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
#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 */
#include <w2c/config.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "mplib.h"
#include "mplibps.h" /* external header */
#include "mplibsvg.h" /* external header */
#include "mpmp.h" /* internal header */
#include "mppsout.h" /* internal header */
#include "mpsvgout.h" /* internal header */
#include "mpmath.h" /* internal header */
  ⟨ Preprocessor definitions ⟩
  ⟨ Types in the outer block 39 ⟩⟨ Declarations 29 ⟩
```

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 **typedef struct** *svgout_data_struct* ***svgout_data**.

3. ⟨ mpsvgout.h 3 ⟩ ≡

```
#ifndef MPSVGOUT_H
#define MPSVGOUT_H 1
#include "mplib.h"
#include "mpmp.h"
#include "mplibps.h"
  typedef struct svgout_data_struct {
    ⟨ Globals 6 ⟩
  } svgout_data_struct; ⟨ Exported function headers 4 ⟩
#endif
```

4. ⟨ Exported function headers 4 ⟩ ≡

```
  void mp_svg_backend_initialize(MP mp);
  void mp_svg_backend_free(MP mp);
```

See also section 60.

This code is used in section 3.

5. **void** *mp_svg_backend_initialize*(MP *mp*)
{
mp→*svg* = *mp_xmalloc*(*mp*, 1, **sizeof** (**svgout_data_struct**));
⟨ Set initial values 7 ⟩;
}
void *mp_svg_backend_free*(MP *mp*)
{
mp_xfree(*mp*→*svg*→*buf*);
mp_xfree(*mp*→*svg*);
mp→*svg* = Λ;
}

6. Writing to SVG files

This variable holds the number of characters on the current SVG file line. It could also be a boolean because right now the only interesting thing it does is keep track of whether or not we are start-of-line.

⟨ Globals 6 ⟩ ≡
size_t *file_offset*;

See also sections 12, 20, 27, and 56.

This code is used in section 3.

7. ⟨ Set initial values 7 ⟩ ≡
mp→*svg*→*file_offset* = 0;

See also sections 13, 21, and 57.

This code is used in section 5.

8. Print a newline.

static void *mp_svg_print_ln*(MP *mp*)
{
(*mp*→*write_ascii_file*)(*mp*, *mp*→*output_file*, "\n");
mp→*svg*→*file_offset* = 0;
}

9. Print a single character.

static void *mp_svg_print_char*(MP *mp*, **int** *s*)
{
char *ss*[2];
ss[0] = (**char**) *s*;
ss[1] = 0;
(*mp*→*write_ascii_file*)(*mp*, *mp*→*output_file*, (**char** ∗) *ss*);
mp→*svg*→*file_offset*++;
}

10. Print a string.

In PostScript, this used to be done in terms of *mp_svg_print_char*, but that is very expensive (in other words: slow). It should be ok to print whole strings here because line length of an XML file should not be an issue to any respectable XML processing tool.

```
static void mp_svg_print(MP mp, const char *ss)
{
  (mp->write_ascii_file)(mp, mp->output_file, ss);
  mp->svg->file_offset += strlen(ss);
}
```

11. 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_svg_print_nl(MP mp, const char *s)
{
  if (mp->svg->file_offset > 0) mp_svg_print_ln(mp);
  mp_svg_print(mp, s);
}
```

12. Many of the printing routines use a print buffer to store partial strings in before feeding the attribute value to *mp_svg_attribute*.

```
⟨ Globals 6 ⟩ +≡
  char *buf;
  unsigned loc;
  unsigned bufsize;
```

13. Start with a modest size of 256. the buffer will grow automatically when needed.

```
⟨ Set initial values 7 ⟩ +≡
  mp->svg->loc = 0;
  mp->svg->bufsize = 256;
  mp->svg->buf = mp_xmalloc(mp, mp->svg->bufsize, 1);
  memset(mp->svg->buf, 0, 256);
```

14. How to append a character or a string of characters to the end of the buffer.

```
#define append_char(A) do
  {
    if (mp→svg→loc ≡ (mp→svg→bufsize − 1)) {
      char *buffer;
      unsigned l;
      l = (unsigned)(mp→svg→bufsize + (mp→svg→bufsize ≫ 4));
      if (l > (#3FFFFFF)) {
        mp_confusion(mp, "svg buffer size");
      }
      buffer = mp_xmalloc(mp, l, 1);
      memset(buffer, 0, l);
      memcpy(buffer, mp→svg→buf, (size_t) mp→svg→bufsize);
      mp_xfree(mp→svg→buf);
      mp→svg→buf = buffer;
      mp→svg→bufsize = l;
    }
    mp→svg→buf[mp→svg→loc++] = (A);
  }
  while (0)
#define append_string(A) do
  {
    const char *ss = (A);
    while (*ss ≠ '\0') {
      append_char(*ss);
      ss++;
    }
  }
  while (0)
```

15. This function resets the buffer in preparation of the next string. The *memset* is an easy way to make sure that the old string is forgotten completely and that the new string will be zero-terminated.

```
static void mp_svg_reset_buf(MP mp)
{
  mp→svg→loc = 0;
  memset(mp→svg→buf, 0, mp→svg→bufsize);
}
```

16. Printing the buffer is a matter of printing its string, then it is reset.

```
static void mp_svg_print_buf(MP mp)
{
  mp_svg_print(mp, (char *) mp→svg→buf);
  mp_svg_reset_buf(mp);
}
```

17. The following procedure, which stores the decimal representation of a given integer n in the buffer, has been written carefully so that it works properly if $n = 0$ or if $(-n)$ would cause overflow.

```
static void mp_svg_store_int(MP mp, integer n)
{
  unsigned char dig[23];      /* digits in a number, for rounding */
  integer m;      /* used to negate n in possibly dangerous cases */
  int k = 0;      /* index to current digit; we assume that n < 10^23 */
  if (n < 0) {
    append_char('-');
    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) {
    append_char((char)('0' + dig[k]));
  }
}
```

