ≡ mp_used) return true;
  }
  return false;
}
```

157. There used to be a check against *emergency_line_length* here, because it was believed that processing programs might not know how to deal with long lines. Nowadays (1.204), we trust backends to do the right thing.

```
#define mp_link(A) (A)¬link /* the link field of a memory word */
#define sc_factor(A) ((mp_font_size_node)(A))¬sc_factor_
  /* the scale factor stored in a font size node */
```

⟨ Print the `%*Font` comment for font f and advance $cur\_fsize[f]$ 157 ⟩ ≡

```
{
  if (mp_check_ps_marks(mp, f)) {
    double dds;
    mp_ps_print_nl(mp, "%*Font: ");
    mp_ps_print(mp, mp¬font_name[f]);
    mp_ps_print_char(mp, ' ');
    ds = (mp¬font_dsize[f] + 8)/16.0;
    dds = (double) ds/65536.0;
    mp_ps_print_double(mp, mp_take_double(mp, dds, sc_factor(cur_fsize[f])));
    mp_ps_print_char(mp, ' ');
    mp_ps_print_double(mp, dds);
    mp_ps_marks_out(mp, f);
  }
  cur_fsize[f] = mp_link(cur_fsize[f]);
}
```

This code is used in section 146.

158. ⟨ Print the procset 158 ⟩ ≡

```
{
  mp_ps_print_nl(mp,
      "/hlw{0 dtransform exch truncate exch idtransform pop setlinewidth}bd");
  mp_ps_print_nl(mp, "/vlw{0 exch dtransform truncate idtransform setlinewidth pop}bd");
  mp_ps_print_nl(mp,
      "/l{lineto}bd/r{rlineto}bd/c{curveto}bd/m{moveto}bd""/p{closepath}bd/n{n\
      ewpath}bd");
  mp_ps_print_nl(mp,
      "/C{setcmykcolor}bd/G{setgray}bd/R{setrgbcolor}bd""/lj{setlinejoin}bd/\
      ml{setmiterlimit}bd");
  mp_ps_print_nl(mp,
      "/lc{setlinecap}bd/S{stroke}bd/F{fill}bd/q{gsave}bd""/Q{grestore}bd/s{sc\
      ale}bd/t{concat}bd");
  mp_ps_print_nl(mp,
      "/sd{setdash}bd/rd{[] 0 setdash}bd/P{showpage}bd/B{q F Q}bd/W{clip}bd");
}
```

This code is used in sections 144 and 160.

159. The prologue defines `fshow` and corrects for the fact that `fshow` arguments use *font_name* instead of *font_ps_name*. Downloaded bitmap fonts might not have reasonable *font_ps_name* entries, but we just charge ahead anyway. The user should not make **prologues** positive if this will cause trouble.

⟨ Declarations 29 ⟩ +≡

```
static void mp_print_prologue(MP mp, mp_edge_object * h, int prologues, int procset);
```

160. **void** *mp_print_prologue*(MP *mp*, *mp_edge_object* ∗ *h*, **int** *prologues*, **int** *procset*)

```
{
  font_number f;
  font_number ldf;
  ldf = mp_print_font_comments(mp, h, prologues);
  mp_ps_print_ln(mp);
  if ((prologues ≡ 1) ∧ (mp→last_ps_fnum ≡ 0)) mp_read_psname_table(mp);
  mp_ps_print(mp, "%%BeginProlog");
  mp_ps_print_ln(mp);
  if ((prologues > 0) ∨ (procset > 0)) {
    if (ldf ≠ null_font) {
      if (prologues > 0) {
        for (f = null_font + 1; f ≤ mp→last_fnum; f++) {
          if (mp_has_font_size(mp, f)) {
            mp_ps_name_out(mp, mp→font_name[f], true);
            mp_ps_name_out(mp, mp→font_ps_name[f], true);
            mp_ps_print(mp, "␣def");
            mp_ps_print_ln(mp);
          }
        }
        if (procset ≡ 0) {
          mp_ps_print(mp, "/fshow␣{exch␣findfont␣exch␣scalefont␣setfont␣show}bind␣def");
          mp_ps_print_ln(mp);
        }
      }
    }
    if (procset > 0) {
      mp_ps_print_nl(mp, "%%BeginResource:␣procset␣mpost");
      if ((prologues > 0) ∧ (ldf ≠ null_font)) mp_ps_print_nl(mp,
          "/bd{bind␣def}bind␣def/fshow␣{exch␣findfont␣exch␣scalefont␣setfont␣show}bd");
      else mp_ps_print_nl(mp, "/bd{bind␣def}bind␣def");
      ⟨ Print the procset 158 ⟩;
      mp_ps_print_nl(mp, "%%EndResource");
      mp_ps_print_ln(mp);
    }
  }
  mp_ps_print(mp, "%%EndProlog");
  mp_ps_print_nl(mp, "%%Page:␣1␣1");
  mp_ps_print_ln(mp);
}
```

161. METAPOST used to have one single routine to print to both 'write' files and the PostScript output. Web2c redefines "Character k cannot be printed", and that resulted in some bugs where 8-bit characters were written to the PostScript file (reported by Wlodek Bzyl).

Also, Hans Hagen requested spaces to be output as "*0*40" instead of a plain space, since that makes it easier to parse the result file for postprocessing.

⟨ Character k is not allowed in PostScript output 161 ⟩ ≡

```
(k ≤ '␣') ∨ (k > '~')
```

This code is used in section 166.

162. We often need to print a pair of coordinates.

```
void mp_ps_pair_out(MP mp, double x, double y)
{
  ps_room(26);
  mp_ps_print_double(mp, x);
  mp_ps_print_char(mp, ' ');
  mp_ps_print_double(mp, y);
  mp_ps_print_char(mp, ' ');
}
```

163. ⟨ Declarations 29 ⟩ +≡

```
static void mp_ps_pair_out(MP mp, double x, double y);
```

164.

```
void mp_ps_print_cmd(MP mp, const char *l, const char *s)
{
  if (number_positive(internal_value(mp_procset))) {
    ps_room(strlen(s));
    mp_ps_print(mp, s);
  }
  else {
    ps_room(strlen(l));
    mp_ps_print(mp, l);
  }
  ;
}
```

165. ⟨ Declarations 29 ⟩ +≡

```
static void mp_ps_print_cmd(MP mp, const char *l, const char *s);
```

166.

```
void mp_ps_string_out(MP mp, const char *s, size_t l)
{
  ASCII_code k;      /* bits to be converted to octal */
  mp_ps_print(mp, "(");
  while (l-- > 0) {
    k = (ASCII_code) * s++;
    if (mp->ps->ps_offset + 5 > mp->max_print_line) {
      mp_ps_print_char(mp, '\\');
      mp_ps_print_ln(mp);
    }
    if ((⟨ Character k is not allowed in PostScript output 161 ⟩)) {
      mp_ps_print_char(mp, '\\');
      mp_ps_print_char(mp, '0' + (k/64));
      mp_ps_print_char(mp, '0' + ((k/8) % 8));
      mp_ps_print_char(mp, '0' + (k % 8));
    }
    else {
      if ((k ≡ '(') ∨ (k ≡ ')') ∨ (k ≡ '\\')) mp_ps_print_char(mp, '\\');
      mp_ps_print_char(mp, k);
    }
  }
  mp_ps_print_char(mp, ')');
}
```

167. ⟨ Declarations 29 ⟩ +≡
```
static void mp_ps_string_out(MP mp, const char *s, size_t l);
```

168. This is a define because the function does not use its *mp* argument.
```
#define mp_is_ps_name(M, A) mp_do_is_ps_name(A)
  static boolean mp_do_is_ps_name(char *s)
  {
    ASCII_code k;       /* the character being checked */
    while ((k = (ASCII_code) * s++)) {
      if ((k ≤ ' ') ∨ (k > '~')) return false;
      if ((k ≡ '(') ∨ (k ≡ ')') ∨ (k ≡ '<') ∨ (k ≡ '>') ∨ (k ≡ '{') ∨ (k ≡ '}') ∨ (k ≡ '/') ∨ (k ≡ '%'))
        return false;
    }
    return true;
  }
```

169. ⟨ Declarations 29 ⟩ +≡
```
static void mp_ps_name_out(MP mp, char *s, boolean lit);
```

170.
```
void mp_ps_name_out(MP mp, char *s, boolean lit)
{
  ps_room(strlen(s) + 2);
  mp_ps_print_char(mp, ' ');
  if (mp_is_ps_name(mp, s)) {
    if (lit) mp_ps_print_char(mp, '/');
    mp_ps_print(mp, s);
  }
  else {
    mp_ps_string_out(mp, s, strlen(s));
    if (¬lit) mp_ps_print(mp, "cvx ");
    mp_ps_print(mp, "cvn");
  }
}
```

171. These special comments described in the *PostScript Language Reference Manual*, 2nd. edition are understood by some PostScript-reading programs. We can't normally output "conforming" PostScript because the structuring conventions don't allow us to say "Please make sure the following characters are downloaded and define the `fshow` macro to access them."

The exact bounding box is written out if *mp_prologues* < 0, although this is not standard PostScript, since it allows TEX to calculate the box dimensions accurately. (Overfull boxes are avoided if an illustration is made to match a given `\hsize`.)

⟨ Declarations 29 ⟩ +≡
```
static void mp_print_initial_comment(MP mp, mp_edge_object * hh, int prologues);
```

172. **void** *mp_print_initial_comment*(MP *mp*, *mp_edge_object* ∗ *hh*, **int** *prologues*)

```
{
  int t;  /* scaled */
  char *s;
  mp_ps_print(mp, "%!PS");
  if (prologues > 0) mp_ps_print(mp, "-Adobe-3.0 EPSF-3.0");
  mp_ps_print_nl(mp, "%%BoundingBox: ");
  if (hh->minx > hh->maxx) {
    mp_ps_print(mp, "0 0 0 0");
  }
  else if (prologues < 0) {
    mp_ps_pair_out(mp, hh->minx, hh->miny);
    mp_ps_pair_out(mp, hh->maxx, hh->maxy);
  }
  else {
    mp_ps_pair_out(mp, floor(hh->minx), floor(hh->miny));
    mp_ps_pair_out(mp, -floor(-hh->maxx), -floor(-hh->maxy));
  }
  mp_ps_print_nl(mp, "%%HiResBoundingBox: ");
  if (hh->minx > hh->maxx) {
    mp_ps_print(mp, "0 0 0 0");
  }
  else {
    mp_ps_pair_out(mp, hh->minx, hh->miny);
    mp_ps_pair_out(mp, hh->maxx, hh->maxy);
  }
  mp_ps_print_nl(mp, "%%Creator: MetaPost ");
  s = mp_metapost_version( );
  mp_ps_print(mp, s);
  mp_xfree(s);
  mp_ps_print_nl(mp, "%%CreationDate: ");
  mp_ps_print_int(mp, round_unscaled(internal_value(mp_year)));
  mp_ps_print_char(mp, '.');
  mp_ps_print_dd(mp, round_unscaled(internal_value(mp_month)));
  mp_ps_print_char(mp, '.');
  mp_ps_print_dd(mp, round_unscaled(internal_value(mp_day)));
  mp_ps_print_char(mp, ':');
  t = round_unscaled(internal_value(mp_time));
  mp_ps_print_dd(mp, t/60);
  mp_ps_print_dd(mp, t % 60);
  mp_ps_print_nl(mp, "%%Pages: 1");
}
```

173. The most important output procedure is the one that gives the PostScript version of a METAPOST path.

```
⟨ mplibps.h 173 ⟩ ≡
#ifndef MPLIBPS_H
#define MPLIBPS_H 1
#include "mplib.h"
  ⟨ Internal Postscript header information 182 ⟩
#endif
```

174. ⟨ Types 18 ⟩ +≡

```
#define gr_left_type(A)(A)→data.types.left_type
#define gr_right_type(A)(A)→data.types.right_type
#define gr_x_coord(A)(A)→x_coord
#define gr_y_coord(A)(A)→y_coord
#define gr_left_x(A)(A)→left_x
#define gr_left_y(A)(A)→left_y
#define gr_right_x(A)(A)→right_x
#define gr_right_y(A)(A)→right_y
#define gr_next_knot(A)(A)→next
#define gr_originator(A)(A)→originator
```

175. If we want to duplicate a knot node, we can say *copy_knot*:

```
static mp_gr_knot mp_gr_copy_knot(MP mp, mp_gr_knot p)
{
  mp_gr_knot q;      /* the copy */
  q = mp_xmalloc(mp, 1, sizeof(struct mp_gr_knot_data));
  memcpy(q, p, sizeof(struct mp_gr_knot_data));
  gr_next_knot(q) = Λ;
  return q;
}
```

176. The *copy_path* routine makes a clone of a given path.

```
static mp_gr_knot mp_gr_copy_path(MP mp, mp_gr_knot p)
{
  mp_gr_knot q, pp, qq;      /* for list manipulation */
  if (p ≡ Λ) return Λ;
  q = mp_gr_copy_knot(mp, p);
  qq = q;
  pp = gr_next_knot(p);
  while (pp ≠ p) {
    gr_next_knot(qq) = mp_gr_copy_knot(mp, pp);
    qq = gr_next_knot(qq);
    pp = gr_next_knot(pp);
  }
  gr_next_knot(qq) = q;
  return q;
}
```

177. When a cyclic list of knot nodes is no longer needed, it can be recycled by calling the following subroutine.

⟨ Declarations 29 ⟩ +≡

```
void mp_do_gr_toss_knot_list(mp_gr_knot p);
```

178.

```
#define mp_gr_toss_knot_list(B,A) mp_do_gr_toss_knot_list(A)
  void mp_do_gr_toss_knot_list(mp_gr_knot p)
  {
    mp_gr_knot q; /* the node being freed */
    mp_gr_knot r; /* the next node */
    if (p ≡ Λ) return;
    q = p;
    do {
      r = gr_next_knot(q);
      mp_xfree(q);
      q = r;
    } while (q ≠ p);
  }
```

179.

```
  static void mp_gr_ps_path_out(MP mp, mp_gr_knot h)
  {
    mp_gr_knot p, q; /* for scanning the path */
    double d; /* a temporary value */
    boolean curved; /* true unless the cubic is almost straight */
    ps_room(40);
    mp_ps_print_cmd(mp, "newpath␣", "n␣");
    mp_ps_pair_out(mp, gr_x_coord(h), gr_y_coord(h));
    mp_ps_print_cmd(mp, "moveto", "m");
    p = h;
    do {
      if (gr_right_type(p) ≡ mp_endpoint) {
        if (p ≡ h) mp_ps_print_cmd(mp, "␣0␣0␣rlineto", "␣0␣0␣r");
        return;
      }
      q = gr_next_knot(p);
      ⟨ Start a new line and print the PostScript commands for the curve from p to q 180 ⟩;
      p = q;
    } while (p ≠ h);
    mp_ps_print_cmd(mp, "␣closepath", "␣p");
  }
```

180. ⟨ Start a new line and print the PostScript commands for the curve from p to q 180 ⟩ ≡

```
curved = true;
⟨ Set curved: = false if the cubic from p to q is almost straight 181 ⟩;
mp_ps_print_ln(mp);
if (curved) {
  mp_ps_pair_out(mp, gr_right_x(p), gr_right_y(p));
  mp_ps_pair_out(mp, gr_left_x(q), gr_left_y(q));
  mp_ps_pair_out(mp, gr_x_coord(q), gr_y_coord(q));
  mp_ps_print_cmd(mp, "curveto", "c");
}
else if (q ≠ h) {
  mp_ps_pair_out(mp, gr_x_coord(q), gr_y_coord(q));
  mp_ps_print_cmd(mp, "lineto", "l");
}
```

This code is used in section 179.

181. Two types of straight lines come up often in METAPOST paths: cubics with zero initial and final velocity as created by *make_path* or *make_envelope*, and cubics with control points uniformly spaced on a line as created by *make_choices*.