18. METAPOST also makes use of a trivial procedure to output two digits. The following subroutine is usually called with a parameter in the range $0 \le n \le 99$, but the assignments makes sure that only the two least significant digits are printed, just in case.

```
static void mp_svg_store_dd(MP mp, integer n)
{
  char nn = (char) abs(n) % 100;
  append_char((char)('0' + (nn/10)));
  append_char((char)('0' + (nn % 10)));
}
```

19. Conversely, here is a procedure analogous to *mp_svg_store_int*. A decimal point is printed only if the value is not an integer. If there is more than one way to print the result with the optimum number of digits following the decimal point, the closest possible value is given.

The invariant relation in the **do while** loop is that a sequence of decimal digits yet to be printed will yield the original number if and only if they form a fraction f in the range $s - \delta Ł10 \cdot 2^{16} f < s$. We can stop if and only if $f = 0$ satisfies this condition; the loop will terminate before s can possibly become zero.

```
static void mp_svg_store_double(MP mp, double s)
{
  char *value, *c;
  value = mp_xmalloc(mp, 1, 32);
  mp_snprintf(value, 32, "%f", s);
  c = value;
  while (*c) {
    append_char(*c);
    c++;
  }
  free(value);
}
```

20. Output XML tags.

In order to create a nicely indented output file, the current tag nesting level needs to be remembered.

```
⟨ Globals 6 ⟩ +≡
  int level;
```

21. ⟨ Set initial values 7 ⟩ +≡

```
  mp→svg→level = 0;
```

22. Output an XML start tag.

Because start tags may carry attributes, this happens in two steps. The close function is trivial of course, but it looks nicer in the source.

```
#define mp_svg_starttag(A,B)
        {
          mp_svg_open_starttag(A,B);
          mp_svg_close_starttag(A);
        }
static void mp_svg_open_starttag(MP mp,const char *s)
{
  int l=mp->svg->level*2;
  mp_svg_print_ln(mp);
  while (l--> 0) {
    append_char(' ');
  }
  append_char('<');
  append_string(s);
  mp_svg_print_buf(mp);
  mp->svg->level++;
}
static void mp_svg_close_starttag(MP mp)
{
  mp_svg_print_char(mp,'>');
}
```

23. Output an XML end tag.

If the *indent* is true, then the end tag will appear on the next line of the SVG file, correctly indented for the current XML nesting level. If it is false, the end tag will appear immediatelu after the preceding output.

```
static void mp_svg_endtag(MP mp,const char *s,boolean indent)
{
  mp->svg->level--;
  if (indent) {
    int l=mp->svg->level*2;
    mp_svg_print_ln(mp);
    while (l--> 0) {
      append_char(' ');
    }
  }
  append_string("</");
  append_string(s);
  append_char('>');
  mp_svg_print_buf(mp);
}
```

24. Attribute. Can't play with the buffer here becase it is likely that that is the v argument.

```
static void mp_svg_attribute(MP mp, const char *s, const char *v)
{
  mp_svg_print_char(mp, ' ');
  mp_svg_print(mp, s);
  mp_svg_print(mp, "=\"");
  mp_svg_print(mp, v);
  mp_svg_print_char(mp, '"');
}
```

25. This is a test to filter out characters that are illegal in XML.

```
⟨ Character k is illegal in SVG output 25 ⟩ ≡
  (k ≤ #8) ∨ (k ≡ #B) ∨ (k ≡ #C) ∨ (k ≥ #E ∧ k ≤ #1F) ∨ (k ≥ #7F ∧ k ≤ #84) ∨ (k ≥ #86 ∧ k ≤ #9F)
```

This code is used in section 51.

26. This is test is used to switch between direct representation of characters and character references. Just in case the input string is UTF-8, allow everything except the characters that have to be quoted for XML well-formedness.

```
⟨ Character k is not allowed in SVG output 26 ⟩ ≡
  (k ≡ '&') ∨ (k ≡ '>') ∨ (k ≡ '<')
```

This code is used in section 51.

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

Because of bugs in svg rendering software, it is necessary to change the point coordinates so that there are all in the "positive" quadrant of the SVG field. This means an shift and a vertical flip.

The two correction values are calculated by the function that writes the initial $< svg >$ tag, and are stored in two globals:

```
⟨ Globals 6 ⟩ +≡
  integer dx;
  integer dy;
```

28.

```
void mp_svg_pair_out(MP mp, double x, double y)
{
  mp_svg_store_double(mp, (x + mp→svg→dx));
  append_char(' ');
  mp_svg_store_double(mp, (−(y + mp→svg→dy)));
}
```

29.

```
⟨ Declarations 29 ⟩ ≡
  void mp_svg_font_pair_out(MP mp, double x, double y);
```

See also sections 31, 33, 34, 38, 40, 43, 48, 50, 52, 54, and 58.

This code is used in section 1.

30.

```
void mp_svg_font_pair_out(MP mp, double x, double y)
{
  mp_svg_store_double(mp, (x));
  append_char(' ');
  mp_svg_store_double(mp, −(y));
}
```

31. When stroking a path with an elliptical pen, it is necessary to distort the path such that a circular pen can be used to stroke the path. The path itself is wrapped in another transformation to restore the points to their correct location (but now with a modified pen stroke).

Because all the points in the path need fixing, it makes sense to have a specific helper to write such distorted pairs of coordinates out.

```
⟨ Declarations 29 ⟩ +≡
  void mp_svg_trans_pair_out(MP mp, mp_pen_info * pen, double x, double y);
```

32.
```
void mp_svg_trans_pair_out(MP mp, mp_pen_info * pen, double x, double y)
{
  double sx, sy, rx, ry, px, py, retval, divider;
  sx = (pen→sx);
  sy = (pen→sy);
  rx = (pen→rx);
  ry = (pen→ry);
  px = ((x + mp→svg→dx));
  py = ((−(y + mp→svg→dy)));
  divider = (sx * sy − rx * ry);
  retval = (sy * px − ry * py)/divider;
  mp_svg_store_double(mp, (retval));
  append_char('␣');
  retval = (sx * py − rx * px)/divider;
  mp_svg_store_double(mp, (retval));
}
```

33.
```
⟨ Declarations 29 ⟩ +≡
  static void mp_svg_pair_out(MP mp, double x, double y);
```

34.
```
⟨ Declarations 29 ⟩ +≡
  static void mp_svg_print_initial_comment(MP mp, mp_edge_object * hh);
```

35. **void** *mp_svg_print_initial_comment*(MP *mp*, *mp_edge_object* ∗ *hh*)

```
{
  double tx, ty;
  ⟨ Print the MetaPost version and time 36 ⟩;
  mp_svg_open_starttag(mp, "svg");
  mp_svg_attribute(mp, "version", "1.1");
  mp_svg_attribute(mp, "xmlns", "http://www.w3.org/2000/svg");
  mp_svg_attribute(mp, "xmlns:xlink", "http://www.w3.org/1999/xlink");
  if (hh→minx > hh→maxx) {
    tx = 0;
    ty = 0;
    mp→svg→dx = 0;
    mp→svg→dy = 0;
  }
  else {
    tx = (hh→minx < 0 ? −hh→minx : 0) + hh→maxx;
    ty = (hh→miny < 0 ? −hh→miny : 0) + hh→maxy;
    mp→svg→dx = (hh→minx < 0 ? −hh→minx : 0);
    mp→svg→dy = (hh→miny < 0 ? −hh→miny : 0) − ty;
  }
  mp_svg_store_double(mp, tx);
  mp_svg_attribute(mp, "width", mp→svg→buf);
  mp_svg_reset_buf(mp);
  mp_svg_store_double(mp, ty);
  mp_svg_attribute(mp, "height", mp→svg→buf);
  mp_svg_reset_buf(mp);
  append_string("0␣0␣");
  mp_svg_store_double(mp, tx);
  append_char('␣');
  mp_svg_store_double(mp, ty);
  mp_svg_attribute(mp, "viewBox", mp→svg→buf);
  mp_svg_reset_buf(mp);
  mp_svg_close_starttag(mp);
  mp_svg_print_nl(mp, "<!--␣Original␣BoundingBox:␣");
  mp_svg_store_double(mp, hh→minx);
  append_char('␣');
  mp_svg_store_double(mp, hh→miny);
  append_char('␣');
  mp_svg_store_double(mp, hh→maxx);
  append_char('␣');
  mp_svg_store_double(mp, hh→maxy);
  mp_svg_print_buf(mp);
  mp_svg_print(mp, "␣-->");
}
```

36. ⟨ Print the MetaPost version and time 36 ⟩ ≡

```
{
  char *s;
  int tt;      /* scaled */
  mp_svg_print_nl(mp, "<!-- Created by MetaPost ");
  s = mp_metapost_version( );
  mp_svg_print(mp, s);
  mp_xfree(s);
  mp_svg_print(mp, " on ");
  mp_svg_store_int(mp, round_unscaled(internal_value(mp_year)));
  append_char('.');
  mp_svg_store_dd(mp, round_unscaled(internal_value(mp_month)));
  append_char('.');
  mp_svg_store_dd(mp, round_unscaled(internal_value(mp_day)));
  append_char(':');
  tt = round_unscaled(internal_value(mp_time));
  mp_svg_store_dd(mp, tt/60);
  mp_svg_store_dd(mp, tt % 60);
  mp_svg_print_buf(mp);
  mp_svg_print(mp, " -->");
}
```

This code is used in section 35.

37. Outputting a color specification.

```
#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;
  static void mp_svg_color_out(MP mp, mp_graphic_object * p){ 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_no_model) {
        append_string("black");
      }
      else {
        if (object_color_model ≡ mp_grey_model) {
          object_color_b = object_color_a;
          object_color_c = object_color_a;
        }
        else if (object_color_model ≡ mp_cmyk_model) {
          int c, m, y, k;
          c = object_color_a;
          m = object_color_b;
          y = object_color_c;
          k = object_color_d;
          object_color_a = unity − (c + k > unity ? unity : c + k);
          object_color_b = unity − (m + k > unity ? unity : m + k);
          object_color_c = unity − (y + k > unity ? unity : y + k);
        }
        append_string("rgb(");
        mp_svg_store_double(mp, (object_color_a * 100));
        append_char('%');
        append_char(',');
        mp_svg_store_double(mp, (object_color_b * 100));
        append_char('%');
        append_char(',');
        mp_svg_store_double(mp, (object_color_c * 100));
        append_char('%');
        append_char(')');
      }
      }
```

38. ⟨ Declarations 29 ⟩ +≡

```
  static void mp_svg_color_out(MP mp, mp_graphic_object * p);
```

39. This is the information that comes from a pen

⟨ Types in the outer block 39 ⟩ ≡
```
typedef struct mp_pen_info {
  double tx, ty;
  double sx, rx, ry, sy;
  double ww;
} mp_pen_info;
```
This code is used in section 1.