```
#define bend_tolerance (131/65536.0)   /* allow rounding error of 2 · 10^-3 */
```

⟨ Set *curved*: = *false* if the cubic from p to q is almost straight 181 ⟩ ≡

```
if (gr_right_x(p) ≡ gr_x_coord(p))
  if (gr_right_y(p) ≡ gr_y_coord(p))
    if (gr_left_x(q) ≡ gr_x_coord(q))
      if (gr_left_y(q) ≡ gr_y_coord(q)) curved = false;
d = gr_left_x(q) − gr_right_x(p);
if (fabs(gr_right_x(p) − gr_x_coord(p) − d) ≤ bend_tolerance)
  if (fabs(gr_x_coord(q) − gr_left_x(q) − d) ≤ bend_tolerance) {
    d = gr_left_y(q) − gr_right_y(p);
    if (fabs(gr_right_y(p) − gr_y_coord(p) − d) ≤ bend_tolerance)
      if (fabs(gr_y_coord(q) − gr_left_y(q) − d) ≤ bend_tolerance) curved = false;
  }
```

This code is used in section 180.

182. The colored objects use a struct with anonymous fields to express the color parts:

⟨ Internal Postscript header information 182 ⟩ ≡

```
typedef struct {
  double a_val, b_val, c_val, d_val;
} mp_color;
```

See also sections 183, 188, 235, 240, and 243.

This code is used in section 173.

183. The exported form of a dash pattern is simpler than the internal format, it is closely modelled to the PostScript model. The array of dashes is ended by a single negative value, because this is not allowed in PostScript.

⟨ Internal Postscript header information 182 ⟩ +≡

```
typedef struct {
  double offset;
  double *array;
} mp_dash_object;
```

184.

```
#define mp_gr_toss_dashes(A,B) mp_do_gr_toss_dashes(B)
⟨ Declarations 29 ⟩ +≡
  static void mp_do_gr_toss_dashes(mp_dash_object *dl);
```

185.

```
void mp_do_gr_toss_dashes(mp_dash_object *dl)
{
  if (dl ≡ Λ) return;
  mp_xfree(dl→array);
  mp_xfree(dl);
}
```

186.

```
static mp_dash_object *mp_gr_copy_dashes(MP mp, mp_dash_object *dl)
{
  mp_dash_object *q = Λ;
  (void) mp;
  if (dl ≡ Λ) return Λ;
  q = mp_xmalloc(mp, 1, sizeof(mp_dash_object));
  memcpy(q, dl, sizeof(mp_dash_object));
  if (dl→array ≠ Λ) {
    size_t i = 0;
    while (*(dl→array + i) ≠ -1) i++;
    q→array = mp_xmalloc(mp, i, sizeof(double));
    memcpy(q→array, dl→array, (i * sizeof(double)));
  }
  return q;
}
```

187. Now for outputting the actual graphic objects. First, set up some structures and access macros.

```
#define gr_has_color(A) (gr_type((A)) < mp_start_clip_code)
⟨ Types 18 ⟩ +≡
#define gr_type(A)(A)→type
#define gr_link(A)(A)→next
#define gr_color_model(A)(A)→color_model
#define gr_red_val(A)(A)→color.a_val
#define gr_green_val(A)(A)→color.b_val
#define gr_blue_val(A)(A)→color.c_val
#define gr_cyan_val(A)(A)→color.a_val
#define gr_magenta_val(A)(A)→color.b_val
#define gr_yellow_val(A)(A)→color.c_val
#define gr_black_val(A)(A)→color.d_val
#define gr_grey_val(A)(A)→color.a_val
#define gr_path_p(A)(A)→path_p
#define gr_htap_p(A) ( ( mp_fill_object * ) A ) → htap_p
#define gr_pen_p(A)(A)→pen_p
#define gr_ljoin_val(A)(A)→ljoin
#define gr_lcap_val(A) ( ( mp_stroked_object * ) A ) → lcap
#define gr_miterlim_val(A)(A)→miterlim
#define gr_pre_script(A)(A)→pre_script
#define gr_post_script(A)(A)→post_script
#define gr_dash_p(A) ( ( mp_stroked_object * ) A ) → dash_p
#define gr_size_index(A) ( ( mp_text_object * ) A ) → size_index
#define gr_text_p(A) ( ( mp_text_object * ) A ) → text_p
#define gr_text_l(A) ( ( mp_text_object * ) A ) → text_l
#define gr_font_n(A) ( ( mp_text_object * ) A ) → font_n
#define gr_font_name(A) ( ( mp_text_object * ) A ) → font_name
#define gr_font_dsize(A) ( ( mp_text_object * ) A ) → font_dsize
#define gr_width_val(A) ( ( mp_text_object * ) A ) → width
#define gr_height_val(A) ( ( mp_text_object * ) A ) → height
#define gr_depth_val(A) ( ( mp_text_object * ) A ) → depth
#define gr_tx_val(A) ( ( mp_text_object * ) A ) → tx
#define gr_ty_val(A) ( ( mp_text_object * ) A ) → ty
#define gr_txx_val(A) ( ( mp_text_object * ) A ) → txx
#define gr_txy_val(A) ( ( mp_text_object * ) A ) → txy
#define gr_tyx_val(A) ( ( mp_text_object * ) A ) → tyx
#define gr_tyy_val(A) ( ( mp_text_object * ) A ) → tyy
```

188. ⟨ Internal Postscript header information 182 ⟩ +≡

```
#define GRAPHIC_BODY
    int type; struct mp_graphic_object *next
    typedef struct mp_graphic_object {
      GRAPHIC_BODY;
    } mp_graphic_object;
    typedef struct mp_text_object {
      GRAPHIC_BODY;
      char *pre_script;
      char *post_script;
      mp_color color;
      unsigned char color_model;
      unsigned char size_index;
      char *text_p;
      size_t text_l;
      char *font_name;
      double font_dsize;
      unsigned int font_n;
      double width;
      double height;
      double depth;
      double tx;
      double ty;
      double txx;
      double txy;
      double tyx;
      double tyy;
    } mp_text_object;
    typedef struct mp_fill_object {
      GRAPHIC_BODY;
      char *pre_script;
      char *post_script;
      mp_color color;
      unsigned char color_model;
      unsigned char ljoin;
      mp_gr_knot path_p;
      mp_gr_knot htap_p;
      mp_gr_knot pen_p;
      double miterlim;
    } mp_fill_object;
    typedef struct mp_stroked_object {
      GRAPHIC_BODY;
      char *pre_script;
      char *post_script;
      mp_color color;
      unsigned char color_model;
      unsigned char ljoin;
      unsigned char lcap;
      mp_gr_knot path_p;
      mp_gr_knot pen_p;
```

```
  double miterlim;
  mp_dash_object *dash_p;
} mp_stroked_object;
typedef struct mp_clip_object {
  GRAPHIC_BODY;
  mp_gr_knot path_p;
} mp_clip_object;
typedef struct mp_bounds_object {
  GRAPHIC_BODY;
  mp_gr_knot path_p;
} mp_bounds_object;
typedef struct mp_special_object {
  GRAPHIC_BODY;
  char *pre_script;
} mp_special_object;
typedef struct mp_edge_object {
  struct mp_graphic_object *body;
  struct mp_edge_object *next;
  char *filename;
  MP parent;
  double minx, miny, maxx, maxy;
  double width, height, depth, ital_corr;
  int charcode;
} mp_edge_object;
```

189. ⟨ Exported function headers 5 ⟩ +≡

```
mp_graphic_object *mp_new_graphic_object(MP mp, int type);
```

190. **mp_graphic_object** **mp_new_graphic_object*(MP *mp*, **int** *type*)

```
{
  mp_graphic_object *p;
  size_t size;
  switch (type) {
  case mp_fill_code: size = sizeof(mp_fill_object);
    break;
  case mp_stroked_code: size = sizeof(mp_stroked_object);
    break;
  case mp_text_code: size = sizeof(mp_text_object);
    break;
  case mp_start_clip_code: size = sizeof(mp_clip_object);
    break;
  case mp_start_bounds_code: size = sizeof(mp_bounds_object);
    break;
  case mp_special_code: size = sizeof(mp_special_object);
    break;
  default: size = sizeof(mp_graphic_object);
    break;
  }
  p = (mp_graphic_object *) mp_xmalloc(mp, 1, size);
  memset(p, 0, size);
  gr_type(p) = type;
  return p;
}
```

191. We need to keep track of several parameters from the PostScript graphics state. This allows us to be sure that PostScript has the correct values when they are needed without wasting time and space setting them unnecessarily.

```
#define gs_red mp→ps→gs_state→red_field
#define gs_green mp→ps→gs_state→green_field
#define gs_blue mp→ps→gs_state→blue_field
#define gs_black mp→ps→gs_state→black_field
#define gs_colormodel mp→ps→gs_state→colormodel_field
#define gs_ljoin mp→ps→gs_state→ljoin_field
#define gs_lcap mp→ps→gs_state→lcap_field
#define gs_adj_wx mp→ps→gs_state→adj_wx_field
#define gs_miterlim mp→ps→gs_state→miterlim_field
#define gs_dash_p mp→ps→gs_state→dash_p_field
#define gs_dash_init_done mp→ps→gs_state→dash_done_field
#define gs_previous mp→ps→gs_state→previous_field
#define gs_width mp→ps→gs_state→width_field
⟨ Types 18 ⟩ +≡
  typedef struct _gs_state {
    double red_field;
    double green_field;
    double blue_field;
    double black_field; /* color from the last setcmykcolor or setrgbcolor or setgray command */
    quarterword colormodel_field; /* the current colormodel */
    quarterword ljoin_field;
    quarterword lcap_field; /* values from the last setlinejoin and setlinecap commands */
    quarterword adj_wx_field; /* what resolution-dependent adjustment applies to the width */
    double miterlim_field; /* the value from the last setmiterlimit command */
    mp_dash_object *dash_p_field; /* edge structure for last setdash command */
    boolean dash_done_field; /* to test for initial setdash */
    struct _gs_state *previous_field; /* backlink to the previous _gs_state structure */
    double width_field; /* width setting or −1 if no setlinewidth command so far */
  } _gs_state;
```

192. ⟨ Globals 7 ⟩ +≡

```
  struct _gs_state *gs_state;
```

193. ⟨ Set initial values 8 ⟩ +≡

```
  mp→ps→gs_state = Λ;
```

194. ⟨ Dealloc variables 62 ⟩ +≡

```
  mp_xfree(mp→ps→gs_state);
```

195. To avoid making undue assumptions about the initial graphics state, these parameters are given special values that are guaranteed not to match anything in the edge structure being shipped out. On the other hand, the initial color should be black so that the translation of an all-black picture will have no **setcolor** commands. (These would be undesirable in a font application.) Hence we use $c = 0$ when initializing the graphics state and we use $c < 0$ to recover from a situation where we have lost track of the graphics state.

```
#define mp_void (mp_node)(null + 1) /* a null pointer different from null */
  static void mp_gs_unknown_graphics_state(MP mp, int c)
  {
    struct _gs_state *p; /* to shift graphic states around */
    if ((c ≡ 0) ∨ (c ≡ −1)) {
      if (mp→ps→gs_state ≡ Λ) {
        mp→ps→gs_state = mp_xmalloc(mp, 1, sizeof(struct _gs_state));
        gs_previous = Λ;
      }
      else {
        while (gs_previous ≠ Λ) {
          p = gs_previous;
          mp_xfree(mp→ps→gs_state);
          mp→ps→gs_state = p;
        }
      }
      gs_red = c;
      gs_green = c;
      gs_blue = c;
      gs_black = c;
      gs_colormodel = mp_uninitialized_model;
      gs_ljoin = 3;
      gs_lcap = 3;
      gs_miterlim = 0.0;
      gs_dash_p = Λ;
      gs_dash_init_done = false;
      gs_width = −1.0;
    }
    else if (c ≡ 1) {
      p = mp→ps→gs_state;
      mp→ps→gs_state = mp_xmalloc(mp, 1, sizeof(struct _gs_state));
      memcpy(mp→ps→gs_state, p, sizeof(struct _gs_state));
      gs_previous = p;
    }
    else if (c ≡ 2) {
      p = gs_previous;
      mp_xfree(mp→ps→gs_state);
      mp→ps→gs_state = p;
    }
  }
```

196. When it is time to output a graphical object, *fix_graphics_state* ensures that PostScript's idea of the graphics state agrees with what is stored in the object.

```
⟨ Declarations 29 ⟩ +≡
  static void mp_gr_fix_graphics_state(MP mp, mp_graphic_object *p);
```

197. **void** *mp_gr_fix_graphics_state*(MP *mp*, **mp_graphic_object** ∗*p*)
{ /∗ get ready to output graphical object *p* ∗/
mp_gr_knot pp, *path_p*; /∗ for list manipulation ∗/
mp_dash_object ∗*hh*;
double *wx*, *wy*, *ww*; /∗ dimensions of pen bounding box ∗/
quarterword adj_wx; /∗ whether pixel rounding should be based on *wx* or *wy* ∗/
double *tx*, *ty*; /∗ temporaries for computing *adj_wx* ∗/
if (*gr_has_color*(*p*)) ⟨ Make sure PostScript will use the right color for object *p* 200 ⟩;
if ((*gr_type*(*p*) ≡ *mp_fill_code*) ∨ (*gr_type*(*p*) ≡ *mp_stroked_code*)) {
if (*gr_type*(*p*) ≡ *mp_fill_code*) {
pp = *gr_pen_p*((**mp_fill_object** ∗) *p*);
path_p = *gr_path_p*((**mp_fill_object** ∗) *p*);
}
else {
pp = *gr_pen_p*((**mp_stroked_object** ∗) *p*);
path_p = *gr_path_p*((**mp_stroked_object** ∗) *p*);
}
if (*pp* ≠ Λ)
if (*pen_is_elliptical*(*pp*)) {
⟨ Generate PostScript code that sets the stroke width to the appropriate rounded value 201 ⟩;
⟨ Make sure PostScript will use the right dash pattern for *dash_p*(*p*) 207 ⟩;
⟨ Decide whether the line cap parameter matters and set it if necessary 198 ⟩;
⟨ Set the other numeric parameters as needed for object *p* 199 ⟩;
}
}
if (*mp*→*ps*→*ps_offset* > 0) *mp_ps_print_ln*(*mp*);
}

198. ⟨ Decide whether the line cap parameter matters and set it if necessary 198 ⟩ ≡
if (*gr_type*(*p*) ≡ *mp_stroked_code*) {
mp_stroked_object ∗*ts* = (**mp_stroked_object** ∗) *p*;
if ((*gr_left_type*(*gr_path_p*(*ts*)) ≡ *mp_endpoint*) ∨ (*gr_dash_p*(*ts*) ≠ Λ))
if (*gs_lcap* ≠ (*quarterword*)*gr_lcap_val*(*ts*)) {
ps_room(13);
mp_ps_print_char(*mp*, '␣');
mp_ps_print_char(*mp*, '0' + *gr_lcap_val*(*ts*));
mp_ps_print_cmd(*mp*, "␣setlinecap", "␣lc");
gs_lcap = (*quarterword*)*gr_lcap_val*(*ts*);
}
}
This code is used in section 197.

199.

```
#define set_ljoin_miterlim(p)
        if (gs_ljoin ≠ (quarterword)gr_ljoin_val(p)) {
          ps_room(14);
          mp_ps_print_char(mp, '␣');
          mp_ps_print_char(mp, '0' + gr_ljoin_val(p));
          mp_ps_print_cmd(mp, "␣setlinejoin", "␣lj");
          gs_ljoin = (quarterword)gr_ljoin_val(p);
        }
        if (gs_miterlim ≠ gr_miterlim_val(p)) {
          ps_room(27);
          mp_ps_print_char(mp, '␣');
          mp_ps_print_double(mp, gr_miterlim_val(p));
          mp_ps_print_cmd(mp, "␣setmiterlimit", "␣ml");
          gs_miterlim = gr_miterlim_val(p);
        }
```

⟨ Set the other numeric parameters as needed for object p 199 ⟩ ≡

```
if (gr_type(p) ≡ mp_stroked_code) {
  mp_stroked_object *ts = (mp_stroked_object *) p;
  set_ljoin_miterlim(ts);
}
else {
  mp_fill_object *ts = (mp_fill_object *) p;
  set_ljoin_miterlim(ts);
}
```

This code is used in section 197.

200.