40. (Re)discover the characteristics of an elliptical pen

⟨ Declarations 29 ⟩ +≡
```
mp_pen_info *mp_svg_pen_info(MP mp, mp_gr_knot pp, mp_gr_knot p);
```

41. The next two constants come from the original web source. Together with the two helper functions, they will tell whether the x or the y direction of the path is the most important

```
#define aspect_bound (10/65536.0)
#define aspect_default 1
  static double coord_range_x(mp_gr_knot h, double dz)
  {
    double z;
    double zlo = 0, zhi = 0;
    mp_gr_knot f = h;
    while (h ≠ Λ) {
      z = gr_x_coord(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_right_x(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_left_x(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      h = gr_next_knot(h);
      if (h ≡ f) break;
    }
    return (zhi − zlo ≤ dz ? aspect_bound : aspect_default);
  }
  static double coord_range_y(mp_gr_knot h, double dz)
  {
    double z;
    double zlo = 0, zhi = 0;
    mp_gr_knot f = h;
    while (h ≠ Λ) {
      z = gr_y_coord(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_right_y(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      z = gr_left_y(h);
      if (z < zlo) zlo = z;
      else if (z > zhi) zhi = z;
      h = gr_next_knot(h);
      if (h ≡ f) break;
    }
    return (zhi − zlo ≤ dz ? aspect_bound : aspect_default);
  }
```

42.

```
mp_pen_info *mp_svg_pen_info(MP mp, mp_gr_knot pp, mp_gr_knot p)
{
  double wx, wy;      /* temporary pen widths, in either direction */
  struct mp_pen_info *pen;      /* return structure */
  if (p ≡ Λ) return Λ;
  pen = mp_xmalloc(mp, 1, sizeof(mp_pen_info));
  pen→rx = unity;
  pen→ry = unity;
  pen→ww = unity;
  if ((gr_right_x(p) ≡ gr_x_coord(p)) ∧ (gr_left_y(p) ≡ gr_y_coord(p))) {
    wx = fabs(gr_left_x(p) − gr_x_coord(p));
    wy = fabs(gr_right_y(p) − gr_y_coord(p));
  }
  else {
    double a, b;
    a = gr_left_x(p) − gr_x_coord(p);
    b = gr_right_x(p) − gr_x_coord(p);
    wx = sqrt(a * a + b * b);
    a = gr_left_y(p) − gr_y_coord(p);
    b = gr_right_y(p) − gr_y_coord(p);
    wy = sqrt(a * a + b * b);
  }
  if ((wy/coord_range_x(pp, wx)) ≥ (wx/coord_range_y(pp, wy))) pen→ww = wy;
  else pen→ww = wx;
  pen→tx = gr_x_coord(p);
  pen→ty = gr_y_coord(p);
  pen→sx = gr_left_x(p) − pen→tx;
  pen→rx = gr_left_y(p) − pen→ty;
  pen→ry = gr_right_x(p) − pen→tx;
  pen→sy = gr_right_y(p) − pen→ty;
  if (pen→ww ≠ unity) {
    if (pen→ww ≡ 0) {
      pen→sx = unity;
      pen→sy = unity;
    }
    else {      /* this negation is needed because the svg coordinate system differs from postscript's. */
      pen→rx = −(pen→rx/pen→ww);
      pen→ry = −(pen→ry/pen→ww);
      pen→sx = pen→sx/pen→ww;
      pen→sy = pen→sy/pen→ww;
    }
  }
  return pen;
}
```

43. 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*.

```
⟨ Declarations 29 ⟩ +≡
  static boolean mp_is_curved(mp_gr_knot p, mp_gr_knot q);
```

44.

```
#define bend_tolerance (131/65536.0) /* allow rounding error of 2 · 10^-3 */
  boolean mp_is_curved(mp_gr_knot p, mp_gr_knot q)
  {
    double d; /* a temporary value */
    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)) return 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) return false;
      }
    return true;
  }
```

45.

```
  static void mp_svg_path_out(MP mp, mp_gr_knot h)
  {
    mp_gr_knot p, q; /* for scanning the path */
    append_char('M');
    mp_svg_pair_out(mp, gr_x_coord(h), gr_y_coord(h));
    p = h;
    do {
      if (gr_right_type(p) ≡ mp_endpoint) {
        if (p ≡ h) {
          append_string("10␣0");
        }
        return;
      }
      q = gr_next_knot(p);
      if (mp_is_curved(p, q)) {
        append_char('C');
        mp_svg_pair_out(mp, gr_right_x(p), gr_right_y(p));
        append_char(',');
        mp_svg_pair_out(mp, gr_left_x(q), gr_left_y(q));
        append_char(',');
        mp_svg_pair_out(mp, gr_x_coord(q), gr_y_coord(q));
      }
      else if (q ≠ h) {
        append_char('L');
        mp_svg_pair_out(mp, gr_x_coord(q), gr_y_coord(q));
      }
      p = q;
    } while (p ≠ h);
    append_char('Z');
    append_char(0);
  }
```

46. **static void** *mp_svg_path_trans_out*(MP *mp*, *mp_gr_knot h*, **mp_pen_info** ∗*pen*)

```
{
  mp_gr_knot p, q;      /* for scanning the path */
  append_char('M');
  mp_svg_trans_pair_out(mp, pen, gr_x_coord(h), gr_y_coord(h));
  p = h;
  do {
    if (gr_right_type(p) ≡ mp_endpoint) {
      if (p ≡ h) {
        append_string("l0␣0");
      }
      return;
    }
    q = gr_next_knot(p);
    if (mp_is_curved(p, q)) {
      append_char('C');
      mp_svg_trans_pair_out(mp, pen, gr_right_x(p), gr_right_y(p));
      append_char(',');
      mp_svg_trans_pair_out(mp, pen, gr_left_x(q), gr_left_y(q));
      append_char(',');
      mp_svg_trans_pair_out(mp, pen, gr_x_coord(q), gr_y_coord(q));
    }
    else if (q ≠ h) {
      append_char('L');
      mp_svg_trans_pair_out(mp, pen, gr_x_coord(q), gr_y_coord(q));
    }
    p = q;
  } while (p ≠ h);
  append_char('Z');
  append_char(0);
}
```

47. **static void** *mp_svg_font_path_out*(MP *mp*, *mp_gr_knot h*)
{
mp_gr_knot p, *q*; /* for scanning the path */
append_char('M');
mp_svg_font_pair_out(*mp*, *gr_x_coord*(*h*), *gr_y_coord*(*h*));
p = *h*;
do {
if (*gr_right_type*(*p*) ≡ *mp_endpoint*) {
if (*p* ≡ *h*) {
append_char('l');
mp_svg_font_pair_out(*mp*, 0, 0);
}
return;
}
q = *gr_next_knot*(*p*);
if (*mp_is_curved*(*p*, *q*)) {
append_char('C');
mp_svg_font_pair_out(*mp*, *gr_right_x*(*p*), *gr_right_y*(*p*));
append_char(',');
mp_svg_font_pair_out(*mp*, *gr_left_x*(*q*), *gr_left_y*(*q*));
append_char(',');
mp_svg_font_pair_out(*mp*, *gr_x_coord*(*q*), *gr_y_coord*(*q*));
}
else if (*q* ≠ *h*) {
append_char('L');
mp_svg_font_pair_out(*mp*, *gr_x_coord*(*q*), *gr_y_coord*(*q*));
}
p = *q*;
} **while** (*p* ≠ *h*);
append_char(0);
}

48. If *prologues* := 3, any glyphs in labels will be converted into paths.

#define *do_mark*(*A*, *B*) **do**
{
if (*mp_chars* ≡ Λ) {
mp_chars = *mp_xmalloc*(*mp*, *mp*→*font_max* + 1, **sizeof**(**int** *));
memset(*mp_chars*, 0, ((*mp*→*font_max* + 1) * **sizeof**(**int** *)));
}
if (*mp_chars*[(*A*)] ≡ Λ) {
int **glfs* = *mp_xmalloc*(*mp*, 256, **sizeof**(**int**));
memset(*glfs*, 0, (256 * **sizeof**(**int**)));
mp_chars[(*A*)] = *glfs*;
}
mp_chars[(*A*)][(**int**)(*B*)] = 1;
}
while (0)

⟨ Declarations 29 ⟩ +≡
void *mp_svg_print_glyph_defs*(MP *mp*, *mp_edge_object* * *h*);

49. **void** *mp_svg_print_glyph_defs*(MP *mp*, *mp_edge_object* ∗ *h*){ *mp_graphic_object* ∗ *p*;
/∗ object index ∗/
int *k*; /∗ general purpose index ∗/
size_t *l*; /∗ a string length ∗/
int ∗∗*mp_chars* = Λ; /∗ a twodimensional array of used glyphs ∗/
mp_ps_font ∗ *f* = Λ;
mp_edge_object ∗ *ch*;
p = *h*→*body*;
while (*p* ≠ Λ) {
if ((*gr_type*(*p*) ≡ *mp_text_code*) ∧ (*gr_font_n*(*p*) ≠ *null_font*) ∧ ((*l* = *gr_text_l*(*p*)) > 0)) {
unsigned char ∗*s* = (**unsigned char** ∗) *gr_text_p*(*p*);
while (*l*−− > 0) {
do_mark(*gr_font_n*(*p*), ∗*s*);
s++;
}
}
p = *gr_link*(*p*);
}
if (*mp_chars* ≠ Λ) { *mp_svg_starttag*(*mp*, "defs"); **for** (*k* = 0; *k* ≤ (**int**) *mp*→*font_max*; *k*++) { **if**
(*mp_chars*[*k*] ≠ Λ) { **double** *scale*; /∗ the next gives rounding errors ∗/
double *ds*, *dx*, *sk*;
ds = (*mp*→*font_dsize*[*k*] + 8)/16;
scale = (1/1000.0) ∗ (*ds*);
ds = (*scale*);
dx = *ds*;
sk = 0; **for** (*l* = 0; *l* < 256; *l*++) { **if** (*mp_chars*[*k*][*l*] ≡ 1) {
if (*f* ≡ Λ) {
f = *mp_ps_font_parse*(*mp*, *k*);
if (*f* ≡ Λ) **continue**;
if (*f*→*extend* ≠ 0) {
dx = (((**double**) *f*→*extend*/1000.0) ∗ *scale*);
}
if (*f*→*slant* ≠ 0) {
sk = (((**double**) *f*→*slant*/1000.0) ∗ 90);
}
}
mp_svg_open_starttag(*mp*, "g");
append_string("scale(");
mp_svg_store_double(*mp*, *dx*/65536);
append_char(',');
mp_svg_store_double(*mp*, *ds*/65536);
append_char(')');
if (*sk* ≠ 0) {
append_string("␣skewX(");
mp_svg_store_double(*mp*, −*sk*);
append_char(')');
}
mp_svg_attribute(*mp*, "transform", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
append_string("GLYPH");
append_string(*mp*→*font_name*[*k*]);

```
append_char('_');
mp_svg_store_int(mp,(int) l);
mp_svg_attribute(mp,"id",mp→svg→buf);
mp_svg_reset_buf(mp);
mp_svg_close_starttag(mp); if (f ≠ Λ) { ch = mp_ps_font_charstring(mp,f,(int) l); if (ch ≠ Λ) {
    p = ch→body;
mp_svg_open_starttag(mp,"path");
mp_svg_attribute(mp,"style","fill-rule:␣evenodd;"); while (p ≠ Λ) {
if (mp→svg→loc > 0) mp→svg→loc−−;      /* drop a '0' */
mp_svg_font_path_out (mp, gr_path_p ( ( mp_fill_object * ) p ) ) ;
p = p→next; } mp_svg_attribute(mp,"d",mp→svg→buf);
mp_svg_reset_buf(mp);
mp_svg_close_starttag(mp);
mp_svg_endtag(mp,"path",false); } mp_gr_toss_objects(ch); } mp_svg_endtag(mp,"g",true); } }
if (f ≠ Λ) {
  mp_ps_font_free(mp,f);
  f = Λ;
}
} } mp_svg_endtag(mp,"defs",true);      /* cleanup */
for (k = 0; k < (int) mp→font_max; k++) {
  mp_xfree(mp_chars[k]);
}
mp_xfree(mp_chars); } }
```