```
#define set_color_objects(pq) object_color_model = pq→color_model;
        object_color_a = pq→color.a_val;
        object_color_b = pq→color.b_val;
        object_color_c = pq→color.c_val;
        object_color_d = pq→color.d_val;
⟨ Make sure PostScript will use the right color for object p 200 ⟩ ≡
  {
    int object_color_model;
    double object_color_a, object_color_b, object_color_c, object_color_d;
    if (gr_type(p) ≡ mp_fill_code) {
      mp_fill_object *pq = (mp_fill_object *) p;
      set_color_objects(pq);
    }
    else if (gr_type(p) ≡ mp_stroked_code) {
      mp_stroked_object *pq = (mp_stroked_object *) p;
      set_color_objects(pq);
    }
    else {
      mp_text_object *pq = (mp_text_object *) p;
      set_color_objects(pq);
    }
    if (object_color_model ≡ mp_rgb_model) {
      if ((gs_colormodel ≠ mp_rgb_model)∨(gs_red ≠ object_color_a)∨(gs_green ≠ object_color_b)∨(gs_blue ≠
          object_color_c)) {
        gs_red = object_color_a;
        gs_green = object_color_b;
        gs_blue = object_color_c;
        gs_black = -1.0;
        gs_colormodel = mp_rgb_model;
        {
          ps_room(36);
          mp_ps_print_char(mp, '␣');
          mp_ps_print_double(mp, gs_red);
          mp_ps_print_char(mp, '␣');
          mp_ps_print_double(mp, gs_green);
          mp_ps_print_char(mp, '␣');
          mp_ps_print_double(mp, gs_blue);
          mp_ps_print_cmd(mp, "␣setrgbcolor", "␣R");
        }
      }
    }
    else if (object_color_model ≡ mp_cmyk_model) {
      if ((gs_red ≠ object_color_a) ∨ (gs_green ≠ object_color_b) ∨ (gs_blue ≠ object_color_c) ∨ (gs_black ≠
          object_color_d) ∨ (gs_colormodel ≠ mp_cmyk_model)) {
        gs_red = object_color_a;
        gs_green = object_color_b;
        gs_blue = object_color_c;
        gs_black = object_color_d;
        gs_colormodel = mp_cmyk_model;
        {
```

```
        ps_room(45);
        mp_ps_print_char(mp, '␣');
        mp_ps_print_double(mp, gs_red);
        mp_ps_print_char(mp, '␣');
        mp_ps_print_double(mp, gs_green);
        mp_ps_print_char(mp, '␣');
        mp_ps_print_double(mp, gs_blue);
        mp_ps_print_char(mp, '␣');
        mp_ps_print_double(mp, gs_black);
        mp_ps_print_cmd(mp, "␣setcmykcolor", "␣C");
      }
    }
  }
  else if (object_color_model ≡ mp_grey_model) {
    if ((gs_red ≠ object_color_a) ∨ (gs_colormodel ≠ mp_grey_model)) {
      gs_red = object_color_a;
      gs_green = −1.0;
      gs_blue = −1.0;
      gs_black = −1.0;
      gs_colormodel = mp_grey_model;
      {
        ps_room(16);
        mp_ps_print_char(mp, '␣');
        mp_ps_print_double(mp, gs_red);
        mp_ps_print_cmd(mp, "␣setgray", "␣G");
      }
    }
  }
  else if (object_color_model ≡ mp_no_model) {
    gs_colormodel = mp_no_model;
  }
}
```

This code is used in section 197.

201. In order to get consistent widths for horizontal and vertical pen strokes, we want PostScript to use an integer number of pixels for the **setwidth** parameter. We set *gs_width* to the ideal horizontal or vertical stroke width and then generate PostScript code that computes the rounded value. For non-circular pens, the pen shape will be rescaled so that horizontal or vertical parts of the stroke have the computed width.

Rounding the width to whole pixels is not likely to improve the appearance of diagonal or curved strokes, but we do it anyway for consistency. The **truncate** command generated here tends to make all the strokes a little thinner, but this is appropriate for PostScript's scan-conversion rules. Even with truncation, an ideal with of w pixels gets mapped into $\lfloor w \rfloor + 1$. It would be better to have $\lceil w \rceil$ but that is ridiculously expensive to compute in PostScript.

```
⟨ Generate PostScript code that sets the stroke width to the appropriate rounded value 201 ⟩ ≡
  ⟨ Set wx and wy to the width and height of the bounding box for pen_p(p) 202 ⟩;
  ⟨ Use pen_p(p) and path_p(p) to decide whether wx or wy is more important and set adj_wx and ww
      accordingly 203 ⟩;
  if ((ww ≠ gs_width) ∨ (adj_wx ≠ gs_adj_wx)) {
    if (adj_wx ≠ 0) {
      ps_room(13);
      mp_ps_print_char(mp, '␣');
      mp_ps_print_double(mp, ww);
      mp_ps_print_cmd(mp, "␣0␣dtransform␣exch␣truncate␣exch␣idtransform␣pop␣setlinewidth",
          "␣hlw");
    }
    else {
      if (number_positive(internal_value(mp_procset))) {
        ps_room(13);
        mp_ps_print_char(mp, '␣');
        mp_ps_print_double(mp, ww);
        mp_ps_print(mp, "␣vlw");
      }
      else {
        ps_room(15);
        mp_ps_print(mp, "␣0␣");
        mp_ps_print_double(mp, ww);
        mp_ps_print(mp, "␣dtransform␣truncate␣idtransform␣setlinewidth␣pop");
      }
    }
    gs_width = ww;
    gs_adj_wx = adj_wx;
  }
```

This code is used in section 197.

202. ⟨ Set wx and wy to the width and height of the bounding box for $pen\_p(p)$ 202 ⟩ ≡

```
if ((gr_right_x(pp) ≡ gr_x_coord(pp)) ∧ (gr_left_y(pp) ≡ gr_y_coord(pp))) {
  wx = fabs(gr_left_x(pp) − gr_x_coord(pp));
  wy = fabs(gr_right_y(pp) − gr_y_coord(pp));
}
else {
  double a, b;
  a = gr_left_x(pp) − gr_x_coord(pp);
  b = gr_right_x(pp) − gr_x_coord(pp);
  wx = sqrt(a ∗ a + b ∗ b);
  a = gr_left_y(pp) − gr_y_coord(pp);
  b = gr_right_y(pp) − gr_y_coord(pp);
  wy = sqrt(a ∗ a + b ∗ b);
}
```

This code is used in section 201.

203. The path is considered "essentially horizontal" if its range of y coordinates is less than the y range wy for the pen. "Essentially vertical" paths are detected similarly. This code ensures that no component of the pen transformation is more that $aspect\_bound ∗ (ww + 1)$.

```
#define aspect_bound 10.0/65536.0
        /∗ "less important" of wx, wy cannot exceed the other by more than this factor ∗/
#define do_x_loc 1
#define do_y_loc 2
```

⟨ Use $pen\_p(p)$ and $path\_p(p)$ to decide whether wx or wy is more important and set adj_wx and ww accordingly 203 ⟩ ≡

```
tx = 1.0/65536.0;
ty = 1.0/65536.0;
if (mp_gr_coord_rangeOK(path_p, do_y_loc, wy)) tx = aspect_bound;
else if (mp_gr_coord_rangeOK(path_p, do_x_loc, wx)) ty = aspect_bound;
if (wy/ty ≥ wx/tx) {
  ww = wy;
  adj_wx = 0;
}
else {
  ww = wx;
  adj_wx = 1;
}
```

This code is used in section 201.

204. This routine quickly tests if path h is "essentially horizontal" or "essentially vertical," where $zoff$ is $x\_loc(0)$ or $y\_loc(0)$ and dz is allowable range for x or y. We do not need and cannot afford a full bounding-box computation.

⟨ Declarations 29 ⟩ +≡

```
static boolean mp_gr_coord_rangeOK(mp_gr_knot h, quarterword zoff, double dz);
```

205. *boolean mp_gr_coord_rangeOK* (*mp_gr_knot h*, *quarterword zoff* , **double** *dz*)

```
{
  mp_gr_knot p;      /* for scanning the path form h */
  double zlo, zhi;      /* coordinate range so far */
  double z;      /* coordinate currently being tested */
  if (zoff ≡ do_x_loc) {
    zlo = gr_x_coord(h);
    zhi = zlo;
    p = h;
    while (gr_right_type(p) ≠ mp_endpoint) {
      z = gr_right_x(p);
      ⟨ Make zlo .. zhi include z and return false if zhi − zlo > dz 206 ⟩;
      p = gr_next_knot(p);
      z = gr_left_x(p);
      ⟨ Make zlo .. zhi include z and return false if zhi − zlo > dz 206 ⟩;
      z = gr_x_coord(p);
      ⟨ Make zlo .. zhi include z and return false if zhi − zlo > dz 206 ⟩;
      if (p ≡ h) break;
    }
  }
  else {
    zlo = gr_y_coord(h);
    zhi = zlo;
    p = h;
    while (gr_right_type(p) ≠ mp_endpoint) {
      z = gr_right_y(p);
      ⟨ Make zlo .. zhi include z and return false if zhi − zlo > dz 206 ⟩;
      p = gr_next_knot(p);
      z = gr_left_y(p);
      ⟨ Make zlo .. zhi include z and return false if zhi − zlo > dz 206 ⟩;
      z = gr_y_coord(p);
      ⟨ Make zlo .. zhi include z and return false if zhi − zlo > dz 206 ⟩;
      if (p ≡ h) break;
    }
  }
  return true;
}
```

206. ⟨ Make *zlo* .. *zhi* include *z* and **return** *false* if *zhi* − *zlo* > *dz* 206 ⟩ ≡

```
if (z < zlo) zlo = z;
else if (z > zhi) zhi = z;
if (zhi − zlo > dz) return false
```

This code is used in section 205.

207. Filling with an elliptical pen is implemented via a combination of **stroke** and **fill** commands and a nontrivial dash pattern would interfere with this. Note that we don't use *delete_edge_ref* because *gs_dash_p* is not counted as a reference.

```
⟨ Make sure PostScript will use the right dash pattern for dash_p(p) 207 ⟩ ≡
  if (gr_type(p) ≡ mp_fill_code ∨ gr_dash_p(p) ≡ Λ) {
    hh = Λ;
  }
  else {
    hh = gr_dash_p(p);
  }
  if (hh ≡ Λ) {
    if (gs_dash_p ≠ Λ ∨ gs_dash_init_done ≡ false) {
      mp_ps_print_cmd(mp, "␣[]␣0␣setdash", "␣rd");
      gs_dash_p = Λ;
      gs_dash_init_done = true;
    }
  }
  else if (¬mp_gr_same_dashes(gs_dash_p, hh)) {
    ⟨ Set the dash pattern from dash_list(hh) scaled by scf 208 ⟩;
  }
```

This code is used in section 197.

208. The original code had a check here to ensure that the result from *mp_take_scaled* did not go out of bounds.

```
⟨ Set the dash pattern from dash_list(hh) scaled by scf 208 ⟩ ≡
  {
    gs_dash_p = hh;
    if ((gr_dash_p(p) ≡ Λ) ∨ (hh ≡ Λ) ∨ (hh→array ≡ Λ)) {
      mp_ps_print_cmd(mp, "␣[]␣0␣setdash", "␣rd");
    }
    else {
      int i;
      ps_room(28);
      mp_ps_print(mp, "␣[");
      for (i = 0; ∗(hh→array + i) ≠ −1; i++) {
        ps_room(13);
        mp_ps_print_double(mp, ∗(hh→array + i));
        mp_ps_print_char(mp, '␣');
      }
      ps_room(22);
      mp_ps_print(mp, "]␣");
      mp_ps_print_double(mp, hh→offset);
      mp_ps_print_cmd(mp, "␣setdash", "␣sd");
    }
  }
```

This code is used in section 207.

209. ⟨ Declarations 29 ⟩ +≡

```
  static boolean mp_gr_same_dashes(mp_dash_object ∗h, mp_dash_object ∗hh);
```

210. This function test if h and hh represent the same dash pattern.

```
boolean mp_gr_same_dashes(mp_dash_object *h, mp_dash_object *hh)
{
  boolean ret = false;
  int i = 0;
  if (h ≡ hh) ret = true;
  else if ((h ≡ Λ) ∨ (hh ≡ Λ)) ret = false;
  else if (h→offset ≠ hh→offset) ret = false;
  else if (h→array ≡ hh→array) ret = true;
  else if (h→array ≡ Λ ∨ hh→array ≡ Λ) ret = false;
  else {
    ⟨ Compare dash_list(h) and dash_list(hh) 211 ⟩;
  }
  return ret;
}
```

211. ⟨ Compare *dash_list*(*h*) and *dash_list*(*hh*) 211 ⟩ ≡

```
{
  while (*(h→array + i) ≠ −1 ∧ *(hh→array + i) ≠ −1 ∧ *(h→array + i) ≡ *(hh→array + i)) i++;
  if (i > 0) {
    if (*(h→array + i) ≡ −1 ∧ *(hh→array + i) ≡ −1) ret = true;
  }
}
```

This code is used in section 210.

212. When stroking a path with an elliptical pen, it is necessary to transform the coordinate system so that a unit circular pen will have the desired shape. To keep this transformation local, we enclose it in a

gsave . . . **grestore**

block. Any translation component must be applied to the path being stroked while the rest of the transformation must apply only to the pen. If *fill_also* = *true*, the path is to be filled as well as stroked so we must insert commands to do this after giving the path.

⟨ Declarations 29 ⟩ +≡

```
static void mp_gr_stroke_ellipse(MP mp, mp_graphic_object *h, boolean fill_also);
```

213.