50. Now for outputting the actual graphic objects.

⟨ Declarations 29 ⟩ +≡

```
static void mp_svg_text_out(MP mp, mp_text_object * p, int prologues);
```

51. **void** *mp_svg_text_out*(MP *mp*, *mp_text_object* ∗ *p*, **int** *prologues*){ /∗ -Wunused: char *fname; ∗/
unsigned char ∗*s*;
int *k*; /∗ a character ∗/
size_t *l*; /∗ string length ∗/
boolean transformed;
double *ds*; /∗ design size and scale factor for a text node ∗/
/∗ clang: never read: *fname* = *mp*→*font_ps_name*[*gr_font_n*(*p*)]; ∗/
s = (**unsigned char** ∗) *gr_text_p*(*p*);
l = *gr_text_l*(*p*);
transformed = (*gr_txx_val*(*p*) ≠ *unity*) ∨ (*gr_tyy_val*(*p*) ≠ *unity*) ∨ (*gr_txy_val*(*p*) ≠ 0) ∨ (*gr_tyx_val*(*p*) ≠ 0);
mp_svg_open_starttag(*mp*, "g");
if (*transformed*) {
append_string("matrix(");
mp_svg_store_double(*mp*, *gr_txx_val*(*p*));
append_char(',');
mp_svg_store_double(*mp*, −*gr_tyx_val*(*p*));
append_char(',');
mp_svg_store_double(*mp*, −*gr_txy_val*(*p*));
append_char(',');
mp_svg_store_double(*mp*, *gr_tyy_val*(*p*));
append_char(',');
}
else {
append_string("translate(");
}
mp_svg_pair_out(*mp*, *gr_tx_val*(*p*), *gr_ty_val*(*p*));
append_char(')');
mp_svg_attribute(*mp*, "transform", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
append_string("fill:␣"); *mp_svg_color_out* (*mp*, (*mp_graphic_object* ∗) *p*) ;
append_char(';');
mp_svg_attribute(*mp*, "style", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
mp_svg_close_starttag(*mp*);
if (*prologues* ≡ 3) {
double *charwd*;
double *wd* = 0.0; /∗ this is in PS design units ∗/
while (*l*−− > 0) {
k = (**int**) ∗*s*++;
mp_svg_open_starttag(*mp*, "use");
append_string("#GLYPH");
append_string(*mp*→*font_name*[*gr_font_n*(*p*)]);
append_char('_');
mp_svg_store_int(*mp*, *k*);
mp_svg_attribute(*mp*, "xlink:href", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
charwd = ((*wd*/100));
if (*charwd* ≠ 0) {
mp_svg_store_double(*mp*, *charwd*);
mp_svg_attribute(*mp*, "x", *mp*→*svg*→*buf*);

```
        mp_svg_reset_buf(mp);
      }
      wd += mp_get_char_dimension(mp, mp→font_name[gr_font_n(p)], k, 'w');
      mp_svg_close_starttag(mp);
      mp_svg_endtag(mp, "use", false);
    }
  }
  else {
    mp_svg_open_starttag(mp, "text");
    ds = (mp→font_dsize[gr_font_n(p)] + 8)/16/65536.0;
    mp_svg_store_double(mp, ds);
    mp_svg_attribute(mp, "font-size", mp→svg→buf);
    mp_svg_reset_buf(mp);
    mp_svg_close_starttag(mp);
    while (l-- > 0) {
      k = (int) *s++;
      if (⟨ Character k is illegal in SVG output 25 ⟩) {
        char S[100];
        mp_snprintf(S, 99,
            "The␣character␣%d␣cannot␣be␣output␣in␣SVG␣""unless␣prologues:=3;", k);
        mp_warn(mp, S);
      }
      else if ((⟨ Character k is not allowed in SVG output 26 ⟩)) {
        append_string("&#");
        mp_svg_store_int(mp, k);
        append_char(';');
      }
      else {
        append_char((char) k);
      }
    }
    mp_svg_print_buf(mp);
    mp_svg_endtag(mp, "text", false);
  }
  mp_svg_endtag(mp, "g", true); }
```

52. 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

$$\text{¡g¿} \cdots \text{¡/g¿}$$

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_svg_stroke_out(MP mp, mp_graphic_object * h, mp_pen_info *pen, boolean fill_also);
```

53. **void** *mp_svg_stroke_out*(MP *mp*, *mp_graphic_object* ∗ *h*, **mp_pen_info** ∗*pen*, *boolean fill_also*){
boolean transformed = *false*;
if (*pen* ≠ Λ) {
transformed = *true*;
if ((*pen*→*sx* ≡ *unity*)∧(*pen*→*rx* ≡ 0)∧(*pen*→*ry* ≡ 0)∧(*pen*→*sy* ≡ *unity*)∧(*pen*→*tx* ≡ 0)∧(*pen*→*ty* ≡ 0))
{
transformed = *false*;
}
}
if (*transformed*) {
mp_svg_open_starttag(*mp*, "g");
append_string("matrix(");
mp_svg_store_double(*mp*, *pen*→*sx*);
append_char(',');
mp_svg_store_double(*mp*, *pen*→*rx*);
append_char(',');
mp_svg_store_double(*mp*, *pen*→*ry*);
append_char(',');
mp_svg_store_double(*mp*, *pen*→*sy*);
append_char(',');
mp_svg_store_double(*mp*, *pen*→*tx*);
append_char(',');
mp_svg_store_double(*mp*, *pen*→*ty*);
append_char(')');
mp_svg_attribute(*mp*, "transform", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
mp_svg_close_starttag(*mp*);
}
mp_svg_open_starttag(*mp*, "path"); **if** (*false*) { **if** (*transformed*) *mp_svg_path_trans_out* (*mp*, *gr_path_p* ((*mp_fill_object* ∗) *h*) , *pen*) ; **else** *mp_svg_path_out* (*mp*, *gr_path_p* ((*mp_fill_object* ∗) *h*)) ;
mp_svg_attribute(*mp*, "d", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
append_string("fill:␣");
mp_svg_color_out(*mp*, *h*);
append_string(";␣stroke:␣none;");
mp_svg_attribute(*mp*, "style", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*); } **else** { **if** (*transformed*) *mp_svg_path_trans_out* (*mp*, *gr_path_p* ((*mp_stroked_object* ∗) *h*) , *pen*) ; **else** *mp_svg_path_out* (*mp*, *gr_path_p* ((*mp_stroked_object* ∗) *h*)) ;
mp_svg_attribute(*mp*, "d", *mp*→*svg*→*buf*);
mp_svg_reset_buf(*mp*);
append_string("stroke:");
mp_svg_color_out(*mp*, *h*);
append_string(";␣stroke-width:␣");
if (*pen* ≠ Λ) {
mp_svg_store_double(*mp*, *pen*→*ww*);
}
else {
append_char('0');
}
append_char(';');

```
if (gr_lcap_val(h) ≠ 0) {
  append_string("stroke-linecap:␣");
  switch (gr_lcap_val(h)) {
  case 1: append_string("round");
    break;
  case 2: append_string("square");
    break;
  default: append_string("butt");
    break;
  }
  append_char(';');
}
if (gr_type(h) ≠ mp_fill_code) { mp_dash_object * hh;
hh = gr_dash_p(h);
if (hh ≠ Λ ∧ hh→array ≠ Λ) {
  int i;
  append_string("stroke-dasharray:␣");      /* svg doesn't accept offsets */
  for (i = 0; *(hh→array + i) ≠ -1; i++) {
    mp_svg_store_double(mp, *(hh→array + i));
    append_char('␣');
  }
  append_char(';');
}
if ( gr_ljoin_val ( ( mp_stroked_object * ) h ) ≠ 0 ) { append_string("stroke-linejoin:␣"); switch
    ( gr_ljoin_val ( ( mp_stroked_object * ) h ) )
{
case 1: append_string("round");
  break;
case 2: append_string("bevel");
  break;
default: append_string("miter");
  break;
}
append_char(';'); } if ( gr_miterlim_val ( ( mp_stroked_object * ) h ) ≠ 4 * unity ) {
    append_string("stroke-miterlimit:␣"); mp_svg_store_double (mp, gr_miterlim_val ( (
    mp_stroked_object * ) h ) ) ;
append_char(';'); } } append_string("fill:␣");
if (fill_also) {
  mp_svg_color_out(mp, h);
}
else {
  append_string("none");
}
append_char(';');
mp_svg_attribute(mp, "style", mp→svg→buf);
mp_svg_reset_buf(mp); } mp_svg_close_starttag(mp);
mp_svg_endtag(mp, "path", false);
if (transformed) {
  mp_svg_endtag(mp, "g", true);
}
}
```

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

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

55.

```
void mp_svg_fill_out(MP mp, mp_gr_knot p, mp_graphic_object * h)
{
  mp_svg_open_starttag(mp, "path");
  mp_svg_path_out(mp, p);
  mp_svg_attribute(mp, "d", mp→svg→buf);
  mp_svg_reset_buf(mp);
  append_string("fill:␣");
  mp_svg_color_out(mp, h);
  append_string(";stroke:␣none;");
  mp_svg_attribute(mp, "style", mp→svg→buf);
  mp_svg_reset_buf(mp);
  mp_svg_close_starttag(mp);      /* path */
  mp_svg_endtag(mp, "path", false);
}
```

56. Clipping paths use IDs, so an extra global is needed:

```
⟨ Globals 6 ⟩ +≡
  int clipid;
```

57.

```
⟨ Set initial values 7 ⟩ +≡
  mp→svg→clipid = 0;
```

58.

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

59. **void** *mp_svg_clip_out*(MP *mp*, *mp_clip_object* ∗ *p*)

```
{
  mp→svg→clipid++;
  mp_svg_starttag(mp,"g");
  mp_svg_starttag(mp,"defs");
  mp_svg_open_starttag(mp,"clipPath");
  append_string("CLIP");
  mp_svg_store_int(mp,mp→svg→clipid);
  mp_svg_attribute(mp,"id",mp→svg→buf);
  mp_svg_reset_buf(mp);
  mp_svg_close_starttag(mp);
  mp_svg_open_starttag(mp,"path");
  mp_svg_path_out(mp,gr_path_p(p));
  mp_svg_attribute(mp,"d",mp→svg→buf);
  mp_svg_reset_buf(mp);
  mp_svg_attribute(mp,"style","fill:␣black;␣stroke:␣none;");
  mp_svg_close_starttag(mp);      /* path */
  mp_svg_endtag(mp,"path",false);
  mp_svg_endtag(mp,"clipPath",true);
  mp_svg_endtag(mp,"defs",true);
  mp_svg_open_starttag(mp,"g");
  append_string("url(#CLIP");
  mp_svg_store_int(mp,mp→svg→clipid);
  append_string(");");
  mp_svg_attribute(mp,"clip-path",mp→svg→buf);
  mp_svg_reset_buf(mp);
  mp_svg_close_starttag(mp);
}
```

60. The main output function

#define *gr_has_scripts*(*A*) (*gr_type*((*A*)) < *mp_start_clip_code*)
#define *pen_is_elliptical*(*A*) ((*A*) ≡ *gr_next_knot*((*A*)))

⟨ Exported function headers 4 ⟩ +≡