```
void mp_gr_stroke_ellipse(MP mp, mp_graphic_object *h, boolean fill_also)
{   /* generate an elliptical pen stroke from object h */
  double txx, txy, tyx, tyy;   /* transformation parameters */
  mp_gr_knot p;   /* the pen to stroke with */
  double d1, det;   /* for tweaking transformation parameters */
  double s;   /* also for tweaking transformation paramters */
  boolean transformed;   /* keeps track of whether gsave/grestore are needed */
  transformed = false;
  ⟨ Use pen_p(h) to set the transformation parameters and give the initial translation 214 ⟩;
  ⟨ Tweak the transformation parameters so the transformation is nonsingular 217 ⟩;
  if (gr_type(h) ≡ mp_fill_code) {
    mp_gr_ps_path_out(mp, gr_path_p((mp_fill_object *) h));
  }
  else {
    mp_gr_ps_path_out(mp, gr_path_p((mp_stroked_object *) h));
  }
  if (number_zero(internal_value(mp_procset))) {
    if (fill_also) mp_ps_print_nl(mp, "gsave␣fill␣grestore");
    ⟨ Issue PostScript commands to transform the coordinate system 216 ⟩;
    mp_ps_print(mp, "␣stroke");
    if (transformed) mp_ps_print(mp, "␣grestore");
  }
  else {
    if (fill_also) mp_ps_print_nl(mp, "B");
    else mp_ps_print_ln(mp);
    if ((txy ≠ 0.0) ∨ (tyx ≠ 0.0)) {
      mp_ps_print(mp, "␣[");
      mp_ps_pair_out(mp, txx, tyx);
      mp_ps_pair_out(mp, txy, tyy);
      mp_ps_print(mp, "0␣0]␣t");
    }
    else if ((txx ≠ unity) ∨ (tyy ≠ unity)) {
      mp_ps_print(mp, "␣");
      mp_ps_pair_out(mp, txx, tyy);
      mp_ps_print(mp, "␣s");
    }
    ;
    mp_ps_print(mp, "␣S");
    if (transformed) mp_ps_print(mp, "␣Q");
  }
  mp_ps_print_ln(mp);
}
```

214. ⟨ Use *pen_p*(*h*) to set the transformation parameters and give the initial translation 214 ⟩ ≡

```
if (gr_type(h) ≡ mp_fill_code) {
  p = gr_pen_p((mp_fill_object *) h);
}
else {
  p = gr_pen_p((mp_stroked_object *) h);
}
txx = gr_left_x(p);
tyx = gr_left_y(p);
txy = gr_right_x(p);
tyy = gr_right_y(p);
if ((gr_x_coord(p) ≠ 0.0) ∨ (gr_y_coord(p) ≠ 0.0)) {
  mp_ps_print_nl(mp, "");
  mp_ps_print_cmd(mp, "gsave␣", "q␣");
  mp_ps_pair_out(mp, gr_x_coord(p), gr_y_coord(p));
  mp_ps_print(mp, "translate␣");
  txx −= gr_x_coord(p);
  tyx −= gr_y_coord(p);
  txy −= gr_x_coord(p);
  tyy −= gr_y_coord(p);
  transformed = true;
}
else {
  mp_ps_print_nl(mp, "");
}
⟨ Adjust the transformation to account for gs_width and output the initial gsave if transformed should
    be true 215 ⟩
```

This code is used in section 213.

215.

#define *mp_make_double*(*A*, *B*, *C*) ((*B*)/(*C*))
#define *mp_take_double*(*A*, *B*, *C*) ((*B*) ∗ (*C*))

⟨ Adjust the transformation to account for *gs_width* and output the initial **gsave** if *transformed* should be *true* 215 ⟩ ≡

```
if (gs_width ≠ unity) {
  if (gs_width ≡ 0.0) {
    txx = unity;
    tyy = unity;
  }
  else {
    txx = mp_make_double(mp, txx, gs_width);
    txy = mp_make_double(mp, txy, gs_width);
    tyx = mp_make_double(mp, tyx, gs_width);
    tyy = mp_make_double(mp, tyy, gs_width);
  }
}
if ((txy ≠ 0.0) ∨ (tyx ≠ 0.0) ∨ (txx ≠ unity) ∨ (tyy ≠ unity)) {
  if ((¬transformed)) {
    mp_ps_print_cmd(mp, "gsave␣", "q␣");
    transformed = true;
  }
}
```

This code is used in section 214.

216. ⟨ Issue PostScript commands to transform the coordinate system 216 ⟩ ≡

```
if ((txy ≠ 0.0) ∨ (tyx ≠ 0.0)) {
  mp_ps_print_ln(mp);
  mp_ps_print_char(mp, '[');
  mp_ps_pair_out(mp, txx, tyx);
  mp_ps_pair_out(mp, txy, tyy);
  mp_ps_print(mp, "0␣0]␣concat");
}
else if ((txx ≠ unity) ∨ (tyy ≠ unity)) {
  mp_ps_print_ln(mp);
  mp_ps_pair_out(mp, txx, tyy);
  mp_ps_print(mp, "scale");
}
```

This code is used in section 213.

217. The PostScript interpreter will probably abort if it encounters a singular transformation matrix. The determinant must be large enough to ensure that the printed representation will be nonsingular. Since the printed representation is always within 2^{-17} of the internal *scaled* value, the total error is at most $4T_{\max}2^{-17}$, where $T_{\max}$ is a bound on the magnitudes of $txx/65536$, $txy/65536$, etc.

The $aspect\_bound * (gs\_width + 1)$ bound on the components of the pen transformation allows $T_{\max}$ to be at most $2 * aspect\_bound$.

```
⟨ Tweak the transformation parameters so the transformation is nonsingular 217 ⟩ ≡
  det = mp_take_double(mp, txx, tyy) - mp_take_double(mp, txy, tyx);
  d1 = 4 * (aspect_bound + 1/65536.0);
  if (fabs(det) < d1) {
    if (det ≥ 0) {
      d1 = d1 - det;
      s = 1;
    }
    else {
      d1 = -d1 - det;
      s = -1;
    }
    ;
    d1 = d1 * unity;
    if (fabs(txx) + fabs(tyy) ≥ fabs(txy) + fabs(tyy)) {
      if (fabs(txx) > fabs(tyy)) tyy = tyy + (d1 + s * fabs(txx))/txx;
      else txx = txx + (d1 + s * fabs(tyy))/tyy;
    }
    else {
      if (fabs(txy) > fabs(tyx)) tyx = tyx + (d1 + s * fabs(txy))/txy;
      else txy = txy + (d1 + s * fabs(tyx))/tyx;
    }
  }
```

This code is used in section 213.

218. Here is a simple routine that just fills a cycle.

```
⟨ Declarations 29 ⟩ +≡
  static void mp_gr_ps_fill_out(MP mp, mp_gr_knot p);
```

219.

```
void mp_gr_ps_fill_out(MP mp, mp_gr_knot p)
{
  /* fill cyclic path p */
  mp_gr_ps_path_out(mp, p);
  mp_ps_print_cmd(mp, " fill", " F");
  mp_ps_print_ln(mp);
}
```

220. A text node may specify an arbitrary transformation but the usual case involves only shifting, scaling, and occasionally rotation. The purpose of *choose_scale* is to select a scale factor so that the remaining transformation is as "nice" as possible. The definition of "nice" is somewhat arbitrary but shifting and 90° rotation are especially nice because they work out well for bitmap fonts. The code here selects a scale factor equal to $1/\sqrt{2}$ times the Frobenius norm of the non-shifting part of the transformation matrix. It is careful to avoid additions that might cause undetected overflow.

```
⟨ Declarations 29 ⟩ +≡
  static double mp_gr_choose_scale(MP mp, mp_graphic_object *p);
```

221. **double** *mp_gr_choose_scale*(MP *mp*, **mp_graphic_object** ∗*p*)

```
{   /* p should point to a text node */
  double a, b, c, d, ad, bc;   /* temporary values */
  double r;
  a = gr_txx_val(p);
  b = gr_txy_val(p);
  c = gr_tyx_val(p);
  d = gr_tyy_val(p);
  if (a < 0) negate(a);
  if (b < 0) negate(b);
  if (c < 0) negate(c);
  if (d < 0) negate(d);
  ad = (a − d)/2.0;
  bc = (b − c)/2.0;
  a = (d + ad);
  b = ad;
  d = sqrt(a ∗ a + b ∗ b);
  a = (c + bc);
  b = bc;
  c = sqrt(a ∗ a + b ∗ b);
  r = sqrt(c ∗ c + d ∗ d);
  return r;
}
```

222. The potential overflow here is caused by the fact the returned value has to fit in a *name_type*, which is a quarterword.

```
#define fscale_tolerance (65/65536.0)   /* that's .001 × 2^16 */
⟨ Declarations 29 ⟩ +≡
  static quarterword mp_size_index(MP mp, font_number f, double s);
```

223. *quarterword mp_size_index* (MP *mp*, *font_number f*, **double** *s*)

```
{
  mp_node p, q;  /* the previous and current font size nodes */
  int i;  /* the size index for q */
  p = Λ;
  q = mp→font_sizes[f];
  i = 0;
  while (q ≠ null) {
    if (fabs(s − sc_factor(q)) ≤ fscale_tolerance) return (quarterword)i;
    else {
      p = q;
      q = mp_link(q);
      incr(i);
    }
    ;
    if (i ≡ max_quarterword) mp_overflow(mp, "sizes␣per␣font", max_quarterword);
  }
  q = (mp_node)mp_xmalloc(mp, 1, font_size_size);
  mp_link(q) = Λ;
  sc_factor(q) = s;
  if (i ≡ 0) mp→font_sizes[f] = q;
  else mp_link(p) = q;
  return (quarterword)i;
}
```

224. ⟨ Declarations 29 ⟩ +≡

```
static double mp_indexed_size(MP mp, font_number f, quarterword j);
```

225. **double** *mp_indexed_size* (MP *mp*, *font_number f*, *quarterword j*)

```
{  /* return scaled */
  mp_node p;  /* a font size node */
  int i;  /* the size index for p */
  p = mp→font_sizes[f];
  i = 0;
  if (p ≡ null) mp_confusion(mp, "size");
  while ((i ≠ j)) {
    incr(i);  /* clang: dereference null pointer 'p' */
    assert(p);
    p = mp_link(p);
    if (p ≡ null) mp_confusion(mp, "size");
  }  /* clang: dereference null pointer 'p' */
  assert(p);
  return sc_factor(p);
}
```

226. ⟨ Declarations 29 ⟩ +≡

```
static void mp_clear_sizes(MP mp);
```

227. **void** *mp_clear_sizes*(MP *mp*)
{
font_number f; /* the font whose size list is being cleared */
mp_node p; /* current font size nodes */
for (*f* = *null_font* + 1; *f* ≤ *mp*→*last_fnum*; *f*++) {
while (*mp*→*font_sizes*[*f*] ≠ *null*) {
p = *mp*→*font_sizes*[*f*];
mp→*font_sizes*[*f*] = *mp_link*(*p*);
mp_xfree(*p*);
}
}
}

228. There may be many sizes of one font and we need to keep track of the characters used for each size. This is done by keeping a linked list of sizes for each font with a counter in each text node giving the appropriate position in the size list for its font.

#define *font_size_size* **sizeof**(**struct** *mp_font_size_node_data*) /* size of a font size node */

⟨ Types 18 ⟩ +≡
typedef struct mp_font_size_node_data {
NODE_BODY;
double *sc_factor_*; /* scaled */
} **mp_font_size_node_data**;
typedef struct mp_font_size_node_data ∗**mp_font_size_node**;

229. ⟨ Declarations 29 ⟩ +≡
static void *mp_apply_mark_string_chars*(MP *mp*, **mp_edge_object** ∗*h*, **int** *next_size*);

230. **void** *mp_apply_mark_string_chars*(MP *mp*, **mp_edge_object** ∗*h*, **int** *next_size*)
{
mp_graphic_object ∗*p*;
p = *h*→*body*;
while (*p* ≠ Λ) {
if (*gr_type*(*p*) ≡ *mp_text_code*) {
if (*gr_font_n*(*p*) ≠ *null_font*) {
if (*gr_size_index*(*p*) ≡ (**unsigned char**) *next_size*)
mp_mark_string_chars(*mp*, *gr_font_n*(*p*), *gr_text_p*(*p*), *gr_text_l*(*p*));
}
}
p = *gr_link*(*p*);
}
}

231. ⟨ Unmark all marked characters 231 ⟩ ≡
for (*f* = *null_font* + 1; *f* ≤ *mp*→*last_fnum*; *f*++) {
if (*mp*→*font_sizes*[*f*] ≠ *null*) {
mp_unmark_font(*mp*, *f*);
mp→*font_sizes*[*f*] = *null*;
}
}
This code is used in section 234.

232. ⟨ Scan all the text nodes and mark the used characters 232 ⟩ ≡

```
p = hh→body;
while (p ≠ null) {
  if (gr_type(p) ≡ mp_text_code) {
    f = gr_font_n(p);
    if (f ≠ null_font) {
      switch (prologues) {
      case 2: case 3: mp→font_sizes[f] = mp_void;
        mp_mark_string_chars(mp, f, gr_text_p(p), gr_text_l(p));
        if (mp_has_fm_entry(mp, f, Λ)) {
          if (mp→font_enc_name[f] ≡ Λ) mp→font_enc_name[f] = mp_fm_encoding_name(mp, f);
          mp_xfree(mp→font_ps_name[f]);
          mp→font_ps_name[f] = mp_fm_font_name(mp, f);
        }
        break;
      case 1: mp→font_sizes[f] = mp_void;
        break;
      default: gr_size_index(p) = (unsigned char) mp_size_index(mp, f, mp_gr_choose_scale(mp, p));
        if (gr_size_index(p) ≡ 0) mp_mark_string_chars(mp, f, gr_text_p(p), gr_text_l(p));
      }
    }
  }
  p = gr_link(p);
}
```

This code is used in section 234.

233.

```
#define pen_is_elliptical(A) ((A) ≡ gr_next_knot((A)))
```

⟨ Exported function headers 5 ⟩ +≡

```
int mp_gr_ship_out(mp_edge_object *hh, int prologues, int procset, int standalone);
```

234. **int** *mp_gr_ship_out*(**mp_edge_object** ∗*hh*, **int** *qprologues*, **int** *qprocset*, **int** *standalone*)

```
{
  mp_graphic_object *p;
  double ds, scf;      /* design size and scale factor for a text node */
  font_number f;      /* for loops over fonts while (un)marking characters */
  boolean transformed;      /* is the coordinate system being transformed? */
  int prologues, procset;
  MP mp = hh->parent;
  if (standalone) {
    mp->jump_buf = malloc(sizeof(jmp_buf));
    if (mp->jump_buf ≡ Λ ∨ setjmp(*(mp->jump_buf))) return 0;
  }
  if (mp->history ≥ mp_fatal_error_stop) return 1;
  if (qprologues < 0)
    prologues = (int)((unsigned) number_to_scaled(internal_value(mp_prologues)) ≫ 16);
  else prologues = qprologues;
  if (qprocset < 0) procset = (int)((unsigned) number_to_scaled(internal_value(mp_procset)) ≫ 16);
  else procset = qprocset;
  mp_open_output_file(mp);
  mp_print_initial_comment(mp, hh, prologues);      /* clang: never read: p = hh-¿body; */
  ⟨ Unmark all marked characters 231 ⟩;
  if (prologues ≡ 2 ∨ prologues ≡ 3) {
    mp_reload_encodings(mp);
  }
  ⟨ Scan all the text nodes and mark the used characters 232 ⟩;
  if (prologues ≡ 2 ∨ prologues ≡ 3) {
    mp_print_improved_prologue(mp, hh, prologues, procset);
  }
  else {
    mp_print_prologue(mp, hh, prologues, procset);
  }
  mp_gs_unknown_graphics_state(mp, 0);
  p = hh->body;
  while (p ≠ Λ) {
    if (gr_has_color(p)) {
      ⟨ Write pre_script of p 237 ⟩;
    }
    mp_gr_fix_graphics_state(mp, p);
    switch (gr_type(p)) {
    case mp_fill_code:
      if (gr_pen_p((mp_fill_object *) p) ≡ Λ) {
        mp_gr_ps_fill_out(mp, gr_path_p((mp_fill_object *) p));
      }
      else if (pen_is_elliptical(gr_pen_p((mp_fill_object *) p))) {
        mp_gr_stroke_ellipse(mp, p, true);
      }
      else {
        mp_gr_ps_fill_out(mp, gr_path_p((mp_fill_object *) p));
        mp_gr_ps_fill_out(mp, gr_htap_p(p));
      }
      if (gr_post_script((mp_fill_object *) p) ≠ Λ) {
```

```
    mp_ps_print_nl(mp, gr_post_script((mp_fill_object *) p));
    mp_ps_print_ln(mp);
  }
  break;
case mp_stroked_code:
  if (pen_is_elliptical(gr_pen_p((mp_stroked_object *) p))) mp_gr_stroke_ellipse(mp, p, false);
  else {
    mp_gr_ps_fill_out(mp, gr_path_p((mp_stroked_object *) p));
  }
  if (gr_post_script((mp_stroked_object *) p) ≠ Λ) {
    mp_ps_print_nl(mp, gr_post_script((mp_stroked_object *) p));
    mp_ps_print_ln(mp);
  }
  break;
case mp_text_code:
  if ((gr_font_n(p) ≠ null_font) ∧ (gr_text_l(p) > 0)) {
    if (prologues > 0) scf = mp_gr_choose_scale(mp, p);
    else scf = mp_indexed_size(mp, gr_font_n(p), (quarterword) gr_size_index(p));
    ⟨ Shift or transform as necessary before outputting text node p at scale factor scf; set
        transformed: = true if the original transformation must be restored 239 ⟩;
    mp_ps_string_out(mp, gr_text_p(p), gr_text_l(p));
    mp_ps_name_out(mp, mp→font_name[gr_font_n(p)], false);
    ⟨ Print the size information and PostScript commands for text node p 238 ⟩;
    mp_ps_print_ln(mp);
  }
  if (gr_post_script((mp_text_object *) p) ≠ Λ) {
    mp_ps_print_nl(mp, gr_post_script((mp_text_object *) p));
    mp_ps_print_ln(mp);
  }
  break;
case mp_start_clip_code: mp_ps_print_nl(mp, "");
  mp_ps_print_cmd(mp, "gsave␣", "q␣");
  mp_gr_ps_path_out(mp, gr_path_p((mp_clip_object *) p));
  mp_ps_print_cmd(mp, "␣clip", "␣W");
  mp_ps_print_ln(mp);
  if (number_positive(internal_value(mp_restore_clip_color))) mp_gs_unknown_graphics_state(mp, 1);
  break;
case mp_stop_clip_code: mp_ps_print_nl(mp, "");
  mp_ps_print_cmd(mp, "grestore", "Q");
  mp_ps_print_ln(mp);
  if (number_positive(internal_value(mp_restore_clip_color))) mp_gs_unknown_graphics_state(mp, 2);
  else mp_gs_unknown_graphics_state(mp, −1);
  break;
case mp_start_bounds_code: case mp_stop_bounds_code: break;
case mp_special_code:
  {
    mp_special_object *ps = (mp_special_object *) p;
    mp_ps_print_nl(mp, gr_pre_script(ps));
    mp_ps_print_ln(mp);
  }
  break;
}    /* all cases are enumerated */
```

```
    p = gr_link(p);
  }
  mp_ps_print_cmd(mp, "showpage", "P");
  mp_ps_print_ln(mp);
  mp_ps_print(mp, "%%EOF");
  mp_ps_print_ln(mp);
  (mp→close_file)(mp, mp→output_file);
  if (prologues ≤ 0) mp_clear_sizes(mp);
  return 1;
}
```

235. ⟨ Internal Postscript header information 182 ⟩ +≡

```
int mp_ps_ship_out(mp_edge_object *hh, int prologues, int procset);
```

236.