```
int mp_svg_gr_ship_out(mp_edge_object ∗ hh, int prologues, int standalone);
```

61. **int** *mp_svg_gr_ship_out*(*mp_edge_object* ∗ *hh*, **int** *qprologues*, **int** *standalone*){ *mp_graphic_object* ∗ *p*;
mp_pen_info ∗*pen* = Λ;
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;
mp_open_output_file(*mp*);
if ((*qprologues* ≥ 1) ∧ (*mp*→*last_ps_fnum* ≡ 0) ∧ *mp*→*last_fnum* > 0) *mp_read_psname_table*(*mp*);
/∗ The next seems counterintuitive, but calls from *mp_svg_ship_out* * set standalone to true, and because embedded use is likely, it is * better not to output the XML declaration in that case. ∗/
if (¬*standalone*) *mp_svg_print*(*mp*, "<?xml␣version=\"1.0\"?>");
mp_svg_print_initial_comment(*mp*, *hh*);
if (*qprologues* ≡ 3) {
mp_svg_print_glyph_defs(*mp*, *hh*);
}
p = *hh*→*body*; **while** (*p* ≠ Λ) {
if (*gr_has_scripts*(*p*)) {
⟨ Write *pre_script* of *p* 64 ⟩;
}
switch (*gr_type*(*p*)) { **case** *mp_fill_code*: { *mp_fill_object* ∗ *ph* = (*mp_fill_object* ∗) *p*;
if (*gr_pen_p*(*ph*) ≡ Λ) {
mp_svg_fill_out(*mp*, *gr_path_p*(*ph*), *p*);
}
else if (*pen_is_elliptical*(*gr_pen_p*(*ph*))) {
pen = *mp_svg_pen_info*(*mp*, *gr_path_p*(*ph*), *gr_pen_p*(*ph*));
mp_svg_stroke_out(*mp*, *p*, *pen*, *true*);
mp_xfree(*pen*);
}
else {
mp_svg_fill_out(*mp*, *gr_path_p*(*ph*), *p*);
mp_svg_fill_out(*mp*, *gr_htap_p*(*ph*), *p*);
}
} **break**; **case** *mp_stroked_code*: { *mp_stroked_object* ∗ *ph* = (*mp_stroked_object* ∗) *p*;
if (*pen_is_elliptical*(*gr_pen_p*(*ph*))) {
pen = *mp_svg_pen_info*(*mp*, *gr_path_p*(*ph*), *gr_pen_p*(*ph*));
mp_svg_stroke_out(*mp*, *p*, *pen*, *false*);
mp_xfree(*pen*);
}
else {
mp_svg_fill_out(*mp*, *gr_path_p*(*ph*), *p*);
}
} **break**; **case** *mp_text_code*: **if** ((*gr_font_n*(*p*) ≠ *null_font*) ∧ (*gr_text_l*(*p*) > 0)) { *mp_svg_text_out* (*mp*, (*mp_text_object* ∗) *p*, *qprologues*) ; } **break**; **case** *mp_start_clip_code*: *mp_svg_clip_out* (*mp*, (*mp_clip_object* ∗) *p*) ;
break;
case *mp_stop_clip_code*: *mp_svg_endtag*(*mp*, "g", *true*);
mp_svg_endtag(*mp*, "g", *true*);
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_svg_print_nl(mp, gr_pre_script(ps));
  mp_svg_print_ln(mp); } break; }      /* all cases are enumerated */
  if (gr_has_scripts(p)) {
    ⟨ Write post_script of p 65 ⟩;
  }
  p = gr_link(p); } mp_svg_endtag(mp, "svg", true);
  mp_svg_print_ln(mp);
  (mp->close_file)(mp, mp->output_file);
  return 1; }
```

62. ⟨ mplibsvg.h 62 ⟩ ≡

```
#ifndef MPLIBSVG_H
#define MPLIBSVG_H 1
#include "mplibps.h"
  int mp_svg_ship_out(mp_edge_object *hh, int prologues);
#endif
```

63.

```
  int mp_svg_ship_out(mp_edge_object *hh, int prologues)
  {
    return mp_svg_gr_ship_out(hh, prologues, (int) true);
  }
```

64.

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

⟨ Write pre_script of p 64 ⟩ ≡

```
  {
    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 61.

65.

```
#define do_write_postscript(a,b)
        { if ( ( gr_post_script ( ( b * ) a ) ) ≠ Λ ) { mp_svg_print_nl (mp, gr_post_script ( ( b * ) a ) ) ;
        mp_svg_print_ln(mp); } }
⟨ Write post_script of p 65 ⟩ ≡
  {
    if (gr_type(p) ≡ mp_fill_code) {
      do_write_postscript(p, mp_fill_object);
    }
    else if (gr_type(p) ≡ mp_stroked_code) {
      do_write_postscript(p, mp_stroked_object);
    }
    else if (gr_type(p) ≡ mp_text_code) {
      do_write_postscript(p, mp_text_object);
    }
  }
```

This code is used in section 61.