```
int mp_ps_ship_out(mp_edge_object *hh, int prologues, int procset)
{
  return mp_gr_ship_out(hh, prologues, procset, (int) true);
}
```

237.

```
#define do_write_prescript(a, b)
        { if ( ( gr_pre_script ( ( b * ) a ) ) ≠ Λ ) { mp_ps_print_nl (mp, gr_pre_script ( ( b * ) a ) ) ;
        mp_ps_print_ln(mp); } }
```

⟨ Write *pre_script* of *p* 237 ⟩ ≡

```
{
  if (gr_type(p) ≡ mp_fill_code) {
    do_write_prescript(p, mp_fill_object);
  }
  else if (gr_type(p) ≡ mp_stroked_code) {
    do_write_prescript(p, mp_stroked_object);
  }
  else if (gr_type(p) ≡ mp_text_code) {
    do_write_prescript(p, mp_text_object);
  }
}
```

This code is used in section 234.

238. ⟨ Print the size information and PostScript commands for text node *p* 238 ⟩ ≡

```
ps_room(18);
mp_ps_print_char(mp, '␣');
ds = (mp→font_dsize[gr_font_n(p)] + 8)/16;
mp_ps_print_double(mp, (mp_take_double(mp, ds, scf)/65536.0));
mp_ps_print(mp, "␣fshow"); if (transformed) mp_ps_print_cmd(mp, "␣grestore", "␣Q")
```

This code is used in section 234.

239. ⟨ Shift or transform as necessary before outputting text node p at scale factor scf; set $transformed$:= $true$ if the original transformation must be restored 239 ⟩ ≡

```
transformed = (gr_txx_val(p) ≠ scf) ∨ (gr_tyy_val(p) ≠ scf) ∨ (gr_txy_val(p) ≠ 0) ∨ (gr_tyx_val(p) ≠ 0);
if (transformed) {
  mp_ps_print_cmd(mp, "gsave␣[", "q␣[");
  mp_ps_pair_out(mp, mp_make_double(mp, gr_txx_val(p), scf), mp_make_double(mp, gr_tyx_val(p), scf));
  mp_ps_pair_out(mp, mp_make_double(mp, gr_txy_val(p), scf), mp_make_double(mp, gr_tyy_val(p), scf));
  mp_ps_pair_out(mp, gr_tx_val(p), gr_ty_val(p));
  mp_ps_print_cmd(mp, "]␣concat␣0␣0␣moveto", "]␣t␣0␣0␣m");
}
else {
  mp_ps_pair_out(mp, gr_tx_val(p), gr_ty_val(p));
  mp_ps_print_cmd(mp, "moveto", "m");
}
mp_ps_print_ln(mp)
```

This code is used in section 234.

240. ⟨ Internal Postscript header information 182 ⟩ +≡

```
void mp_gr_toss_objects(mp_edge_object *hh);
void mp_gr_toss_object(mp_graphic_object *p);
```

241. **void** *mp_gr_toss_object*(**mp_graphic_object** ∗*p*)

```
{
  mp_fill_object ∗tf;
  mp_stroked_object ∗ts;
  mp_text_object ∗tt;
  switch (gr_type(p)) {
  case mp_fill_code: tf = (mp_fill_object ∗) p;
    mp_xfree(gr_pre_script(tf));
    mp_xfree(gr_post_script(tf));
    mp_gr_toss_knot_list(mp, gr_pen_p(tf));
    mp_gr_toss_knot_list(mp, gr_path_p(tf));
    mp_gr_toss_knot_list(mp, gr_htap_p(p));
    break;
  case mp_stroked_code: ts = (mp_stroked_object ∗) p;
    mp_xfree(gr_pre_script(ts));
    mp_xfree(gr_post_script(ts));
    mp_gr_toss_knot_list(mp, gr_pen_p(ts));
    mp_gr_toss_knot_list(mp, gr_path_p(ts));
    if (gr_dash_p(p) ≠ Λ) mp_gr_toss_dashes(mp, gr_dash_p(p));
    break;
  case mp_text_code: tt = (mp_text_object ∗) p;
    mp_xfree(gr_pre_script(tt));
    mp_xfree(gr_post_script(tt));
    mp_xfree(gr_text_p(p));
    mp_xfree(gr_font_name(p));
    break;
  case mp_start_clip_code: mp_gr_toss_knot_list(mp, gr_path_p((mp_clip_object ∗) p));
    break;
  case mp_start_bounds_code: mp_gr_toss_knot_list(mp, gr_path_p((mp_bounds_object ∗) p));
    break;
  case mp_stop_clip_code: case mp_stop_bounds_code: break;
  case mp_special_code: mp_xfree(gr_pre_script((mp_special_object ∗) p));
    break;
  }     /∗ all cases are enumerated ∗/
  mp_xfree(p);
}
```

242. **void** *mp_gr_toss_objects*(**mp_edge_object** ∗*hh*)

```
{
  mp_graphic_object ∗p, ∗q;
  p = hh→body;
  while (p ≠ Λ) {
    q = gr_link(p);
    mp_gr_toss_object(p);
    p = q;
  }
  mp_xfree(hh→filename);
  mp_xfree(hh);
}
```

243. ⟨ Internal Postscript header information 182 ⟩ +≡
mp_graphic_object ∗*mp_gr_copy_object*(MP *mp*, **mp_graphic_object** ∗*p*);

244. **mp_graphic_object** *mp_gr_copy_object(MP mp, mp_graphic_object *p)*

```
{
  mp_fill_object *tf;
  mp_stroked_object *ts;
  mp_text_object *tt;
  mp_clip_object *tc;
  mp_bounds_object *tb;
  mp_special_object *tp;
  mp_graphic_object *q = Λ;
  switch (gr_type(p)) {
  case mp_fill_code: tf = (mp_fill_object *) mp_new_graphic_object(mp, mp_fill_code);
    gr_pre_script(tf) = mp_xstrdup(mp, gr_pre_script((mp_fill_object *) p));
    gr_post_script(tf) = mp_xstrdup(mp, gr_post_script((mp_fill_object *) p));
    gr_path_p(tf) = mp_gr_copy_path(mp, gr_path_p((mp_fill_object *) p));
    gr_htap_p(tf) = mp_gr_copy_path(mp, gr_htap_p(p));
    gr_pen_p(tf) = mp_gr_copy_path(mp, gr_pen_p((mp_fill_object *) p));
    q = (mp_graphic_object *) tf;
    break;
  case mp_stroked_code: ts = (mp_stroked_object *) mp_new_graphic_object(mp, mp_stroked_code);
    gr_pre_script(ts) = mp_xstrdup(mp, gr_pre_script((mp_stroked_object *) p));
    gr_post_script(ts) = mp_xstrdup(mp, gr_post_script((mp_stroked_object *) p));
    gr_path_p(ts) = mp_gr_copy_path(mp, gr_path_p((mp_stroked_object *) p));
    gr_pen_p(ts) = mp_gr_copy_path(mp, gr_pen_p((mp_stroked_object *) p));
    gr_dash_p(ts) = mp_gr_copy_dashes(mp, gr_dash_p(p));
    q = (mp_graphic_object *) ts;
    break;
  case mp_text_code: tt = (mp_text_object *) mp_new_graphic_object(mp, mp_text_code);
    gr_pre_script(tt) = mp_xstrdup(mp, gr_pre_script((mp_text_object *) p));
    gr_post_script(tt) = mp_xstrdup(mp, gr_post_script((mp_text_object *) p));
    gr_text_p(tt) = mp_xstrldup(mp, gr_text_p(p), gr_text_l(p));
    gr_text_l(tt) = gr_text_l(p);
    gr_font_name(tt) = mp_xstrdup(mp, gr_font_name(p));
    q = (mp_graphic_object *) tt;
    break;
  case mp_start_clip_code: tc = (mp_clip_object *) mp_new_graphic_object(mp, mp_start_clip_code);
    gr_path_p(tc) = mp_gr_copy_path(mp, gr_path_p((mp_clip_object *) p));
    q = (mp_graphic_object *) tc;
    break;
  case mp_start_bounds_code:
    tb = (mp_bounds_object *) mp_new_graphic_object(mp, mp_start_bounds_code);
    gr_path_p(tb) = mp_gr_copy_path(mp, gr_path_p((mp_bounds_object *) p));
    q = (mp_graphic_object *) tb;
    break;
  case mp_special_code: tp = (mp_special_object *) mp_new_graphic_object(mp, mp_special_code);
    gr_pre_script(tp) = mp_xstrdup(mp, gr_pre_script((mp_special_object *) p));
    q = (mp_graphic_object *) tp;
    break;
  case mp_stop_clip_code: q = mp_new_graphic_object(mp, mp_stop_clip_code);
    break;
  case mp_stop_bounds_code: q = mp_new_graphic_object(mp, mp_stop_bounds_code);
    break;
  }     /* all cases are enumerated */
```

```
    return q;
  }
```


__FILE__: 17.
__LINE__: 17.
_array: 90.
_entry: 90.
_gs_state: 191, 192, 195.
_limit: 90.
_ltype: 36.
_mode: 36.
_ptr: 90.
_tfmavail: 36.
a: 4, 50, 52, 93, 202, 221.
a_val: 182, 187, 200.
abs: 1, 15, 50.
abyte: 20.
achar: 113.
ad: 221.
add_curve_segment: 111, 112, 113.
add_line_segment: 111, 112, 113.
adj_wx: 197, 201, 203.
adj_wx_field: 191.
adx: 113.
ady: 113.
all_glyphs: 40, 115.
alloc_array: 71, 74, 90, 93, 98.
append_char_to_buf: 17, 21, 52, 90.
append_eol: 17, 21, 90.
applied_reencoding: 132, 134.
args: 98.
array: 183, 185, 186, 208, 210, 211.
asb: 113.
ASCENT_CODE: 93.
ASCII_code: 166, 168.
aspect_bound: 203, 217.
assert: 27, 46, 48, 49, 50, 52, 53, 56, 57, 58, 94, 111, 225.
atoi: 40, 52.
avail_tfm_found: 37.
avl_create: 27, 48.
avl_del: 49.
avl_destroy: 30, 64.
avl_do_entry: 49, 52.
avl_false: 27, 49, 57.
avl_find: 27, 49, 56, 57.
avl_ins: 27, 49, 57.
avl_tree: 23, 43.
a1: 98, 113.
a2: 98, 113.
b: 52, 90, 93, 98, 113, 151, 202, 221.
b_val: 182, 187, 200.
bad_line: 52.
basefont_names: 51.
bc: 140, 151, 152, 153, 154, 221.
bchar: 113.
begin_buf: 89.
bend_tolerance: 181.
black_field: 191.
blue_field: 191.
body: 111, 113, 188, 230, 232, 234, 242.
boolean: 4, 18, 21, 31, 32, 41, 42, 50, 51, 74, 79, 81, 87, 89, 90, 91, 96, 97, 99, 107, 110, 113, 115, 116, 117, 118, 123, 124, 126, 128, 130, 144, 146, 155, 156, 168, 169, 170, 179, 191, 204, 205, 209, 210, 212, 213, 234.
bottom: 81, 97, 98, 113.
buf: 17, 21, 40, 52, 74, 98.
buf_size: 17.
byte: 81, 89, 90, 97, 98, 99, 113.
Byte: 68, 74.
byte_ptr: 20, 35, 77.
Bytef: 68, 74.
b3: 74, 138, 140, 142, 151, 152, 154, 156.
c: 4, 16, 21, 52, 89, 90, 93, 108, 109, 195, 221.
c_val: 182, 187, 200.
CAPHEIGHT_CODE: 93.
cc: 98, 113.
cc_clear: 97, 98, 113.
cc_entry: 81, 97, 98, 113.
cc_get: 97, 98, 113.
cc_init: 97, 99, 103.
cc_pop: 97, 98, 113.
cc_push: 97, 98, 113.
cc_stack: 97, 98, 113.
CC_STACK_SIZE: 97.
cc_tab: 97, 98, 113.
cdecrypt: 89, 98.
cencrypt: 89, 99.
char_array: 69, 70, 74, 101.
char_base: 74, 140, 142, 152, 154, 156.
char_entry: 68, 69.
char_info: 138.
char_limit: 69.
char_ptr: 69, 71, 74.
charcode: 188.
charset: 40, 74, 99, 115.
charsetstr: 79.
charstringname: 78, 89, 99.
check_basefont: 51, 52.
check_buf: 17, 40.
check_cs_token_pair: 97.
check_ff_exist: 57, 96.

check_fm_entry: 50, 52.
check_subr: 97, 98.
choose_scale: 220.
cipher: 89.
clear: 81, 97, 98, 113.
close_file: 20, 35, 77, 234.
close_name_suffix: 90.
CLOSEPATH: 113.
cmp_return: 40, 46.
color: 187, 188, 200.
color_model: 187, 188, 200.
colormodel_field: 191.
comp_enc_entry: 26, 27.
comp_ff_entry: 47, 48.
comp_fm_entry_ps: 46, 48.
comp_fm_entry_tfm: 45, 48.
copy_enc_entry: 27.
copy_ff_entry: 40, 48.
copy_fm_entry: 40, 48.
copy_glyph_names: 93.
copy_knot: 175.
copy_path: 176.
count: 99.
counter: 93.
cr: 89, 98, 99, 113.
crc: 74.
crc32: 74.
create_avl_trees: 48, 53.
cs: 98.
CS_CALLOTHERSUBR: 80, 97, 98, 113.
CS_CALLSUBR: 80, 97, 98, 113.
CS_CLOSEPATH: 80, 97, 113.
cs_count: 84, 98, 99.
cs_debug: 112, 113.
cs_dict_end: 84, 98, 99, 101.
cs_dict_start: 84, 98, 99, 101.
CS_DIV: 80, 97, 98, 113.
cs_do_debug: 112, 113.
CS_DOTSECTION: 80, 97, 113.
CS_ENDCHAR: 80, 97, 113.
cs_entry: 81, 84, 97, 98, 99, 102, 105, 113.
cs_error: 97, 98, 113.
CS_ESCAPE: 80, 97, 98, 113.
cs_getchar: 98, 113.
CS_HLINETO: 80, 97, 113.
CS_HMOVETO: 80, 97, 113.
CS_HSBW: 80, 97, 113.
CS_HSTEM: 80, 97, 113.
CS_HSTEM3: 80, 97, 113.
CS_HVCURVETO: 80, 97, 113.
cs_init: 98, 99, 101, 103.
cs_len: 98, 99, 113.
cs_mark: 98.
CS_MAX: 80, 97, 98, 113.
cs_name: 98, 110, 113.
cs_no_debug: 112.
cs_notdef: 84, 98, 99.
cs_parse: 108, 110, 113.
CS_POP: 80, 97, 98, 113.
cs_ptr: 84, 97, 98, 99, 102, 103, 105, 113.
CS_RETURN: 80, 97, 99, 113.
CS_RLINETO: 80, 97, 113.
CS_RMOVETO: 80, 97, 113.
CS_RRCURVETO: 80, 97, 113.
CS_SBW: 80, 97, 113.
CS_SEAC: 80, 97, 98, 113.
CS_SETCURRENTPOINT: 80, 97, 113.
cs_size: 84, 97, 98, 99.
cs_size_pos: 84, 98, 99.
cs_start: 84, 90, 97.
cs_store: 97.
cs_tab: 84, 97, 98, 99, 102, 103, 105, 113.
cs_token_pair: 87, 97, 98, 99.
cs_token_pairs_list: 86, 97.
CS_VHCURVETO: 80, 97, 113.
CS_VLINETO: 80, 97, 113.
CS_VMOVETO: 80, 97, 113.
CS_VSTEM: 80, 97, 113.
CS_VSTEM3: 80, 97, 113.
cs_warn: 98, 113.
CS_1BYTE_MAX: 80, 98, 113.
CS_2BYTE_MAX: 80.
cslen: 81, 97, 98, 99, 113.
cur_enc_name: 75.
cur_file_name: 90.
cur_fsize: 136, 144, 146, 147, 157.
cur_x: 107, 108, 111, 113.
cur_y: 107, 108, 111, 113.
curved: 179, 180, 181.
d: 52, 151, 179, 221.
d_val: 182, 187, 200.
dash_done_field: 191.
dash_p: 187, 188.
dash_p_field: 191.
data: 81, 97, 98, 99, 105, 111, 113, 174.
dds: 157.
decr: 1.
decrypt: 90.
delete_edge_ref: 207.
delete_ff_entry: 40, 48, 57.
delete_fm_entry: 40, 48, 52.
delta: 16.
den: 113.
depth: 187, 188.

DESCENT_CODE: 93.
destroy_enc_entry: 26, 27.
det: 213, 217.
dig: 14.
div: 14.
dl: 184, 185, 186.
do_strdup: 40.
do_write_prescript: 237.
do_x_loc: 203, 205.
do_y_loc: 203.
DONE: 21, 40, 52.
done_fonts: 136, 144, 146.
DOTSECTION: 113.
ds: 146, 157, 234, 238.
dsc: 12.
dvips_extra_charset: 75, 76.
dx: 111, 112, 113.
dx0: 113.
dx1: 111, 112, 113.
dx2: 111, 112, 113.
dx3: 111, 112, 113.
dy: 111, 112, 113.
dy0: 113.
dy1: 111, 112, 113.
dy2: 111, 112, 113.
dy3: 111, 112, 113.
dz: 204, 205, 206.
d1: 213, 217.
e: 21, 22, 32, 56, 120.
ec: 140, 151, 152, 154.
edecrypt: 89, 90.
eencrypt: 89, 90.
eexec_len: 90.
eexec_scan: 90.
eexec_str: 90.
eid: 40, 52, 115.
eight_bits: 74, 75, 151.
embed_all_glyphs: 75, 99.
emergency_line_length: 157.
ENC_BUF_SIZE: 19, 21.
ENC_BUILTIN: 19, 93.
enc_close: 20, 21.
enc_encname: 21.
enc_entry: 18, 21, 22, 26, 27, 32, 115, 120.
enc_eof: 20, 21.
enc_file: 19, 20, 21.
enc_free: 6, 29, 30.
enc_getchar: 20, 21.
enc_line: 19, 21.
enc_name: 18, 21, 22, 26, 27, 120.
ENC_STANDARD: 19, 93, 98.
enc_tree: 23, 25, 27, 30.
encoding: 32, 40, 52, 74, 75, 93, 103, 115, 117, 120.
encodings_only: 31, 32.
end_buf: 89.
end_hexline: 75, 89.
end_last_eexec_line: 75, 90.
end_tab: 99.
ENDCHAR: 113.
endloc: 98.
EOF: 17, 35, 89, 90.
eof_file: 20.
eol: 89, 90, 93, 99.
equal: 1.
err: 21, 96, 108.
exit: 49.
EXIT: 90.
ext_glyph_names: 75.
extend: 40, 46, 50, 52, 102, 103, 115, 117, 122.
external_enc: 75, 98, 103.
extra_charset: 75, 99.
f: 35, 89, 103, 105, 106, 108, 109, 110, 111, 112, 113.
F_BASEFONT: 117.
F_INCLUDED: 117.
F_SUBSETTED: 117.
F_TRUETYPE: 117.
fabs: 181, 202, 217, 223.
false: 1, 21, 27, 40, 51, 74, 75, 89, 90, 92, 96, 97, 98, 99, 113, 118, 124, 126, 128, 130, 136, 146, 156, 168, 181, 195, 206, 207, 210, 213, 234.
fd_objnum: 40, 115.
ff: 35, 40, 57, 96, 126, 128, 130, 146, 148.
ff_entry: 40, 47, 57, 95, 96.
ff_name: 40, 47, 52, 57, 95, 96, 103, 115, 117, 118, 124.
ff_objnum: 40, 115.
ff_path: 40, 57, 95, 96.
ff_tree: 43, 44, 47, 48, 57, 64.
File: 35.
file_name: 18, 21, 22, 26, 27, 74.
filename: 113, 188, 242.
fill command: 207.
fill_also: 212, 213.
find_file: 61.
finish_subpath: 108, 111, 112, 113.
first_r_x: 113.
first_r_y: 113.
first_x: 113.
first_y: 113.
firstitem: 126, 128, 130.
fix_graphics_state: 196.
fixedcontent: 75, 96, 99.
flags: 40, 52, 115.

flex_hint_data: 107, 113.
flex_hint_index: 107, 108, 113.
floor: 172.
fm: 32, 40, 41, 42, 49, 50, 52, 54, 56, 57, 99, 117, 118, 120, 122.
FM_BUF_SIZE: 33, 52.
fm_byte_length: 33, 34, 35.
fm_byte_waiting: 33, 34, 35.
fm_bytes: 33, 34, 35.
fm_close: 35, 53.
fm_cur: 32, 37, 74, 75, 90, 93, 96, 98, 99, 103, 124.
FM_DELETE: 36, 49, 58.
FM_DUPIGNORE: 36, 49, 58, 61, 66.
fm_entry: 32, 37, 40, 41, 42, 45, 46, 49, 50, 52, 54, 55, 56, 57, 74, 90, 93, 96, 98, 99, 103, 115, 118, 120, 122, 124.
fm_eof: 34, 35, 53.
fm_extend: 93, 117.
fm_file: 33, 35, 53.
fm_fontfile: 52, 117.
fm_free: 6, 63, 64.
fm_getchar: 34, 35, 52.
fm_line: 52.
fm_read_info: 53, 58, 66.
FM_REPLACE: 36, 49, 58.
fm_scan_line: 52, 53.
fm_slant: 93, 117.
fmt: 98.
fn_objnum: 40, 115.
fnstr_append: 71, 74.
foffset: 22.
font_bc: 74, 140, 142, 150, 151.
font_dsize: 157, 187, 188, 238.
font_ec: 74, 140, 142, 151, 156.
font_enc_name: 32, 126, 128, 132, 134, 232.
font_info: 74, 140, 142, 151, 152, 154, 156.
font_keys: 92, 93.
FONT_KEYS_NUM: 92.
font_max: 144, 146.
font_n: 187, 188.
font_name: 56, 103, 120, 126, 128, 130, 132, 134, 144, 157, 159, 160, 187, 188, 234.
font_num: 102, 103.
font_number: 31, 32, 41, 42, 54, 55, 56, 66, 74, 93, 96, 98, 99, 103, 115, 117, 118, 119, 120, 121, 122, 123, 124, 126, 128, 130, 131, 132, 133, 134, 139, 140, 141, 142, 144, 145, 146, 150, 151, 155, 156, 160, 222, 223, 224, 225, 227, 234.
font_ps_name: 65, 66, 126, 128, 130, 132, 134, 148, 159, 160, 232.
font_ps_name_fixed: 120.
font_size_size: 223, 228.
font_sizes: 144, 146, 147, 148, 223, 225, 227, 231, 232.
FONTBBOX1_CODE: 93.
FONTBBOX2_CODE: 93.
FONTBBOX3_CODE: 93.
FONTBBOX4_CODE: 93.
fontfile_found: 96, 116.
fontname_buf: 93, 99, 116.
FONTNAME_BUF_SIZE: 93, 116.
FONTNAME_CODE: 93.
FOUND: 98, 126, 128, 130, 148.
FOUND2: 126, 128, 130.
fp: 40, 49.
fprintf: 113.
free: 4, 16, 27, 48.
fscale_tolerance: 222, 223.
fseek: 35, 77.
ftell: 35, 77.
g: 22, 99.
glyph_name: 81, 97, 98, 99, 105, 113.
glyph_names: 18, 21, 22, 26, 27, 74, 75, 93.
gr_black_val: 187.
gr_blue_val: 187.
gr_color_model: 187.
gr_cyan_val: 187.
gr_dash_p: 187, 198, 207, 208, 241, 244.
gr_depth_val: 187.
gr_font_dsize: 187.
gr_font_n: 187, 230, 232, 234, 238.
gr_font_name: 187, 241, 244.
gr_green_val: 187.
gr_grey_val: 187.
gr_has_color: 187, 197, 234.
gr_height_val: 187.
gr_htap_p: 187, 234, 241, 244.
gr_lcap_val: 187, 198.
gr_left_type: 174, 198.
gr_left_x: 174, 180, 181, 202, 205, 214.
gr_left_y: 174, 180, 181, 202, 205, 214.
gr_link: 111, 187, 230, 232, 234, 242.
gr_ljoin_val: 187, 199.
gr_magenta_val: 187.
gr_miterlim_val: 187, 199.
gr_next_knot: 174, 175, 176, 178, 179, 205, 233.
gr_originator: 174.
gr_path_p: 111, 187, 197, 198, 213, 234, 241, 244.
gr_pen_p: 187, 197, 214, 234, 241, 244.
gr_post_script: 187, 234, 241, 244.
gr_pre_script: 187, 234, 237, 241, 244.
gr_red_val: 187.
gr_right_type: 174, 179, 205.
gr_right_x: 174, 180, 181, 202, 205, 214.

gr_right_y: 174, 180, 181, 202, 205, 214.
gr_size_index: 187, 230, 232, 234.
gr_text_l: 187, 230, 232, 234, 244.
gr_text_p: 187, 230, 232, 234, 241, 244.
gr_tx_val: 187, 239.
gr_txx_val: 187, 221, 239.
gr_txy_val: 187, 221, 239.
gr_ty_val: 187, 239.
gr_type: 187, 190, 197, 198, 199, 200, 207, 213, 214, 230, 232, 234, 237, 241, 244.
gr_tyx_val: 187, 221, 239.
gr_tyy_val: 187, 221, 239.
gr_width_val: 187.
gr_x_coord: 174, 179, 180, 181, 202, 205, 214.
gr_y_coord: 174, 179, 180, 181, 202, 205, 214.
gr_yellow_val: 187.
GRAPHIC_BODY: 188.
graphics state: 191.
greater: 1.
green_field: 191.
grid: 75.
gs_adj_wx: 191, 201.
gs_black: 191, 195, 200.
gs_blue: 191, 195, 200.
gs_colormodel: 191, 195, 200.
gs_dash_init_done: 191, 195, 207.
gs_dash_p: 191, 195, 207, 208.
gs_green: 191, 195, 200.
gs_lcap: 191, 195, 198.
gs_ljoin: 191, 195, 199.
gs_miterlim: 191, 195, 199.
gs_previous: 191, 195.
gs_red: 191, 195, 200.
gs_state: 191, 192, 193, 194, 195.
gs_width: 191, 195, 201, 215, 217.
h: 209, 210, 212, 213, 229, 230.
halfword: 150, 151.
handling: 49.
has_pslink: 49.
has_tfmlink: 49.
height: 187, 188.
hexdigits: 90.
hexline_length: 75, 87, 88, 89, 90.
HEXLINE_WIDTH: 75, 88, 89, 90.
hexval: 89.
hh: 171, 172, 197, 207, 208, 209, 210, 211, 232, 233, 234, 235, 236, 240, 242.
history: 234.
HLINETO: 113.
HMOVETO: 113.
HSBW: 113.
HSTEM: 113.
HSTEM3: 113.
htap_p: 187, 188.
HVCURVETO: 113.
i: 16, 22, 26, 27, 40, 46, 74, 90, 93, 94, 97, 98, 99, 112, 113, 186, 208, 210, 223, 225.
ignore_flex_hint: 107, 108, 113.
incr: 1, 10, 11, 14, 223, 225.
Index: 51.
init_cs_entry: 98, 99.
init_fm: 54, 56.
integer: 7, 14, 15, 18, 74, 98, 113, 115, 121, 122, 140.
internal: 65.
internal_value: 71, 164, 172, 201, 213, 234.
is_basefont: 50, 117.
is_cc_init: 97.
is_cfg_comment: 52.
is_fontfile: 50, 52, 117.
is_included: 50, 90, 93, 99, 103, 117, 118, 124.
is_otf_font: 116.
is_reencoded: 32, 50, 98, 103, 117, 118, 120.
is_subr: 97, 99.
is_subsetted: 32, 50, 93, 98, 99, 117, 118, 120, 124.
is_troff: 59, 61.
is_truetype: 50, 103, 117, 124.
is_t1fontfile: 117.
is_used: 81, 98, 99, 113.
is_used_char: 75, 98, 99.
isread: 66.
ital_corr: 188.
ITALIC_ANGLE_CODE: 93.
j: 11, 52, 98.
job_id_string: 69, 70, 71, 73, 74.
job_name: 71.
join_l_x: 113.
join_l_y: 113.
join_r_x: 113.
join_r_y: 113.
join_x: 113.
join_y: 113.
jump_buf: 234.
k: 14, 51, 93, 101, 140, 142.
key: 93.
key_entry: 91, 92, 93.
kpse_find_file: 57.
l: 14, 74, 90, 139, 140, 164, 165, 166, 167.
last_fnum: 32, 65, 66, 126, 128, 130, 135, 136, 144, 146, 147, 148, 160, 227, 231.
last_hexbyte: 87, 89, 90.
last_l_x: 113.
last_l_y: 113.
last_ps_fnum: 65, 66, 160.

last_ptr_index: 90.
last_x: 113.
last_y: 113.
lastargOtherSubr3: 98, 113.
lastfnum: 31, 32.
lcap: 187, 188.
lcap_field: 191.
ldf: 126, 128, 130, 146, 148, 160.
left_type: 111, 174.
left_x: 111, 174.
left_y: 111, 174.
len: 11, 20, 74, 81, 97, 98, 99.
lenIV: 93.
line_end: 99.
lineno: 36, 53.
link: 157.
LINK_PS: 49.
LINK_TFM: 49.
links: 40, 49, 115.
lit: 169, 170.
ljoin: 187, 188.
ljoin_field: 191.
loaded: 18, 21, 27.
loaded_tfm_found: 37.
make_choices: 181.
make_envelope: 181.
make_path: 181.
make_subset_tag: 74, 93.
malloc: 27, 40, 48, 234.
map_line: 36, 52, 53, 58, 61, 62, 66.
MAPFILE: 36, 52, 53, 58, 60, 61, 66.
mapitem: 36, 37, 61, 66.
MAPLINE: 36, 52, 53, 58, 60.
mark_cs: 98, 99.
mark_subr: 98.
math: 1.
math_data: 1.
math_mode: 16.
MAX_KEY_CODE: 93.
max_print_line: 12, 126, 128, 130, 148, 166.
max_quarterword: 1, 223.
maxx: 113, 172, 188.
maxy: 113, 172, 188.
memcpy: 21, 40, 97, 99, 175, 186, 195.
memset: 6, 16, 27, 190.
  sizes per font: 223.
minx: 113, 172, 188.
miny: 113, 172, 188.
mitem: 36, 37, 38, 52, 53, 58, 61, 62, 66.
miterlim: 187, 188.
miterlim_field: 191.
mk_base_tfm: 40, 56.
mod: 14.
mode: 36, 49, 52, 58, 61, 66.
mp: 1, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 20, 21, 22, 25, 27, 29, 30, 31, 32, 34, 35, 38, 40, 41, 42, 44, 48, 49, 50, 52, 53, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 70, 71, 72, 73, 74, 75, 76, 77, 85, 88, 89, 90, 93, 94, 96, 97, 98, 99, 100, 101, 103, 104, 105, 106, 108, 109, 110, 111, 112, 113, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 175, 176, 179, 180, 186, 189, 190, 191, 193, 194, 195, 196, 197, 198, 199, 200, 201, 207, 208, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 229, 230, 231, 232, 234, 237, 238, 239, 241, 243, 244.
MP: 5, 6, 9, 10, 11, 12, 13, 14, 15, 16, 20, 21, 22, 27, 29, 30, 31, 32, 35, 40, 41, 42, 48, 49, 50, 52, 53, 55, 56, 57, 58, 59, 60, 61, 63, 64, 65, 66, 71, 72, 74, 77, 89, 90, 93, 96, 97, 98, 99, 100, 101, 103, 104, 105, 106, 108, 109, 110, 111, 112, 113, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 139, 140, 141, 142, 143, 144, 145, 146, 149, 150, 151, 155, 156, 159, 160, 162, 163, 164, 165, 166, 167, 169, 170, 171, 172, 175, 176, 179, 186, 188, 189, 190, 195, 196, 197, 212, 213, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 229, 230, 234, 243, 244.
mp_add_enc: 27, 52.
mp_apply_mark_string_chars: 137, 229, 230.
mp_bounds_object: 188, 190, 241, 244.
mp_char_mark_state: 138.
mp_char_marked: 74, 75.
mp_check_ps_marks: 155, 156, 157.
mp_clear_sizes: 226, 227, 234.
mp_clip_object: 188, 190, 234, 241, 244.
mp_cmyk_model: 200.
mp_color: 182, 188.
mp_confusion: 225.
mp_dash_object: 183, 184, 185, 186, 188, 191, 197, 209, 210.
mp_day: 71, 172.
mp_do_gr_toss_dashes: 184, 185.
mp_do_gr_toss_knot_list: 177, 178.
mp_do_is_ps_name: 168.
mp_do_ps_font: 123, 124, 136.
mp_edge_object: 107, 108, 109, 113, 143, 144, 145, 146, 159, 160, 171, 172, 188, 229, 230, 233, 234, 235, 236, 240, 242.

mp_enc_getline: 21.
mp_enc_open: 21.
mp_endpoint: 179, 198, 205.
mp_error: 21, 120, 136.
mp_explicit: 111.
mp_fatal_error: 17, 74, 89, 90, 93, 97, 98, 99.
mp_fatal_error_stop: 234.
mp_filetype_encoding: 21.
mp_filetype_font: 96.
mp_filetype_fontmap: 53, 61.
mp_fill_code: 111, 190, 197, 200, 207, 213, 214, 234, 237, 241, 244.
mp_fill_object: 111, 187, 188, 190, 197, 199, 200, 213, 214, 234, 237, 241, 244.
mp_fm_encoding_name: 119, 120, 232.
mp_fm_font_extend: 121, 122, 132, 134.
mp_fm_font_name: 66, 119, 120, 232.
mp_fm_font_slant: 121, 122, 132, 134.
mp_fm_font_subset_name: 119, 120, 126, 128, 132, 134.
mp_fm_lookup: 42, 55, 56.
mp_font_encodings: 31, 32, 135.
mp_font_is_included: 117, 118, 120, 128, 130, 132, 134.
mp_font_is_reencoded: 117, 118, 126, 128, 132.
mp_font_is_subsetted: 117, 118, 126, 128, 132, 134.
mp_font_size_node: 157, 228.
mp_font_size_node_data: 228.
mp_gr_choose_scale: 220, 221, 232, 234.
mp_gr_coord_rangeOK: 203, 204, 205.
mp_gr_copy_dashes: 186, 244.
mp_gr_copy_knot: 175, 176.
mp_gr_copy_object: 243, 244.
mp_gr_copy_path: 176, 244.
mp_gr_fix_graphics_state: 196, 197, 234.
mp_gr_knot: 107, 111, 175, 176, 177, 178, 179, 188, 197, 204, 205, 213, 218, 219.
mp_gr_knot_data: 111, 175.
mp_gr_ps_fill_out: 218, 219, 234.
mp_gr_ps_path_out: 179, 213, 219, 234.
mp_gr_same_dashes: 207, 209, 210.
mp_gr_ship_out: 233, 234, 236.
mp_gr_stroke_ellipse: 212, 213, 234.
mp_gr_toss_dashes: 184, 241.
mp_gr_toss_knot_list: 178, 241.
mp_gr_toss_object: 240, 241, 242.
mp_gr_toss_objects: 108, 240, 242.
mp_graphic_object: 107, 111, 188, 189, 190, 196, 197, 212, 213, 220, 221, 230, 234, 240, 241, 242, 243, 244.
mp_grey_model: 200.
mp_gs_unknown_graphics_state: 195, 234.
mp_has_fm_entry: 32, 41, 42, 66, 103, 118, 120, 122, 124, 136, 144, 232.
mp_has_font_size: 32, 49, 118, 126, 128, 130, 144, 160.
mp_hex_digit_out: 149, 153, 154.
mp_indexed_size: 224, 225, 234.
mp_init_map_file: 59, 61.
mp_is_ps_name: 168, 170.
mp_isdigit: 4, 40, 52, 89, 99.
mp_link: 136, 157, 223, 225, 227.
mp_list_needed_resources: 129, 130, 144.
mp_list_supplied_resources: 127, 128, 144.
mp_list_used_resources: 125, 126, 144.
mp_load_enc: 21.
mp_make_double: 215, 239.
mp_map_file: 59, 60.
mp_map_line: 59, 60.
mp_mark_string_chars: 139, 140, 230, 232.
mp_math_scaled_mode: 16.
mp_metapost_version: 172.
mp_month: 71, 172.
mp_new_graphic_object: 111, 189, 190, 244.
mp_no_model: 200.
mp_node: 144, 146, 195, 223, 225, 227.
mp_normalize_selector: 21, 53, 99.
mp_open_output_file: 234.
mp_overflow: 223.
mp_print: 21, 53, 75.
mp_print_font_comments: 145, 146, 160.
mp_print_improved_prologue: 143, 144, 234.
mp_print_initial_comment: 171, 172, 234.
mp_print_prologue: 159, 160, 234.
mp_process_map_item: 58, 60.
mp_procset: 164, 201, 213, 234.
mp_prologues: 171, 234.
mp_ps_backend_free: 5, 6.
mp_ps_backend_initialize: 5, 6.
mp_ps_do_font_charstring: 108, 109.
mp_ps_do_print: 11, 12.
mp_ps_dsc_print: 12, 126, 128, 130.
mp_ps_font: 102, 103, 104, 105, 106, 108, 109, 110, 111, 112, 113.
mp_ps_font_charstring: 108, 109.
mp_ps_font_free: 105, 106.
mp_ps_font_parse: 103, 104.
mp_ps_marks_out: 150, 151, 157.
mp_ps_name_out: 132, 144, 160, 169, 170, 234.
mp_ps_pair_out: 162, 163, 172, 179, 180, 213, 214, 216, 239.
mp_ps_print: 12, 13, 22, 124, 130, 132, 134, 144, 148, 157, 160, 164, 166, 170, 172, 201, 208, 213, 214, 216, 234, 238.

mp_ps_print_char: 10, 11, 12, 14, 15, 16, 22, 124, 126, 128, 130, 148, 149, 153, 157, 162, 166, 170, 172, 198, 199, 200, 201, 208, 216, 238.
mp_ps_print_cmd: 164, 165, 179, 180, 198, 199, 200, 201, 207, 208, 214, 215, 219, 234, 238, 239.
mp_ps_print_dd: 15, 172.
mp_ps_print_defined_name: 132, 133, 134, 144.
mp_ps_print_double: 16, 157, 162, 199, 200, 201, 208, 238.
mp_ps_print_double_new: 16.
mp_ps_print_double_scaled: 16.
mp_ps_print_int: 14, 16, 132, 134, 172.
mp_ps_print_ln: 9, 12, 13, 22, 124, 126, 128, 130, 132, 144, 160, 166, 180, 197, 213, 216, 219, 234, 237, 239.
mp_ps_print_nl: 13, 22, 124, 126, 128, 130, 144, 148, 157, 158, 160, 172, 213, 214, 234, 237.
mp_ps_ship_out: 235, 236.
mp_ps_string_out: 166, 167, 170, 234.
mp_read_enc: 21, 32, 103.
mp_read_psname_table: 56, 65, 66, 160.
mp_reload_encodings: 31, 32, 234.
mp_restore_clip_color: 234.
mp_rgb_model: 200.
mp_set_job_id: 71, 72.
mp_size_index: 222, 223, 232.
mp_snprintf: 16, 17, 21, 49, 50, 52, 53, 89, 90, 93, 96, 97, 98, 99, 103, 108, 120, 144.
mp_special_code: 190, 234, 241, 244.
mp_special_object: 188, 190, 234, 241, 244.
mp_start_bounds_code: 190, 234, 241, 244.
mp_start_clip_code: 187, 190, 234, 241, 244.
mp_stop_bounds_code: 234, 241, 244.
mp_stop_clip_code: 234, 241, 244.
mp_str: 60.
mp_strcasecmp: 4, 52.
mp_strdup: 4.
mp_string: 59, 60.
mp_stroked_code: 190, 197, 198, 199, 200, 234, 237, 241, 244.
mp_stroked_object: 187, 188, 190, 197, 198, 199, 200, 213, 214, 234, 237, 241, 244.
mp_take_double: 157, 215, 217, 238.
mp_take_scaled: 208.
mp_text_code: 190, 230, 232, 234, 237, 241, 244.
mp_text_object: 187, 188, 190, 200, 234, 237, 241, 244.
mp_time: 71, 172.
mp_tolower: 4.
mp_uninitialized_model: 195.
mp_unmark_font: 136, 141, 142, 146, 231.
mp_unused: 138, 142, 151, 152, 154.
mp_used: 138, 140, 156.
mp_void: 136, 195, 232.
mp_warn: 49, 50, 52, 53, 90, 96, 97, 98, 99, 103, 108, 144.
mp_write_enc: 22, 32.
mp_write_font_definition: 131, 132, 144.
mp_xfree: 6, 21, 26, 32, 35, 40, 53, 58, 60, 61, 62, 66, 71, 73, 74, 77, 93, 98, 99, 101, 105, 111, 126, 128, 132, 134, 144, 146, 172, 178, 185, 194, 195, 227, 232, 241, 242.
mp_xmalloc: 6, 16, 21, 27, 35, 40, 61, 66, 71, 77, 90, 97, 98, 99, 103, 111, 113, 120, 144, 146, 175, 186, 190, 195, 223.
mp_xrealloc: 90.
mp_xstrcmp: 21, 126, 128, 130, 132, 134, 148.
mp_xstrdup: 21, 27, 40, 57, 60, 61, 66, 71, 74, 93, 97, 98, 99, 120, 244.
mp_xstrldup: 244.
mp_year: 71, 172.
MPLIBPS_H: 173.
MPPSOUT_H: 3.
msg: 21, 103, 120.
myname: 21.
n: 21, 51, 53, 71, 98, 109, 113.
nam: 108.
name_string: 71.
name_type: 222.
names_count: 21.
nargs: 81, 97, 98, 113.
negate: 1, 14, 16, 221.
new_ff_entry: 40, 57.
new_fm_entry: 40, 52.
new_line: 98.
next: 111, 113, 174, 187, 188.
next_size: 136, 137, 144, 146, 229, 230.
nn: 90.
NODE_BODY: 228.
non_tfm_found: 37.
noninteractive: 35.
nontfm: 39, 49, 56.
not_avail_tfm_found: 37.
notdef: 21, 24, 26, 27, 74, 78, 93, 94, 98, 99, 101, 105.
null: 1, 136, 146, 148, 195, 223, 225, 227, 231, 232.
null_font: 1, 32, 40, 54, 126, 128, 130, 136, 144, 146, 147, 148, 160, 227, 230, 231, 232, 234.
num: 113.
number_greater: 1.
number_positive: 1, 164, 201, 234.
number_to_scaled: 1, 71, 234.
number_zero: 1, 213.
object_color_a: 200.

object_color_b: 200.
object_color_c: 200.
object_color_d: 200.
object_color_model: 200.
objnum: 18, 22, 27, 32.
odd: 1.
offset: 183, 208, 210.
oldcrc: 74.
open_file: 21, 53, 96.
open_name_prefix: 90, 96.
orig_x: 103, 107, 108, 113.
orig_y: 103, 107, 108, 113.
originator: 174.
outbuf: 11, 14.
output_file: 9, 11, 14, 90, 234.
p: 21, 26, 27, 40, 45, 46, 47, 52, 58, 89, 90, 93, 97, 98, 99, 151, 156, 190, 195, 196, 197, 220, 221, 230, 234, 240, 241, 242, 243, 244.
pa: 26, 27, 45, 46, 47.
parent: 113, 188, 234.
path_p: 187, 188, 197, 203.
pb: 26, 45, 46, 47.
pdfname: 91.
pen_is_elliptical: 197, 233, 234.
pen_p: 187, 188.
pid: 40, 50, 52, 115.
plain: 89.
post_script: 187, 188.
POST_SUBRS_SCAN: 98.
pp: 107, 108, 111, 113, 176, 197, 202.
pq: 200.
pre_script: 187, 188.
previous_field: 191.
print: 13.
print_buf: 75.
print_int: 16.
print_nl: 13.
printf: 113.
PRINTF_BUF_SIZE: 75.
procset: 125, 126, 127, 128, 143, 144, 159, 160, 233, 234, 235, 236.
prologues: 65, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 144, 145, 146, 159, 160, 171, 172, 232, 233, 234, 235, 236.
prologues primitive: 159.
ps: 6, 8, 9, 10, 11, 12, 13, 20, 21, 25, 27, 30, 34, 35, 38, 44, 48, 49, 52, 53, 56, 57, 58, 61, 62, 64, 66, 70, 71, 73, 74, 75, 76, 77, 85, 88, 89, 90, 93, 94, 96, 97, 98, 99, 101, 103, 126, 128, 130, 148, 166, 191, 193, 194, 195, 197, 234.
ps_name: 32, 40, 46, 49, 50, 52, 74, 98, 99, 103, 115, 118, 120, 122, 124.
ps_offset: 7, 8, 9, 10, 11, 12, 13, 126, 128, 130, 148, 166, 197.
ps_print_double_scaled: 16.
ps_room: 12, 162, 164, 170, 179, 198, 199, 200, 201, 208, 238.
ps_tab_file: 65.
ps_tab_name: 65, 66.
ps_tree: 43, 44, 46, 48, 49, 57, 64, 115.
psout_data: 2.
psout_data_struct: 2, 3, 6.
ptr: 97, 98, 99, 105, 113.
p1: 46, 143.
p2: 46.
q: 27, 40, 52, 186, 242, 244.
qprocset: 234.
qprologues: 234.
qq: 176.
qqqq: 74, 140, 142, 151, 152, 154, 156.
quarterword: 1, 144, 146, 149, 153, 154, 191, 197, 198, 199, 204, 205, 222, 223, 224, 225, 234.
r: 4, 21, 40, 52, 61, 89, 93, 99, 221.
read_binary_file: 20, 35, 77.
read_encoding_only: 79, 90, 99.
read_field: 40, 52.
read_only: 98.
red_field: 191.
ref_x: 113.
ref_y: 113.
remove_eol: 17, 21, 89, 93.
res: 42, 113.
RESTART: 21, 90.
ret: 74, 210, 211.
return_cs: 99.
right_type: 111, 174.
right_x: 111, 174.
right_y: 111, 174.
RLINETO: 113.
RMOVETO: 113.
round_unscaled: 1, 172.
rr: 111.
RRCURVETO: 113.
S: 17, 99.
s: 10, 13, 16, 22, 27, 49, 50, 51, 52, 53, 58, 60, 66, 71, 89, 90, 93, 97, 98, 99, 108, 112, 113, 120, 126, 128, 132, 134, 139, 140, 144, 164, 165, 166, 167, 168, 169, 170, 172, 213, 222, 223.
save_selector: 21, 53, 99.
SBW: 113.
sbx: 113.
sby: 113.
sc_factor: 157, 223, 225.
*sc_factor_*: 157, 228.

scaled: 217.
scf: 234, 238, 239.
SEAC: 113.
SEEK_END: 35, 77.
SEEK_SET: 35, 77.
selector: 21, 53, 99.
set_basefont: 52, 117.
set_cc: 97.
set_color_objects: 200.
set_field: 40, 52.
set_included: 52, 117.
set_ljoin_miterlim: 199.
set_pslink: 49.
SET_SUBR_MAX: 99.
set_subsetted: 52, 117.
set_tfmlink: 49.
set_truetype: 52, 117.
SETCURRENTPOINT: 113.
setjmp: 234.
setwidthcommand: 201.
size: 17, 190.
size_index: 187, 188.
size_pos: 99.
skip: 17, 21, 40, 52, 89, 93.
slant: 40, 46, 50, 52, 102, 103, 115, 117, 122.
slen: 71.
SMALL_ARRAY_SIZE: 71, 74.
SMALL_BUF_SIZE: 40, 71, 98.
sprintf: 71.
sqrt: 202, 221.
ss: 9, 11, 12, 16, 60, 71, 90.
ss_cur: 90.
sscanf: 52, 89, 93.
ss1: 4.
ss2: 4.
stack_error: 97.
stack_ptr: 97, 98, 113.
standalone: 233, 234.
standard_glyph_names: 78, 93, 98, 113.
start_line: 99.
start_subpath: 111, 112, 113.
stdout: 113.
STEMV_CODE: 93.
store_cs: 97, 99.
store_subr: 97, 98.
str_prefix: 17, 21, 52, 89, 93.
str_suffix: 89.
strcat: 71, 98, 99.
strchr: 17, 21, 93.
strcmp: 4, 26, 45, 46, 47, 49, 51, 56, 74, 93, 98, 99, 113.
strcpy: 74, 90, 98.
strdup: 27, 40, 94.
strend: 17, 40, 52, 71, 74, 89, 90, 99.
strlen: 12, 17, 22, 51, 52, 71, 74, 90, 93, 98, 99, 120, 126, 128, 130, 148, 164, 170.
strncat: 98.
strncmp: 17.
strncpy: 11, 40, 93.
stroke command: 207.
strstr: 89, 93, 98, 99.
subr: 97, 98, 110, 113.
SUBR: 97.
subr_array_end: 84, 98, 99, 101.
subr_array_start: 84, 98, 99, 101.
subr_max: 84, 98, 99.
subr_size: 84, 97, 98, 99, 102, 103, 105.
subr_size_pos: 84, 98, 99, 103.
subr_tab: 84, 97, 98, 99, 102, 103, 105, 113.
subset_tag: 40, 74, 93, 98, 115, 120, 124.
s1: 4, 17, 89.
s2: 4, 17, 89.
t: 172.
tab: 99.
tag: 74.
tb: 244.
tc: 244.
tex_font: 74, 75, 93, 96, 98, 99, 103, 104.
text_l: 187, 188.
text_p: 187, 188.
tf: 241, 244.
tfm: 56.
tfm_avail: 40, 54, 115.
TFM_FOUND: 36, 54.
tfm_name: 40, 45, 46, 49, 50, 52, 56, 74, 115.
TFM_NOTFOUND: 36.
tfm_num: 40, 49, 54, 115.
tfm_tree: 43, 44, 45, 48, 49, 53, 56, 57, 64, 115.
TFM_UNCHECKED: 36, 40.
tfmname: 40.
tmp: 27, 56, 57.
to_scaled: 1.
tounicode: 18, 27.
tp: 244.
transformed: 213, 214, 215, 234, 238, 239.
true: 1, 21, 51, 52, 74, 75, 89, 90, 93, 96, 97, 98, 99, 103, 113, 118, 120, 124, 126, 128, 130, 132, 136, 144, 146, 156, 160, 168, 179, 180, 205, 207, 210, 211, 212, 214, 215, 234, 236.
truncate command: 201.
ts: 198, 199, 241, 244.
tt: 241, 244.
tx: 187, 188, 197, 203.
txx: 187, 188, 213, 214, 215, 216, 217.

txy: 187, 188, 213, 214, 215, 216, 217.
ty: 187, 188, 197, 203.
type: 36, 40, 52, 53, 58, 61, 66, 115, 117, 187, 188, 189, 190.
types: 111, 174.
tyx: 187, 188, 213, 214, 215, 216, 217.
tyy: 187, 188, 213, 214, 215, 216, 217.
t1_block_length: 87, 89, 90.
t1_buf: 90, 98.
t1_buf_array: 84, 85, 89, 90, 93, 97, 98, 101.
t1_buf_entry: 83, 84.
t1_buf_limit: 84, 101.
t1_buf_prefix: 89, 97.
t1_buf_ptr: 84, 89, 90, 97, 101.
T1_BUF_SIZE: 80, 90, 93, 98.
t1_buf_suffix: 89, 97.
t1_builtin_enc: 93, 98, 103.
t1_builtin_glyph_names: 79, 93, 94, 98, 101, 103.
t1_byte_length: 75, 76, 77.
t1_byte_waiting: 75, 76, 77.
t1_bytes: 75, 76, 77.
t1_char: 75, 98.
t1_charstrings: 89, 96, 98.
t1_check_block_len: 90.
t1_check_end: 93, 96, 99.
t1_check_pfa: 89, 90.
t1_cleartomark: 89, 96, 99.
t1_close: 77, 90.
t1_close_font_file: 90, 99, 103.
t1_cs: 87, 90, 96, 98.
t1_cs_flush: 98, 99.
t1_cslen: 82, 90, 97, 98, 99.
t1_c1: 82, 89.
t1_c2: 82, 89.
t1_do_subset_charstrings: 99, 103.
t1_dr: 82, 89, 90.
t1_eexec_encrypt: 75, 87, 90.
t1_encoding: 79, 93, 98.
t1_end_eexec: 89, 96, 99.
t1_eof: 77, 89, 90, 93.
t1_er: 82, 89, 90.
t1_file: 77, 87, 96.
t1_flush_cs: 98, 99.
t1_free: 6, 100, 101, 105.
t1_getbyte: 89, 90.
t1_getchar: 77, 89.
t1_getline: 90, 93, 96, 98, 99, 103.
t1_glyph_names: 79, 98, 99, 102, 103, 108.
t1_in_eexec: 87, 90, 96, 98, 103.
t1_include: 96, 99.
t1_init_params: 90, 96.
t1_lenIV: 82, 90, 93, 98, 99, 102, 103, 113.
t1_line: 90, 93.
t1_line_array: 84, 85, 89, 90, 93, 97, 98, 99, 101.
t1_line_entry: 83, 84, 90.
t1_line_limit: 84, 90, 98, 99, 101.
t1_line_ptr: 84, 89, 90, 93, 98, 99, 101.
t1_log: 75, 90.
t1_mark_glyphs: 98, 99.
t1_modify_fm: 90, 93.
t1_modify_italic: 90, 93.
t1_open_fontfile: 96, 99, 103.
t1_pfa: 87, 89, 90.
t1_prefix: 89, 93, 98, 103.
t1_putchar: 75.
t1_putline: 90, 93, 96, 98, 99.
t1_puts: 90, 98.
t1_read_subrs: 98, 99, 103.
t1_scan: 87, 90, 93, 98.
t1_scan_keys: 93.
t1_scan_num: 89, 90, 93, 97, 98, 99.
t1_scan_only: 96, 99.
t1_scan_param: 93, 96, 98, 103.
t1_start_eexec: 90, 96, 99, 103.
t1_stop_eexec: 90, 96, 99.
t1_subr_flush: 98, 99.
t1_subrs: 89, 96, 98.
t1_subset_ascii_part: 98, 99.
t1_subset_charstrings: 99.
t1_subset_end: 99.
t1_suffix: 89, 90, 93.
t1_synthetic: 87, 90, 98, 99, 103.
t1_ungetchar: 77, 89.
t1_updatefm: 99, 120.
t1name: 91, 93.
u: 52.
unity: 1, 213, 215, 216, 217.
unityold: 16.
unwrap_file: 35, 77.
update_subset_tag: 75.
v: 52.
va_end: 98.
va_start: 98.
valid: 81, 91, 93, 97, 98, 99, 113.
valid_code: 35, 77, 93.
value: 16, 91, 93.
VHCURVETO: 113.
VLINETO: 113.
VMOVETO: 113.
vsprintf: 98.
VSTEM: 113.
VSTEM3: 113.
warn: 50.
warn_s: 52.

width: 187, 188.
width_field: 191.
wps: 9.
wps_chr: 9, 10, 75.
wps_cr: 9, 10, 89.
wps_ln: 9.
write_ascii_file: 9, 11, 14, 90.
write_enc: 22.
writet1: 99, 124.
wterm_cr: 9.
ww: 197, 201, 203.
wx: 113, 197, 202, 203.
wy: 113, 197, 202, 203.
x: 162, 163.
x_coord: 111, 174.
x_loc: 204.
XHEIGHT_CODE: 93.
xstrdup: 52.
xtalloc: 98.
x0: 113.
x1: 113.
x2: 113.
y: 162, 163.
y_coord: 111, 174.
y_loc: 204.
y0: 113.
y1: 113.
y2: 113.
z: 205.
Z_NULL: 71, 74.
ZCONF_H: 68.
zero_t: 1.
zhi: 205, 206.
zlib: 68.
zlo: 205, 206.
zoff: 204, 205.

⟨ Adjust the transformation to account for *gs_width* and output the initial **gsave** if *transformed* should be *true* 215 ⟩ Used in section 214.
⟨ Character *k* is not allowed in PostScript output 161 ⟩ Used in section 166.
⟨ Compare *dash_list*(*h*) and *dash_list*(*hh*) 211 ⟩ Used in section 210.
⟨ Dealloc variables 62, 73, 194 ⟩ Used in section 6.
⟨ Decide whether the line cap parameter matters and set it if necessary 198 ⟩ Used in section 197.
⟨ Declarations 29, 31, 39, 41, 63, 92, 100, 110, 112, 117, 119, 121, 123, 125, 127, 129, 131, 133, 139, 141, 143, 145, 150, 155, 159, 163, 165, 167, 169, 171, 177, 184, 196, 204, 209, 212, 218, 220, 222, 224, 226, 229 ⟩ Used in section 1.
⟨ Embed fonts that are available 136 ⟩ Used in section 135.
⟨ Exported function headers 5, 55, 59, 65, 72, 104, 106, 109, 189, 233 ⟩ Used in section 3.
⟨ Generate PostScript code that sets the stroke width to the appropriate rounded value 201 ⟩ Used in section 197.
⟨ Give a `DocumentFonts` comment listing all fonts with non-null *font_sizes* and eliminate duplicates 148 ⟩ Used in section 146.
⟨ Globals 7, 19, 23, 33, 37, 43, 69, 75, 79, 82, 84, 87, 116, 192 ⟩ Used in section 3.
⟨ Include encodings and fonts for edge structure *h* 135 ⟩ Used in section 144.
⟨ Increment *next_size* and apply *mark_string_chars* to all text nodes with that size index 137 ⟩ Used in sections 136 and 146.
⟨ Internal Postscript header information 182, 183, 188, 235, 240, 243 ⟩ Used in section 173.
⟨ Issue PostScript commands to transform the coordinate system 216 ⟩ Used in section 213.
⟨ Make sure PostScript will use the right color for object *p* 200 ⟩ Used in section 197.
⟨ Make sure PostScript will use the right dash pattern for *dash_p*(*p*) 207 ⟩ Used in section 197.
⟨ Make *cur_fsize* a copy of the *font_sizes* array 147 ⟩ Used in sections 136 and 146.
⟨ Make *zlo* .. *zhi* include *z* and **return** *false* if *zhi* − *zlo* > *dz* 206 ⟩ Used in section 205.
⟨ Print a hexadecimal encoding of the marks for characters *bc* .. *ec* 154 ⟩ Used in section 151.
⟨ Print the `%*Font` comment for font *f* and advance *cur_fsize*[*f*] 157 ⟩ Used in section 146.
⟨ Print the initial label indicating that the bitmap starts at *bc* 153 ⟩ Used in section 151.
⟨ Print the procset 158 ⟩ Used in sections 144 and 160.
⟨ Print the size information and PostScript commands for text node *p* 238 ⟩ Used in section 234.
⟨ Restrict the range *bc* .. *ec* so that it contains no unused characters at either end 152 ⟩ Used in section 151.
⟨ Scan all the text nodes and mark the used characters 232 ⟩ Used in section 234.
⟨ Set initial values 8, 25, 34, 38, 44, 70, 76, 85, 88, 94, 193 ⟩ Used in section 6.
⟨ Set the dash pattern from *dash_list*(*hh*) scaled by *scf* 208 ⟩ Used in section 207.
⟨ Set the other numeric parameters as needed for object *p* 199 ⟩ Used in section 197.
⟨ Set *curved*: = *false* if the cubic from *p* to *q* is almost straight 181 ⟩ Used in section 180.
⟨ Set *wx* and *wy* to the width and height of the bounding box for *pen_p*(*p*) 202 ⟩ Used in section 201.
⟨ Shift or transform as necessary before outputting text node *p* at scale factor *scf*; set *transformed*: = *true* if the original transformation must be restored 239 ⟩ Used in section 234.
⟨ Start a new line and print the PostScript commands for the curve from *p* to *q* 180 ⟩ Used in section 179.
⟨ Static variables in the outer block 24, 78, 86 ⟩ Used in section 1.
⟨ Tweak the transformation parameters so the transformation is nonsingular 217 ⟩ Used in section 213.
⟨ Types 18, 36, 68, 81, 83, 91, 95, 102, 115, 138, 174, 187, 191, 228 ⟩ Used in section 3.
⟨ Unmark all marked characters 231 ⟩ Used in section 234.
⟨ Use *pen_p*(*h*) to set the transformation parameters and give the initial translation 214 ⟩ Used in section 213.
⟨ Use *pen_p*(*p*) and *path_p*(*p*) to decide whether *wx* or *wy* is more important and set *adj_wx* and *ww* accordingly 203 ⟩ Used in section 201.
⟨ Variables for the charstring parser 107 ⟩ Used in section 102.
⟨ Write *pre_script* of *p* 237 ⟩ Used in section 234.
⟨ `mplibps.h` 173 ⟩
⟨ `mppsout.h` 3 ⟩

	Section	Page
44*a* Dealing with font encodings	17	11
44*b* Parsing font map files	33	20
44*c* Helper functions for Type1 fonts	68	38
Embedding Charstrings	102	75
44*d* Embedding fonts	114	87